Responsive Parallel Computation: Bridging Competitive and Cooperative Threading

Stefan K. Muller **Umut A. Acar**
Robert Harper

June 2017
CMU-CS-17-107

School of Computer Science
Carnegie Mellon University
Pittsburgh, PA 15213

A version of this work appears in the proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI 2017).

This research is partially supported grants from the National Science Foundation (CCF-1320563, CCF-1408940, CCF-1629444) and European Research Council (grant ERC-2012-StG-308246), and by a gift from Microsoft Research.

Keywords: Parallelism, Cost Models, Cost Semantics, Operational Semantics, Scheduling

Abstract

Competitive and cooperative threading are widely used abstractions in computing. In competitive threading, threads are scheduled preemptively with the goal of minimizing response time, usually of interactive applications. In cooperative threading, threads are scheduled non-preemptively with the goal of maximizing throughput or minimizing the completion time, usually in compute-intensive applications, e.g. scientific computing, machine learning and AI.

Although both of these forms of threading rely on the same abstraction of a thread, they have, to date, remained largely separate forms of computing. Motivated by the recent increase in the mainstream use of multicore computers, we propose a threading model that aims to unify competitive and cooperative threading. To this end, we extend the classic graph-based cost model for cooperative threading to allow for competitive threading, and describe how such a cost model may be used in a programming language by presenting a language and a corresponding cost semantics. Finally, we show that the cost model and the semantics are realizable by presenting an operational semantics for the language that specifies the behavior of an implementation, as well as an implementation and a small empirical evaluation.

1 Introduction

The idea of multiple threads sharing an address space is one of the most widely applicable abstractions in computer science. Over many years of research and practice, two forms of threading have emerged: competitive threading and cooperative threading. Although they both rely on essentially the same abstraction of threads, these two forms of threading differ and complement each other in their domain of applications, the form of scheduling that they use, and their performance goals, as summarized by the table below.

	Application	Scheduling	Goal
Competitive	Interactive	Preempt.	Responsiveness
Cooperative	Parallel	Non-preemp.	Throughput

Broadly used in interactive systems [33], the work on competitive threads goes back to early systems such as Xerox's STAR [56] and Cedar [59]. Such systems rely on threads to implement responsive interaction between the different components of the systems (e.g., I/O subsystem, the network) and between the system and the users [33]. Maximizing *responsiveness* is the main performance goal in interactive systems, since this is key to the user experience. To this end, threads are scheduled preemptively, often based on priorities [33, 24, 7, 27].

Cooperative threading is broadly used in fine-grained parallelism, and its use goes back to early parallel programming languages such as Id [5] and Multilisp [30], but it has regained fresh popularity with the increasing mainstream availability of multicore computers. Parallel applications, usually drawn from areas such scientific computing, physical simulations, machine learning and AI, and discrete optimization, are usually compute-intensive and use threads to reduce execution time. To this end, they break up the computation into smaller threads that can be run in parallel and rely on a non-preemptive scheduler to map the threads onto processors. The goal of the scheduler is to minimize the execution time of a parallel application by maximizing *throughput*.

It is technically possible to use competitive threading to implement parallel programs by, for example, creating a small number of system threads and manually scheduling the work of an application over them. This approach, however, can result in complex, low-level, and error-prone code. There has therefore been much work on specialized programming languages and language extensions for parallel systems, including NESL [10], OpenMP, Cilk [26], Fork/Join Java [41], X10 [18], TBB [36], TPL [42], parallel Haskell [17, 39], parallel ML [25, 37, 50] and Habanero Java [35]. To ensure high performance, these systems rely on non-preemptive schedulers such as work stealing [13, 4, 2], depth-first schedulers [11], and priority schedulers [34].

As shared memory multicore computers have become the common platform for essentially all applications, ranging from compute-intensive to interactive, many applications would benefit from a threading model that bridges competitive and cooperative threading. In such a model, an application can create both competitive and cooperative threads and expect them to be scheduled optimally, that is, to maximize both throughput and responsiveness. For example, an application that interacts with a user as it also performs parallel compute-intensive tasks mixes throughput-oriented parallel computation with responsiveness-oriented interaction.

In this paper, we propose a language and accompanying cost model that combines the competitive and cooperative threading models. We build on a popular graph-based cost model for parallel computing (e.g. [38, 13]), which goes back to the 1960s [28], in which an execution of a parallel program is represented with a Directed Acyclic Graph (DAG or simply dag). We extend this model (Section 2) to allow instructions to be assigned priorities, *foreground* and *background*, which correspond to high-priority and low-priority. We then present a scheduling principle, called *prompt scheduling*, which generalizes the standard *greedy scheduling* [15, 4] to bound both the run-time and responsiveness of a computation. To establish the bound, we make an important assumption that requires the absence of priority inversions in which high-priority computations depend on low-priority ones.

Like all other dag-based cost models, our model allows reasoning about run-time and responsiveness but it leaves an important gap: it only applies to a specific execution of the program rather than the program. To close this gap, we present a small core language (Section 3), called λ^{ip}, which we equip with a cost semantics, following prior language-based cost models [8, 9, 29]. The language supports cooperative threading based on the popular fork-join paradigm and competitive threading based on two constructs for associating priorities with computations. Furthermore, it has a type system based on linear temporal logic that guarantees that well-typed programs avoid the above-mentioned priority inversions. The result is the ability to reason about cost at the level of the program rather than that of the execution.

The dag-based cost models and the cost semantics are both abstract notions of cost that have little value unless they can be realized by an implementation. We show that the cost semantics of λ^{ip} is theoretically *realizable* by giving a transition system (Section 4) that specifies the implementation of a language runtime, and proving that the operational semantics matches the cost semantics. The operational semantics captures important details of an implementation and can be implemented on a modern multicore machine by providing a scheduling algorithm. We briefly describe (Section 5) how such a scheduling algorithm may be implemented.

Finally, we present a prototype implementation of the proposed techniques as an extension to the MLton compiler for Standard ML and perform a small empirical evaluation. Our results show that our theoretical bounds predict the practical run-time and responsiveness of a number of interactive parallel programs.

2 The DAG Model and Prompt Scheduling

The Standard DAG Model. It is common to represent parallel computations using directed acyclic graphs or *dags*. Vertices of the dag represent instructions of the computation, each of which executes in one unit of time, which we call a *step*. Edges represent dependencies between instructions: an edge from u to u' indicates that the instruction represented by u must execute before u'. For a dag g, we write $u \leq_g u'$ to indicate that u is an ancestor of u' in g. When it is clear from the context, we drop g and simply write $u \leq u'$.

For example, consider the function `fib(n)`, which computes the n^{th} Fibonacci number by performing the two recursive subcalls `fib(n-1)` and `fib(n-2)` in parallel[1]. Figure 1 shows the code for `fib`. We can represent an execution of `fib(3)` as a dag, as shown in Figure 2. For brevity,

[1] While inefficient, this algorithm is commonly used in the literature to illustrate a simple compute-intensive parallel computation.

```
function fib n =
  if n <= 1 then n
  else
    let (a, b) = par (fib (n - 1), fib (n - 2))
    in a + b
```

Figure 1: Code for parallel Fibonnacci.

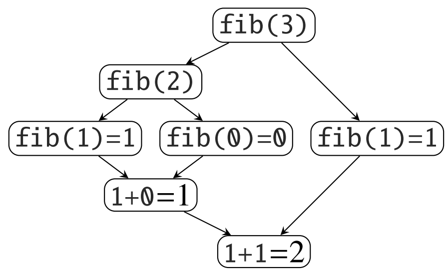

Figure 2: A dag representation of `fib(3)`.

each vertex represents a call to `fib` instead of an individual instruction, but can be expanded into a chain of instructions if desired. Vertices with out-degree two "fork" two parallel computations, which may be executed in two (cooperative) threads. Vertices with in-degree two "join" two parallel computations; a join vertex synchronizes its two in-neighbors by waiting for both of them to complete before executing.

Our Model. To model responsiveness concerns, we extend the standard dag model to allow certain portions of a dag, called *foreground blocks*, to be specified as *foreground* or high-priority computations. A foreground block is specified by its source and sink vertices. A foreground block with source s and sink t is written $\langle {s \atop t} \rangle$ and is the vertex-induced subdag of g consisting of all u such that $s \preceq_g u \preceq_g t$.

To account for the latency incurred by input operations, we weight edges with the number $\delta \in \mathbb{N}$ of steps by which the operation represented by the source vertex is delayed [45]. More specifically, for a weighted edge (u, u', δ), if $\delta = 1$, then u incurs no latency and u' may execute on the next step. If $\delta > 1$, then u incurred a latency of δ and u' may execute anytime δ steps after u starts executing.

Mathematically speaking, a dag is a tuple (s, t, V, E, F) consisting of a source vertex s, a sink vertex t, a set V of vertices (where $s, t \in V$ and $s \preceq_g t$), a set E of weighted, directed edges, and a set F of foreground blocks. We will derive run-time and responsiveness properties for dags that have no priority inversions, in which a high-priority computation depends on a low-priority one. Without this property, which we call *well-formedness*, we cannot ensure responsiveness. We say that a dag is *well-formed* if each foreground block satisfies the condition that no vertex in the block except for the source has an incoming edge from outside the block. That is, for all $\langle {s \atop t} \rangle \in F$ and all $u \neq s \in \langle {s \atop t} \rangle$, there does not exist $(u', u, \delta) \in E$ for any $u' \notin \langle {s \atop t} \rangle$.

```
function hello i =
  if i <= 0 then bg ()
  else
    let _ = output(``What is your name?'')
        x = input ()
        _ = output(``Hello, '' ^ x)
    in
      hello (i-1)

function fib_hello () = par(fib 3, fg (hello 1))
```

Figure 3: Fibonacci composed with an interactive process.

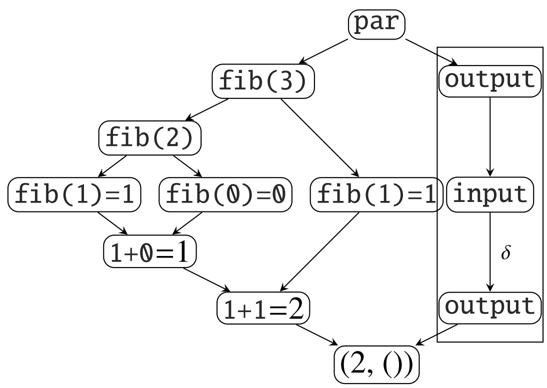

Figure 4: A dag representation of `fib_hello`.

As an example, consider the simple program shown in Figure 3. The function `fib_hello` mixes computation and interaction by computing `fib(3)` and, in parallel, asking the user a question and responding to the user's answer. The keyword `fg` indicates that the interaction should be given high priority (i.e. is a foreground computation). Figure 4 illustrates the dag for this program. The foreground computation is drawn within a box. The edge weight δ stands for the latency incurred by the input instruction. For all other edges, where the edge weight is 1, we don't explicitly write the weight.

Cost Metrics: Work, Span, Width. In parallel computing with cooperative threads, the *work* of a dag g, which we write $W(g)$, is defined as the number of vertices in the dag and *span* $S(g)$ is defined as the length of the longest path in the dag. When edge weights are used to account for latency, work remains the same as in the traditional model, because time spent blocking on inputs requires delay but no computational work. The span, on the other hand, is now the longest *weighted* path in the dag. The span takes the delays into account since the computation cannot complete until all of the inputs are available [45]. As usual, span corresponds to the time needed to complete the computation with infinitely many processors. Work now corresponds to the total active processing time of the computation. With latencies, it may not be possible to complete the computation in time $W(g)$.

The rest of this section extends the model to account for prioritized computations and uses the extensions to bound both the completion time and the responsiveness of interactive parallel computations. No additional changes are necessary to the notions of work and span beyond including edge weights in the span. However, we distinguish between total and foreground-only work and span. The bounds on the response time will involve the work and span of only the foreground blocks, reflecting the desire that the amount of low-priority computation should not affect responsiveness. For a foreground block f (which, recall, is itself a subdag of the overall dag of the computation), we write $W(f)$ and $S(f)$ for the work and span, respectively of the block. For a graph $g = (s, t, V, E, F)$, the *foreground work* $W^\circ(g)$ and *foreground span* $S^\circ(g)$ are the sum over all foreground blocks:

$$\begin{array}{lcl} W^\circ(g) & \triangleq & \sum_{f \in F} W(f) \\ S^\circ(g) & \triangleq & \sum_{f \in F} S(f) \end{array}$$

To bound the response time, we define a new notion, called *foreground width*, which intuitively corresponds to the maximum number of foreground blocks that can be executing at the same time. Formally, we say that two foreground blocks f_1 and f_2 are *serial* if there exists a directed path in the graph from a vertex of f_1 to a vertex of f_2 or vice versa. A set of foreground blocks $F' \subset F$ is *independent* if for all $f_1, f_2 \in F'$, f_1 and f_2 are not serial. The *foreground width* $D(g)$ of a graph g is

$$D(g) \triangleq \max \{|F'| \mid F' \subset F \land F' \text{ is independent}\}$$

Prompt Schedules. A schedule is an assignment of vertices to processors at each step such that if a vertex u is executed at step i, it is *ready* at step i. A vertex u is ready if all of its ancestors have executed and its latency requirements (if any) have expired. A schedule is *greedy* if as many ready vertices as possible are executed at each step. Greedy schedules suffice to minimize run-time (to within a constant factor of optimal), but not response time of high-priority computations. To reduce response time, we propose a generalization of greedy scheduling which we call *prompt scheduling*. We say that a schedule is *prompt* if it is greedy and it also gives priority to foreground blocks, executing as many foreground vertices as possible at each step (up to the number of ready foreground vertices or the number of processors).

Let T_P denote the time to execute a given parallel computation on P processors using a given schedule. Let $f = \langle {s \atop t} \rangle$ be a foreground block. Given a schedule, we define the P-processor *response time*, $R_P(f)$ as the number of steps between when s becomes ready and when t is executed (inclusive). We define the total P-processor response time, R_P, as the sum of $R_P(f)$ for all foreground blocks in the dag.

The run-time of a greedy schedule of a dag g is bounded by $\frac{W(g)}{P} + S(g)\frac{P-1}{P}$ [20, 12]. Results such as this are well-studied in the literature, and often attributed to Brent [15], who proved a similar result for "level-by-level" schedules. A similar bound exists for weighted dags such as the ones we use [45], but without the $\frac{P-1}{P}$ factor, since in this case it is possible for all processors to be idle at once, which is not possible in the traditional setting without latencies. We generalize this bound to prompt schedules, taking into account both the run-time and the response time. The intuition behind this proof, and many proofs of Brent-type theorems, is that, by definition, a greedy schedule (all prompt schedules are greedy) will either execute P instructions or execute all ready instructions (an entire "level" of the dag), decreasing the critical path by 1. To show the bound on the response time, we similarly show that, if any foreground blocks are ready, each step decreases the foreground work by P or the foreground span by the number of ready foreground blocks.

Theorem 1. *Consider a parallel computation represented by a well-formed dag g with foreground width D. If this computation is scheduled with a prompt schedule, then $T_P \leq \frac{W(g)}{P} + S(g)$ and $R_P \leq D\frac{W^\circ(g)}{P} + S^\circ(g)$.*

Proof. Since all prompt schedules are greedy, the bound on T_P follows from the run-time bound for weighted dags [45]. We now show the bound on the response time.

We split the total response time into two components R_B (for steps when all processors are **B**usy with foreground work) and R_I (for when some processors are **I**dle or not busy with foreground work), which we will bound separately by visualizing each quantity as a bucket to which tokens are added. The total response time is the total number of tokens in R_B and R_I at the end of the computation. We will also need a bucket W_B to track the "busy" component of the work. At a step i, suppose there are n_i ready foreground vertices which come from N_i foreground blocks. If $n_i \geq P$, then this is a busy step: place this step's N_i tokens in bucket R_B and P tokens in W_B. If $n_i < P$, then place N_i tokens in R_I.

Since $N_i \leq D$, for every P tokens placed in W_B, at most D tokens are placed in R_B. So, at any time, $\frac{R_B}{D} \leq \frac{W_B}{P}$.

At the end of the computation, the total number of tokens in the work bucket is at most $W^\circ(g)$ since at busy steps, a prompt schedule will execute only foreground vertices. Thus, at the end of the computation, $R_B \leq D\frac{W^\circ(g)}{P}$.

Now consider a token placed in R_I at step i. This token corresponds to a foreground block f for which at least one vertex is ready at step i. Let g_i be the sub-dag consisting of vertices of f that have not been executed after step i. Extend this in the following way to form a dag g_i^*. All vertices and edges in g_i are also in g_i^*. In addition, for all edges (u, u', δ) where u is in $f \setminus g_i$ and u' is in g_i (that is, u has been executed by the end of step i and u' has not), if u was executed in step $i - j$, add to g_i^* vertices $u_1, \ldots, u_{\delta-j-1}$ and edges $(u_1, u_2, 1), \ldots, (u_{\delta-j-1}, u, 1)$ (that is, add a chain of length $\delta - j - 1$ before u). Note that because g is well-formed, no vertex of g_i may have edges from outside f in g (except the source of f, but the source must be ready or executed at step i or no vertex of f would be ready), and so the vertices of f that are ready at the start of step $i + 1$ are exactly those vertices that are contained in g_i and have in-degree zero in g_i^*. By the definition of a prompt schedule, it must be the case that all ready vertices of f at step i are executed at step i, and so do not appear in g_{i+1}^*. In addition, for any vertex that is incurring latency at the start of step i and so has a chain before it in g_i^*, the chain is decreased by one vertex in g_{i+1}^*. Together, these facts mean that every vertex in g_i^* with in-degree zero is not present in g_{i+1}^*, and so the longest path in g_{i+1}^* is one shorter than the longest path in g_i^*. Since the longest path in g_0^*, by definition, has length $S(f)$, at most $S(f)$ tokens can be placed in R_I corresponding to f. In total, $R_I \leq S^\circ(g)$. Take the total response time to be $R_B + R_I$. □

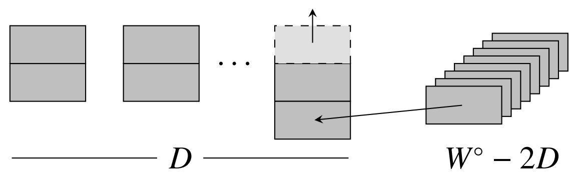

Figure 5: Intuition for the lower bound proof.

Lower Bounds for Online Scheduling. Given a parallel computation represented by a well-formed dag g, Theorem 1 gives an upper bound on the running time and the responsiveness of a prompt schedule. The run-time bound, like the similar bound for greedy schedules, is within a factor of two of optimal, because $W(g)/P$ and $S(g)$ are both, individually, lower bounds on the computation time. We now show a similar result for the bound on the response time under certain conditions: for the bound, we assume an online scheduling algorithm that has no prior knowledge of the computation dag. Specifically, we show that no matter what decisions the scheduler makes, there exists a dag whose response time is no lower than half of the given bound.

Recall that the response time is the sum over all foreground blocks f of the time taken to execute f. Since $S(f)$ is a lower bound on the time to execute f, $S^{\circ}(g)$, which is the sum of the spans over all blocks, is a lower bound on response time. Thus, to establish a 2-approximation, it suffices to show that DW°/P is also a lower bound on response time. This is the only part of the argument that relies on the online assumption on the scheduling algorithm. Consider a computation with total work $W^{\circ} + 2D$ which consists only of $D \ll W^{\circ}$ foreground blocks, each of which is sequential, and the two trees of vertices necessary to fork off and join these foreground blocks. Once all of the foreground blocks have been spawned, think of the work of the computation as W° "bricks" which are distributed arbitrarily into D stacks, as illustrated in Figure 5. At each step, a prompt scheduler will remove one brick from each of $\min(D, P)$ stacks (foreground blocks). When a stack is empty, that block is complete and no longer counts toward the response time. Since, by assumption, the scheduler only knows which blocks are ready (which stacks have a brick on top) and cannot base its decisions on how large each stack is (this would require knowing how long a block will take to execute), we may play a game against the scheduler. Start by placing two bricks on each stack. Keep the rest of the bricks hidden. At each step, when the scheduler removes a brick from a stack, place another brick at the bottom of that stack until you run out of bricks. In this way, all D blocks will be ready for at least $\frac{W^{\circ}-2D}{\min(D,P)}$ steps (the number of steps it will take to run out of bricks), which will cause the response time to be at least $D\frac{W^{\circ}-2D}{\min(D,P)} \approx D\frac{W^{\circ}}{\min(D,P)} \geq D\frac{W^{\circ}}{P}$.

3 A Language for Responsive Parallelism

We introduce a core calculus called λ^{ip}, which extends a functional core with constructs for I/O, parallelism and priority. The type system of λ^{ip} separates subcomputations by priority and enforces that high-priority computations do not depend on low-priority ones. The dynamics of λ^{ip} is given by a *cost semantics*, which computes not only the value of an expression, but also an execution dag

$$
\begin{array}{llll}
\textit{Types} & \tau & ::= & \mathtt{unit} \mid \mathtt{nat} \mid \tau_1 \to \tau_2 \mid \tau_1 \times \tau_2 \mid \tau_1 + \tau_2 \mid \bigcirc\, \tau \\
\textit{Exprs.} & e & ::= & x \mid \langle\rangle \mid \mathtt{n} \mid \lambda x{:}\tau.e \mid e\ e \mid \langle e, e\rangle \mid \mathtt{fst}(e) \mid \\
& & & \mathtt{snd}(e) \mid \mathtt{inl}(e) \mid \mathtt{inr}(e) \mid \mathtt{case}(e)\{x.e; y.e\} \mid \\
& & & \mathtt{fix}\, x{:}\tau\ \mathtt{is}\ e \mid e \parallel e \mid \\
& & & \mathtt{out}(e) \mid \mathtt{inp}[d](x.e) \mid \mathtt{bg}(e) \mid \mathtt{fg}(e)
\end{array}
$$

Figure 6: Syntax of λ^{ip}.

of the kind described in Section 2, which allows us to reason about cost at the level of the language, and to apply the prompt scheduling theorem to the run-time and responsiveness of programs.

To introduce the features of the language, we consider several simple examples, wherein we use, for convenience, "syntactic sugar," such as let binding, that can be easily expressed in λ^{ip}. We also make use of base types such as strings and booleans that are not described in the formalism.

3.1 Syntax and Examples

The syntax of λ^{ip} is given in Figure 6. Expressions e include the standard introduction and elimination forms for base types, functions, pairs and sums: natural numbers $\mathtt{n}$, unit values, λ-abstractions, application, pairs, projection, injection, and case analysis. We do not include operations (such as + and <) on natural numbers for simplicity, but these could be added in a straightforward way. The fixed point operator $\mathtt{fix}\, x{:}\tau\ \mathtt{is}\ e$ allows expressing recursion. *Parallel tuples* $e_1 \parallel e_2$ (written `par(e1, e2)` in examples) allow for fork-join parallelism: the expressions e_1 and e_2 denote parallel expressions that may be evaluated in parallel.

Input and Output. The construct $\mathtt{inp}[d](x.e)$ binds user input to the variable x in evaluating e and $\mathtt{out}(e)$ outputs the value of e to the user. The annotation d relates to the cost semantics; we ignore it for now. Our techniques do not make assumptions about how exactly input/output is performed (e.g., via a console, through GUI operations, over a network). We therefore leave these details unspecified for simplicity. Because natural numbers are the only interesting base type of λ^{ip}, only natural numbers can be input/output; generalization to other base types such as strings, as used in our examples, is straightforward. Figure 3 shows an example interactive function `hello` that asks the user questions, repeating for a number of times specified by the argument `i`.

Prioritized Computation. Now consider a parallel interactive program combining `hello` and `fib` from Figure 3:

```
function fib_hello () = par(fib 43, hello 15)
```

This code cannot guarantee responsiveness because it does not distinguish between the competitive thread, executing `hello 15`, and the many low-priority computation threads created by `fib 43`. A

```
function fib_server () =
  let n = input () in
    if n < 0 then ()
    else
      output (fib n);
      fib_server ()

function main () =
  fib_server ()
```

```
function fib_server () =
  let n = input () in
    if n < 0 then bg ()
    else
      bg (output (fib n));
      fib_server ()

function main () =
  fg (fib_server ())
```

Figure 7: Server (l) without priorities and (r) with priorities.

scheduler might get lucky, but in general, responses to the user could get arbitrarily delayed as the computation threads starve the interaction.[2]

As the `fib_hello` example illustrates, we would like to enable the programmer to distinguish between high-priority and low-priority computations. To this end, λ^{ip} provides two language constructs, `fg`(e) and `bg`(e), that represent, respectively, a *foreground computation* that runs with high priority, and a *background computation* that runs with low priority from within a foreground block. Using these constructs, we can write `fib_hello` so that it runs `hello` in the foreground as shown in Figure 3.

In the example `fib_hello`, foreground and background computations do not interact in interesting ways. For an example where they do, consider a "Fibonacci server", `fib_server`, shown in Figure 7 on the left. The function asks the user for an input n (a natural number) and computes the n^{th} Fibonacci number using `fib`. Because a Fibonacci computation performs a large amount of work, the input loop could become sluggish. In λ^{ip}, the programmer can solve this problem by running `fib_server` in the foreground and `fib` in the background, as shown on the right in Figure 7. The expression `bg (output (fib n))` spawns a new background thread to perform the Fibonacci computation asynchronously and output the result. The foreground computation can spawn many background computations, each of which computes the requested Fibonacci number in parallel with other background computations as well as the foreground interactive server loop.

3.2 Type System

As presented so far, the language allows "priority inversions" in which foreground code blocks on background computation. As an example, consider the following variant of the Fibonacci server:

```
function fib_server_bad () =
  let n = input () in
    if n < 0 then bg ()
    else let fibn = bg (fib n) in
           output (fg (fibn));
```

[2]Although we do not discuss the details in the paper, we confirmed that indeed such an implementation has poor responsiveness.

```
6             fib_server_bad ()
7
8 function main () = fg (fib_server_bad ())
```

The function `fib_server_bad` receives the input n from the user and then creates a background computation `fibn` to compute the n^{th} Fibonacci number (Line 4). It then immediately demands the result for output (Line 5). This program might not be responsive because a foreground computation (function `fib_server_bad`) is waiting on a potentially long-running background computation.

To prevent such responsiveness problems, the type system of λ^{ip} enforces a clean separation between foreground and background code, using techniques inspired by prior type systems for staged computation (Section 6). This separation is sufficient to show (in Section 3.3) that the dags corresponding to well-typed λ^{ip} programs are well-formed in the sense of Section 2 and thus, by Theorem 1, admit prompt schedules that bound responsiveness and completion time. In this section, we describe the salient aspects of the type system.

The types τ include unit and natural numbers as base types, as well as functions, binary tuples and binary sums and the circle type $\bigcirc\,\tau$, which represents background computations. Figure 8 shows the typing rules. The typing judgment has the form $\Gamma \vdash e : \tau @ w$ indicating that e has type τ at "world" w. The world is either $\mathbb{F}$ or $\mathbb{B}$, indicating that the expression is suitable for the foreground or background, respectively. Contexts Γ have entries of the form $x : \tau \mathbin{@} w$, indicating that variable x is in the context with type τ at world w.

Most of the rules allow expressions to type at any world, but require all subexpressions to be at the same world as the whole expression. Transitions between worlds are effected by the $\mathtt{bg}(e)$ and $\mathtt{fg}(e)$ operations. If e has type τ in the background (world $\mathbb{B}$), then the expression $\mathtt{bg}(e)$ has the type $\bigcirc\,\tau$ in the foreground (world $\mathbb{F}$). This allows encapsulated background computations to be created and passed around in the foreground. If e has type $\bigcirc\,\tau$ in world $\mathbb{F}$, then the expression $\mathtt{fg}(e)$ has type τ in $\mathbb{B}$. This means that the result of an encapsulated computation can *only* be demanded in the background, which precludes priority inversions. For example, this restriction will rule out the function `fib_server_bad` above, since this function is called in the foreground and the expression `fg fibn` cannot be assigned a type in the foreground.

There are two rules for typing variables. If $x : \tau \mathbin{@} w$ is in the context, the variable x has type τ at world w. We also allow variables of type `nat` to type at either world, allowing foreground code to make use of variables (of type `nat`) bound in the background and vice versa. The restriction to type `nat` ensures that code can't "escape" to the wrong world encapsulated in a function or thread. This is related to the mobility restriction of Murphy et al. [46], and could easily be expanded to allow any "mobile" type, including sums and products (but not functions or encapsulations of type $\bigcirc\,\tau$).

3.3 Cost Semantics

We now define a *cost semantics* for λ^{ip}, which both computes the value of an expression and determines an execution dag of the kind described in Section 2 for a λ^{ip} program. The parallel structure of the program, as well as the cost metrics such as work and span, can be read off from the resulting dag, and are used to reason about the run-time and responsiveness of parallel programs.

$$
\frac{}{\Gamma, x : \tau @ w \vdash x : \tau @ w}
\qquad
\frac{}{\Gamma, x : \mathtt{nat} @ w \vdash x : \tau @ w'}
\qquad
\frac{}{\Gamma \vdash \langle\rangle : \mathtt{unit} @ w}
\qquad
\frac{}{\Gamma \vdash \mathtt{n} : \mathtt{nat} @ w}
\qquad
\frac{\Gamma, x : \tau @ w \vdash e : \tau' @ w}{\Gamma \vdash \lambda x{:}\tau.e : \tau \to \tau' @ w}
$$

$$
\frac{\Gamma \vdash e_1 : \tau \to \tau' @ w \qquad \Gamma \vdash e_2 : \tau @ w}{\Gamma \vdash e_1\ e_2 : \tau' @ w}
\qquad
\frac{\Gamma \vdash e_1 : \tau_1 @ w \qquad \Gamma \vdash e_2 : \tau_2 @ w}{\Gamma \vdash \langle e_1, e_2 \rangle : \tau_1 \times \tau_2 @ w}
\qquad
\frac{\Gamma \vdash e_1 : \tau_1 @ w \qquad \Gamma \vdash e_2 : \tau_2 @ w}{\Gamma \vdash e_1 \parallel e_2 : \tau_1 \times \tau_2 @ w}
$$

$$
\frac{\Gamma \vdash e : \tau_1 \times \tau_2 @ w}{\Gamma \vdash \mathtt{fst}(e) : \tau_1 @ w}
\qquad
\frac{\Gamma \vdash e : \tau_1 \times \tau_2 @ w}{\Gamma \vdash \mathtt{snd}(e) : \tau_2 @ w}
\qquad
\frac{\Gamma \vdash e : \tau_1 @ w}{\Gamma \vdash \mathtt{inl}(e) : \tau_1 + \tau_2 @ w}
\qquad
\frac{\Gamma \vdash e : \tau_2 @ w}{\Gamma \vdash \mathtt{inr}(e) : \tau_1 + \tau_2 @ w}
$$

$$
\frac{\Gamma \vdash e : \tau_1 + \tau_2 @ w \qquad \Gamma, x : \tau_1 @ w \vdash e_1 : \tau' @ w \qquad \Gamma, y : \tau_2 @ w \vdash e_2 : \tau' @ w}{\Gamma \vdash \mathtt{case}(e)\{x.e_1; y.e_2\} : \tau' @ w}
\qquad
\frac{\Gamma, x : \tau @ w \vdash e : \tau @ w}{\Gamma \vdash \mathtt{fix}\ x{:}\tau\ \mathtt{is}\ e : \tau @ w}
$$

$$
\frac{\Gamma \vdash e : \mathtt{nat} @ w}{\Gamma \vdash \mathtt{out}(e) : \mathtt{unit} @ w}
\qquad
\frac{\Gamma, x : \mathtt{nat} @ w \vdash e : \tau @ w}{\Gamma \vdash \mathtt{inp}[d](x.e) : \tau @ w}
\qquad
\frac{\Gamma \vdash e : \tau @ \mathbb{B}}{\Gamma \vdash \mathtt{bg}(e) : \bigcirc\, \tau @ \mathbb{F}}
\qquad
\frac{\Gamma \vdash e : \bigcirc\, \tau @ \mathbb{F}}{\Gamma \vdash \mathtt{fg}(e) : \tau @ \mathbb{B}}
$$

Figure 8: Static semantics of λ^{ip}

$$
\frac{}{v \Downarrow^{\Delta} v; \emptyset}
\qquad
\frac{e_1 \Downarrow^{\Delta} \lambda x{:}\tau.e; g_1 \qquad e_2 \Downarrow^{\Delta} v; g_2 \qquad [v/x]e \Downarrow^{\Delta} v'; g_3 \qquad u\ \mathtt{fresh}}{e_1\ e_2 \Downarrow^{\Delta} v'; g_1 \oplus g_2 \oplus [u] \oplus g_3}
\qquad
\frac{e \Downarrow^{\Delta} \langle v_1, v_2 \rangle; g \qquad u\ \mathtt{fresh}}{\mathtt{fst}(e) \Downarrow^{\Delta} v_1; g \oplus [u]}
$$

$$
\frac{e \Downarrow^{\Delta} \langle v_1, v_2 \rangle; g \qquad u\ \mathtt{fresh}}{\mathtt{snd}(e) \Downarrow^{\Delta} v_2; g \oplus [u]}
\qquad
\frac{e_1 \Downarrow^{\Delta} v_1; g_1 \qquad e_2 \Downarrow^{\Delta} v_2; g_2}{\langle e_1, e_2 \rangle \Downarrow^{\Delta} \langle v_1, v_2 \rangle; g_1 \oplus g_2}
\qquad
\frac{e_1 \Downarrow^{\Delta} v_1; g_1 \qquad e_2 \Downarrow^{\Delta} v_2; g_2}{e_1 \parallel e_2 \Downarrow^{\Delta} \langle v_1, v_2 \rangle; g_1 \otimes g_2}
$$

$$
\frac{e \Downarrow^{\Delta} \mathtt{inl}(v); g_1 \qquad [v/x]e_1 \Downarrow^{\Delta} v'; g_2 \qquad u\ \mathtt{fresh}}{\mathtt{case}(e)\{x.e_1; y.e_2\} \Downarrow^{\Delta} v'; g_1 \oplus [u] \oplus g_2}
\qquad
\frac{e \Downarrow^{\Delta} \mathtt{inr}(v); g_1 \qquad [v/y]e_2 \Downarrow^{\Delta} v'; g_2 \qquad u\ \mathtt{fresh}}{\mathtt{case}(e)\{x.e_1; y.e_2\} \Downarrow^{\Delta} v'; g_1 \oplus [u] \oplus g_2}
$$

$$
\frac{e \Downarrow^{\Delta} v; g \qquad g = (s, t, V, E, F) \qquad u\ \mathtt{fresh}}{\mathtt{bg}(e) \Downarrow^{\Delta} \mathtt{thread}[t](v); g \oslash^{u}}
\qquad
\frac{e \Downarrow^{\Delta} \mathtt{thread}[u_1](v); g \qquad u_2\ \mathtt{fresh}}{\mathtt{fg}(e) \Downarrow^{\Delta} v; (g \odot) \oplus [u_2] \cup \{(u_1, u_2, 1)\}}
\qquad
\frac{e \Downarrow^{\Delta} v; g \qquad u\ \mathtt{fresh}}{\mathtt{out}(e) \Downarrow^{\Delta} \langle\rangle; g \oplus [u]}
$$

$$
\frac{[\mathtt{n}/x]e \Downarrow^{\Delta} v; g \qquad u_1\ \mathtt{fresh} \qquad u_2\ \mathtt{fresh} \qquad \delta \in \Delta(d)}{\mathtt{inp}[d](x.e) \Downarrow^{\Delta} v; [u_1] \oplus_{\delta} [u_2] \oplus g}
\qquad
\frac{[\mathtt{fix}\ x{:}\tau\ \mathtt{is}\ e/x]e \Downarrow^{\Delta} v; g \qquad u\ \mathtt{fresh}}{\mathtt{fix}\ x{:}\tau\ \mathtt{is}\ e \Downarrow^{\Delta} v; [u] \oplus g}
$$

Figure 9: Cost Semantics.

The cost semantics is given in Figure 9. The judgment $e \Downarrow^{\Delta} v; g$ states that the expression e evaluates to v and has cost graph g. The judgment is parametrized by $\Delta : \mathit{InputIDs} \to 2^{\mathbb{N}}$, a mapping which assigns a set of possible delays to each input identifier d (recall from the syntax that input operations are tagged with such identifiers). Values v consist of the unit value, numerals, lambda abstractions, pairs and injections of values, and a new form of thread handle which abstractly represents a thread as the value to which it will evaluate and a handle to the sink of its expression's cost graph:

$$
v ::= \langle\rangle \mid \mathtt{n} \mid \lambda x{:}\tau.e \mid \langle v, v \rangle \mid \mathtt{inl}(v) \mid \mathtt{inr}(v) \mid \mathtt{thread}[u](v)
$$

$$
\begin{array}{lcl}
[u] & = & (u, u, \{u\}, \emptyset, \emptyset) \\
[(u_1, u_2, \delta)] & = & (u_1, u_2, \{u_1, u_2\}, \{(u_1, u_2, \delta)\}, \emptyset) \\
(s_1, t_1, V_1, E_1, F_1) \oplus_\delta (s_2, t_2, V_2, E_2, F_2) & = & (s_1, t_2, V_1 \cup V_2, E_1 \cup E_2 \cup \{(t_1, s_2, \delta)\}, F_1 \cup F_2) \\
g_1 \oplus g_2 & = & g_1 \oplus_1 g_2 \\
(s_1, t_1, V_1, E_1, F_1) \otimes (s_2, t_2, V_2, E_2, F_2) & = & (s, t, V_1 \cup V_2 \cup \{s, t\}, E_1 \cup E_2 \cup \\
& & \quad \{(s, s_1, 1), (s, s_2, 1), (t_1, t, 1), (t_2, t, 1)\}, F_1 \cup F_2) \quad s, t \texttt{ fresh} \\
(s, t, V, E, F)\oslash^u & = & (u, u, V \cup \{u\}, E \cup \{(u, s, 1)\}, F) \\
(s, t, V, E, F)\odot & = & (s, t, V, E, F \cup \{\langle {s \atop t} \rangle\})
\end{array}
$$

Figure 10: Graph building and composition operations.

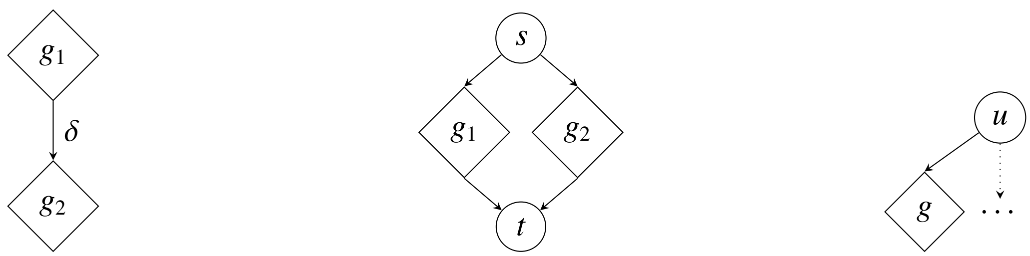

Figure 11: From left: $g_1 \oplus_\delta g_2$, $g_1 \otimes g_2$, and $g\oslash^u$.

Many of the rules for the sequential components of the language and parallel tuples are based on the cost semantics of Spoonhower et al. [58]. The rules for generating and joining with background threads (bg(e) and fg(e), respectively), are based on Spoonhower's treatment of futures [57], which share the property that an asynchronous expression is spawned in one part of a computation and demanded in another.

The generation of cost graphs is defined as part of the derivation of the evaluation judgment. A graph may consist of a single vertex, written $[u]$, or a single edge, written $[(u_1, u_2, \delta)]$, or may be formed by combining smaller graphs, which are generally produced from evaluating subexpressions. In most cases, subexpressions are evaluated sequentially, represented in the cost graph by combining the cost graphs of the subexpressions using serial composition $g_1 \oplus g_2$ which joins the sink of g_1 to the source of g_2 by an edge of weight 1 (a more general form, $\oplus_\delta$, uses an edge of weight δ, as shown in Figure 11). The empty graph $\emptyset$ is an identity for $\oplus$. In the rule for $e_1 \parallel e_2$, the cost graphs for e_1 and e_2 are combined using parallel composition $g_1 \otimes g_2$, which joins the graphs in parallel with new vertices s and t as the source and sink (Figure 11). If one of the graphs is empty, the other is simply composed with s and t.

The rule for bg(e) uses the left parallel composition operator [57]. The graph $g\oslash^u$ "hangs g off of" vertex u (Figure 11). For the purposes of sequentially composing this graph with other graphs, u is both the source and the sink, reflecting the fact that the new thread is executed concurrently with the continuation of the current thread.

The rule for fg(e) evaluates e to a background thread and also gets a handle to the sink of the cost graph for the thread's expression. The rule adds an edge between the sink and the vertex representing the fg instruction. In the rule for $\mathtt{fg}(e)$, the cost graph for e is marked as foreground with the operation $g\odot$. This operation produces a foreground block $\langle {s \atop t} \rangle$ where s and t are the source and sink of g. Finally, the input rule adds an edge of weight δ, where δ is chosen nondeterministically from $\Delta(d)$.

Figure 10 formally defines the graph formation and composition operations.

Recall that, in order to apply the results of Section 2 to the dags generated by the cost semantics, we need to show that such dags are well-formed. The well-formedness assumption requires that there are no edges to internal nodes of foreground blocks. Such an edge would correspond to a priority inversion in the language, and is ruled out by the type system, as we will now show. We first show that an expression that types in the foreground will correspond to a dag with no nested foreground blocks or external dependencies.

Lemma 1. *If $\cdot \vdash e : \tau@\mathbb{F}$ and $e \Downarrow^{\Delta} v; (s, t, V, E, F)$, then $F = \emptyset$ and for all $(u', u, \delta) \in E$, we have $u \in V$.*

Proof. By induction on the derivation of $\cdot \vdash e : \tau@\mathbb{F}$. □

This result can then be easily extended to show that well-typed programs produce well-formed dags.

Theorem 2. *If $\cdot \vdash e : \tau@w$ and $e \Downarrow^{\Delta} v; g$, g is well-formed.*

Proof. Let $g = (s, t, V, E, F)$. Proceed by induction on the derivation of $e \Downarrow^{\Delta} v; g$. The interesting case is the rule for $\mathtt{fg}(e')$, which adds a foreground block. By inversion, $e' \Downarrow^{\Delta} v'; (s', t', V', E', F')$ and by inversion on the typing rules, $\cdot \vdash e' : \bigcirc\, \tau@\mathbb{F}$. By Lemma 1, $F' = \emptyset$ and for all $(u', u, \delta) \in E'$, we have $u' \in V'$. By the cost semantics, we have $F = \{\langle {s' \atop t'} \rangle\}$ and $E = E' \cup \{(t, u_2, 1), (u_1, u_2, 1)\}$ Since no edge is added with a target in $\langle {s' \atop t'} \rangle$, there is no $u \in \langle {s' \atop t'} \rangle$ such that $(u', u, \delta) \in E$ for $u' \notin \langle {s' \atop t'} \rangle$. No other rule adds an edge to a vertex of a subdag except to its source, so well-formedness is preserved. □

4 Semantic Realization

We have thus far established bounds on the responsiveness and run-time of prompt schedules of well-formed execution dags (Theorem 1), and defined a language for prioritized interactive parallelism whose cost semantics generates only such dags. The cost model provides a theory of the responsiveness and efficiency of λ^{ip} programs with which we can derive results about programs, but these results remain abstract until we validate them with respect to a lower-level model. In this section, we give a transition semantics that specifies an implementation of λ^{ip}, and show that the cost attributed to a program by the cost model corresponds to a more concrete notion of cost in terms of steps of the transition system.

4.1 Operational Semantics

Since the operational semantics is a transition system, it must be able to represent intermediate states of computation. In particular, such an intermediate state may have many active threads of execution. We will name these threads with thread symbols, for which we use the metavariables a, b, c and variants. We use these symbols for threads generated by parallel pairs $e_1 \parallel e_2$ as well as background threads. We also introduce three new expression forms which are not needed for source programs (i.e. programs that have not begun evaluation):

$$e ::= \cdots \mid \texttt{tid}[a] \mid \texttt{join}[a, b] \mid \texttt{in}(x.e)$$

The first, $\texttt{tid}[a]$, is a runtime representation of a background thread identified by thread symbol a. The second, $\texttt{join}[a, b]$, is a parallel tuple whose components are being evaluated by threads a and b. Finally, $\texttt{in}(x.e)$ will be used to represent an input expression whose latency has expired but which has not yet produced an input value.

We also introduce *thread pools*. A thread pool μ is a mapping from thread identifiers a to pairs (δ, e) of a delay and an expression, indicating that thread a may run command e after δ steps. We write a thread pool as

$$a_1 \hookrightarrow (\delta_1, e_1) \uplus \ldots \uplus a_n \hookrightarrow (\delta_n, e_n)$$

and the concatenation of two disjoint thread pools as $\mu_1 \uplus \mu_2$.

We extend the type system to account for threads. The new typing judgment is $\Gamma \vdash_\Sigma e : \tau @ w$, which includes a thread signature Σ. Thread signatures have entries of the form $a \sim \tau \mathbin{@} w$, indicating that thread a is running an expression of type τ at world w. The modified typing rules are shown in Figure 12. Most are unchanged from Figure 8 and simply pass the thread signature through. The rules for $\texttt{join}[a, b]$ and $\texttt{tid}[b]$ look up the thread identifiers in the signature and produce the appropriate types.

A new typing judgment, $\Gamma \vdash_{\Sigma'} \mu : \Sigma$, indicates that the thread pool μ has the signature Σ. The rules require that $a \sim \tau \mathbin{@} w \in \Sigma$ if and only if $a \hookrightarrow (\delta, e) \in \mu$ and e has type τ at world w. For the purposes of typing, thread pools are ordered and threads may only refer to threads that come later in μ. This ensures that the references between threads are acyclic. However, we will occasionally treat the thread pool as the unordered set of its threads when this property is not important. Expressions are also allowed to refer to threads in Σ', which must be disjoint from Σ, allowing us to type just a part of a thread pool, whose expressions may refer to threads outside this part. Whenever μ is the entire thread pool, Σ' will be empty.

Lemma 2 states a property of thread pool typing which will be useful later: the concatenation of two thread pools is well-typed with the concatenation of the two signatures.

Lemma 2. *If* $\Gamma \vdash_{\Sigma_2, \Sigma'} \mu_1 : \Sigma_1$ *and* $\Gamma \vdash_{\Sigma'} \mu_2 : \Sigma_2$ *then* $\Gamma \vdash_{\Sigma'} \mu_1 \uplus \mu_2 : \Sigma_1, \Sigma_2$.

Proof. By induction on the derivation of $\Gamma \vdash_{\Sigma'} \mu_1 : \Sigma_1$. If $\mu_1 = \emptyset$, then the result is trivial. Otherwise, $\mu_1 = a \hookrightarrow (\delta, e) \uplus \mu_1'$ and $\Sigma_1 = \Sigma_1', a \sim \tau \mathbin{@} w$ and $\Gamma \vdash_{\Sigma', \Sigma_1'} e : \tau @ w$ and $\Gamma \vdash_{\Sigma_2, \Sigma'} \mu_1' : \Sigma_1'$. By induction, $\Gamma \vdash_{\Sigma'} \mu_1' \uplus \mu_2 : \Sigma_1', \Sigma_2$. By weakening, $\Gamma \vdash_{\Sigma', \Sigma_1', \Sigma_2} e : \tau @ w$. The result follows from the thread pool typing rules. $\square$

Expression typing $\Gamma \vdash_\Sigma e : \tau @ w$

$$\frac{}{\Gamma, x : \tau \mathrel{@} w \vdash_\Sigma x : \tau @ w} \qquad \frac{}{\Gamma, x : \mathtt{nat} \mathrel{@} w \vdash_\Sigma x : \tau @ w'} \qquad \frac{}{\Gamma \vdash_\Sigma \langle\rangle : \mathtt{unit} @ w} \qquad \frac{}{\Gamma \vdash_\Sigma \mathtt{n} : \mathtt{nat} @ w}$$

$$\frac{\Gamma, x : \tau \mathrel{@} w \vdash_\Sigma e : \tau' @ w}{\Gamma \vdash_\Sigma \lambda x{:}\tau.e : \tau \to \tau' @ w} \qquad \frac{\Gamma \vdash_\Sigma e_1 : \tau \to \tau' @ w \qquad \Gamma \vdash_\Sigma e_2 : \tau @ w}{\Gamma \vdash_\Sigma e_1\ e_2 : \tau' @ w} \qquad \frac{\Gamma \vdash_\Sigma e_1 : \tau_1 @ w \qquad \Gamma \vdash_\Sigma e_2 : \tau_2 @ w}{\Gamma \vdash_\Sigma \langle e_1, e_2 \rangle : \tau_1 \times \tau_2 @ w}$$

$$\frac{\Gamma \vdash_\Sigma e_1 : \tau_1 @ w \qquad \Gamma \vdash_\Sigma e_2 : \tau_2 @ w}{\Gamma \vdash_\Sigma e_1 \parallel e_2 : \tau_1 \times \tau_2 @ w} \qquad \frac{\Gamma \vdash_\Sigma e : \tau_1 \times \tau_2 @ w}{\Gamma \vdash_\Sigma \mathtt{fst}(e) : \tau_1 @ w} \qquad \frac{\Gamma \vdash_\Sigma e : \tau_1 \times \tau_2 @ w}{\Gamma \vdash_\Sigma \mathtt{snd}(e) : \tau_2 @ w}$$

$$\frac{}{\Gamma \vdash_{\Sigma, a \sim \tau_1 @ w, b \sim \tau_2 @ w} \mathtt{join}[a, b] : \tau_1 \times \tau_2 @ w} \qquad \frac{\Gamma \vdash_\Sigma e : \tau_1 @ w}{\Gamma \vdash_\Sigma \mathtt{inl}(e) : \tau_1 + \tau_2 @ w} \qquad \frac{\Gamma \vdash_\Sigma e : \tau_2 @ w}{\Gamma \vdash_\Sigma \mathtt{inr}(e) : \tau_1 + \tau_2 @ w}$$

$$\frac{\Gamma \vdash_\Sigma e : \tau_1 + \tau_2 @ w \qquad \Gamma, x : \tau_1 \mathrel{@} w \vdash_\Sigma e_1 : \tau' @ w \qquad \Gamma, y : \tau_2 \mathrel{@} w \vdash_\Sigma e_2 : \tau' @ w}{\Gamma \vdash_\Sigma \mathtt{case}(e)\{x.e_1; y.e_2\} : \tau' @ w}$$

$$\frac{\Gamma, x : \tau \mathrel{@} w \vdash_\Sigma e : \tau @ w}{\Gamma \vdash_\Sigma \mathtt{fix}\ x{:}\tau\ \mathtt{is}\ e : \tau @ w} \qquad \frac{\Gamma \vdash_\Sigma e : \mathtt{nat} @ w}{\Gamma \vdash_\Sigma \mathtt{out}(e) : \mathtt{unit} @ w} \qquad \frac{\Gamma, x : \mathtt{nat} \mathrel{@} w \vdash_\Sigma e : \tau @ w}{\Gamma \vdash_\Sigma \mathtt{inp}[d](x.e) : \tau @ w}$$

$$\frac{\Gamma, x : \mathtt{nat} \mathrel{@} w \vdash_\Sigma e : \tau @ w}{\Gamma \vdash_\Sigma \mathtt{in}(x.e) : \tau @ w} \qquad \frac{\Gamma \vdash_\Sigma e : \tau @ \mathbb{B}}{\Gamma \vdash_\Sigma \mathtt{bg}(e) : \bigcirc \tau @ \mathbb{F}} \qquad \frac{}{\Gamma \vdash_{\Sigma, b \sim \tau @ \mathbb{B}} \mathtt{tid}[b] : \bigcirc \tau @ \mathbb{F}} \qquad \frac{\Gamma \vdash_\Sigma e : \bigcirc \tau @ \mathbb{F}}{\Gamma \vdash_\Sigma \mathtt{fg}(e) : \tau @ \mathbb{B}}$$

Thread pool typing $\Gamma \vdash_\Sigma \mu : \Sigma'$

$$\frac{}{\Gamma \vdash_{\Sigma'} \emptyset : \cdot} \qquad \frac{\Gamma \vdash_{\Sigma, \Sigma'} e : \tau @ w \qquad \Gamma \vdash_{\Sigma'} \mu : \Sigma}{\Gamma \vdash_{\Sigma'} a \hookrightarrow (\delta, e) \uplus \mu : \Sigma, a \sim \tau \mathrel{@} w}$$

Figure 12: Extended static semantics of λ^{ip}

The operational semantics of λ^{ip} consists of two components: local and global [32]. The local semantics concerns individual threads, and indicates how expressions transition. Selected rules are presented in Figure 13. The rules in this figure correspond to two judgments. The judgment e `val` indicates that e is an irreducible value. Values are the unit value, numerals, functions, pairs and injections of values, and thread handles $\mathtt{tid}[b]$. The local transition judgment is

$$e \mid \mu \mapsto_a^\Delta (\delta', e') \mid \mu \uplus \mu'$$

which states that thread a running e transitions to e', possibly spawning new threads, which are collected in μ'. The original thread pool μ is unchanged; threads are never altered or removed by local transitions. The thread identifier a is not important for the local transition, but will be used in some of the global definitions and results. The new expression e' will be able to run after a delay of δ' steps (if $\delta' = 0$, it can run immediately). As with the cost semantics, the judgment is parametrized by a delay assignment Δ.

$$\frac{}{\langle\rangle\ \mathtt{val}} \qquad \frac{}{\mathtt{n}\ \mathtt{val}} \qquad \frac{}{\lambda x{:}\tau.e\ \mathtt{val}} \qquad \frac{e_1\ \mathtt{val} \qquad e_2\ \mathtt{val}}{\langle e_1, e_2\rangle\ \mathtt{val}} \qquad \frac{e\ \mathtt{val}}{\mathtt{inl}(e)\ \mathtt{val}} \qquad \frac{e\ \mathtt{val}}{\mathtt{inr}(e)\ \mathtt{val}} \qquad \frac{}{\mathtt{tid}[a]\ \mathtt{val}}$$

$$\frac{e_1 \mid \mu \mapsto_a^{\Delta} (\delta, e_1') \mid \mu \uplus \mu'}{e_1\ e_2 \mid \mu \mapsto_a^{\Delta} (\delta, e_1'\ e_2) \mid \mu \uplus \mu'} \qquad \frac{e_2 \mid \mu \mapsto_a^{\Delta} (\delta, e_2') \mid \mu \uplus \mu'}{(\lambda x{:}\tau.e_1)\ e_2 \mid \mu \mapsto_a^{\Delta} (\delta, (\lambda x{:}\tau.e_1)\ e_2') \mid \mu \uplus \mu'} \qquad \frac{e_2\ \mathtt{val}}{(\lambda x{:}\tau.e_1)\ e_2 \mid \mu \mapsto_a^{\Delta} (0, [e_2/x]e_1) \mid \mu}$$

$$\frac{b\ \mathtt{fresh} \qquad c\ \mathtt{fresh}}{e_1 \parallel e_2 \mid \mu \mapsto_a^{\Delta} (0, \mathtt{join}[b,c]) \mid \mu \uplus b \hookrightarrow (0, e_1) \uplus c \hookrightarrow (0, e_2)} \qquad \frac{\mu = b \hookrightarrow (\delta_b, e_b) \uplus c \hookrightarrow (\delta_c, e_c) \uplus \mu' \qquad e_b\ \mathtt{val} \qquad e_c\ \mathtt{val}}{\mathtt{join}[b,c] \mid \mu \mapsto_a^{\Delta} (0, \langle e_b, e_c\rangle) \mid \mu}$$

$$\frac{b\ \mathtt{fresh}}{\mathtt{bg}(e) \mid \mu \mapsto_a^{\Delta} (0, \mathtt{tid}[b]) \mid \mu \uplus b \hookrightarrow (0, e)} \qquad \frac{e \mid \mu \mapsto_a^{\Delta} (\delta, e') \mid \mu \uplus \mu'}{\mathtt{fg}(e) \mid \mu \mapsto_a^{\Delta} (\delta, \mathtt{fg}(e')) \mid \mu \uplus \mu'} \qquad \frac{\mu = b \hookrightarrow (\delta, e) \uplus \mu' \qquad e\ \mathtt{val}}{\mathtt{fg}(\mathtt{tid}[b]) \mid \mu \mapsto_a^{\Delta} (0, e) \mid \mu}$$

$$\frac{e \mid \mu \mapsto_a^{\Delta} (\delta, e') \mid \mu \uplus \mu'}{\mathtt{out}(e) \mid \mu \mapsto_a^{\Delta} (\delta, \mathtt{out}(e')) \mid \mu \uplus \mu'} \qquad \frac{e\ \mathtt{val}}{\mathtt{out}(e) \mid \mu \mapsto_a^{\Delta} (0, \langle\rangle) \mid \mu}$$

$$\frac{\delta \in \Delta(d)}{\mathtt{inp}[d](x.e) \mid \mu \mapsto_a^{\Delta} (\delta - 1, \mathtt{in}(x.e)) \mid \mu} \qquad \frac{}{\mathtt{in}(x.e) \mid \mu \mapsto_a^{\Delta} (0, [\mathtt{n}/x]e) \mid \mu}$$

Figure 13: Selected local dynamic rules.

Most of the transition rules are straightforward and are omitted. The complete rules for function application are given as an example: in $e_1\ e_2$, the subexpression e_1 is stepped until it is a lambda abstraction, then e_2 is stepped until it is a value, which is then substituted for the variable in the body of the abstraction using standard capture-avoiding substitution. A parallel tuple $e_1 \parallel e_2$ spawns two new threads b and c to execute e_1 and e_2, respectively. The local thread a steps to $\mathtt{join}[b, c]$, indicating that this thread is now waiting for b and c to complete. When both threads have stepped to irreducible values, $\mathtt{join}[b, c]$ steps to a pair of the two values. In the same vein, $\mathtt{bg}(e)$ spawns a new thread b to evaluate e and returns the thread handle $\mathtt{tid}[b]$. Note that, while threads spawned by parallel tuples and threads spawned by $\mathtt{bg}(e)$ are treated identically by the semantics (i.e. they are stepped with the same transitions and not distinguished in the thread pool), the threads b and c spawned by a parallel tuple are never referred to by thread handles (e.g. $\mathtt{tid}[b]$) because these threads are not first class.

The expression $\mathtt{fg}(e)$ steps e until it reaches $\mathtt{fg}(\mathtt{tid}[b])$, which then blocks until thread b has evaluated its expression down to an irreducible value e', at which point $\mathtt{fg}(\mathtt{tid}[b])$ steps to e'. The input rule is the only one which results in a delay, which is chosen nondeterministically from $\Delta(d)$. After the delay, the new expression $\mathtt{in}(x.e)$ nondeterministically chooses a natural number $\mathtt{n}$ to substitute for x in e, representing the uncertainty in the input from the user or environment.

We can prove type safety at the local level by showing that progress and preservation results hold for the local dynamics. Both lemmas have some unusual features. Local progress states that if an expression is well-typed, it is either fully evaluated or can take a step, or is waiting for some other thread using `join` or `fg` (which can take a step or is delayed).

$$\frac{}{\emptyset\ \texttt{final}} \qquad \frac{e\ \texttt{val} \qquad \mu\ \texttt{final}}{a \hookrightarrow (\delta, e) \uplus \mu\ \texttt{final}}$$

$$\frac{\begin{array}{c}\mu = a_1 \hookrightarrow (\delta_1, e_1) \uplus \ldots \uplus a_n \hookrightarrow (\delta_n, e_n) \qquad \forall N < i \leq n.\delta'_i = \max(0, \delta_i - 1) \\ N \leq n \qquad \forall 1 \leq i \leq N.\delta_i = 0 \qquad \forall 1 \leq i \leq N.e_i \mid \mu \mapsto^{\Delta}_{a_i} (\delta'_i, e'_i) \mid \mu \uplus \mu'_i\end{array}}{r; \mu \mapsto_{\texttt{glo}} r + |RFB(\mu)|; a_1 \hookrightarrow (\delta'_1, e'_1) \uplus \ldots \uplus a_N \hookrightarrow (\delta'_N, e'_N) \uplus a_{N+1} \hookrightarrow (\delta'_{N+1}, e_{N+1}) \uplus \ldots \uplus a_n \hookrightarrow (\delta'_n, e_n) \uplus \mu'_1 \uplus \ldots \uplus \mu'_N}$$

Figure 14: Global Dynamics.

Lemma 3 (Local Progress)**.** *If* $\cdot \vdash_{\Sigma} e : \tau@w$ *and* $\cdot \vdash_{\cdot} \mu : \Sigma, a \sim \tau\ @\ w$, *then either* e `val` *or* $e \mid \mu \mapsto^{\Delta}_{a} (\delta, e') \mid \mu \uplus \mu'$ *or there exists* $b \hookrightarrow (\delta_b, e_b) \in \mu$ *such that* $\delta_b > 0$ *or* $e_b \mid \mu \mapsto^{\Delta}_{b} (\delta'_b, e'_b) \mid \mu \uplus \mu'$.

Proof. By induction on the derivations of $\cdot \vdash_{\Sigma} e : \tau@w$ and $\cdot \vdash_{\cdot} \mu : \Sigma, a \sim \tau\ @\ w$. In most of the base cases, e either is a value or can step. The interesting cases are $e = \texttt{fg}(\texttt{tid}[b])$ and $e = \texttt{join}[b, c]$. Consider the case for foreground. By inversion, $b \sim \tau\ @\ \mathbb{B} \in \Sigma$ and $b \hookrightarrow (\delta_b, e_b) \in \mu$. If $\delta_b > 0$, the case is proven, so suppose $\delta_b = 0$. By induction, either (1) e_b `val` and $e \mid \mu \mapsto^{\Delta}_{a} (0, e_b) \mid \mu$ or (2) $e_b \mid \mu \mapsto^{\Delta}_{b} (\delta'_b, e'_b) \mid \mu \uplus \mu'$ or (3) there exists $c \hookrightarrow (\delta_c, e_c) \in \mu$ such that $\delta_c > 0$ or $e_c \mid \mu \mapsto^{\Delta}_{c} (\delta'_c, e'_c) \mid \mu \uplus \mu'$. In cases (2) or (3), b and c, respectively, meet the conditions in the theorem. The case for join is similar. □

The statement of preservation is more standard but requires finding a signature Σ' which accounts for the new threads that are created when e takes a step.

Lemma 4 (Local Preservation)**.** *If* $\cdot \vdash_{\Sigma} e : \tau@w$ *and* $e \mid \mu \mapsto^{\Delta}_{a} (\delta, e') \mid \mu \uplus \mu'$, *then there exists* Σ' *such that* $\cdot \vdash_{\Sigma, \Sigma'} e' : \tau@w$ *and* $\cdot \vdash_{\Sigma} \mu' : \Sigma'$.

Proof. Induction on the derivation of $e \mid \mu \mapsto^{\Delta}_{a} (\delta, e') \mid \mu \uplus \mu'$. □

The global rules in Figure 14 define the transitions of entire thread pools, i.e. the entire state of the computation. The judgment μ `final` states that μ has completed evaluating and its rules simply require that all threads in μ be irreducible. The global step relation is

$$r; \mu \mapsto_{\texttt{glo}} r'; \mu'$$

and has only one rule, which allows some number N of threads whose delay is 0 to step using the local dynamics. There is also a counter for the total response time r, which at each step is incremented by the number of ready foreground blocks[3] (the formal definition of $RFB(\mu)$, which counts the ready foreground blocks in a thread pool, is straightforward and omitted for simplicity). The new thread pool consists of the updated threads 1 through N, and the unaltered threads $N + 1$ through n with their delays (if nonzero) decremented.

Note that the global step relation does not specify a scheduling strategy, nor does it enforce any constraints on schedules other than that only ready threads may step. In our results, we will quantify

[3]Commuting the summations, counting the number of blocks at each step is equivalent to counting the number of steps taken to execute each block, which is the response time.

over valid, prompt schedules: those that step as many threads as possible, prioritizing threads that are executing foreground blocks, bounded by the number of available processors.

We can now prove progress and preservation for the global semantics. Most of the work is done by Lemmas 3 and 4.

Lemma 5 (Progress)**.** *If* $\cdot \vdash_{\cdot} \mu : \Sigma$*, then either* μ $\mathtt{final}$ *or there exist* r' *and* μ' *such that* $r; \mu \mapsto_{\mathtt{glo}} r'; \mu \uplus \mu'$.

Proof. Let $\mu = a_1 \hookrightarrow (\delta_1, e_1) \uplus \ldots \uplus a_m \hookrightarrow (\delta_m, e_m)$. If any $\delta_i > 0$, then the configuration can take a step to reduce δ_i, so consider the case where $\delta_1 = \cdots = \delta_m = 0$. By inversion on the configuration typing derivation, we have $\Sigma = a_1 \sim \tau_1 \ @\ w, \ldots, a_n \sim \tau_n \ @\ w$ and for all i, there exists Σ_i such that $\cdot \vdash_{\Sigma_i} e_i : \tau_i @ w$. By Lemma 3, either there exists some i such that $e_i \mid \mu \mapsto^{\Delta}_{a_i} (\delta'_i, e'_i) \mid \mu \uplus \mu'_i$ or for all i, e_i $\mathtt{final}$. In the former case, the configuration can take a step, and in the latter case, μ $\mathtt{final}$. ◻

Lemma 6 (Preservation)**.** *If* $\cdot \vdash_{\cdot} \mu : \Sigma$ *and* $r; \mu \mapsto_{\mathtt{glo}} r'; \mu'$*, then there exists* Σ' *such that* $\cdot \vdash_{\cdot} \mu' : \Sigma'$.

Proof. Let $\mu = a_1 \hookrightarrow (\delta_1, e_1) \uplus \ldots \uplus a_n \hookrightarrow (\delta_n, e_n)$ and $\mu' = a_1 \hookrightarrow (\delta'_1, e'_1) \uplus \mu'_1 \uplus \ldots \uplus a_n \hookrightarrow (\delta'_n, e'_n) \uplus \mu'_n$. Note that, since we are dealing with typing, we can no longer treat the thread pool as unordered, and so the indices may not correspond to those in the step rule. For each i, by inversion on typing, there exists Σ_i such that $\cdot \vdash_{\Sigma_i} e_i : \tau_i @ w$. Either $e'_i = e_i$ and $\mu'_i = \emptyset$ or, by Lemma 4, we have, for some Σ'_i, $\cdot \vdash_{\Sigma_i, \Sigma'_i} e'_i : \tau_i @ w_i$ and $\cdot \vdash_{\Sigma_i} \mu'_i : \Sigma'_i$. Let $\Sigma' = \Sigma, \Sigma'_1, \ldots, \Sigma'_n$. By weakening, $\cdot \vdash_{\Sigma_i, \Sigma'_{i+1}, \ldots, \Sigma'_n} e'_i : \tau_i @ w_i$ and $\cdot \vdash_{\Sigma_i, \Sigma'_{i+1}, \ldots, \Sigma'_n} \mu'_i : \Sigma'_i$. The result follows from the thread pool typing rules and Lemma 2. ◻

We could now show a fairly standard type safety theorem, showing that a well-typed thread pool will not become "stuck". However, there is one additional property, in addition to well-typedness, which we wish to ensure is preserved during execution. We call this property "well-joinedness". It is defined by the judgment e $\mathtt{wj}$ ("e is well-joined") in Figure 15. Intuitively, well-joinedness is the property that `join` expressions appear only in the part of an expression which is currently being evaluated[4]. In particular, they may not appear encapsulated in functions, or in expressions which have not yet been evaluated. The auxiliary judgment e $\mathtt{nj}$ ("no joins") indicates that e contains no `join` expressions. Its straightforward definition is omitted.

We first show that well-joinedness is preserved by local transitions: if all expressions of a thread pool are well-joined and one thread steps, then all resulting expressions are well-joined.

Lemma 7. *If*

$$e_0 \mid a_1 \hookrightarrow (\delta_1, e_1) \uplus \ldots \uplus a_n \hookrightarrow (\delta_n, e_n) \mapsto^{\Delta}_{a} (\delta, e'_0) \mid a_1 \hookrightarrow (\delta_1, e_1) \uplus \ldots \uplus a_m \hookrightarrow (\delta_m, e_m)$$

and for all $0 \le i \le n$*, we have* e_i $\mathtt{wj}$*, then* e'_0 $\mathtt{wj}$ *and for all* $n < i \le m$*, we have* e_i $\mathtt{wj}$.

Proof. By induction on the derivation of the transition judgment. Consider some representative cases:

[4] For those familiar with evaluation contexts or stack machine semantics, `join` can only appear in the "hole" of an evaluation context or at the top of a stack.

$$\frac{}{x\ \mathtt{wj}} \quad \frac{}{\langle\rangle\ \mathtt{wj}} \quad \frac{}{\mathtt{n}\ \mathtt{wj}} \quad \frac{e\ \mathtt{nj}}{\lambda x{:}\tau.e\ \mathtt{wj}} \quad \frac{e_1\ \mathtt{wj} e_2\ \mathtt{nj}}{\langle e_1, e_2\rangle\ \mathtt{wj}} \quad \frac{e_1\ \mathtt{val} \qquad e_1\ \mathtt{nj} \qquad e_2\ \mathtt{nj}}{\langle e_1, e_2\rangle\ \mathtt{wj}} \quad \frac{e_1\ \mathtt{wj} \qquad e_2\ \mathtt{nj}}{e_1\ e_2\ \mathtt{wj}}$$

$$\frac{e_1\ \mathtt{val} \qquad e_1\ \mathtt{nj} \qquad e_2\ \mathtt{wj}}{e_1\ e_2\ \mathtt{wj}} \quad \frac{e_1\ \mathtt{nj} \qquad e_2\ \mathtt{nj}}{e_1 \parallel e_2\ \mathtt{wj}} \quad \frac{}{\mathtt{join}[b,c]\ \mathtt{wj}} \quad \frac{e\ \mathtt{wj}}{\mathtt{fst}(e)\ \mathtt{wj}}$$

$$\frac{e\ \mathtt{wj}}{\mathtt{snd}(e)\ \mathtt{wj}} \quad \frac{e\ \mathtt{wj}}{\mathtt{inl}(e)\ \mathtt{wj}} \quad \frac{e\ \mathtt{wj}}{\mathtt{inr}(e)\ \mathtt{wj}} \quad \frac{e\ \mathtt{wj} \qquad e_1\ \mathtt{nj} \qquad e_2\ \mathtt{nj}}{\mathtt{case}(e)\{x.e_1; y.e_2\}\ \mathtt{wj}}$$

$$\frac{e\ \mathtt{nj}}{\mathtt{bg}(e)\ \mathtt{wj}} \quad \frac{}{\mathtt{tid}[a]\ \mathtt{wj}} \quad \frac{e\ \mathtt{wj}}{\mathtt{fg}(e)\ \mathtt{wj}} \quad \frac{e\ \mathtt{wj}}{\mathtt{out}(e)\ \mathtt{wj}}$$

$$\frac{e\ \mathtt{nj}}{\mathtt{inp}[d](x.e)\ \mathtt{wj}} \quad \frac{e\ \mathtt{nj}}{\mathtt{fix}\ x{:}\tau\ \mathtt{is}\ e\ \mathtt{wj}}$$

Figure 15: Rules for well-joinedness

- $e_1\ e_2 \mid \mu \mapsto_a^\Delta (\delta, e_1'\ e_2) \mid \mu \uplus \mu'$. By inversion, $e_1\ \mathtt{wj}$ and $e_2\ \mathtt{nj}$, so by induction, $e_1'\ \mathtt{wj}$. This gives $e_1'\ e_2\ \mathtt{wj}$.
- $e_1\ e_2 \mid \mu \mapsto_a^\Delta (\delta, e_1\ e_2') \mid \mu$. By inversion, $e_1\ \mathtt{val}$ and $e_2\ \mathtt{wj}$, so by induction, $e_2'\ \mathtt{wj}$. This gives $e_1\ e_2'\ \mathtt{wj}$.
- $(\lambda x{:}\tau.e_1)\ e_2 \mid \mu \mapsto_a^\Delta (\delta, [e_2/x]e_1) \mid \mu$. By inversion, $e_2\ \mathtt{val}$ and $e_2\ \mathtt{wj}$. It can be shown by a straightforward induction on these two derivations that $e_2\ \mathtt{nj}$, so $[e_2/x]e_1\ \mathtt{wj}$.
- $e_1 \parallel e_2 \mid \mu \mapsto_a^\Delta (0, \mathtt{join}[b,c]) \mid \mu \uplus b \hookrightarrow (0, e_1) \uplus c \hookrightarrow (0, e_2)$. By inversion, $e_1\ \mathtt{nj}$ and $e_2\ \mathtt{nj}$ (and therefore, are well-joined). We also have $\mathtt{join}[b,c]\ \mathtt{wj}$.
- $\mathtt{join}[b,c] \mid \mu \uplus b \hookrightarrow (\delta_b, e_b) \uplus c \hookrightarrow (\delta_c, e_c) \mapsto_a^\Delta (0, \langle e_b, e_c\rangle) \mid \mu \uplus b \hookrightarrow (\delta_b, e_b) \uplus c \hookrightarrow (\delta_c, e_c) \mapsto_a^\Delta$. By assumption, $e_b\ \mathtt{wj}$ and $e_c\ \mathtt{wj}$.

□

Finally, we prove a theorem which encompasses type safety and well-joinedness. If an initial thread pool consisting of a single source expression (which is well-typed under the empty context and signature) evaluates to μ' after some number of steps, then μ' is well-typed, not stuck and all of its expressions are well-joined.

Theorem 3 (Type Safety and Well-Joinedness)**.** *If $\cdot \vdash_\cdot e : \tau@w$ and $0; a \hookrightarrow (0, e) \mapsto^*_{\mathtt{glo}} r; \mu'$, then*

1. *there exists Σ' such that $\cdot \vdash_\cdot \mu' : \Sigma'$*
2. *either μ' $\mathtt{final}$ or there exist r'' and μ'' such that $r; \mu' \mapsto_{\mathtt{glo}} r''; \mu''$*
3. *For all $b \hookrightarrow (\delta, e_b) \in \mu'$, we have $e_b\ \mathtt{wj}$.*

Expression cost semantics $e;\mu \Downarrow^{\Delta} v; g$

$$\frac{e\ \mathtt{val}}{e;\mu \Downarrow^{\Delta} e;\emptyset} \qquad \frac{e_1;\mu \Downarrow^{\Delta} \lambda x{:}\tau.e; g_1 \qquad e_2;\mu \Downarrow^{\Delta} v; g_2 \qquad [v/x]e;\mu \Downarrow^{\Delta} v'; g_3 \qquad u\ \mathtt{fresh}}{e_1\ e_2;\mu \Downarrow^{\Delta} v'; g_1 \oplus g_2 \oplus [u] \oplus g_3} \qquad \frac{e;\mu \Downarrow^{\Delta} \langle v_1, v_2\rangle; g \qquad u\ \mathtt{fresh}}{\mathtt{fst}(e);\mu \Downarrow^{\Delta} v_1; g \oplus [u]}$$

$$\frac{e;\mu \Downarrow^{\Delta} \langle v_1, v_2\rangle; g \qquad u\ \mathtt{fresh}}{\mathtt{snd}(e);\mu \Downarrow^{\Delta} v_2; g \oplus [u]} \qquad \frac{e_1;\mu \Downarrow^{\Delta} v_1; g_1 \qquad e_2;\mu \Downarrow^{\Delta} v_2; g_2}{\langle e_1, e_2\rangle;\mu \Downarrow^{\Delta} \langle v_1, v_2\rangle; g_1 \oplus g_2} \qquad \frac{e_1;\mu \Downarrow^{\Delta} v_1; g_1 \qquad e_2;\mu \Downarrow^{\Delta} v_2; g_2}{e_1 \parallel e_2;\mu \Downarrow^{\Delta} \langle v_1, v_2\rangle; g_1 \otimes g_2}$$

$$\frac{\mu = \mu' \uplus b \hookrightarrow (\delta_b, e_b) \uplus c \hookrightarrow (\delta_c, e_c) \qquad e_b;\mu \Downarrow^{\Delta} v_1; g_1 \qquad e_c;\mu \Downarrow^{\Delta} v_2; g_2 \qquad u\ \mathtt{fresh}}{\mathtt{join}[b,c];\mu \Downarrow^{\Delta} \langle v_1, v_2\rangle; (u, u, \{u\}, \{(b,u,1),(c,u,1)\}, \emptyset)}$$

$$\frac{e;\mu \Downarrow^{\Delta} \mathtt{inl}(v); g_1 \qquad [v/x]e_1;\mu \Downarrow^{\Delta} v'; g_2 \qquad u\ \mathtt{fresh}}{\mathtt{case}(e)\{x.e_1; y.e_2\};\mu \Downarrow^{\Delta} v'; g_1 \oplus [u] \oplus g_2} \qquad \frac{e;\mu \Downarrow^{\Delta} \mathtt{inr}(v); g_1 \qquad [v/y]e_2;\mu \Downarrow^{\Delta} v'; g_2 \qquad u\ \mathtt{fresh}}{\mathtt{case}(e)\{x.e_1; y.e_2\};\mu \Downarrow^{\Delta} v'; g_1 \oplus [u] \oplus g_2}$$

$$\frac{e;\mu \Downarrow^{\Delta} v; g \qquad g = (s,t,V,E,F) \qquad u\ \mathtt{fresh}}{\mathtt{bg}(e);\mu \Downarrow^{\Delta} \mathtt{thread}[t](v); g \oslash^{u}} \qquad \frac{e;\mu \Downarrow^{\Delta} \mathtt{thread}[u_1](v); g \qquad u_2\ \mathtt{fresh}}{\mathtt{fg}(e);\mu \Downarrow^{\Delta} v; (g\odot) \oplus [u_2] \cup \{(u_1, u_2, 1)\}}$$

$$\frac{\mu = \mu' \uplus b \hookrightarrow (\delta, e_b) \qquad e;\mu \Downarrow^{\Delta} \mathtt{tid}[b]; g \qquad e_b;\mu \Downarrow^{\Delta} v; g_b \qquad u\ \mathtt{fresh}}{\mathtt{fg}(e);\mu \Downarrow^{\Delta} v; (g\odot) \oplus [u] \cup \{(b, u, 1)\}} \qquad \frac{e;\mu \Downarrow^{\Delta} v; g \qquad u\ \mathtt{fresh}}{\mathtt{out}(e);\mu \Downarrow^{\Delta} \langle\rangle; g \oplus [u]}$$

$$\frac{[\mathtt{n}/x]e;\mu \Downarrow^{\Delta} v; g \qquad u_1\ \mathtt{fresh} \qquad u_2\ \mathtt{fresh} \qquad \delta \in \Delta(d)}{\mathtt{inp}[d](x.e);\mu \Downarrow^{\Delta} v; [u_1] \oplus_{\delta} [u_2] \oplus g} \qquad \frac{[\mathtt{n}/x]e;\mu \Downarrow^{v}; g \qquad u\ \mathtt{fresh}}{\mathtt{in}(x.e);\mu \Downarrow^{v}; [u] \oplus g}$$

$$\frac{[\mathtt{fix}\ x{:}\tau\ \mathtt{is}\ e/x]e;\mu \Downarrow^{\Delta} v; g \qquad u\ \mathtt{fresh}}{\mathtt{fix}\ x{:}\tau\ \mathtt{is}\ e;\mu \Downarrow^{\Delta} v; [u] \oplus g}$$

Thread pool cost semantics $\mu_l;\mu_g \Downarrow^{\Delta} v; g$

$$\frac{}{\emptyset;\mu_g \Downarrow^{\Delta}_{\mathsf{C}} \{\}} \qquad \frac{\mu_l;\mu_g \Downarrow^{\Delta}_{\mathsf{C}} \{G\} \qquad e;\mu_g \Downarrow^{\Delta} g \qquad g \neq \emptyset}{a \hookrightarrow (\delta, e) \uplus \mu_l;\mu_g \Downarrow^{\Delta}_{\mathsf{C}} \{a \hookrightarrow g \downharpoonright_{\delta} \uplus G\}}$$

Figure 16: Extended cost semantics.

Proof. Parts 1 and 2 are simply an inductive application of Lemmas 5 and 6. We prove part 3 by induction on the derivation of $0; a \hookrightarrow (0, e) \mapsto^{*}_{\mathtt{glo}} r;\mu'$. If $\mu' = a \hookrightarrow (0, e)$, then we must have e $\mathtt{nj}$ since e types with an empty signature, and this implies that e $\mathtt{wj}$ (these facts can be shown by a straightforward induction on the typing derivation and the derivation of e $\mathtt{nj}$, respectively.

Otherwise, suppose $0; a \hookrightarrow (0, e) \mapsto^{*}_{\mathtt{glo}} r'';\mu''$ and $r'';\mu'' \mapsto_{\mathtt{glo}} r;\mu'$. By induction, e''_b $\mathtt{wj}$ for all $b \hookrightarrow (\delta'', e''_b) \in \mu''$. Let $b \hookrightarrow (\delta', e'_b) \in \mu'$. We have three cases: (1) $b \hookrightarrow (\delta'', e'_b) \in \mu''$ or (2) $b \hookrightarrow (0, e''_b) \in \mu''$ and $e''_b \mid \mu'' \mapsto^{\Delta}_{b} (\delta', e'_b) \mid \mu'' \uplus \mu_b$ or (3) there exists $c \hookrightarrow (0, e_c) \in \mu''$ such that $e_c \mid \mu'' \mapsto^{\Delta}_{c} (\delta'_c, e'_c) \mid \mu'' \uplus \mu_c$ and $b \hookrightarrow (\delta', e'_b) \in \mu_c$. In case (1), the result is clear by induction. In cases (2) and (3), e'_b $\mathtt{wj}$ by Lemma 7. □

$$
\begin{array}{lcll}
g \downharpoonleft_0 & = & g & \\
(s,t,V,E,F) \downharpoonleft_\delta & = & [\alpha] \oplus_\delta g & \delta > 0, \alpha\ \texttt{fresh} \\
(s,t,V,E,F)\odot & = & (s,t,V,E,F \cup \{\langle {}^s_t \rangle\}) & \nexists a, \delta.(a,s,\delta) \in E \\
(s,t,V,E,F)\odot & = & (s,t,V,E,F \cup \{\langle {}_t \rangle\}) & \exists a, \delta.(a,s,\delta) \in E
\end{array}
$$

Figure 17: Extended graph operations.

4.2 Extended Cost Model

In order to show a correspondence between the cost semantics and the operational semantics, we must extend the cost semantics to generate cost graphs not just for expressions but also for thread pools which can represent programs that have already begun to execute. As such, dags may no longer have a single source vertex, though they will continue to have a single sink vertex (the final instruction of the initial thread). They will have a source vertex for each ready thread. This modification is relatively straightforward: for each thread, we will generate a standard dag like those of Section 3.3, which we now call a *thread graph* or *thread dag*, with a single source and single sink. These are then composed to form a *configuration graph* or *configuration dag* by adding edges that correspond to the inter-thread dependencies created by `join` and `fg`.

The cost semantics in Figure 16 generates thread graphs for expressions. The judgment $e; \mu \Downarrow^{\Delta} v; g$ indicates that the expression e evaluates to v and has cost graph g in the presence of μ. The expression being evaluated may refer to threads in μ. These threads are included so that the value can be generated, but their cost is not included in g. There are two major differences between the extended version of this judgment and the version in Figure 9. First, there is an additional rule for $\texttt{fg}(e)$ handling the case in which e evaluates to a thread handle $\texttt{tid}[b]$. In this case, the expression corresponding to b in the thread pool is evaluated. Second, the definition of $g\odot$ is extended for the case in which the source of g is a join point, i.e. has incoming edges from outside g. This is handled using a new form of foreground block $\langle {}_t \rangle$, which has only a sink t and no source. All vertices that are ancestors of t are part of the foreground block $\langle {}_t \rangle$. The extended graph operations are given in Figure 17.

A configuration graph G mirrors the structure of the thread pool μ; it is a mapping from thread symbols to thread graphs:

$$G = a_1 \hookrightarrow g_1 \uplus \ldots \uplus a_n \hookrightarrow g_n$$

The vertices, edges and foreground blocks of a configuration graph are the union of the vertices, edges and foreground blocks of the component thread graphs. If $a \hookrightarrow g_a \in G$, an edge (a, u, δ) may be viewed as an edge from the sink of g_a to u. If $g_a = \emptyset$, this edge is ignored. The metrics such as work, span and foreground width extend in the natural way to configuration graphs.

The judgment $\mu_l; \mu_g \Downarrow^{\Delta}_{\mathsf{C}} \{G\}$ generates a portion of a configuration graph from the threads in a partial thread pool μ_l by generating a thread graph for each thread and composing any non-empty graphs that result. As above, the whole thread pool μ_g is included so that threads may refer to other threads which are not currently under attention, but these threads are not included in G. If a thread is delayed with delay $\delta > 0$, its cost graph is composed serially after a fresh auxiliary vertex using an edge of weight δ.

The extended cost semantics allows us to assign costs (work, span, etc.) to programs, as represented by thread pools. The work and span of a thread pool that is in the middle of execution can be thought of as the *remaining* work and span of the program. The work of a thread pool μ under Δ is written $W(\mu, \Delta)$ and is defined as the maximum work over all dags that can be generated from μ:

$$W(\mu, \Delta) = \max\{W(G) \mid \mu; \mu \Downarrow^{\Delta}_{\mathsf{C}} \{G\}\}$$

We take the maximum since the cost semantics is nondeterministic. The definitions of $S(\mu, \Delta)$, $W^{\circ}(\mu, \Delta)$ and $S^{\circ}(\mu, \Delta)$ are similar.

$$\begin{array}{lcl} W(\mu, \Delta) & = & \max\{W(G) \mid \mu; \mu \Downarrow^{\Delta}_{\mathsf{C}} \{G\}\} \\ S(\mu, \Delta) & = & \max\{S(G) \mid \mu; \mu \Downarrow^{\Delta}_{\mathsf{C}} \{G\}\} \\ W^{\circ}(\mu, \Delta) & = & \max\{W^{\circ}(G) \mid \mu; \mu \Downarrow^{\Delta}_{\mathsf{C}} \{G\}\} \\ S^{\circ}(\mu, \Delta) & = & \max\{S^{\circ}(G) \mid \mu; \mu \Downarrow^{\Delta}_{\mathsf{C}} \{G\}\} \\ D(\mu, \Delta) & = & \max\{D(G) \mid \mu; \mu \Downarrow^{\Delta}_{\mathsf{C}} \{G\}\} \end{array}$$

Lemma 8 extends Lemma 1 to handle programs that have begun to execute. It shows that cost graphs generated by $\mathbb{F}$ expressions have no nested foreground blocks and edges from other threads occur only at joins.

(TODO: Definition of J)

Lemma 8. *If* $\cdot \vdash_{\Sigma} e : \tau @ \mathbb{F}$ *and* $\cdot \vdash. a \hookrightarrow (\delta, e) \uplus \mu : \Sigma, a \sim \tau @ \mathbb{F}$ *and* $a \hookrightarrow (\delta, e) \uplus \mu; \mu \Downarrow^{\Delta}_{\mathsf{C}} \{G\}$ *where*

$$G = a \hookrightarrow g \uplus \biguplus_{i=1}^{n} a_i \hookrightarrow (s_i, t_i, V_i, E_i, F_i)$$

and $g = (s, t, V, E, F)$*, then*

1. $F = \emptyset$
2. *if* e $\texttt{nj}$*, then there does not exist* $(b, u, \delta) \in E$ *(where* b *is a thread identifier)*
3. *if* e $\texttt{wj}$*, then there does not exist* $(b, u, \delta) \in E$ *for any* $u \neq s$*.*
4. *for all* a_i*, we have* $s_i \preceq_G t$ *if and only if* $a_i \in J_{\mu}(e)$

Proof. 1. By induction on the derivation of $\cdot \vdash_{\Sigma} e : \tau @ \mathbb{F}$.

2. By induction on the derivation of $\cdot \vdash_{\Sigma} e : \tau @ \mathbb{F}$. We need not consider the case for $e = \texttt{join}[b, c]$ since e $\texttt{nj}$.
3. By induction on the derivation of $\cdot \vdash_{\Sigma} e : \tau @ \mathbb{F}$. If $e = \texttt{join}[b, c]$, then the result is clear. As a representative set of cases, consider the cases for $e = e_1\ e_2$. By inversion, $e_1; \mu \Downarrow^{\Delta} v_1; g_1$ and $e_2; \mu \Downarrow^{\Delta} v_2; g_2$. If e_1 $\texttt{wj}$ and e_2 $\texttt{nj}$, then by induction, g_2 has no edges from other threads and g_1 has them only to its source, which is s, so the result holds. If e_1 $\texttt{val}$ and e_2 $\texttt{wj}$, then $g_1 = \emptyset$ and by induction, g_2 has edges from other threads only to its source, which is s.
4. By induction on the derivations of $\cdot \vdash_{\Sigma} e : \tau @ \mathbb{F}$ and $\cdot \vdash. a \hookrightarrow (0, e) \uplus \mu : \Sigma, a \sim \tau @ \mathbb{F}$. The interesting case is $e = \texttt{join}[a_i, a_j]$. Suppose $\mu = a_i \hookrightarrow (\delta_i, e_i) \uplus a_j \hookrightarrow (\delta_j, e_j) \uplus \mu'$. We have $J_{\mu}(e) = \{a_i, a_j\} \cup J_{\mu}(e_i) \cup J_{\mu}(e_j)$. By inversion on the typing rules, we have $\cdot \vdash_{\Sigma} e_i : \tau_i @ \mathbb{F}$ and

$\cdot \vdash_\Sigma e_j : \tau_j@\mathbb{F}$, so by induction, for any s_k, $s_k \preceq_G t_i$ if and only if $a_k \in J_\mu(e_i)$ and $s_k \preceq_G t_j$ if and only if $a_k \in J_\mu(e_j)$. Let a_k be such that $s_k \preceq_G t$. By inversion on the cost semantics, we have $a_k = a_i$ or $a_k = a_j$ or $s_k \preceq_G t_i$ or $s_k \preceq_G t_j$. In any of these cases, $a_k \in J_\mu(e)$. Now suppose $a_k \in J_\mu(e)$. We have that $a_k = a_i$ or $a_k = a_j$ or $a_k \in J_\mu(e_i)$ or $a_k \in J_\mu(e_j)$. In any case, $s_k \preceq_G t_i$ or $s_j \preceq_G t_j$ and, by transitivity, $s_k \preceq_G t$.

□

Next, we show that the operational semantics and cost semantics agree on the values produced by an expression. One complication in showing such a result is accounting for the value $\texttt{thread}[v](u)$ which is produced by the cost semantics but not the operational semantics[5]. We therefore show that the cost semantics and the operational semantics are equivalent up to a relation $\rightsquigarrow_\mu$ which relates the two forms of thread handle. We define $\rightsquigarrow_\mu$ inductively. The important rules are the ones for thread handles:

$$\frac{\mu = b \hookrightarrow (\delta, e) \uplus \mu' \qquad e; \mu \Downarrow^\Delta v; g}{\texttt{thread}[u](v) \rightsquigarrow_\mu \texttt{tid}[b]} \qquad \frac{}{\texttt{tid}[b] \rightsquigarrow_\mu \texttt{tid}[b]}$$

$$\frac{v \rightsquigarrow_\mu v'}{\texttt{thread}[u](v) \rightsquigarrow_\mu \texttt{thread}[u](v')}$$

All other rules simply preserve $\rightsquigarrow_\mu$. It can be shown that $\rightsquigarrow_\mu$ is reflexive, transitive and respects substitution.

In order to show how individual steps of the operational semantics change the cost graph (which we will in turn use to show the correspondence between the two versions of the semantics), we generalize serial composition to allow thread graphs to be composed with configuration graphs. In $g_1 \overline{\oplus} G_2$, the sink vertex of g_1 is joined to all source vertices of G_2 with edges of weight 1. Source vertices of G_2 which are auxiliary vertices are eliminated in the process.

If $G_2 = \biguplus_{i=1}^n a_i \hookrightarrow (s_i, t_i, V_i, E_i, F_i)$ and t_{G_2} is the sink vertex of G_2, then $(s, t, V, E, F) \overline{\oplus} G_2$ is defined as

$$\begin{array}{l}(s, t_{G_2}, V \cup V_1 \cup \cdots \cup V_n,\\ E \cup E_1 \cup \cdots \cup E_n \cup \{(t, s', 1) \mid s' \textit{ has no ancestors in } G_2\}\\ \cup\{(t, s', \delta + 1) \mid (\alpha, s', \delta) \in E_1 \cup \cdots \cup E_n\},\\ F \cup F_1 \cup \cdots \cup F_n)\end{array}$$

Note that the operation $G_1 \overline{\oplus} g_2$ is not necessary; ordinary serial composition works in this case since G_1 has a unique sink.

Lemma 9 states that related expressions evaluate to related values and isomorphic cost graphs.

Lemma 9. *If $e \rightsquigarrow_\mu e'$ and $e'; \mu \Downarrow^\Delta v'; g'$, then there exist v and g such that $e; \mu \Downarrow^\Delta v; g$ and $v \rightsquigarrow_\mu v'$ and $g \cong g'$.*

Proof. By induction on the derivation of $e'; \mu \Downarrow^\Delta v'; g'$. The interesting case is $\texttt{fg}(e_0'); \mu \Downarrow^\Delta v'; g'$, where $e = \texttt{fg}(e_0)$ and $e_0 \rightsquigarrow_\mu e_0'$. By inversion, either (1) $e_0'; \mu \Downarrow^\Delta \texttt{thread}[u](v'); g_0'$ and $g' =$

[5] The form $\texttt{tid}[a]$ is not explicitly produced by the cost semantics, but can be carried through since it is an irreducible expression and therefore evaluates to itself under the cost semantics.

$g'_0 \oplus [u_2] \cup \{(u, u_2, 1)\}$ or (2) $e'_0; \mu \Downarrow^{\Delta} \mathtt{tid}[b]; g'_0$ and $g' = g'_0 \oplus [u] \cup \{(b, u, 1)\}$. In case (1), by induction, $e_0; \mu \Downarrow^{\Delta} \mathtt{thread}[u](v); g_0$ where $v \rightsquigarrow_{\mu} v'$ and $g_0 \cong g'_0$. In case (2), by induction, either $e_0; \mu \Downarrow^{\Delta} \mathtt{tid}[b]; g_0$ (in which case the result is trivial) or $e_0; \mu \Downarrow^{\Delta} \mathtt{thread}[u](v); g_0$, where $g_0 \cong g'_0$ and $\mu = \mu' \uplus b \hookrightarrow (\delta, e_b)$ and $e_b; \mu \Downarrow^{\Delta} v; g$. In this case, by the cost semantics, $\mathtt{fg}(e_0); \mu \Downarrow^{\Delta} v; g$ where $g = g_0 \oplus [u_2] \cup \{(u, u_2, 1)\} \cong g'$. ◻

Lemma 10 examines the effect of a transition $r; \mu \mapsto_{\mathtt{glo}} r'; \mu'$ on the cost graph of a specified thread $a \hookrightarrow (\delta, e)$ of μ'. In other words, the lemma shows how the cost semantics behaves under "converse evaluation" or "head expansion", a standard step in relating transition and evaluation semantics. Part 1 considers the case in which a is not one of the threads that transitions. In this case, if e evaluates to v' with cost graph g' under μ', it evaluates to a related value and isomorphic cost graph under μ. Part 2 considers the more complex case in which a steps from e to e', adding the threads in μ'_a. In this case, if e' evaluates to v' under μ' and $a \hookrightarrow (\delta', e') \uplus \mu'_a$ produces the cost graph G', then e evaluates to a value related to v' with a cost graph that adds at least one vertex as an ancestor of G'. This lemma will then be used to show that the cost semantics and the operational semantics correspond on the final values, and will later be used to show the Brent-type theorem that the cost graph is an accurate representation of the length of a prompt schedule.

Lemma 10. *Fix Δ and suppose that $\cdot \vdash_{\Sigma} e : \tau @ w$ and $\cdot \vdash_{\cdot} \mu_0 : \Sigma$. Let $\mu = a \hookrightarrow (\delta, e) \uplus \mu_0$.*

1. *Suppose $r; \mu \mapsto_{\mathtt{glo}} r'; \mu'$ and $e; \mu' \Downarrow^{\Delta} v'; g'$. There exist v and g such that $e; \mu \Downarrow^{\Delta} v; g$ and $g \cong g'$ and $v \rightsquigarrow_{\mu'} v'$.*
2. *Suppose $\mu' = a \hookrightarrow (\delta', e') \uplus \mu'_0 \uplus \mu'_a$ and $r; \mu \mapsto_{\mathtt{glo}} r'; \mu'$ where $e \mid \mu \mapsto^{\Delta}_{a} (\delta', e') \mid \mu \uplus \mu'_a$ and $e'; \mu' \Downarrow^{\Delta} v'; g'_a$ and $\mu'_a; \mu \Downarrow^{\Delta}_{\mathtt{C}} \{G'_0\}$. Let $G' = a \hookrightarrow g'_a \downharpoonright_{\delta'} \uplus G'_0$. There exist v and g and g'', where g'' is nonempty, such that $e; \mu \Downarrow^{\Delta} v; g$ and $g \cong g'' \overline{\oplus} G'$ and $v \rightsquigarrow_{\mu'} v'$.*

Proof. By induction on the derivation of $e; \mu' \Downarrow^{\Delta} v'; g'$ and $(0, e) \mid \mu \mapsto^{\Delta}_{a} (\delta', e') \mid \mu \uplus \mu'_a$.

1. The interesting cases are those for $\mathtt{join}[b, c]$ and $\mathtt{fg}(e)$ where $e; \mu' \Downarrow^{\Delta} \mathtt{tid}[b]; g'_1$. We consider the latter as an example. By inversion, $b \hookrightarrow (\delta'_b, e'_b) \in \mu'$ and $e'_b; \mu' \Downarrow^{\Delta} v'; g'_2$. Since $\cdot \vdash_{\Sigma} e : \tau @ w$, we must have $b \in \Sigma$ and so $b \hookrightarrow (\delta_b, e_b) \in \mu_0$ and either (1) $e_b = e'_b$ or (2) $\delta_b = 0$ and $e_b \mid \mu \mapsto^{\Delta}_{b} (\delta'_b, e'_b) \mid \mu \uplus \mu'_b$. In either case, by induction, we have $e_b; \mu \Downarrow^{\Delta} v; g_2$ and $v \rightsquigarrow_{\mu'} v'$. Also by induction, $e; \mu \Downarrow^{\Delta} \mathtt{tid}[b]; g_1$ where $g_1 \cong g'_1$. By the cost semantics, $\mathtt{fg}(e); \mu \Downarrow^{\Delta} v; g$ where $g = (g_1 \odot) \oplus [u] \cup \{(b, u, 1)\} \cong (g'_1 \odot) \oplus [u] \cup \{(b, u, 1)\} = g'$.
2. Consider cases for which rule the step invokes.
   - $e_1\ e_2 \mid \mu \mapsto^{\Delta}_{a} (\delta', e'_1\ e_2) \mid \mu \uplus \mu'_a$. By inversion, $e_1 \mid \mu \mapsto^{\Delta}_{a} (\delta', e'_1) \mid \mu \uplus \mu'_a$. By inversion on the cost semantics, we have $e'_1; \mu' \Downarrow^{\Delta} \lambda x{:}\tau.e'_0; g'_1$ and $e_2; \mu' \Downarrow^{\Delta} v'_2; g'_2$ and $[v'_2/x]e'_0; \mu' \Downarrow^{\Delta} v'; g'_3$. By induction, there exist g_1, e_0, g'', g_2 and v_2 such that $e_1; \mu \Downarrow^{\Delta} \lambda x{:}\tau.e_0; g_1$ and $e_0 \rightsquigarrow_{\mu'} e'_0$ and $g_1 \cong g'' \overline{\oplus}(G'_0 \uplus a \hookrightarrow g'_1 \downharpoonright_{\delta'})$ and $e_2; \mu \Downarrow^{\Delta} v_2; g_2$ and $g_2 \cong g'_2$ and $v_2 \rightsquigarrow_{\mu'} v'_2$. By Lemma 9, there exist v and g_3 such that $[v_2/x]e_0; \mu \Downarrow^{\Delta} v; g_3$ and $v \rightsquigarrow_{\mu \uplus \mu'} v'$ and $g_3 \cong g'_3$. By this fact, and the cost semantics, $e_1\ e_2; \mu \uplus \mu' \Downarrow^{\Delta} v; g$ where

$$g = g_1 \oplus g_2 \oplus [u] \oplus g_3 \cong g'' \overline{\oplus}(G'_0 \uplus a \hookrightarrow g'_1 \downharpoonright_{\delta'}) \oplus g'_2 \oplus [u] \oplus g'_3$$

     so g'' satisfies the requirement of the conclusion.

- $e_1\ e_2 \mid \mu \mapsto_a^\Delta (\delta', e_1\ e_2') \mid \mu \uplus \mu_a'$. By inversion, $e_2 \mid \mu \mapsto_a^\Delta (\delta', e_2') \mid \mu \uplus \mu_a'$. By inversion on the cost semantics, $e_2'; \mu' \Downarrow^\Delta v_2'; g_2'$ and $[v_2'/x]e_1; \mu' \Downarrow^\Delta v'; g_3'$. By induction, there exist v_2 and g_2 and g'' such that $e_2; \mu \Downarrow^\Delta v_2; g_2$ and $v_2 \rightsquigarrow_{\mu'} v_2'$ and $g_2 \cong g'' \mathbin{\overline{\oplus}} (G_0' \uplus a \hookrightarrow g_2' \downharpoonright_{\delta'})$. By Lemma 9, there exist v and g_3 such that $[v_2/x]e_1; \mu \Downarrow^\Delta v; g_3$ and $v \rightsquigarrow_{\mu'} v'$ and $g_3 \cong g_3'$. By this fact and the cost semantics, $(\lambda x{:}\tau.e_1)\ e_2; \mu \Downarrow^\Delta v; g$ where

$$g = g_2 \oplus [u] \oplus g_3 \cong g'' \mathbin{\overline{\oplus}} (G_0' \uplus a \hookrightarrow g_2' \downharpoonright_{\delta'}) \oplus [u] \oplus g_3'$$

  so g'' satisfies the requirement of the conclusion.
- $(\lambda x{:}\tau.e_1)\ e_2 \mid \mu \mapsto_a^\Delta (0, [e_2/x]e_1) \mid \mu$. Since $\mu_a' = \emptyset$, we have that $G' = a \hookrightarrow g_a'$, where $[e_2/x]e_1; \mu' \Downarrow^\Delta v'; g_a'$. By part 1, there exist v and g_a such that $[e_2/x]e_1; \mu \Downarrow^\Delta v; g_a$ and $g_a \cong g_a'$ and $v \rightsquigarrow_{\mu'} v'$. By the cost semantics, we also have that $(\lambda x{:}\tau.e_1)\ e_2; \mu \Downarrow^\Delta v; [u] \oplus g_a$.
- $e_1 \| e_2 \mid \mu \mapsto_a^\Delta (0, \mathtt{join}[b, c]) \mid \mu \uplus \mu_a'$. We have $G' = a \hookrightarrow [u] \cup \{(b, u, 1), (c, u, 1)\} \uplus b \hookrightarrow g_b' \uplus c \hookrightarrow g_c'$ and $e_1; \mu' \Downarrow^\Delta v_1'; g_b'$ and $e_2; \mu' \Downarrow^\Delta v_2'; g_c'$ and $v' = \langle v_1', v_2' \rangle$. By induction, there exist $v_1 \rightsquigarrow_{\mu'} v_1'$ and $g_b \cong g_b'$ and $v_2 \rightsquigarrow_{\mu'} v_2'$ and $g_c \cong g_c'$ such that $e_1; \mu \Downarrow^\Delta v_1; g_b$ and $e_2; \mu \Downarrow^\Delta v_2; g_c$. By the cost semantics, $e_1 \parallel e_2; \mu \Downarrow^\Delta \langle v_1, v_2 \rangle; g_b \otimes g_c$. Identify u with the sink vertex of $g_b \otimes g_c$ and let s be the source vertex of $g_b \otimes g_c$. We have $g_b \otimes g_c \cong [s] \mathbin{\overline{\oplus}} G'$.
- $\mathtt{join}[b, c] \mid \mu \mapsto_a^\Delta (0, \langle e_1, e_2 \rangle) \mid \mu$ where $b \hookrightarrow (\delta_b, e_b), c \hookrightarrow (\delta_c, e_c) \in \mu$. Since $\mu_a' = \emptyset$, we have that $G' = \emptyset$. By the cost semantics, $\mathtt{join}[b, c]; \mu \Downarrow^\Delta \langle e_b, e_c \rangle; g$, where g is nonempty, so the case is trivial.
- $\mathtt{bg}(e) \mid \mu \mapsto_a^\Delta (0, \mathtt{tid}[b]) \mid \mu \uplus b \hookrightarrow (0, e)$. We have $G' = b \hookrightarrow g_b'$ and $e; \mu' \Downarrow^\Delta v'; g_b'$. By part 1, there exist $v \rightsquigarrow_{\mu'} v'$ and $g_b \cong g_b'$ such that $e; \mu \Downarrow^\Delta v; g_b$. By the cost semantics, $\mathtt{bg}(e); \mu \Downarrow^\Delta \mathtt{thread}[u](v); g$, where $g = g_b \oslash^u \cong [u] \oplus g_b'$. Since $\mu' = b \hookrightarrow (0, e)$, we have $\mathtt{thread}[u](v) \rightsquigarrow_{\mu'} \mathtt{thread}[u](v') \rightsquigarrow_{\mu'} \mathtt{tid}[b]$.
- $\mathtt{fg}(e) \mid \mu \mapsto_a^\Delta (\delta', \mathtt{fg}(e')) \mid \mu \uplus \mu_a'$. By induction.
- $\mathtt{fg}(\mathtt{tid}[b]) \mid \mu \mapsto_a^\Delta (0, e) \mid \mu$. Since $\mu_a' = \emptyset$, we have that $G' = \emptyset$. By the cost semantics, $\mathtt{fg}(\mathtt{tid}[b]); \mu \Downarrow^\Delta e; g$ where g is nonempty and so the case is trivial.
- $\mathtt{out}(e) \mid \mu \mapsto_a^\Delta (\delta', \mathtt{out}(e')) \mid \mu \uplus \mu_a'$. By induction.
- $\mathtt{out}(e) \mid \mu \mapsto_a^\Delta (0, \langle\rangle) \mid \mu$. Since $\mu_a' = \emptyset$, we have that $G' = \emptyset$. By the cost semantics, $\mathtt{out}(e); \mu \Downarrow^\Delta \langle\rangle; g$, where g is nonempty and so the case is trivial.
- $\mathtt{inp}[d](x.e) \mid \mu \mapsto_a^\Delta (\delta - 1, \mathtt{in}(x.e)) \mid \mu$. Since $\mu_a' = \emptyset$, we have that $G' = a \hookrightarrow g_a' \downharpoonright_{\delta-1}$, where $\mathtt{in}(x.e); \mu' \Downarrow^\Delta v'; g_a'$. By inversion, $[\mathtt{n}/x]e; \mu' \Downarrow^\Delta v'; g'$ and $g_a' = [u] \oplus g'$. By induction, there exist $v \rightsquigarrow_{\mu'} v'$ and $g \cong g'$ such that $[\mathtt{n}/x]e; \mu \Downarrow^\Delta v; g$. By the cost semantics, $\mathtt{inp}[d](x.e); \mu \Downarrow^\Delta v; [u_1] \oplus_\delta [u] \oplus g$. The result follows from the definition of generalized serial composition.
- $\mathtt{in}(x.e) \mid \mu \mapsto_a^\Delta (0, [\mathtt{n}/x]e) \mid \mu$. Since $\mu_a' = \emptyset$, we have that $G' = a \hookrightarrow g_a'$, where $[\mathtt{n}/x]e; \mu' \Downarrow^\Delta v'; g_a'$. By part 1 and the cost semantics, there exist $v \rightsquigarrow_{\mu'} v'$ and $g_a \cong g_a'$ such that $\mathtt{in}(x.e); \mu \Downarrow^\Delta v; [u] \oplus g_a$.

$\square$

We can now show the final result of this section: that if a well-typed λ^{ip} program evaluates to a value using the operational semantics, the cost semantics will produce a cost graph for that program, along with the same final value.

Theorem 4. *If* $\cdot \vdash_{.} a \hookrightarrow (\delta, e) \uplus \mu : \Sigma, a \sim \tau \mathbin{@} w$ *and*

$$r; a \hookrightarrow (\delta, e) \uplus \mu \mapsto^{*}_{\mathtt{glo}} r'; a \hookrightarrow (0, e') \uplus \mu'$$

and $a \hookrightarrow (0, e') \uplus \mu'$ $\mathtt{final}$*, then there exist* v *and* g *such that* $e; \mu \Downarrow^{\Delta} v; g$ *and* $v \rightsquigarrow_{a \hookrightarrow (0,e') \uplus \mu'} e'$*.*

Proof. Since e' $\mathtt{val}$, we have $e'; \mu' \Downarrow^{\Delta} e'; \emptyset$. Proceed by an inductive application of Lemma 10. $\square$

4.3 Cost Bounds for Prompt Scheduling Principle

The main result of this section is showing that the cost bounds predicted by the cost semantics can be realized by the operational semantics in that, given a prompt schedule, a λ^{ip} program can be evaluated using the operational semantics in the number of steps and response time predicted by the prompt scheduling theorem (Theorem 1).

The key step in showing the bound on the computation time is showing that a global transition step decreases the total work by P or the total span by 1. The intuition behind this proof is the same as that of Theorem 1: the scheduler will either execute P (foreground) instructions or execute all ready (foreground) instructions. The proof of this lemma makes heavy use of part 2 of Lemma 10, which shows that a local transition on a thread decreases the work and span of the thread's dag by at least 1.

We also require one more technical lemma, stating that any path to a vertex u of a foreground block from an ancestor u' outside the foreground block must pass through the source of the block. This is a direct result of the fact, shown in Lemma 8, that there are no edges from outside a foreground block into it, except to the source.

Lemma 11. *Fix* Δ*. Suppose* $\mu = a_1 \hookrightarrow (\delta_1, e_1) \uplus \ldots \uplus a_n \hookrightarrow (\delta_n, e_n)$ *and* $\cdot \vdash_{.} \mu : \Sigma$ *and* e_i $\mathtt{wj}$ *for all* i *and* $\mu; \mu \Downarrow^{\Delta}_{\mathsf{C}} \{G\}$*, where*

$$G = a_1 \hookrightarrow (s_1, t_1, V_1, E_1, F_1) \uplus \ldots a_n \hookrightarrow (s_n, t_n, V_n, E_n, F_n)$$

Let $f \in F_i$*. If* $u \in f$ *and* $u' \notin f$ *and* $u' \preceq_G u$*, then* $f = \langle {}^{s}_{t} \rangle$ *for some* s, t *and* $u' \preceq_G s$*.*

Proof. By induction on the derivation of

$$e_i; \mu \Downarrow^{\Delta} v_i; (s_i, t_i, V_i, E_i, F_i)$$

If $e_i = \mathtt{fg}(e)$, then $e; \mu \Downarrow^{\Delta} v; g$ and $g_i = (g\odot) \oplus [u]$. We must have $g \neq \emptyset$ (since otherwise $F_i = \emptyset$), so by Lemma 8, we have $g = (s_i, t, V, E, \emptyset)$. Let $f \in F_i$. We have $f = \langle {}^{s_i}_{t} \rangle$ or $f = \langle {}_{t} \rangle$. Let $u_1 \in f$. If $u' \preceq_G u_1$ and $u' \notin f$, then there exist u_2 and u_3 such that $(u_2, u_3, \delta) \in G$ and $u' \preceq_G u_2$ and $u_3 \preceq_G u_1$ and $u_2 \notin f$ and $u_3 \in f$. We must have $u_2 \notin V$, so there is some a_i such that $(a_i, u_3, \delta) \in E$ and $u_2 \in V_i$. By Lemma 8, $u_3 = s_i$, so $u' \preceq_G s_i$. It remains to be shown that $f = \langle {}^{s_i}_{t} \rangle$. Suppose for the sake of contradiction that $f = \langle {}_{t} \rangle$. If this is the case, then since $u_1 \in f$, we must have $u_1 \preceq_G u$, and so $u' \preceq_G u$, but then $u' \in f$, which gives the required contradiction. $\square$

Lemma 12. *Fix Δ and suppose that $\cdot \vdash_{\cdot} \mu : \Sigma$ and that e $\texttt{wj}$ for all $a \hookrightarrow (\delta, e) \in \mu$. If $r; \mu \mapsto_{\texttt{glo}} r'; \mu'$ using a prompt scheduling policy, then*

1. $W(\mu', \Delta) \leq W(\mu, \Delta)$
2. $S(\mu', \Delta) \leq S(\mu, \Delta)$
3. $W(\mu, \Delta) - W(\mu', \Delta) \geq P$ *or* $S(\mu, \Delta) - S(\mu', \Delta) \geq 1$
4. $W^\circ(\mu', \Delta) \leq W^\circ(\mu, \Delta)$
5. $S^\circ(\mu', \Delta) \leq S^\circ(\mu, \Delta)$
6. $W^\circ(\mu, \Delta) - W^\circ(\mu', \Delta) \geq P$ *or* $S^\circ(\mu, \Delta) - S^\circ(\mu', \Delta) \geq r' - r$.

Proof. Suppose $\mu'; \mu' \Downarrow^{\Delta}_{\mathsf{C}} \{G'\}$. We first show conclusions 1-3, and then 4-6. By the definition of a prompt schedule, if N is the number of threads that step, either $N = P$ (where P is the number of processors) or $a_1, \ldots, a_N$ are the only threads in μ that are ready. If $N = P$, then for each $1 \leq i \leq N$, we have $e_i \mid \mu \mapsto^{\Delta}_{a_i} (\delta_i, e'_i) \mid \mu \uplus \mu_i$. We also have $G' = G_0 \uplus G'_1 \uplus \ldots G'_i$ where, for each i, $a_i \hookrightarrow (\delta_i, e'_i) \uplus \mu_i; \mu \Downarrow^{\Delta}_{\mathsf{C}} \{G'_i\}$. By Lemma 10, for each i, there exists g_i such that $e_i; \mu \Downarrow^{\Delta} v_i; g_i$ and $W(g_i) - W(G'_i) \geq 1$. Let $G = G_0 \uplus a_1 \hookrightarrow g_1 \uplus \ldots \uplus a_n \hookrightarrow g_n$. Since $\mu; \mu \Downarrow^{\Delta}_{\mathsf{C}} \{G\}$, we have $W(\mu, \Delta) - W(\mu', \Delta) \geq P$.

Now suppose $a_1, \ldots, a_N$ are the only threads in μ that are ready. The longest path in G' starts at a vertex u with no ancestor. Suppose u is the source of the thread dag of a (or an auxiliary vertex delaying the thread dag of a). If a is one of the a_i, then, by Lemma 10, there exists a g_i such that $e_i; \mu \Downarrow^{\Delta} v_i; g_i$ and u has an ancestor in g_i. If a is not one of the a_i and $\mu; \mu \Downarrow^{\Delta}_{\mathsf{C}} \{G\}$ then u has an ancestor in G or is an auxiliary vertex with a delay (since otherwise a would be ready in μ). In the latter case, the delay is increased by one in G. In any case, there exists a G with a longer path and $S(\mu, \Delta) - S(\mu', \Delta) \geq 1$.

We now show conclusions 4-6. By the definition of a prompt schedule, either $N = P$ and all N threads stepped are foreground threads (i.e. are running foreground blocks) or, for some $j < N$, $a_1, \ldots, a_j$ are the only ready foreground threads in μ. Suppose $N = P$. For each $1 \leq i \leq j$, we have $e_i \mid \mu \mapsto^{\Delta}_{a_i} (\delta_i, e'_i) \mid \mu \uplus \mu_i$. We also have $G' = G_0 \uplus G'_1 \uplus \ldots G'_i$ where, for each i, we have $a_i \hookrightarrow (\delta_i, e'_i) \uplus \mu_i; \mu \Downarrow^{\Delta}_{\mathsf{C}} \{G'_i\}$. By Lemma 10, for each i, there exists g_i such that $e_i; \mu \Downarrow^{\Delta} v_i; g_i$ and there exists a vertex u_i which is in g_i and is a proper ancestor in g_i of all vertices of G'_i. Since a_i is a foreground thread, u_i must be a foreground vertex in g_i. Let $G = G'_0 \uplus a_1 \hookrightarrow g_1 \uplus \ldots \uplus a_n \hookrightarrow g_n$. Since $\mu; \mu \Downarrow^{\Delta}_{\mathsf{C}} \{G\}$, there are P foreground vertices that are in G and not G', so $W^\circ(\mu, \Delta) - W^\circ(\mu', \Delta) \geq P$.

Suppose $a_1, \ldots, a_j$ are the only ready foreground threads in μ. If $j = 0$, then $RFB(\mu) = \emptyset$ and $r' - r = 0$. Otherwise, let $f \in RFB(G')$. The longest path in the subgraph of G' induced by f starts at a vertex u with no ancestor (in f or G'). Suppose $G' = a \hookrightarrow (u, u', V_a, E_a, F_a) \uplus G'_0$. By assumption, u has no ancestor and is a foreground vertex, so a is a ready foreground thread and must be one of the a_i for $1 \leq i \leq j$. By Lemma 10, there exists a g_i such that $e_i; \mu \Downarrow^{\Delta} v_i; g_i$ and u has an ancestor u' in g_i. Consider two cases: (1) If $u' \in f$, then for all G such that $\mu; \mu \Downarrow^{\Delta}_{\mathsf{C}} \{G\}$, we have that $f \in RFB(G) \cap RFB(G')$ and $S_G(f) - S_{G'}(f) \geq 1$. (2) If $u' \notin f$, then by Lemma 11, $f = \langle {s \atop t} \rangle$ and for all G such that $\mu; \mu \Downarrow^{\Delta}_{\mathsf{C}} \{G\}$, we have $u' \leq_{g_i} s$ and $f \notin RFB(G)$. This shows that for all G, $S(\mu, \Delta) - S(\mu', \Delta) \geq |RFB(G) \cap RFB(G')|$.

Now let G be such that $\mu; \mu \Downarrow^{\Delta}_{\mathtt{C}} \{G\}$ and let $f \in RFB(G) \setminus RFB(G')$. There must be $|RFB(\mu)| - |RFB(G) \cap RFB(G')|$ such blocks. Since the span of f in G is at least 1, each such f also contributes a decrease of at least one from $S^{\circ}(\mu, \Delta)$ to $S^{\circ}(\mu', \Delta)$. Together with the contribution from the previous paragraph, this completes the proof. □

The proof of the response time and computation time bounds is then straightforward.

Theorem 5. *Fix Δ and let e be such that $\cdot \vdash_{\cdot} e : \tau@\mathbb{B}$. Suppose $e; \emptyset \Downarrow^{\Delta} v; g$ If $0; a \hookrightarrow (0, e) \mapsto^{T}_{\mathtt{glo}} r; \mu$ using a prompt scheduling policy and μ `final`, then $T \leq \frac{W(g)}{P} + S(g)$ and $r \leq D(g)\frac{W^{\circ}(g)}{P} + S^{\circ}(g)$.*

Proof. Let $\mu_0 = a \hookrightarrow (0, e)$ and $\mu_T = \mu$ and $r_0 = 0$ and $r_T = r$. We have a sequence $0; \mu_0 \mapsto_{\mathtt{glo}} r_1; \mu_1 \mapsto_{\mathtt{glo}} \cdots \mapsto_{\mathtt{glo}} r_T; \mu_T$.

For each i, let $W_i = W(\mu_i, \Delta)$ (and similar for S_i, W_i° and S_i°. Note that $W_0 = W(g)$ (and similar for S, W°, S°) and that $W_T = S_T = W_T^{\circ} = S_T^{\circ} = 0$.

By Theorem 3, e_b `wj` for all $b \hookrightarrow (\delta, e_b) \in \mu_i$. By Lemma 12,

$$\frac{W_0}{P} + S_0 \geq 1 + \frac{W_1}{P} + S_1 \geq \cdots \geq 1 + \frac{W_T}{P} + S_T = 1$$

This immediately gives $\frac{W_0}{P} + S_0 \geq T$.

Let $D = D(g)$. For each i, consider the quantity $D\frac{W_i^{\circ}}{P} + S_i^{\circ} + r_i$. Note that for $i = 0$, $D\frac{W_i^{\circ}}{P} + S_i^{\circ} + r_i = D\frac{W^{\circ}(g)}{P} + S^{\circ}(g)$ and for $i = T$, $D\frac{W_i^{\circ}}{P} + S_i^{\circ} + r_i = r$. When $r_i; \mu_i \mapsto_{\mathtt{glo}} r_{i+1}; \mu_{i+1}$, by Lemma 12, either

1. $W_i^{\circ} - W_{i+1}^{\circ} \geq P$ and $r_{i+1} - r_i = |RFB(\mu_i)| \leq D$ (the last inequality is by definition of D) or
2. $S_i^{\circ} - S_{i+1}^{\circ} \geq |RFB(\mu_i)|$ and $r_{i+1} - r_i = |RFB(\mu_i)|$

In both cases, the quantity above decreases or remains the same, so $r \leq D(g)\frac{W_0^{\circ}}{P} + S_0^{\circ}$. □

5 Implementation and Evaluation

The operational semantics (Section 4) specifies an implementation at the level of threads and scheduling decisions. To realize the semantics in practice, we must implement the global scheduling step by giving a prompt scheduling algorithm. In order to approximate the operational semantics, which reschedules at each step, it is necessary to perform some preemption, using periodic interrupts, so that low-priority threads can be switched out for high-priority threads.

Next, prompt scheduling requires that, whenever the scheduler runs, it maps high-priority threads onto the available processors, followed by low-priority threads if any processors remain. A naïve implementation could use a global priority queue, but this would not scale beyond just a few processors due to the cost of synchronization at the queue. A realistic implementation therefore would have to distribute the queues. There are many ways to achieve this. In this paper, we build on a recently proposed variant of the work-stealing algorithm [2]. In our algorithm, each processor has a private priority queue and a public communication cell, a *mailbox*, to which other processors can send threads. At periodic intervals, each processor attempts to send, or *deal*, a thread to a random processor, in priority order, by atomically writing into the target processor's mailbox. Each processor then checks its own queue and mailbox and begins working on the highest-priority task available. Generalizations of work stealing to support priorities have been considered before [34] but these algorithms are not preemptive.

5.1 Implementation

We implemented the basic primitives of our formal language λ^{ip} as a Standard ML library, and implemented the preemptive priority-based work stealing algorithm described above by building on an existing parallel extension [57, 58] of the MLton [44] compiler for Standard ML. We have not extended SML's type system to implement λ^{ip}'s temporal type system because this is less essential for the performance analysis.

5.2 Experimental Setup

The experiments were performed on a 48-core machine with 125GB of memory and 2.1 GHz AMD CPUs running Ubuntu 16.04. To account for inherent noise in the data, we performed each run between 10 and 20 times and each data point represents the average over the runs. Through empirical analysis, we found that interrupt intervals in the range of 1-25 milliseconds lead to the best throughput and responsiveness. Details can be found in Section 5.5. For the results reported in this paper, we use a 5ms interval.

Measuring Responsiveness. Empirical analysis of interactive programs can be challenging because it requires isolating the completion time of potentially small pieces of computation (such as an interaction with a user) within an application. For example, prior work proposed operating system modifications [21]. We use a simpler approach. In our experiments, a *driver* program, written in C, reads a sequence of interactive events (e.g. mouse clicks, key presses) from a trace file. It simulates these events, records the response of the program to the event and measures the response time.

5.3 Quantitative Benchmarks

Fibonacci-Terminal. This benchmark performs a parallel Fibonacci computation, specifically `fib(45)` (to stand in for an intensive parallel computation), and simultaneously performs user interaction via a terminal. The user interaction consists of a loop that repeatedly reads a name from standard input, and immediately greets the user by name. To ensure responsiveness, the benchmark designates the terminal computation as high priority and the Fibonacci computation as low priority. The benchmark terminates once the `fib(45)` computation completes.

To assess the responsiveness of the Fibonacci-terminal benchmark, we run it while varying the number of processors between 1 and 30 and the rate of interaction between 1 and 50 interactions per second[6]. In the experiments, the driver program sends a name on standard input at uniform

[6]Due to a technical limitation of the thread-pinning library used by the runtime, we were unable to use all of the system's cores.

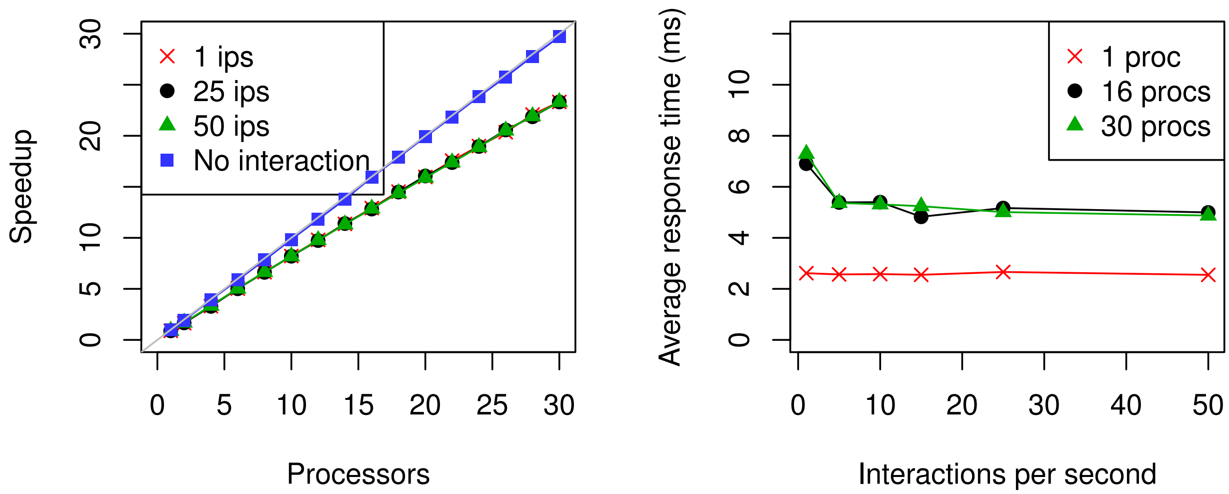

Figure 18: Speedup (l) and response time (r) for the Fibonacci-terminal benchmark.

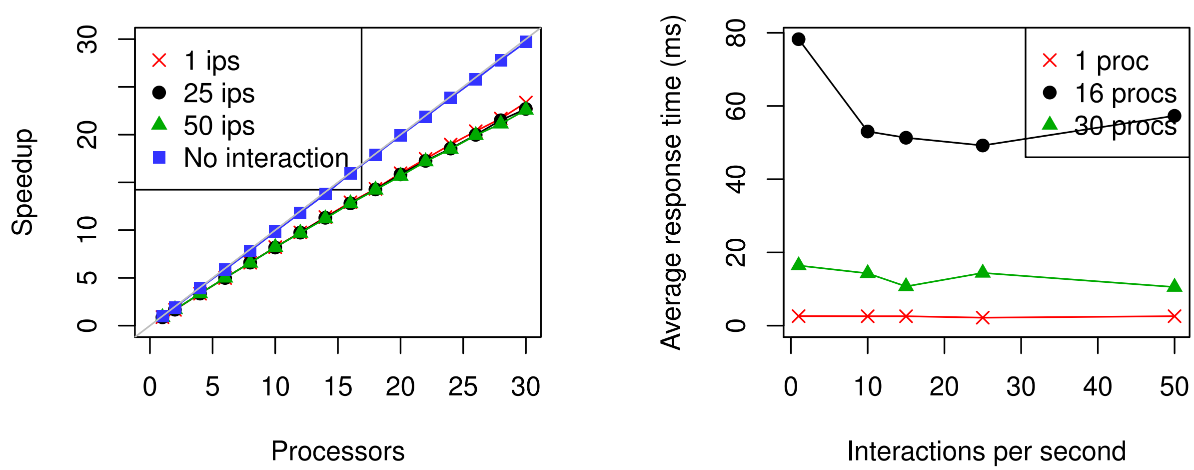

Figure 19: Speedup (l) and response time (r) for the Fibonacci-network benchmark.

intervals to match the desired number of interactions per second. It then waits for the response from the program. The time between the input and the response is the response time. We measure the response time for each input and take the average.

The left plot in Figure 18 shows the speedup (with respect to the sequential version of Fibonacci) of the Fibonacci computation as a function of the number of processors under varying interaction rates. For comparison, the figure also shows (in blue squares) the speedup of a standard work stealing scheduler running a Fibonacci computation only (with no interaction). The results show that interaction decreases the speedup, but not significantly. This is consistent with our bounds because interaction, which is high-priority, takes precedence over the low-priority Fibonacci computation. The right plot in Figure 18 shows the average response time as a function of the number of interactions per second. The average response time remains relatively flat even as the interaction rate increases, which is expected because each interaction involves little work (just echoing the input name). We furthermore see that increasing the number of processors causes an increase in the response time up to a point. This seems counterintuitive but is likely caused by migrations of high-priority computations to other processors via a deal, which can increase response time compared to the local handling of the same interaction. Overall, average response time remains very good, staying well under 10 milliseconds.

Fibonacci-Network. Our next benchmark has the same structure as the Fibonacci-terminal but involves more complex interaction. The benchmark opens a socket and listens for incoming connections. When a connection is received, it starts an interactive channel, implemented as a new high-priority thread. The interaction on each channel proceeds as in Terminal echo above, until the program terminates or the client disconnects. Because there can be many channels, each of which is handled by a thread, active at the same time, this benchmark tests the case where there are many interactive computations, all of which demand responsiveness.

In this benchmark, the driver program opens a number of network connections, and sends a line over each at one-second intervals, staggered so that the messages arrive at uniform intervals. The number of network connections opened is the desired number of interactions per second. Figure 19 shows the results. We see again that the Fibonacci computation scales well with respect to the sequential baseline with varying levels of interaction. As above, the one-processor case shows the best responsiveness and the average response times are good, under 100 milliseconds.

Fibonacci Server. In the above benchmarks, the interactive and computational parts of the benchmark did not interact, apart from the fact that they run together. Our next benchmark, the "Fibonacci server", simulates an application that receives queries, each of which requires performing some compute-intensive task. An interactive, high-priority loop waits for the user to enter a number on the console. When a number n is entered, the loop starts the computation of the n^{th} Fibonacci number in a separate low-priority thread that also prints the result on the console; the loop continues to listen to further inputs immediately after starting the Fibonacci computation.

To assess this benchmark, our driver program runs the benchmark with a trace that inputs the numbers 41 to 45 in increasing order. The driver calculates the response time as the time between the input and the next prompt, and the computation time as the time taken to compute each Fibonacci number. The rate of interaction varies from 0.5 to 2.0 inputs per second. Figure 20 shows the results. As expected, both numbers begin to increase as the interaction becomes frequent enough that the Fibonacci computations overlap. The computations still complete in a timely matter, and the program remains responsive, with average response times not exceeding several milliseconds.

Interactive Convex Hull. Our interactive convex hull benchmark maintains the convex hull of a set of 2D points, as a user inserts new points by clicking on the screen. A high-priority loop polls the mouse and, every time the user adds a point, starts a low-priority thread that computes and draws

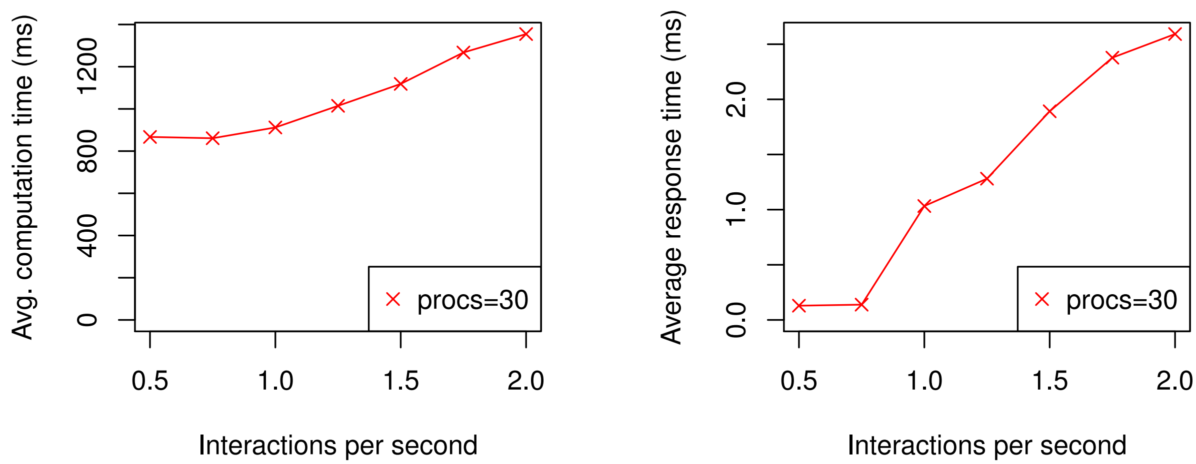

Figure 20: The effect of interaction rate for the Fibonacci server on: (l) Computation time (r) Response time.

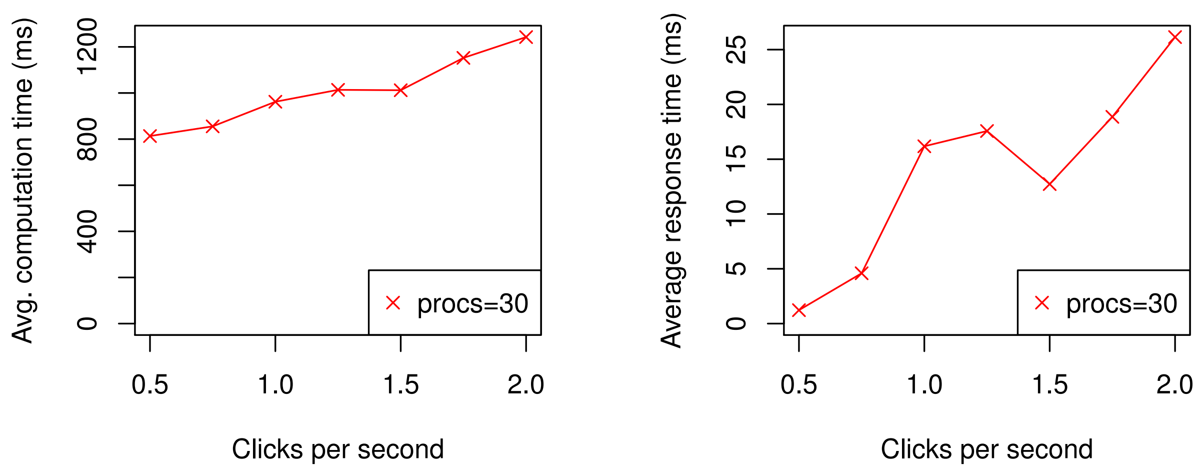

Figure 21: The effect of interaction rate for the convex hull server on: (l) Computation time (r) Response time.

the new convex hull. In our experiments, the driver program simulates five clicks at random points on the screen at regular intervals and calculates the response time for each click as the time between the click and drawing of the point, and the computation time as the time to compute each hull. So that the hull computations are not trivial, each one includes 1,000,000 random points that are chosen at initialization. Figure 21 shows the results. As with the Fibonacci server, the computation times and response times increase with the interaction rate; this is expected because increased interaction rate causes multiple convex hull computations to overlap with each other and with the interaction. The program still remains responsive to clicks, with response times under 30ms.

5.4 Qualitative Benchmarks

In addition to the relatively simple benchmarks above, we considered more sophisticated benchmarks. These benchmarks are more difficult to evaluate quantitatively, but we assess their performance qualitatively by their usability.

Web Server. A high-priority loop listens for connections and starts a new high-priority thread for each one. HTTP requests are logged, and a low-priority thread periodically performs analytics on the log. We simulate a large analytics computation by computing a large Fibonacci number in parallel. As expected, we observe that the background computations do not interfere with the handling of HTTP requests.

Photo Viewer. Our photo viewer benchmark allows the user to navigate through a folder of JPEG images, either by scrolling or jumping to an image. To ensure smooth scrolling, the user interaction is high priority, and the viewer decodes the next several images in the background so they will be ready when requested. If the user selects an image that has not yet been decoded, it is decoded in the foreground and displayed. Our experience shows that the viewer is responsive, indicating that the decoding is proceeding quickly enough to be effective, and that the background decoding processes do not hamper the high-priority interaction.

Music Server. A streaming music server listens for network connections and spawns a thread for each new client. The client requests a music file from the server, which the server streams over the connection until the end of the file is reached. Some clients (perhaps those paying for a higher level of service) are designated high-priority, and are handled by high-priority threads; the remaining clients receive low priority. We tested the server with a relatively small number of clients (up to 10, both low and high priority), and in our experience, it maintains a high quality of service for all clients.

5.5 Interrupts

To faithfully implement prompt scheduling, our scheduler implements preemption by delivering interrupts at regular intervals. By performing the interrupts in user space as part of our scheduler, we keep the overhead of preemption small. Figure 22 reports the effect of varying the interrupt frequency. In both plots, the x-axis is the interrupt interval in microseconds, shown in log scale. The plot on the left shows the computation time of the Fibonacci-terminal and Fibonacci-network benchmarks (on 30 processors). As we would expect, computation time increases with longer intervals (above 25ms) since these long intervals could result in long delays between deals. Shorter intervals (below 1ms) increase computation time due to the overhead of interrupts. The plot on the

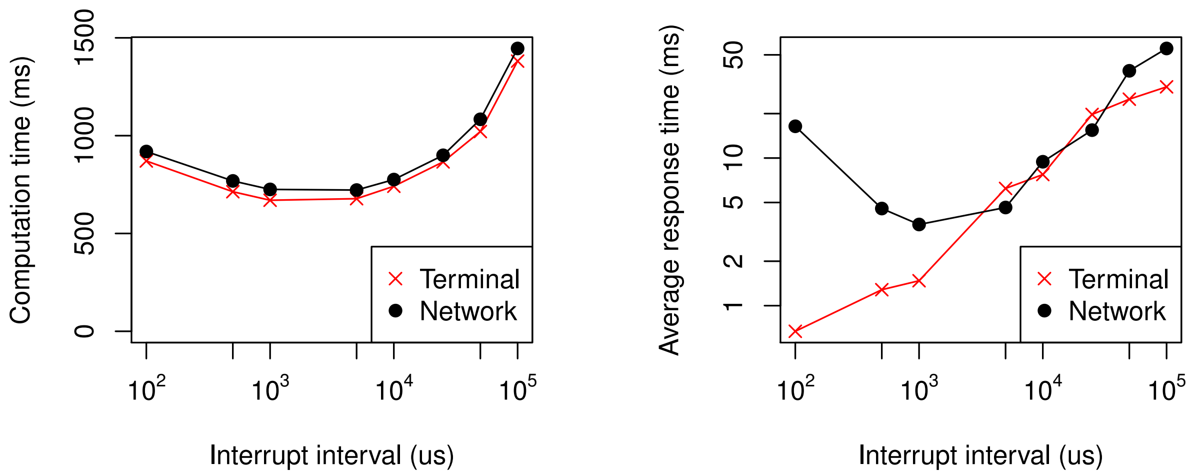

Figure 22: The effect of interrupt period on: (l) Computation time (r) Response time.

right shows the response time (in log scale). As expected, the response time generally increases with the interval at which background tasks are interrupted to handle the interaction.

6 Related Work

We discussed the most closely related work in the main body of the paper. Here we take a broader perspective and briefly describe more remotely related work.

Parallel Computing. Much work has been done on parallel computing with dynamically scheduled, fine-grained and cooperative threads since the 1970s [5, 30, 10, 26, 41, 18, 17, 42, 39, 36, 25, 37, 35, 50]. Nearly all of this work focuses on maximizing throughput in compute-intensive applications and relies on cooperative threading. This paper shows that the language abstractions, dag-based cost models [28, 38, 13] and cost semantics [8, 9, 29], can be extended to include competitive threading, where threads are scheduled preemptively.

Type Systems for Staged Computation. The type system of λ^{ip} is based on that of Davies [19] for binding time analysis, which is derived from linear temporal logic. This work influenced much followup work on metaprogramming and staged computation [40, 49, 60, 47]. These systems allow a computation at a stage to create and manipulate, but not eliminate, a computation in a later stage. For example, a stage 1 computation can create a stage 2 computation as a "black box" but cannot inspect that computation. We use a two-stage variant of the $\bigcirc$ modality of Davies [19], similar to that of Feltman et al. [23], which inspires some of our notation. One important difference between stages and the priorities of our work is that, in our work, computations belonging to different stages (priorities) can be evaluated concurrently, whereas in staged computations, evaluation proceeds monotonically in stage order.

Cost Semantics. The idea of using a cost semantics to reason about efficiency of programs goes back to the early 1990s [51, 53] and has since been applied in a number of contexts [53, 54, 8, 9, 58, 43]. Our approach builds directly on the work of Blelloch and Greiner [9] and Spoonhower et al. [58], who use computation graphs represented as dags (directed acyclic graphs) to reason about time and space in functional parallel programs. These cost models, however, consider cooperatively threaded parallelism only.

Scheduling. Our prompt-scheduling results generalize Brent's classic result for scheduling parallel computations [15]. Since Brent's result, much work has been done on scheduling. Ullman [61], Brent [15], and Eager et al. [20] established the hardness of optimal scheduling and the greedy scheduling principle. These early results have led to many more algorithms [16, 30, 26, 13, 48, 1, 22, 2, 3]. More recent papers showed that priority-based schedulers can improve performance in practice [62, 63, 34]. Our weighted-dag model builds on the model of Muller and Acar [45], who developed an algorithm for scheduling blocking parallel programs to hide latency, but did not consider responsiveness.

Scheduling is also studied extensively in the operating systems community (a book by Silberschatz et al. [55] presents a comprehensive overview). There has been significant interest in making operating systems work well on multicore machines [6, 14]. The focus, however, has been on reducing contention within the OS and, as in the high-performance computing community, distributing resources to jobs so that they can run effectively. Scheduling within a job, which is our main concern, is less central to systems research.

There has been a great deal of work on scheduling for responsiveness in queuing theory (Harchol-Balter [31] presents a comprehensive overview). This line of work assumes a continuous stream of independent jobs arriving for processing according to some stochastic process. Such arrival assumptions do not quite fit the parallel computing model, where work is created by a program. In queuing theory, each job is generally processed by a single processor (or "server") that decides at every point in time which of the current jobs to run. This work, however, typically assumes jobs to be sequential.

Scheduling is also an important concern in real-time computing. Most of this work considers highly structured (usually synchronous) sequential computations. Saifullah et al. [52] consider scheduling a set of real-time tasks where each task is a parallel computation represented by a parallel dag. Their algorithm infers for each vertex in the dag a deadline and schedules the vertices according to their deadlines. Their work assumes that the tasks are independent and are known in advance, as is the dag structure.

7 Conclusion

This paper takes a step toward uniting cooperative and competitive threading. To this end, we consider a programming language with fork-join parallelism, interaction, and priorities, and extend the classic cost models for cooperative threading based on cost graphs and cost semantics to bound both run-time and responsiveness. Our implementation and experiments suggest that the approach can be made practical. We leave a number of questions to future work, including the extension of our techniques to multiple priorities (instead of the two priorities we consider), the development of an efficient scheduling algorithm that implements the prompt-scheduling principle, and a more detailed evaluation.

Acknowledgments

This research is partially supported by grants from the National Science Foundation (CCF-1320563, CCF-1408940, CCF-1629444) and European Research Council (grant ERC-2012-StG-308246), and by a gift from Microsoft Research. We thank Tim Harris and Ziv Scully for their feedback.

References

[1] U. A. Acar, G. E. Blelloch, and R. D. Blumofe. The data locality of work stealing. *Theory of Computing Systems (TOCS)*, 35(3):321–347, 2002.

[2] U. A. Acar, A. Charguéraud, and M. Rainey. Scheduling parallel programs by work stealing with private deques. In *PPoPP '13*, 2013.

[3] U. A. Acar, A. Charguéraud, and M. Rainey. Oracle-guided scheduling for controlling granularity in implicitly parallel languages. *Journal of Functional Programming (JFP)*, 26: e23, 2016.

[4] N. S. Arora, R. D. Blumofe, and C. G. Plaxton. Thread scheduling for multiprogrammed multiprocessors. *Theory of Computing Systems*, 34(2):115–144, 2001.

[5] Arvind and K. P. Gostelow. The Id report: An asychronous language and computing machine. Technical Report TR-114, Department of Information and Computer Science, University of California, Irvine, Sept. 1978.

[6] A. Baumann, P. Barham, P.-E. Dagand, T. Harris, R. Isaacs, S. Peter, T. Roscoe, A. Schüpbach, and A. Singhania. The multikernel: A new OS architecture for scalable multicore systems. In *Proceedings of the ACM SIGOPS 22nd Symposium on Operating Systems Principles*, SOSP '09, pages 29–44, 2009.

[7] G. Blake, R. G. Dreslinski, T. Mudge, and K. Flautner. Evolution of thread-level parallelism in desktop applications. In *Proceedings of the 37th Annual International Symposium on Computer Architecture*, ISCA '10, pages 302–313, 2010.

[8] G. Blelloch and J. Greiner. Parallelism in sequential functional languages. In *Proceedings of the 7th International Conference on Functional Programming Languages and Computer Architecture*, FPCA '95, pages 226–237. ACM, 1995.

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

[10] G. E. Blelloch, J. C. Hardwick, J. Sipelstein, M. Zagha, and S. Chatterjee. Implementation of a portable nested data-parallel language. *J. Parallel Distrib. Comput.*, 21(1):4–14, 1994.

[11] G. E. Blelloch, P. B. Gibbons, and Y. Matias. Provably efficient scheduling for languages with fine-grained parallelism. *J. ACM*, 46:281–321, Mar. 1999.

[12] R. D. Blumofe and C. E. Leiserson. Space-efficient scheduling of multithreaded computations. *SIAM Journal on Computing*, 27(1):202–229, 1998.

[13] R. D. Blumofe and C. E. Leiserson. Scheduling multithreaded computations by work stealing. *J. ACM*, 46:720–748, Sept. 1999.

[14] S. Boyd-Wickizer, H. Chen, R. Chen, Y. Mao, F. Kaashoek, R. Morris, A. Pesterev, L. Stein, M. Wu, Y. Dai, Y. Zhang, and Z. Zhang. Corey: An operating system for many cores. In *Proceedings of the 8th USENIX Conference on Operating Systems Design and Implementation*, pages 43–57, 2008.

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

[16] F. W. Burton and M. R. Sleep. Executing functional programs on a virtual tree of processors. In *Functional Programming Languages and Computer Architecture (FPCA '81)*, pages 187–194. ACM Press, Oct. 1981.

[17] M. M. T. Chakravarty, R. Leshchinskiy, S. L. Peyton Jones, G. Keller, and S. Marlow. 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*, pages 10–18, 2007.

[18] P. Charles, C. Grothoff, V. Saraswat, C. Donawa, A. Kielstra, K. Ebcioglu, C. von Praun, and V. Sarkar. 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, pages 519–538. ACM, 2005.

[19] R. Davies. A temporal-logic approach to binding-time analysis. In *LICS*, pages 184–195, 1996.

[20] D. L. Eager, J. Zahorjan, and E. D. Lazowska. Speedup versus efficiency in parallel systems. *IEEE Transactions on Computing*, 38(3):408–423, 1989.

[21] Y. Endo, Z. Wang, J. B. Chen, and M. Seltzer. Using latency to evaluate interactive system performance. In *Proceedings of the Second USENIX Symposium on Operating Systems Design and Implementation*, OSDI '96, pages 185–199, New York, NY, USA, 1996. ACM.

[22] D. G. Feitelson, L. Rudolph, and U. Schwiegelshohn. Parallel job scheduling - A status report. In *Job Scheduling Strategies for Parallel Processing (JSSPP), 10th International Workshop*, pages 1–16, 2004.

[23] N. Feltman, C. Angiuli, U. A. Acar, and K. Fatahalian. Automatically splitting a two-stage lambda calculus. In *Proceedings of the 25 European Symposium on Programming, ESOP*, pages 255–281, 2016.

[24] K. Flautner, R. Uhlig, S. Reinhardt, and T. Mudge. 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, pages 129–138, 2000.

[25] M. Fluet, M. Rainey, J. Reppy, and A. Shaw. Implicitly threaded parallelism in Manticore. *Journal of Functional Programming*, 20(5-6):1–40, 2011.

[26] M. Frigo, C. E. Leiserson, and K. H. Randall. The implementation of the Cilk-5 multithreaded language. In *PLDI*, pages 212–223, 1998.

[27] C. Gao, A. Gutierrez, R. G. Dreslinski, T. Mudge, K. Flautner, and G. Blake. A study of thread level parallelism on mobile devices. In *Performance Analysis of Systems and Software (ISPASS), 2014 IEEE International Symposium on*, pages 126–127, March 2014.

[28] R. L. Graham. Bounds on multiprocessing timing anomalies. *SIAM Journal on Applied Mathematics*, 17(2):416–429, 1969.

[29] J. Greiner and G. E. Blelloch. A provably time-efficient parallel implementation of full speculation. *ACM Transactions on Programming Languages and Systems*, 21(2):240–285, Mar. 1999.

[30] R. H. Halstead. Multilisp: a language for concurrent symbolic computation. *ACM Transactions on Programming Languages and Systems*, 7:501–538, 1985.

[31] M. Harchol-Balter. *Performance Modeling and Design of Computer Systems: Queueing Theory in Action*. Cambridge University Press, 2013.

[32] R. Harper. *Practical Foundations for Programming Languages*. Cambridge University Press, New York, NY, USA, 2012.

[33] C. Hauser, C. Jacobi, M. Theimer, B. Welch, and M. Weiser. Using threads in interactive systems: A case study. In *Proceedings of the Fourteenth ACM Symposium on Operating Systems Principles*, SOSP '93, pages 94–105, New York, NY, USA, 1993. ACM.

[34] S. Imam and V. Sarkar. Load balancing prioritized tasks via work-stealing. In *Euro-Par 2015: Parallel Processing - 21st International Conference on Parallel and Distributed Computing*, pages 222–234, 2015.

[35] S. M. Imam and V. Sarkar. 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*, pages 75–86, 2014.

[36] Intel. Intel threading building blocks, 2011. `https://www.threadingbuildingblocks.org/`.

[37] S. Jagannathan, A. Navabi, K. Sivaramakrishnan, and L. Ziarek. The design rationale for Multi-MLton. In *ML '10: Proceedings of the ACM SIGPLAN Workshop on ML*. ACM, 2010.

[38] J. Jaja. *An introduction to parallel algorithms*. Addison Wesley Longman Publishing Company, 1992.

[39] G. Keller, M. M. Chakravarty, R. Leshchinskiy, S. Peyton Jones, and B. Lippmeier. Regular, shape-polymorphic, parallel arrays in Haskell. In *Proceedings of the 15th ACM SIGPLAN international conference on Functional programming*, ICFP '10, pages 261–272, 2010.

[40] T. B. Knoblock and E. Ruf. Data specialization. In *Proceedings of the ACM SIGPLAN 1996 Conference on Programming Language Design and Implementation*, PLDI '96, pages 215–225, 1996.

[41] D. Lea. A Java fork/join framework. In *Proceedings of the ACM 2000 conference on Java Grande*, JAVA '00, pages 36–43, 2000.

[42] D. Leijen, W. Schulte, and S. Burckhardt. 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, pages 227–242, 2009.

[43] R. Ley-Wild, U. A. Acar, and M. Fluet. A cost semantics for self-adjusting computation. In *Proceedings of the 26th Annual ACM Symposium on Principles of Programming Languages*, POPL '09, 2009.

[44] MLton. MLton web site. `http://www.mlton.org`.

[45] S. K. Muller and U. A. Acar. 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 '16, pages 71–82, 2016.

[46] T. Murphy, VII, K. Crary, R. Harper, and F. Pfenning. A symmetric modal lambda calculus for distributed computing. In *Proceedings of the 19th IEEE Symposium on Logic in Computer Science (LICS)*, pages 286–295. IEEE Press, 2004.

[47] A. Nanevski and F. Pfenning. Staged computation with names and necessity. *J. Funct. Program.*, 15(5):893–939, 2005.

[48] G. J. Narlikar and G. E. Blelloch. Space-efficient scheduling of nested parallelism. *ACM Transactions on Programming Languages and Systems*, 21, 1999.

[49] F. Pfenning and R. Davies. A judgmental reconstruction of modal logic. *Mathematical Structures in Computer Science*, 11:511–540, 2001.

[50] R. Raghunathan, S. K. Muller, U. A. Acar, and G. Blelloch. Hierarchical memory management for parallel programs. In *Proceedings of the 21st ACM SIGPLAN International Conference on Functional Programming*, ICFP 2016, pages 392–406, New York, NY, USA, 2016. ACM.

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

[52] A. Saifullah, D. Ferry, J. Li, K. Agrawal, C. Lu, and C. D. Gill. Parallel real-time scheduling of DAGs. *IEEE Trans. Parallel Distrib. Syst.*, 25(12):3242–3252, 2014.

[53] D. Sands. Complexity analysis for a lazy higher-order language. In *ESOP '90: Proceedings of the 3rd European Symposium on Programming*, pages 361–376, London, UK, 1990. Springer-Verlag.

[54] P. M. Sansom and S. L. Peyton Jones. Time and space profiling for non-strict, higher-order functional languages. In *Principles of Programming Languages*, pages 355–366, 1995.

[55] A. Silberschatz, P. B. Galvin, and G. Gagne. *Operating system concepts (7. ed.)*. Wiley, 2005.

[56] D. C. Smith, C. Irby, R. Kimball, B. Verplank, and E. Harslem. Designing the star user interface. *BYTE Magazine*, 7(4):242–282, 1982.

[57] D. Spoonhower. *Scheduling Deterministic Parallel Programs*. PhD thesis, Carnegie Mellon University, Pittsburgh, PA, USA, 2009.

[58] D. Spoonhower, G. E. Blelloch, R. Harper, and P. B. Gibbons. Space profiling for parallel functional programs. In *International Conference on Functional Programming*, 2008.

[59] D. C. Swinehart, P. T. Zellweger, R. J. Beach, and R. B. Hagmann. A structural view of the cedar programming environment. *ACM Trans. Program. Lang. Syst.*, 8(4):419–490, Aug. 1986.

[60] W. Taha and T. Sheard. MetaML and multi-stage programming with explicit annotations. *Theoretical Computer Science*, 248(1):211 – 242, 2000.

[61] J. Ullman. NP-complete scheduling problems. *Journal of Computer and System Sciences*, 10 (3):384 – 393, 1975.

[62] M. Wimmer, D. Cederman, J. L. Träff, and P. Tsigas. Work-stealing with configurable scheduling strategies. In *Proceedings of the 18th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPoPP '13, pages 315–316, 2013.

[63] M. Wimmer, F. Versaci, J. L. Träff, D. Cederman, and P. Tsigas. Data structures for task-based priority scheduling. In *Proceedings of the 19th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPoPP '14, pages 379–380, 2014.