Responsive Parallelism with Dynamic and First-Class Priorities

MARELLE LEÓN, Illinois Institute of Technology, USA
MY DINH, Illinois Institute of Technology, USA
STEFAN K. MULLER, University of Connecticut, USA

PriML, a language developed in recent work on *responsive parallelism*, extends traditional fine-grained parallel languages such as Cilk by allowing programmers to annotate threads with *priorities*. Programmers thus get the substantial throughput benefits of lightweight threads scheduled by a user-level runtime, while retaining the ability to use threads for responsive applications typically programmed with lower-level threading systems. PriML's type system guarantees the absence of *priority inversions*, costly performance errors in which high-priority threads are delayed by low-priority threads, enabling formal bounds on throughput and responsiveness, but at the cost of expressiveness: threads may not change priority once spawned and the priority of a thread is specified as a priority literal in the code (i.e., priorities are not *first-class*). This work relaxes the two assumptions above by tracking *sets of priorities* using techniques drawn from the literature on refinement types. We extend the graph-based cost models of responsive parallelism to incorporate first-class and changing priorities, and prove bounds on the throughput and responsiveness of well-typed programs in our extended language. We implement our type system extensions in the PriML compiler, and demonstrate the benefits of the extension using a concurrent web server as a case study.

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

Additional Key Words and Phrases: priority scheduling, priority inversion, liquid types, cost semantics

ACM Reference Format:
Marelle León, My Dinh, and Stefan K. Muller. 2026. Responsive Parallelism with Dynamic and First-Class Priorities. *Proc. ACM Program. Lang.* 10, PLDI, Article 256 (June 2026), 23 pages. https://doi.org/10.1145/3808334

1 Introduction

A recent line of work on *responsive parallelism* has aimed to combine two forms of multithreading: *cooperative threading* in which fine-grained threads divide the work of a program onto multiple processors in order to maximize *throughput*, and *competitive threading* in which heavier-weight threads achieve *responsiveness* by timeslicing on one or more processors. In responsive parallelism, like cooperative threading, fine-grained threads are managed by a user-level scheduler, allowing programs to scale to orders of magnitude more threads than if system-level threads were used. At the same time, as in many competitive threading libraries, these threads can be assigned *priorities*, allowing the scheduler to maintain responsiveness of high-priority threads.

PriML [24], the key language for responsive parallelism, uses a type system to statically track the priorities of threads in order to rule out *priority inversions*, a common bug in which high-priority threads wait (potentially indefinitely) on lower-priority threads. This is necessary for several reasons: first, the absence of priority inversions is necessary in order to maintain provable

Authors' Contact Information: Marelle León, Illinois Institute of Technology, Chicago, USA, mlean@hawk.illinoistech.edu; My Dinh, Illinois Institute of Technology, Chicago, USA, mdinh@hawk.iit.edu; Stefan K. Muller, University of Connecticut, Storrs, USA, stefan.muller@uconn.edu.

ACM 2475-1421/2026/6-ART256
https://doi.org/10.1145/3808334

efficiency bounds, a decades-old cornerstone of cooperative threading. Second, in a more general sense, priority inversions have plagued concurrent systems for decades [18]. In addition, static notions of priority and priority inversion are especially relevant today given the increasing reliance on cloud containers and similar systems in which threads at different priorities share limited virtualized resources. Priority inversions can more easily starve high-priority threads for resources when fewer CPUs are available, and virtualization can thwart existing dynamic mechanisms for identifying and prioritizing foreground threads [35].

In order to keep the type system simple, PriML makes two assumptions which together ensure that each thread can be associated in the type system with one priority. First, priorities are *hard-coded*: when a thread is spawned, the priority is specified as a priority literal, such as `High` or `Low`, and not an arbitrary expression. In other words, priorities are not *first class*. Second, priorities are *fixed*: the priority of a thread does not change once it is spawned (if it did, the type system would need to reason about all of the priorities a thread might take on).

Both of these restrictions run counter to how one typically programs in competitive multithreading systems such as POSIX threads (pthreads), where priorities are often represented as integers in a fixed range. In particular, consider spawning a thread `t` at a slightly higher priority, doing some other work in the meantime, and waiting for `t` to complete. If the priority of the current thread is `curr_prio`, in syntax resembling that of PriML, we would write:

```
1 t <- spawn (curr_prio + 1) { ... (* Work of t *) }; ... (* More work *) ; sync t
```

Unfortunately, this code is not possible to express in PriML: because priorities are not first-class, they cannot be stored in variables like `curr_prio`, nor can they be used in calculations like the increment, two features that programmers take for granted with first-class objects like integers.

Nor is it a simple endeavor to add useful first-class priorities to PriML's type system. It is clear to a human that there is no priority inversion in the above code because `t` has a higher priority than the current thread (assuming no corner cases such as integer overflow). However, for the type system to prove this, it must know both that the variable `curr_prio` indeed contains the current priority of the thread as intended and that the increment operation returns a higher-or-equal priority. Such reasoning requires at least some of the power of dependent types.

The inability to reason about threads that change priority (henceforth *dynamic* priorities) is also quite restrictive. Consider a web server that handles clients using multiple foreground threads while, in the background, processing request logs in parallel to identify clients that are overloading the system [34]. When the system is lightly loaded, the background processing is low priority. However, when the system is heavily loaded (a denial-of-service attack might be occurring), we wish to prioritize classifying clients over accepting new ones, and do this by dynamically adjusting the priority of the processing threads. Even setting aside programs that explicitly change their own priorities, it is important to be able to *reason* about threads that change priorities, because there are many reasons that a scheduler (whether OS or user-level) might want to change the priority of a running thread. This is true even in cooperative threading systems that have no explicit notion of priorities: for example, the scheduler of Concurrent ML [27] uses heuristics to classify threads as compute-intensive or communication-intensive and prioritizes them accordingly.

The state of the art, in which PriML combines some of the benefits of competitive threading (e.g. pthreads) and cooperative threads (e.g., Cilk) and statically prevents priority inversions at the cost of first-class and dynamic priorities, is shown in Table 1. In this paper, we extend PriML to allow threads to change priority, and to make priorities first-class objects of the language that can be computed on (as in the increment example) like any other value. As it happens, there is a large overlap in the technical machinery required to add each of these affordances. The addition of either feature requires the type system to track, for each thread, a *set of possible priorities* as opposed to

Table 1. A comparison of features between commonly used examples of competitive and cooperative threading systems, existing PriML, and our work.

Feature	Pthreads	Cilk	PriML	This work
Fine-grained threads, throughput bounds		✓	✓	✓
Priorities	✓		✓	✓
Static priority inversion prevention			✓	✓
First-class priorities	✓			✓
Dynamic priorities	✓			✓

a single priority. This set indicates uncertainty about the priority of a thread, both because the priority may be determined at runtime and not known statically (due to first-class priorities) and because it may actually change at runtime (due to dynamic priorities). We track sets of priorities by *refining* the types of priorities by a set of constraints. As with most refinement type systems, we must exercise care to ensure that type checking and inference remain decidable. We do so by syntactically restricting the constraints allowed in refinements, in a manner inspired by *liquid types* [29], while still allowing many useful properties to be expressed. With our extensions, given the code `spawn (incr curr_prio)` (generalizing the `+ 1` of the earlier example to an increment function `incr`), the type system is able to reason that the current thread has priority `curr_prio` and `incr` returns a priority greater than that of its argument, making the situation described above safe.

Unfortunately, the task was not as simple as adding a refined type of priorities and calling it a day, because of the way in PriML code composes commands into threads. Suppose we have the code `change (incr curr_prio)`, which uses the `change` keyword, introduced in this paper, to increment the priority of the current thread. In addition to impacting the set of priorities the thread can take on, this change of priority also places additional restrictions on the priority at which this thread may start (e.g., if after incrementing its priority, the thread waits on a thread of priority 10, the pre-increment priority of the thread must be no higher than 9). We also need to track the set of priorities at which a command might *end* execution so that we can compose it with another command. In the end, we must reason about 3 sets of priorities for each command (the full set of priorities, the start set, and the end set), with different interpretations and variance properties.

This paper makes the following contributions:

- A core calculus λ^4_ψ (Section 2) for responsive parallel programs with priorities that are *dynamic* (changeable at runtime) and *first class*. We equip λ^4_ψ with a refinement type system that tracks the priorities of threads and prevents priority inversions.
- A graph-based model for analyzing the cost of λ^4_ψ programs and for guaranteeing throughput and responsiveness under certain assumptions about the program. These assumptions correspond to the absence of priority inversions, and the type system of λ^4_ψ therefore guarantees that programs meet the assumptions for efficient scheduling (Section 3).
- An extension of PriML with the new theory above. The implementation includes an algorithm for inferring the refinement types of priorities (Section 4). It also uses a modernized backend that outputs Multicore OCaml code, allowing PriML programs to interface with OCaml libraries in a straightforward way.
- An implementation in PriML of a small, but working, web server that uses dynamic and first-class priorities as described earlier (Section 5).

Due to space limitations, many proofs and other details of the theoretical contributions are omitted from the body of the paper, but are provided in the appendix available as supplementary material.

$$
\begin{array}{llll}
\textit{Refinements} & \psi & ::= & x :: C \\
\textit{Qualifiers} & C & ::= & v = v \mid v \preceq v \mid C \wedge C \\
\textit{Types} & \tau & ::= & \texttt{unit} \mid x : \tau \to \tau \mid \texttt{prio}[\psi] \mid \tau\ \texttt{thread}[\psi] \mid \tau\ \texttt{cmd}[\psi \to \psi \to \psi] \\
\textit{Values} & v & ::= & x \mid \langle\rangle \mid \texttt{fun}\ f\ x = e \mid \rho \mid \texttt{tid}[a] \mid \texttt{cmd}\ \{m\} \\
\textit{Expressions} & e & ::= & v \mid \texttt{let}\ x = e\ \texttt{in}\ e \mid v\ v \mid \texttt{if}\ v \preceq v\ \texttt{then}\ e\ \texttt{else}\ e \\
\textit{Commands} & m & ::= & x \leftarrow e; m \mid \texttt{spawn}[\psi; \tau](e)\ \{m\} \mid \texttt{sync}\ e \mid \texttt{change}\ e \mid \texttt{ret}\ e
\end{array}
$$

Fig. 1. Syntax of λ^4_ψ

2 The λ^4_ψ Calculus

This section defines λ^4_ψ, a core calculus for fine-grained parallelism with priorities which extends λ^4 [24] with first-class priorities and a mechanism for threads to change their priority. In the presentation that follows, we will focus on the novel aspects of the calculus with respect to λ^4. The syntax of λ^4_ψ appears in Figure 1. We assume a finite set of priorities R equipped with a partial order $\preceq$ (where if $\rho_1 \preceq \rho_2$, then ρ_1 is lower priority than ρ_2). Unlike in prior calculi for responsive parallelism, priorities are first-class values of a priority type, which is constrained by *refinements* ψ. In a refinement of the form $x :: C$, the bound variable x represents the priority being discussed and C constrains the value of x. Qualifiers C are conjunctions of equalities and inequalities between priorities. As an example, the refinement $x :: \mathtt{L} \preceq x \wedge x \preceq \mathtt{H}$ indicates a priority bounded below by $\mathtt{L}$ and above by $\mathtt{H}$ (inclusive).

The syntax for expressions is presented in *monadic* form (e.g., [4]): most expressions are limited to accepting values v as subexpressions. This causes no loss of expressivity as intermediate expressions can be bound with `let`s. Our implementation does not impose these constraints, but using this discipline locally for expressions of priority type is occasionally helpful to express desired priority constraints, as we discuss further in Section 4. Expressions include the base type type `unit`; we explore adding a type `nat`, as well as sums and products, in the appendix. Function types $x : \tau_1 \to \tau_2$ bind the *name* of the argument x so that it can be used in refinements inside τ_2. Functions $\texttt{fun}\ f\ x = e$ are recursive by default and bind the function name f inside e.

Operations involving threads and priorities are included in a monadic layer consisting of *commands* m. The expression $\texttt{cmd}\ \{m\}$ represents an encapsulated command m. The bind operation $x \leftarrow e; m$ evaluates e to an encapsulated command, executes it, and binds its result to x while executing m. The command $\texttt{spawn}[\psi; \tau](e)\ \{m\}$ spawns a new thread to evaluate m and returns a handle to the new thread. The priority of that thread is supplied as an expression e of priority type. This is a key difference with respect to λ^4, in which (because priorities are not first class), a priority literal must be supplied rather than an expression e. For convenience in defining the rules, the command is annotated with a refinement ψ constraining the priorities at which the thread will execute (which will include the starting priority e but also any priorities to which the thread may switch) and the return type τ of the thread. These annotations are inferred in our implementation. The command `sync` e evaluates e to a thread handle and waits until the thread completes, returning its value. The command `change` e evaluates e to a priority and changes the current priority of the current thread to that priority. The command `ret` e evaluates e and returns its value.

An encapsulated command has the type $\tau\ \texttt{cmd}[\psi_s \to \psi \to \psi_e]$, indicating a command returning a value of type τ whose starting priority is bounded by ψ_s, whose ending priority is bounded by ψ_e and which, during execution, may take on any priorities allowed by ψ. We refer to these refinements as the *start*, *end*, and *comprehensive* refinements, respectively. For clarity, in addition to their position in the type, we will use colors to distinguish the refinements. Note that the colors are

simply for added readability: the same information will always also be conveyed through subscripts (generally s for start, e for end, and none for comprehensive) and/or superscripts.

The type of threads, τ `thread`$[\psi]$, includes a refinement indicating at what priorities the thread may run. A thread of type τ `thread`$[x :: \mathtt{L} \preceq x \wedge x \preceq \mathtt{H}]$ may be running at any priority between `L` and `H`, inclusive. Note that this may mean that the thread is running at one priority, known at runtime but not known precisely at compile time, or that the thread takes on many priorities (using `change`) during its execution but its priority always remains bounded by the constraint. For the purpose of preventing priority inversions, we need not distinguish between these scenarios.

Priority constants, for which we use the metavariable ρ, are first-class values in λ^4_ψ. The types of these values are of the form `prio`$[\psi]$, where the refinement ψ bounds the priority level of the priority. Finally, we include a construct `if` $v_1 \preceq v_2$ `then` e_1 `else` e_2 which compares the runtime levels of two priorities and branches on whether one is less than the other. This can be used, for example, to perform a `sync` at runtime only if doing so would not cause a priority inversion.

2.1 Static Semantics

The λ^4_ψ calculus is equipped with a type system that ensures the absence of priority inversions, that is, that threads only perform `sync` operations on threads of higher (or equal) priority. This task is complicated by the addition of first-class priorities and the ability to change priorities at runtime: priorities of threads are no longer static but rather must be approximated using the refinements.

Refinements and Constraints. As shown in the syntax, qualifiers take the form $v = v$ or $v \preceq v$, where v is a value of priority type—by typing of expressions, these are guaranteed to be either program variables or priority constants ρ. We use the judgment $\Gamma \vdash^R C$ to mean that qualifier C is satisfied under the assumptions in Γ (note that we use Γ to store both typing assumptions, as is standard, and asssumptions about priority orderings). We also introduce two constraints on refinements. First, we write $\Gamma \vdash^R \psi \implies \psi'$ to mean that under the constraints contained in Γ, refinement ψ *implies* ψ', in other words that the priorities consistent with ψ are a subset of those in ψ'. Second, we extend the qualifier $v \preceq v$ to allow ψ and ψ to appear in either position. For example, the judgment $\Gamma \vdash^R \psi_1 \preceq \psi_2$ indicates that, under Γ, all priorities allowed by ψ_1 are lower than or equal to all priorities in ψ_2. Because the set of priorities is finite and qualifiers include only variables and constants, not general expressions, these constraint judgments are decidable. We assume that the judgments above obey two fairly intuitive properties: cut elimination (that is, for a judgment J, if $\Gamma, C \vdash J$ and $\Gamma \vdash^R C$, then $\Gamma \vdash J$) and substitution (that is, if $\Gamma_1, x : \tau, \Gamma_2 \vdash J$ and $\Gamma_1, [v/x]\Gamma_2 \vdash^R_\Sigma v : \tau$, then $\Gamma_1, [v/x]\Gamma_2 \vdash [v/x]J$, where substitution for variables in contexts is defined in the natural way).

Typing of Expressions. Figure 2 presents selected rules for the expression typing judgment, $\Gamma \vdash^R_\Sigma e : \tau$. We omit the introduction and elimination rules for sums and products, which can be found in the supplementary appendix. Because types in the variable context Γ can refer to free variables, Γ is ordered and types can refer only to variables bound earlier. In addition, the judgment takes a signature Σ as a parameter, as well as R, the partial order on priorities. Signatures map thread names a and variants to their return types and priority refinements: the signature entry $a : \tau @ \psi$ indicates that thread a returns a value of type τ and has a priority bounded by ψ. Several rules have premises that require facts about constraints or refinements, discussed earlier in this subsection.

We discuss the rules that are new or substantially changed with respect to λ^4. Rule Prio-Var assigns variables of priority type the most specific possible type, `prio`$[y :: y = x]$, that is, a refinement stating that x holds a priority equal to the priority represented by variable x. This allows for specific refinements for uses of x, e.g., the function `fun` $f\ x = x$ can be assigned the

$$\textsc{Subsumption}\ \frac{\Gamma \vdash^R_\Sigma e : \tau \qquad \Gamma \vdash^R \tau <: \tau' \qquad \Gamma \vdash^R \tau' \ \mathsf{wf}}{\Gamma \vdash^R_\Sigma e : \tau'} \qquad \textsc{Var}\ \frac{\Gamma(x) = \tau}{\Gamma \vdash^R_\Sigma x : \tau} \qquad \textsc{Prio-Var}\ \frac{\Gamma(x) = \mathtt{prio}[\psi]}{\Gamma \vdash^R_\Sigma x : \mathtt{prio}[y :: y = x]}$$

$$\textsc{UnitI}\ \frac{}{\Gamma \vdash^R_\Sigma \langle\rangle : \mathtt{unit}} \qquad \textsc{Prio}\ \frac{}{\Gamma \vdash^R_\Sigma \rho : \mathtt{prio}[\{\rho\}]} \qquad \textsc{Tid}\ \frac{}{\Gamma \vdash^R_{\Sigma, a:\tau@\psi} \mathtt{tid}[a] : \tau\ \mathtt{thread}[\psi]}$$

$$\textsc{Let}\ \frac{\Gamma \vdash^R_\Sigma e_1 : \tau_1 \qquad \Gamma, x : \tau_1 \vdash^R_\Sigma e_2 : \tau_2 \qquad \Gamma \vdash^R \tau_2\ \mathsf{wf}}{\Gamma \vdash^R_\Sigma \mathtt{let}\ x = e_1\ \mathtt{in}\ e_2 : \tau_2} \qquad \textsc{If-Refine}\ \frac{\Gamma \vdash^R_\Sigma v_1 : \mathtt{prio}[\psi_1] \qquad \Gamma \vdash^R_\Sigma v_2 : \mathtt{prio}[\psi_2] \qquad \Gamma, v_1 \preceq v_2 \vdash^R_\Sigma e_1 : \tau \qquad \Gamma, v_1 \not\preceq v_2 \vdash^R_\Sigma e_2 : \tau}{\Gamma \vdash^R_\Sigma \mathtt{if}\ v_1 \preceq v_2\ \mathtt{then}\ e_1\ \mathtt{else}\ e_2 : \tau}$$

$$\to\text{I}\ \frac{\Gamma, f : (x\ :\ \tau_1 \to \tau_2), x : \tau_1 \vdash^R_\Sigma e : \tau_2 \qquad \Gamma \vdash^R \tau_1\ \mathsf{wf} \qquad \Gamma, x : \tau_1 \vdash^R \tau_2\ \mathsf{wf}}{\Gamma \vdash^R_\Sigma \mathtt{fun}\ f\ x = e : (x\ :\ \tau_1 \to \tau_2)} \qquad \to\text{E}\ \frac{\Gamma \vdash^R_\Sigma v_1 : (x\ :\ \tau_1 \to \tau_2) \qquad \Gamma \vdash^R_\Sigma v_2 : \tau_1}{\Gamma \vdash^R_\Sigma v_1\ v_2 : [v_2/x]\tau_2}$$

$$\textsc{CmdI}\ \frac{\Gamma \vdash^R_\Sigma m : \tau\ @\ \psi_s \to \psi \to \psi_e}{\Gamma \vdash^R_\Sigma \mathtt{cmd}\ \{m\} : \tau\ \mathtt{cmd}[\psi_s \to \psi \to \psi_e]}$$

Fig. 2. Selected expression typing rules.

type $x\ :\ \mathtt{prio}[\psi] \to \mathtt{prio}[y :: y = x]$. In rule Prio, we use the shorthand $\{\rho\}$ for $x :: x = \rho$, a refinement allowing only one priority, that is, the priority of the priority constant ρ itself. Rule Let requires that the type τ_2 is well-formed under Γ, that is, the variable x does not escape the scope of e_2. The rules for the type well-formedness judgment $\Gamma \vdash^R \tau\ \mathsf{wf}$ as well as an auxiliary judgment for well-formedness of refinements, $\Gamma \vdash^R \psi\ \mathsf{wf}$, are defined in the supplementary appendix and largely require that variables included in types and refinements are bound. Rule If-Refine adds the constraint to the context when checking the "then" branch and its negation when checking the "else" branch. Rule $\to E$ substitutes the argument v_2 for the formal parameter x in the function type; for example, if v_1 has type $x\ :\ \mathtt{prio}[\psi] \to \mathtt{prio}[y :: y = x]$, then $v_1\ \rho$ would have type $\mathtt{prio}[\{\rho\}]$.

Typing of Commands. The rules for typing commands are shown in Figure 3. The judgment $\Gamma \vdash^R_\Sigma m\ :\ \tau\ @\ \psi_s \to \psi \to \psi_e$ indicates the command's return type τ and three refinements on its priorities. The start refinement ψ_s contains priorities at which the command is *allowed* to start. The end refinement ψ_e contains priorities at which the command *might* end. The comprehensive refinement ψ contains priorities at which the command *might* run at any point during its execution.[1]

The Bind rule for a command $x \leftarrow e; m$ ensures that the priorities of the commands compose appropriately. Because the bind command starts by executing the command represented by e, the start refinement of the bind and the start priority of e are the same. Similarly, the end refinement of the bind is taken from the end refinement of m. Running m after e requires that the end refinement of e is the same as the start refinement of m. One might worry that this is too restrictive: m should be allowed to follow e as long as the set of possible end priorities of e is a subset of the set of allowed

[1]This includes priorities at which at least some computation occurs; if a thread is spawned at a priority ρ but then immediately changes priority, ρ need not be included in ψ.

$$\textsc{Bind}\quad \frac{\Gamma \vdash^R_\Sigma e : \tau\ \mathsf{cmd}[\psi_s \to \psi \to \psi_e] \qquad \Gamma, x : \tau \vdash^R_\Sigma m : \tau' \mathbin{@} \psi_e \to \psi \to \psi'_e \qquad \Gamma \vdash^R \tau'\ \mathsf{wf} \qquad \Gamma \vdash^R \psi'_e\ \mathsf{wf}}{\Gamma \vdash^R_\Sigma x \leftarrow e; m : \tau' \mathbin{@} \psi_s \to \psi \to \psi'_e}$$

$$\textsc{Spawn}\quad \frac{\Gamma \vdash^R_\Sigma e : \mathsf{prio}[\psi'_s] \qquad \Gamma \vdash^R_\Sigma m : \tau \mathbin{@} \psi'_s \to \psi' \to \psi'_e \qquad \Gamma \vdash^R \psi'_s \Rightarrow \psi'' \qquad \Gamma \vdash^R \psi' \Rightarrow \psi'' \qquad \Gamma \vdash^R \psi''\ \mathsf{wf} \qquad \Gamma \vdash^R \psi\ \mathsf{wf}}{\Gamma \vdash^R_\Sigma \mathsf{spawn}[\psi''; \tau](e)\ \{m\} : \tau\ \mathsf{thread}[\psi''] \mathbin{@} \psi \to \psi \to \psi}$$

$$\textsc{Change}\quad \frac{\Gamma \vdash^R_\Sigma e : \mathsf{prio}[\psi'_e] \qquad \Gamma \vdash^R \psi'_s\ \mathsf{wf}}{\Gamma \vdash^R_\Sigma \mathsf{change}\ e : \mathsf{unit} \mathbin{@} \psi'_s \to \psi'_e \to \psi'_e} \qquad \textsc{Sync}\quad \frac{\Gamma \vdash^R_\Sigma e : \tau\ \mathsf{thread}[\psi'] \qquad \Gamma \vdash^R \psi \preceq \psi' \qquad \Gamma \vdash^R \psi\ \mathsf{wf}}{\Gamma \vdash^R_\Sigma \mathsf{sync}\ e : \tau \mathbin{@} \psi \to \psi \to \psi}$$

$$\textsc{Ret}\quad \frac{\Gamma \vdash^R_\Sigma e : \tau \qquad \Gamma \vdash^R \psi\ \mathsf{wf}}{\Gamma \vdash^R_\Sigma \mathsf{ret}\ e : \tau \mathbin{@} \psi \to \psi \to \psi} \qquad \textsc{Sub-Cmd}\quad \frac{\Gamma \vdash^R_\Sigma m : \tau \mathbin{@} \psi_s \to \psi \to \psi_e \qquad \Gamma \vdash^R \tau <: \tau' \qquad \Gamma \vdash^R \psi'_s \Rightarrow \psi_s \qquad \Gamma \vdash^R \psi \Rightarrow \psi' \qquad \Gamma \vdash^R \psi_e \Rightarrow \psi'_e \qquad \Gamma \vdash^R \psi'_e \Rightarrow \psi' \qquad \Gamma \vdash^R \tau'\ \mathsf{wf} \qquad \Gamma \vdash^R \psi'_s\ \mathsf{wf} \qquad \Gamma \vdash^R \psi'\ \mathsf{wf} \qquad \Gamma \vdash^R \psi'_e\ \mathsf{wf}}{\Gamma \vdash^R_\Sigma m : \tau' \mathbin{@} \psi'_s \to \psi' \to \psi'_e}$$

Fig. 3. Command typing rules.

$$\textsc{Sub-Refl}\quad \frac{}{\Gamma \vdash^R \tau <: \tau} \qquad \textsc{Sub-Function}\quad \frac{\Gamma \vdash^R \tau'_1 <: \tau_1 \qquad \Gamma \vdash^R \tau_2 <: \tau'_2}{\Gamma \vdash^R (x : \tau_1 \to \tau_2) <: (x : \tau'_1 \to \tau'_2)} \qquad \textsc{Sub-Prio}\quad \frac{\Gamma \vdash^R \psi \Rightarrow \psi'}{\Gamma \vdash^R \mathsf{prio}[\psi] <: \mathsf{prio}[\psi']}$$

$$\textsc{Sub-Thread}\quad \frac{\Gamma \vdash^R \tau <: \tau' \qquad \Gamma \vdash^R \psi \Rightarrow \psi'}{\Gamma \vdash^R \tau\ \mathsf{thread}[\psi] <: \tau'\ \mathsf{thread}[\psi']} \qquad \textsc{Sub-Cmd-Expr}\quad \frac{\Gamma \vdash^R \tau <: \tau' \qquad \Gamma \vdash^R \psi'_s \Rightarrow \psi_s \qquad \Gamma \vdash^R \psi \Rightarrow \psi' \qquad \Gamma \vdash^R \psi_e \Rightarrow \psi'_e \qquad \Gamma \vdash^R \psi'_e \Rightarrow \psi'}{\Gamma \vdash^R \tau\ \mathsf{cmd}[\psi_s \to \psi \to \psi_e] <: \tau'\ \mathsf{cmd}[\psi'_s \to \psi' \to \psi'_e]}$$

Fig. 4. Subtyping rules.

start priorities of m. This less restrictive version will turn out to be admissible after we introduce subtyping on priority refinements, discussed later. The comprehensive priority refinement ψ of the overall command must cover all of the priorities of both commands; this is enforced in the rule by requiring ψ as the refinement for both commands, but again, subtyping allows for the actual priorities of both commands to be subsets of ψ. The return type τ' and refinements must not mention x, as ensured by the well-formedness premises.

For a spawn command $\mathsf{spawn}[\psi''; \tau](e)\ \{m\}$, the command starts running at priority e, so e should have the type $\mathsf{prio}[\psi'_s]$ for some refinement ψ'_s, which must also be the start refinement of the command. The return type is $\tau\ \mathsf{thread}[\psi'']$, that of a handle to a thread whose possible priorities cover the comprehensive refinement ψ' of m as well as its starting priority. This is ensured by requiring that both ψ'_s and ψ' are sub-refinements of ψ''. Note that the spawn command itself

may run at any priority, regardless of the priority of the new thread, and that spawning a thread does not change the priority of the spawning thread. Hence, the start, end, and comprehensive requirements of the spawn command are all identical but otherwise unconstrained.

The change e command changes the priority of the current thread to e, which will fall in ψ'_e. This command is *allowed* to start at any priority. The only priority that change e *might* be throughout its course and once it ends is the priority value that e evaluates to. In rule Sync, the argument e must be a handle to a thread with priority refinement ψ'. As in Spawn, the start, end, and comprehensive refinements of the current thread are all ψ. However, in order to prevent priority inversions, we must constrain the priority of the current thread to be lower at runtime than the priority of thread e. This is enforced by requiring priorities of ψ to be lower than or equal to priorities of ψ'.

Subtyping. We use subtyping to allow for weakening of refinements in types. Subtyping is introduced through the Subsumption rule of Figure 2 and rule Sub-Cmd of Figure 3. Figure 4 gives the rules for the subtyping judgment $\Gamma \vdash^R \tau <: \tau'$. Subtyping is covariant with respect to the priority refinements on priorities and threads, as well as the end and comprehensive refinements of commands (the intuition is that it is safe to assume that a thread or command might take on a wider set of priorities than it does). It is contravariant in the start refinement of commands: a command may start at a *narrower* set of priorities than it is allowed to. The Sub-Cmd-Expr rule also requires that the "new" comprehensive refinement of a command include the "new" end refinement. Reflexivity of subtyping is enforced by the Sub-Refl rule. Transitivity of subtyping is admissible, as we show later. Because subtyping depends only on the refinement implication judgment, which we have argued to be decidable for finite sets of priorities, subtyping is decidable.

Example: Increment. We return briefly to the example from the introduction of the increment function `incr`. From a typing perspective, it suffices to know that the returned priority is higher than or equal to the argument. This can be expressed by the type $p : \texttt{prio}[x :: \top] \to \texttt{prio}[p' :: p \preceq p']$, where $x :: \top$ is a dummy refinement indicating that the argument priority is unconstrained. For a finite set of priority constraints as in our setting, such a function can be implemented as a series of conditionals. For example, with three priorities, `Low`, `Med`, and `High`:

$$\texttt{fun incr } p = \texttt{if } p = \texttt{Low then Med else High}$$

We can show, by applying subsumption and Sub-Prio in both branches and discharging the resulting constraints on priority constants, that this implementation has the required type.

Metatheoretic Results about Yyping. We prove a number of results about typing; these ensure the consistency of the definitions, and some of these lemmas are also needed for the results of Section 3. Due to space restrictions, we leave the formal statements and proofs of these lemmas to the supplementary appendix, and summarize them here. The key definitions and results are:

- A lemma stating that the end refinement of a well-typed command implies its comprehensive refinement, as we have informally motivated.
- Well-formedness judgments over contexts (Γ wf) and signatures ($\Gamma \vdash^R \Sigma$ wf). Informally, a context is well-formed if all of its types are well-formed under the previous bindings (recall that contexts are ordered) and a signature is well-formed if all of its entries are well-typed under the provided context Γ.
- A lemma showing that, under well-formed contexts and signatures, the expression and command typing judgments produce well-formed types.
- Two inversion lemmas allowing us to invert on the subtyping rules and the (expression and command) typing rules, respectively.
- A lemma proving transitivity of the subtyping relation.

- A set of substitution results showing that well-formedness of refinements and types, as well as the typing judgments, are preserved under substitutions of values for free variables.

3 Cost Semantics

In Section 2, we presented the λ^4_ψ calculus and a type system that guarantees lack of priority inversions. Priority inversions are commonly considered a bug in concurrent programs, and so preventing them seems like a worthy goal. However, our end goal in this work is actually guaranteeing efficient and responsive execution of programs. To this end, in Section 3.1, we present a cost model for parallel programs with dynamic priorities, building on a similar model for λ^4 [24]. We show that, for some assumptions on the program and the scheduler, we can provide formal guarantees on the response time of threads. One of these assumptions is closely tied to priority inversions; this gives us our formal definition of a priority inversion which we use (in Section 3.2) to prove that well-typed λ^4_ψ programs are free of priority inversions and can therefore be scheduled efficiently.

3.1 A Cost Model for Prioritized Parallel Code

In this subsection, we model λ^4_ψ execution as a Directed Acyclic Graph (DAG) and show how the response time of threads can be formally defined and theoretically bounded by properties of the graph. The definitions and results in this subsection build on many similar models from prior work, most directly on that of λ^4 [24]. In order to keep this section self-contained, we present all of the definitions and results and highlight the novel features of our model. Vertices of the DAG correspond to sequential portions of computation. Without loss of generality, we assume that each vertex corresponds to a uniform unit of computation, e.g., a single CPU instruction. Edges in the DAG correspond to dependencies: if there is a path from vertex u to u' (we say that u is an *ancestor* of u' and write $u \sqsupseteq u'$) then the computation corresponding to u must occur before u'; on the other hand, if there is no path between u and u', the two computations may happen in parallel.

A vertex u is paired with its priority ρ, written (u, ρ). This differs from the model of λ^4, in which priorities are associated with threads: this difference propagates through the remaining results in subtle but pervasive ways. We use the notation $\vec{u}$ to refer to a sequence of vertex-priority pairs, e.g., $\vec{u} = (u_1, \rho_1) \cdot (u_2, \rho_2) \cdots (u_i, \rho_i)$. This represents a sequence of unit-time operations that are connected by a series of edges, which we will refer to as *continuation edges*, $(u_1, u_2), (u_2, u_3), \ldots, (u_{n-1}, u_n)$ representing sequential dependencies. We write [] for the empty sequence.

We represent a DAG g as a tuple $(\mathcal{T}, E_s, E_j)$. Here, $\mathcal{T}$ is a mapping from thread names to sequences of vertices (as defined above), written $a \hookrightarrow \vec{u}$. We define $Prio_g(u)$ to be the priority of vertex u and $Prios_g(u)$ to be the set of all priorities in the thread to which u belongs. That is, if $a \hookrightarrow (u_1, \rho_1) \cdots (u_n, \rho_n) \in \mathcal{T}$ and $u = u_i$ for some i, then $Prios_{(\mathcal{T}, E_s, E_j)}(u) = \{\rho_1, \ldots, \rho_n\}$. In addition to the set of threads, a DAG contains a set of *spawn edges* E_s and a set of *join edges* E_j. Spawn edges are of the form (u, a) where u is a vertex that spawned a thread a. Join edges are of the form (a, u), where u is a vertex that syncs on (joins with) thread a. In both spawn and join edges, we include the thread name rather than the vertex, but concretely, the target of a spawn edge (u, a) is the first vertex of a and the source of a join edge (a, u) is the last vertex of a.

The Prompt Scheduling Principle. A *schedule* is an assignment of vertices to processors at each time step in a computation, subject to two constraints: at any time, no more than P vertices may be scheduled, where P is the number of processors; and a vertex may not be scheduled before all of its parents have executed. If a vertex's parents have already executed, we say the vertex is *ready*.

We do not attempt to find the optimal (shortest) valid schedule, which is NP-hard, even when scheduling decisions are made offline [36]. Instead, we prove an upper bound on response time for a wide class of schedules, known as *prompt* schedules [23, 24]. A prompt schedule is one that 1)

executes a vertex (u, ρ) only if no additional vertices of priority greater than ρ are ready at that time step and 2) only leaves a processor idle if no additional vertices of any priority are ready.

Toward Bounding Response Time. We now wish to bound the response time of a thread or, indeed, any sequence of vertices, defined following prior work:

Definition 1. We denote the response time of a sequence $\vec{u} = s \cdots t$ as $T(\vec{u})$, which we define to be the number of time steps from when s is ready (exclusive) to when t is executed (inclusive).

Intuitively, $T(\vec{u})$ should depend only on the amount of computation that 1) is at priority not less than that of $\vec{u}$ and 2) may run at the same time as $\vec{u}$. This generalizes the notion of *competitor work* [24]: in prior work, because entire threads were at the same priority, competitor work was defined in a much more coarse-grained way. First, we define a notation for the portion of a graph g that may occur in parallel with a sequence of vertices $\vec{u} = s \cdots t$.

$$\nleftrightarrow \vec{u} \triangleq g \setminus \{u \mid u \sqsupseteq s \wedge u \neq s\} \setminus \{u \mid t \sqsupseteq u \wedge u \neq t\}$$

The *competitor work* of $\vec{u} = s \cdots t$ is the number of vertices that either happen in parallel with some vertex of $\vec{u}$ at priority not less than the priority of that vertex, or are ancestors of t. This is the work that is either in parallel competition with $\vec{u}$ or must be completed before $\vec{u}$ is finished:

$$CW(\vec{u}) \triangleq \left|\{(v, \rho) \mid \exists u\ (u \in \vec{u} \text{ and } v \in \nleftrightarrow u \text{ and } \rho \not\prec Prio_g(u))\} \cup \{(v, \rho) \mid v \in \nleftrightarrow \vec{u} \text{ and } v \sqsupseteq t\}\right|$$

The response time of a sequence also depends upon the critical path of vertices necessary to finish that sequence, referred to in the parallel algorithms literature as *span* or *depth*. Let $S_u(g)$ be the longest path in a graph g ending at vertex u. The *$\vec{u}$-span* of a sequence $\vec{u} = s \cdots t$, is defined as $S_t(\nleftrightarrow \vec{u})$, that is, the longest path ending at t, that doesn't include ancestors of s (intuitively, if a vertex u is an ancestor of s, u cannot delay the response time of the sequence, because u must have already been executed before s became ready).

Well-formed DAGs. Priority inversions emerge in the cost model once we attempt to bound the response time of a portion of a thread, $\vec{u}$, by its competitor work and $\vec{u}$-span. Suppose there is a long chain of low-priority vertices on the $\vec{u}$-span of a higher-priority sequence of vertices. Suppose l is one such low-priority vertex and there exist a large number of other ready high-priority vertices $h_1, \ldots, h_n$ in the program (even if they are unrelated to $\vec{u}$). Because l is on the critical path of $\vec{u}$, finishing $\vec{u}$ quickly will eventually require choosing to execute l rather than the h_i, but a prompt scheduler, by definition, will not make such a choice. As in prior work [23, 24], we restrict our attention to dags that are *well-formed*, i.e., do not have lower-priority vertices on the critical paths of higher-priority vertices. As we show later, a program that is free of priority inversions according to the type system of λ^4_ψ corresponds to a well-formed DAG. The formal definition of well-formedness is below, and accounts for different vertices in a thread having different priorities.

Definition 2. A DAG $g = (\mathcal{T}, E_s, E_j)$ is well-formed if for all threads $a \hookrightarrow (u_1, \rho_1) \cdots (u_n, \rho_n) \in \mathcal{T}$ and all $1 < i \leq n$, if for some (u, ρ) we have $u \sqsupseteq u_i$ and $u \not\sqsupseteq u_{i-1}$, then $\rho_i \preceq \rho$.

Response Time Bound. We can now bound the response time for portions of a thread in a well-formed DAG, under a prompt schedule. Note that Theorem 1 can also be used to bound the overall computation time of the program by letting $\vec{u}$ be the entire main thread of the program. The theorem statement (modulo the definition of competitor work) is similar to that of Muller et al. [24], but allows us to bound the response time of a portion of a thread, rather than just a whole threads, and the change to the definition of competitor work requires a substantially different proof.

Theorem 1. *Let g be a well-formed DAG where $\vec{u} = s \cdots t$ is a portion of some thread $a \in g$.*

$$\textsc{C-Val}\ \frac{}{v \downarrow_\rho v; []} \qquad \textsc{C-Let}\ \frac{u\ \texttt{fresh} \quad e_1 \downarrow_\rho v_1; \vec{u}_1 \quad [v_1/x]e_2 \downarrow_\rho v; \vec{u}_2}{\texttt{let}\ x = e_1\ \texttt{in}\ e_2 \downarrow_\rho v; \vec{u}_1 \cdot (u, \rho) \cdot \vec{u}_2} \qquad \textsc{C-App}\ \frac{u\ \texttt{fresh} \quad [v/x][\texttt{fun}\ f\ x = e/f]e \downarrow_\rho v'; \vec{u}}{(\texttt{fun}\ f\ x = e)\ v \downarrow_\rho v'; (u, \rho) \cdot \vec{u}}$$

$$\textsc{C-If-Refine-T}\ \frac{\vdash^R \rho_1 \preceq \rho_2 \quad e_1 \downarrow_\rho v; \vec{u} \quad u\ \texttt{fresh}}{\texttt{if}\ \rho_1 \preceq \rho_2\ \texttt{then}\ e_1\ \texttt{else}\ e_2 \downarrow_\rho v; (u, \rho) \cdot \vec{u}} \qquad \textsc{C-If-Refine-F}\ \frac{\vdash^R \rho_2 \prec \rho_1 \quad e_2 \downarrow_\rho v; \vec{u} \quad u\ \texttt{fresh}}{\texttt{if}\ \rho_1 \preceq \rho_2\ \texttt{then}\ e_1\ \texttt{else}\ e_2 \downarrow_\rho v; (u, \rho) \cdot \vec{u}}$$

Fig. 5. Cost semantics for expressions (selected rules).

For any prompt schedule of g on P processors,

$$T(\vec{u}) \leq \frac{CW(\vec{u})}{P} + S_t(\nmid\vec{u})$$

Proof. Vertices s and t are the first and last vertices of $\vec{u}$, respectively. Consider the portion of the schedule from the step in which s is ready (exclusive) to the step in which t is executed (inclusive). Let u be the next vertex of a yet to be executed at the current step. For each processor at each step working on vertex v, place a token in one of two buckets. Put a token in the "high" bucket if the processor is working on v such that for some $u' \in u \cdots t$, $Prio_g(v) \not\prec Prio_g(u')$, and a token in the "low" bucket otherwise. Because P tokens are placed per step, we have $T(\vec{u}) = \frac{1}{P}(B_h + B_l)$, where B_h and B_l are the number of tokens in the high and low buckets, respectively, after t is executed. We wish to bound B_h by the competitor work of $\vec{u}$. Each token in B_h corresponds to some processor working on a vertex $v \in \nmid\vec{u}$ since it is placed during a step after s is ready and before any descendant of t is executed. If v is an ancestor of t, it is trivially competitor work of $\vec{u}$, so suppose $v \not\sqsupseteq t$. In this case, since none of $u \cdots t$ has been executed yet, $v \in \nmid u'$ for all $u' \in u \cdots t$, and $Prio_g(v) \not\prec Prio_g(u')$ by construction. Therefore, v is competitor work of $\vec{u}$. So, we have $B_h \leq CW(\vec{u})$ and $T(\vec{u}) \leq \frac{CW(\vec{u})}{P} + \frac{B_l}{P}$.

We now need only bound B_l by $P \cdot S_t(\nmid a)$. Let step 0 be the step after s is ready, and let $Exec(j)$ be the set of vertices executed before the start of step j. Consider a step j where a token is placed in the low bucket. Let $v \sqsupseteq t$ be a ready vertex. Let u' be the first vertex of $s \cdots t$ where $v \sqsupseteq u'$. Note that u' has not been executed at the start of step j since v is ready, and also that $u' \neq s$ (because s must be ready or executed). We have $Prio_g(u') \preceq Prio_g(v)$ by well-formedness. Because a token was placed in the low bucket on step j, either a processor executes a vertex of priority less than all vertices not yet executed on $\vec{u}$ on this step, or a processor is idle on this step. In either case, promptness requires that all vertices of priority $Prio_g(v)$ are executed on this step, and so v is executed. Thus, the length of the path to t starting at v decreases by 1 and so $S_t(g \setminus Exec(j+1)) = S_t(g \setminus Exec(j)) - 1$. The maximum number of such steps is thus $S_t(g \setminus Exec(0))$, and so $B_l \leq P \cdot S_t(g \setminus Exec(0))$. Any path ending at t in $g \setminus Exec(0)$ does not contain vertices that are ancestors of s (because these would be in $Exec(0)$) and, by construction, does not contain descendants of t, so the path is also contained in $\nmid\vec{u}$. Thus, $S_t(g \setminus Exec(0)) \leq S_t(\nmid\vec{u})$ and $B_l \leq S_t(\nmid\vec{u})$. □

3.2 Cost Semantics for λ^4_ψ

A key requirement of soundness for the λ^4_ψ type system is that an execution of a well-typed program corresponds to a well-formed DAG by the definition in Section 3.1, which can therefore be scheduled efficiently by a prompt scheduler (Theorem 1). In order to prove this, we must establish a way of

C-Bind

$$\frac{e \downarrow_\rho \texttt{cmd}\ \{m_1\}; \vec{u}_1 \quad \sigma; \Sigma; m_1; \rho \downarrow_a \sigma_1; \Sigma_1; v; \rho'; g_1 \quad \sigma_1; \Sigma_1; [v/x]m_2; \rho' \downarrow_a \sigma_2; \Sigma_2; v'; \rho''; g_2 \quad u\ \texttt{fresh}}{\sigma; \Sigma; x \leftarrow e; m_2; \rho \downarrow_a \sigma_2; \Sigma_2; v'; \rho''; [\vec{u}_1] \oplus_a g_1 \oplus_a [(u, \rho')] \oplus_a g_2}$$

C-Spawn

$$\frac{b\ \texttt{fresh} \quad e \downarrow_\rho \rho'; \vec{u} \quad \sigma; \Sigma; m; \rho' \downarrow_b \sigma, \sigma'; \Sigma, \Sigma'; v; \rho''; (\mathcal{T}, E_s, E_j) \quad u\ \texttt{fresh}}{\begin{array}{c}\sigma; \Sigma; \texttt{spawn}[\psi''; \tau'](e)\ \{m\}; \rho \\ \downarrow_a \sigma, b \hookrightarrow (v, \sigma', \Sigma'); \Sigma, b : \tau' @ \psi''; \texttt{tid}[b]; \rho; (a \hookrightarrow \vec{u} \cdot (u, \rho) \uplus \mathcal{T}, E_s \cup \{(u, b)\}, E_j)\end{array}}$$

C-Sync

$$\frac{e \downarrow_\rho \texttt{tid}[b]; \vec{u} \quad u\ \texttt{fresh}}{\begin{array}{c}\sigma, b \hookrightarrow (v, \sigma', \Sigma'); \Sigma, b : \tau' @ \psi'; \texttt{sync}\ e; \rho \\ \downarrow_a \sigma, b \hookrightarrow (v, \sigma', \Sigma'), \sigma'; \Sigma, b : \tau' @ \psi', \Sigma'; v; \rho; (a \hookrightarrow \vec{u} \cdot (u, \rho), \emptyset, \{(b, u)\})\end{array}}$$

C-Change

$$\frac{e \downarrow_\rho \rho'; \vec{u} \quad u\ \texttt{fresh}}{\sigma; \Sigma; \texttt{change}\ e; \rho \downarrow_a \sigma; \Sigma; \langle\rangle; \rho'; (a \hookrightarrow \vec{u} \cdot (u, \rho'), \emptyset, \emptyset)}$$

C-Ret

$$\frac{e \downarrow_\rho v; \vec{u}}{\sigma; \Sigma; \texttt{ret}\ e; \rho \downarrow_a \sigma; \Sigma; v; \rho; (a \hookrightarrow \vec{u}, \emptyset, \emptyset)}$$

Fig. 6. Cost semantics for commands.

executing a program and tracking the resulting DAG. This is the purpose of the *cost semantics* for λ^4_ψ we provide in this subsection. The cost semantics is a big-step semantics that, in addition to a value, produces a graph of the form given in Section 3.1.

We use separate judgments for expressions and commands. The expression evaluation judgment $e \downarrow_\rho v; \vec{u}$ states that, if run at priority ρ, the expression e evaluates to v and its execution is represented by the vertex sequence $\vec{u}$. Note that the graph resulting from an expression will always be a single sequence of vertices: expressions, by design, cannot spawn or synchronize threads, which would result in other sorts of dependencies. In fact, the vertices of the resulting sequence will all be at the same priority, but the judgment does not enforce this property. The rules for this judgment are largely straightforward other than propagating the priority ρ to all of the vertices in the resulting sequence. Selected rules are given in Figure 5; other rules are deferred to the supplementary appendix.

The cost semantics judgment for commands is $\sigma; \Sigma; m; \rho \downarrow_a \sigma'; \Sigma'; m; \rho'; g$, and its rules appear in Figure 6. The judgment is parameterized by the thread name a, which is used to build the graph. The judgment also includes signatures Σ which, as in the typing rules, give the return types and priorities of threads, and the priority ρ at which the command starts executing. The judgment also returns a new signature Σ', which may include newly spawned threads, and the ending priority ρ'.

Finally, the judgment includes *thread records* σ and σ' [24]. Thread records are used to hold the results of running threads and pass these through the cost evaluation; on a sync, the thread record of the target thread is used to obtain these results. The thread record entry $a \hookrightarrow (v, \sigma, \Sigma)$ maps a thread a to its return value v and the thread record and signature for threads spawned by a. The thread record is used by the rule C-Sync to capture the value of the target thread b, which must be returned by the sync operation. When thread a syncs on thread b, it also "learns about" threads spawned by b by adding the thread record and signature from b's thread record entry. This reflects the fact that, if b spawned c, there is no way for a to have a handle to c unless b returns it; this

property is useful for proving well-formedness and so we enforce it structurally in the rules.[2] The rule also captures the signature of the new threads that b "knows about", which it adds to the signature, indicating that future operations in thread a now "know about" these threads. Since a "knew about" b to sync on it, a already "knew about" the threads that b "knew about" before it began, so these are already included in the signature.

Rule C-Bind evaluates e to an encapsulated command $\texttt{cmd}\ \{m_1\}$ using the expression rules, then evaluates m_1 and finally substitutes its result into m_2 before evaluating it. The resulting graphs are composed together in order using the sequential composition operator $g_1 \oplus_a g_2$, which composes two graphs together by appending the thread a of g_2 to the end of thread a of g_1. Note that g_1 and g_2 may contain spawns and syncs of other threads, which are also added into the composed graph. This operation is defined formally as follows.

$$(a \hookrightarrow \vec{u} \uplus \mathcal{T}, E_s, E_j) \oplus_a (a \hookrightarrow \vec{u}' \uplus \mathcal{T}', E_s', E_j') \triangleq (a \hookrightarrow \vec{u} \cdot \vec{u}' \uplus \mathcal{T} \uplus \mathcal{T}', E_s \cup E_s', E_j \cup E_j')$$

We use the notation $[\vec{u}]$ to indicate a graph consisting only of a single thread with vertex sequence $\vec{u}$. The name of the thread will generally be evident from context, e.g. because $[\vec{u}]$ is immediately sequentially composed with another graph at thread a, so

$$[\vec{u}] \oplus_a (a \hookrightarrow \vec{u}' \uplus \mathcal{T}, E_s, E_j) \triangleq (a \hookrightarrow \vec{u} \cdot \vec{u}' \uplus \mathcal{T}, E_s, E_j)$$

Rule C-Spawn evaluates e to a priority ρ' and spawns a new thread b at priority ρ' to run command m. The rule evaluates m eagerly in its premise (using b as the thread name) and adds the resulting graph to the computation graph, and the result to the thread record. The signature Σ' that goes in the thread record entry for b contains the threads added to the signature by evaluation of b.

Rule C-Change evaluates the expression to a priority and returns that as the new priority of the thread. In addition, it adds a new vertex to the thread in the graph at the new priority. Finally, rule C-Ret evaluates and returns its argument, without adding any additional vertices to the graph (we do not consider $\texttt{ret}\ v$ where v is a value to perform an additional computation).

Lemma 3.1 shows the standard type preservation result for expression evaluation. The equivalent property for commands is part of the main lemma of this section, which guarantees well-formed DAGs. The proof requires a cut elimination lemma that allows us to remove provable constraints from contexts. We defer this cut elimination lemma and its (straightforward) proof to the appendix.

Lemma 3.1. *If* $\cdot \vdash^R_\Sigma e : \tau$ *and* $e \downarrow_\rho v; \vec{u}$, *then* $\cdot \vdash^R_\Sigma v : \tau$ *and* $\vec{u} = (u_1, \rho) \cdots (u_n, \rho)$ *for* $n \geq 0$.

The proof is a straightforward induction on the typing derivation and we defer it to the appendix.

3.3 Proving Absence of Priority Inversions

Recall that our goal is to prove that well-typed programs yield, under the cost semantics of the previous subsection, well-formed cost graphs. For space reasons, we will only outline the proof here, and leave many of the details to the supplementary appendix.

As it happens, rather than directly show that graphs produced by the cost semantics are well-formed, it is more straightforward to show that these graphs meet a stronger property called *strong* well-formedness. This property was introduced in λ^4 [24]; we extend the definition to account for priorities on individual vertices. Here, we define strong well-formedness and state that it implies the definition of well-formedness from Section 3.1 (the proof appears in the appendix). Intuitively, a graph is strongly well-formed if 1) for all sync edges (b, u), the priority of vertex u is less than all

[2] If we extend the calculus with mutable state, this provides another way to pass thread handles between threads. It is still possible to ensure a suitable variation of well-formedness in this setting, as shown by [25]; because the mechanisms involved are largely orthogonal to our goals in this paper, we leave them out for simplicity.

of the priorities taken on by thread b, and 2) there is a path in the graph from the spawn of b to u that does not go through b; that is, u somehow "learned about" thread b.

Definition 3. A DAG $g = (\mathcal{T}, E_s, E_j)$ is *strongly well-formed* if for all $(b, u) \in E_j$, if $a \hookrightarrow \vec{u}_1 \cdot (u, \rho) \cdot \vec{u}_2,\ b \hookrightarrow \vec{u} \in \mathcal{T}$, we have that

(1) $\rho \preceq Prios_g(b)$
(2) If $(u', b) \in E_s$, then there exists a path from u' to u where the first edge is not a spawn edge.

Lemma 3.2. *If g is strongly well-formed, then g is well-formed.*

Proving that well-typed programs produce (strongly) well-formed cost graphs requires an induction hypothesis that is strengthened substantially with a number of invariants relating the graph, signatures and thread records. A few of the key invariants are:

- The graph and thread record are compatible: every thread b in the thread record exists in the graph and b's signature Σ in the thread record correctly captures the other threads b may sync on based on edges that exist in the graph (i.e., either b spawned the threads or learned about them by syncing on another thread that knew about them).
- The graph is *prioritized* by the thread record: the signatures in the thread record correctly describe (potentially conservatively overapproximating) the priorities actually taken on by the thread's nodes in the graph.
- The thread record is well-formed in that its values are well-typed.
- The graph is strongly well-formed.

These properties and some additional minor invariants are formally defined in the appendix.

Finally, we state and prove a lemma ensuring that if the invariants described above (as well as the additional ones in the appendix) hold and a command (which, for the purposes of induction, may represent a piece of an already-executing program that has already produced a partial cost graph) is well-typed and evaluates, adding additional nodes and edges to the cost graph, then the invariants continue to hold. The lemma is formally stated and proven in the supplementary appendix. Theorem 2, which states simply that if a well-typed command executes under the cost semantics, then the resulting graph is well-formed, follows directly from the lemma.

Theorem 2. *If $\cdot \vdash^R_\cdot m : \tau \ @\ \rho$ and $\cdot; \cdot; m; \rho \downarrow_a \sigma; \Sigma; v; \rho'; g$, then g is well-formed.*

4 Implementation

We have extended the implementation of the PriML language [24] to support dynamic, first-class priorities following the theory of λ^4_ψ. PriML supports fine-grained threads with priorities and has a type system that prevents priority inversions. Like the λ^4 calculus on which it is based (and which λ^4_ψ extends), PriML did not previously support first-class priorities or dynamic changing of priorities. Like λ^4_ψ, PriML syntax is divided into expressions, which support almost all of the constructs of Standard ML, and a monadic command layer for code that must run at a designated priority. In PriML, the set and order of priorities are not fixed ahead of time but is specified using dedicated declarations. For example, Figure 7 shows how to declare three priorities `A`, `B`, and `C`, where `A` is greater than both others but the others are incompatible.

```
priority A
priority B
priority C
order B < A
order C < A
```

Fig. 7. Priority and order declarations.

In addition to the extensions described in this paper, we have added support for mutex locks. The design of a priority-inversion-prevention type system with mutexes has been described in previous work [26] and is largely orthogonal to the extensions for first-class and dynamic priorities, so this

was not included in the formal type system in this work. We discuss the typing of mutexes at a high level in the supplementary appendix.

The largest modification to PriML is extending PriML's type inference to infer refined priority types. In addition, PriML previously compiled to a parallel extension of Standard ML from prior research [33]. In order to modernize and scale up the implementation, we retargeted the code generation pass to instead produce Multicore OCaml code. In the remainder of this section, we describe the type inference and code generation passes in more detail.

4.1 Type Inference

Type inference proceeds in three phases: (1) Hindley-Milner (HM) type inference, (2) Priority constraint generation, and (3) Constraint solving. This structure largely follows the algorithm for inferring liquid types [29] and, because there are many similarities, we focus on parts that are unique to our setting. In the first phase, expressions are assigned ordinary ML types using a standard unification-based type inference algorithm. Priority refinements ψ are simply initialized with a fresh *refinement variable* κ which stands for an as-yet-undetermined refinement—in phases 2 and 3, we generate and solve constraints over these refinement variables. In the type system of λ^4_ψ, subsumption is used only to widen priority refinements, and so it is not needed or used in this first phase.

In the second phase, the program is traversed again to generate a set of constraints on priority refinements induced by the typing rules. There are three types of constraints: (1) well-formedness constraints $\Gamma \vdash^R \psi$, (2) refinement implication constraints $\Gamma \vdash^R \psi \Rightarrow \psi'$ and (3) boundedness constraints $\Gamma \vdash^R \psi \preceq \psi'$. Implication and boundedness constraints appear directly in the typing rules; well-formedness constraints result from premises of the form $\Gamma \vdash^R \tau$ wf.

In the third phase, we run an iterative constraint solver to assign priority refinements to all of the refinement variables. If no such assignment is possible, the program is not typable and inference fails. In the remainder of this subsection, we describe the the implementation of refinements and constraints, followed by more detail on the constraint generation and solving phases.

4.1.1 Refinement and Constraint Representation. In the theoretical presentation, refinements are conjunctions of qualifiers of the form $v_1 \preceq v_2$, where v_1 and v_2 are priority constants or variables.[3] In the context of type inference, we refer to such a conjunction of constraints as a *concrete refinement* $\overline{\psi}$. The goal of inference is to infer such a concrete refinement for each priority refinement in the program. Recall that these have all been initialized, during the first phase, with refinement variables; our goal then is to find an assignment A mapping refinement variables to concrete refinements.

Constraints hold over concrete refinements as well as refinement variables with *pending substitutions* θ [29]. Pending substitutions indicate a replacement of a program variable with either another variable or a concrete refinement. These are used, for example, in typing applications. Rule $\rightarrow$E assigns the application the type $[v_2/x]\tau_2$ which requires performing substitutions on any refinements in τ_2. If, for example, $\tau_2 = \texttt{prio}[\kappa]$, it is not possible to perform the substitution until we solve for κ, and so the substituted type $\texttt{prio}[[v_2/x] \cdot \kappa]$ contains a pending substitution. We can apply pending substitutions to values and qualifiers by performing the appropriate substitutions on variables. We write $\theta \cdot v$ for the application of the pending substitution to v. Figure 8 summarizes the syntax we use to discuss refinements and qualifiers in this section.

4.1.2 Constraint Generation. Our constraint generation algorithm consists of a pass over the AST, which has been annotated with types by HM type inference. The pass applies the subsumption rule

[3] For simplicity of presentation, the presentation of the syntax in Section 2 also used qualifiers $v_1 = v_2$. In the implementation, we represent this as $v_1 \preceq v_2 \wedge v_2 \preceq v_1$.

$$
\begin{array}{lrcl}
\textit{Pending Substitutions} & \theta & ::= & \epsilon \mid [y/x]\theta \mid [\overline{\psi}/x]\theta \\
\textit{Qualifiers} & C & ::= & v \preceq v \mid C \wedge C \\
\textit{Concrete Refinements} & \overline{\psi} & ::= & [v :: C] \\
\textit{Refinements} & \psi & ::= & \overline{\psi} \mid \theta \cdot \kappa
\end{array}
$$

Fig. 8. Syntax for Refinements.

where necessary, which induces subtyping constraints $\Gamma \vdash^R \tau_1 <: \tau_2$; applying the subtyping rules in turn generates constraints on refinements.

The application and `let` cases are worth discussing in more detail. In both cases, we must perform a substitution in the type of the expression: in an application, this is to substitute the argument for the function parameter in the function's return type, and in `let`, it is to replace the variable that is going out of scope. Here we discuss the application case; we employ a similar strategy for `let`. In an application $e_1\ e_2$, the most precise return type would be $[e_2/x]\tau_2$. However, in order to keep inference decidable, we have restricted constraints in refinements to mention only priority constants and variables, not general expressions. Note that, in Figure 2, this doesn't arise because of the restriction to monadic form, a restriction which is not present in our implementation. In our implementation, if e_2 is a variable, we perform the substitution, as in the figure. Otherwise, we proceed based on the type of e_2 (determined during Hindley-Milner inference). If e_2 is not of priority type, then x cannot appear in τ_2, and so it is not necessary to perform any substitution. However, if the type is $\texttt{prio}[\psi]$, we substitute the refinement for x instead, giving the return type $[\psi/x]\tau_2$. This type reflects the fact that the argument, by virtue of its type, is guaranteed to be a priority contained in ψ. If the loss of precision resulting from this substitution is unacceptable, the programmer can let-bind e_2 to a variable (effectively locally reintroducing the monadic form of the formal type system), which can then appear in τ_2 as long as the variable is in scope.

4.1.3 Constraint Solving. The final phase of type inference solves the constraints to produce an assignment A of concrete refinements to refinement variables. As in liquid type inference, for each refinement variable κ, we initialize $A(\kappa) = [v :: C_\kappa]$ where C_κ is the set of all possible qualifiers over priority constants and variables that are in scope at a constraint involving κ. A well-formedness constraint is satisfied if all variables in the constraint are in the context. To check implication and boundedness constraints, we convert the constraint to the form $C_1 \Rightarrow C_2$ and then check that all of the inequalities in C_2 are provable using the assertions in C_1 as well as reflexivity and transitivity of $\preceq$. For example, a boundedness constraint $\Gamma \vdash^R [x :: C_x] \preceq [y :: C_y]$ is converted to $(Q(\Gamma) \cup C_x \cup C_y) \Rightarrow x \preceq y$ where $Q(\Gamma)$ represents the assumptions contained in Γ (e.g., a variable binding $p : \texttt{prio}[z :: C]$ induces the assumptions $[p/z]C$). Before checking a constraint, we apply pending substitutions. The case of substituting a refinement, e.g., $[[y :: C]/x]$ is, to our knowledge, unique to our setting. We handle this by passing the qualifiers in C as additional assumptions (over an existentially-quantified variable y) to the constraint checking procedure described above.

We solve the constraints in 3 phases: (1) solve all of the well-formedness constraints in a single linear pass, weakening unsatisfied constraints by removing qualifiers, (2) use an iterative worklist algorithm to solve the implication constraints, and (3) perform a linear pass over the boundedness constraints. The first two phases have direct analogs in liquid type inference, but boundnedness constraints are unique to our setting. We now describe the three phases in more detail.

For an unsatisfied constraint $\Gamma \vdash^R \theta \cdot \kappa$, we remove all qualifiers $v_1 \preceq v_2$ in $A(\kappa)$ such that either $\theta \cdot v_1$ or $\theta \cdot v_2$ is not typable under Γ. Because future attempts to solve other constraints will only further remove qualifiers, this will never cause a well-formedness constraint to become

unsatisfied again. Next, we solve the implication constraints using a standard iterative worklist algorithm. The solver picks an unsatisfied constraint $\Gamma \vdash^R \psi \Rightarrow \theta \cdot \kappa$ and removes from $A(\kappa)$ all qualifiers C such that $\Gamma \vdash^R \psi \Rightarrow \theta \cdot C$ is not provable. Doing so may cause other implication constraints to become unsatisfied; we add back to the worklist any constraints whose context or antecedent has changed and repeat the process.

Finally, we perform one linear pass to check the boundedness constraints. If any boundedness constraint is unsatisfied, the program is untypable and inference raises an error. The reason is that an unsatisfied boundedness constraint $\Gamma \vdash^R \psi \preceq \psi'$ could only be made satisfied by further constraining (i.e., strengthening) the assignment for ψ or ψ'. However, we are guaranteed to have already found the strongest set of assignments that solves the implication constraints and so this strengthening would cause those constraints to be unsatisfiable.

Running Time of Inference. Inference time is dominated (both asymptotically and in practice) by solving the implication constraints. Because this procedure is very similar to the equivalent phase of liquid type inference, we refer the reader to the paper by Rondon et al. [29] for a detailed analysis. However, the running time of this phase is $O(C \cdot V \cdot Q^2)$, where C is the number of constraints (which is is linear in the size of the typing derivation), V is the maximum number of variables of priority type, and Q is the maximum number of qualifiers in a refinement.

4.2 Compilation to OCaml

After type inference, our implementation transpiles the code to Multicore OCaml. For futures and priorities, the generated code uses Priodomainslib [22], an extension of Multicore OCaml's Domainslib library for fine-grained parallel programming. Priodomainslib provides fine-grained tasks compatible with PriML's threads (including allowing threads to change priority at runtime, as required by our extension) as well as a priority scheduler based on work stealing. The translation from PriML to Multicore OCaml is similar to the translation to Standard ML described by Muller et al. [24]: PriML threading constructs are translated to calls to Priodomainslib, and commands are translated to expressions inside thunks.

Priodomainslib provides a priority type as well as an API for registering and manipulating priorities. Our transpiler collects the set of priorities declared in the PriML code and then emits calls to Priodomainslib's priority creation function for each priority. For example, the priority declaration `priority p` becomes `let p : priority = Priority.new_priority ()`. One wrinkle is that Priodomainslib does not support partially-ordered priorities; rather, priorities must be registered with `Priority.new_priority ()` in a total order from lowest to highest priority. To accomplish this, our transpiler performs a topological sort on the set of collected priorities and orderings to produce a total order of priorities before emitting the OCaml code.

4.3 Quantitative Evaluation

Although a thorough quantitative evaluation is out of the scope of this paper, we report on compilation times for several example programs to give a sense of the scalability of the approach. Table 2 shows, for each program, the number of constraints generated and the time in milliseconds for each phase of type checking as well as the overall compilation time, which also includes the time taken by parsing, translation to OCaml, and bytecode compilation using the OCaml compiler. The table also reports the number of non-blank, non-comment lines of code for each program (including only PriML code; some of the benchmarks are linked with OCaml code that is not checked by the PriML compiler). The four sample programs are as follows:

- **deadline**: a small skeleton of a program that selects and changes to a new priority based on the time to a deadline.

Table 2. Lines of code, number of constraints, and breakdown of compilation time for four programs.

Program	LoC	Constraints	Time (ms)			
			HM Inference	Generation	Solving	Overall
deadline	29	45	0.32	0.15	1.5	28.0
exhaust-sync	98	735	0.34	0.58	53.4	84.0
nqueens-serv	115	508	0.43	2.35	52.2	86.2
web	372	631	1.67	4.08	110.8	169.6

- **exhaust-sync**: a series of unit tests of various instantiations of spawning and synchronizing on a thread at different priorities.
- **nqueens-serv**: a foreground thread in a tight loop accepts numbers n on standard input and spawns background threads to solve the N Queens problem for n (computing the number of ways in which n queens can be placed on an $n \times n$ chess board without attacking each other), at a priority determined based on n to allow small instances to finish quickly.
- **web**: a small web server that forms our case study in Section 5.

For all examples, compilation completes in well under a second. As expected, compilation time is dominated by constraint solving (except for deadline, the smallest example, where it is dominated by the call to the OCaml compiler). The web server benchmark results in a similar number of constraints to smaller benchmarks because most of the web server code doesn't involve priorities. However, constraint solving still takes longer because the code that does handle priorities is more complex and involves more priority variables and larger numbers of qualifiers.

We note that, although the benchmarks studied in this evaluation are relatively small, we expect the compilation time to scale reasonably to larger benchmarks because, as discussed above, most code in larger programs such as the web server doesn't interact with priorities in a meaningful way and therefore generates few to no constraints. As an extreme example of this, we note that the web server benchmark itself is compiled with hundreds of lines of OCaml library code implementing hash tables, network functions, and other libraries. This code does not interact with PriML's threads or priorities at all and therefore, need not be, and isn't, processed by the PriML compiler.

5 Case Study

We have implemented a concurrent web server in our extended PriML, extending a PriML web server from previous work [24], using the new features of our implementation to dynamically adjust the priority of background threads.[4] Here, we describe the web server in detail, including a discussion of a subtle priority inversion bug in the design which was caught by our type checker.

Overall Design. The main functionality of the web server consists of a single thread (the "accept" thread) running a loop that listens for new connections and spawns a new thread (a "serve" thread) to handle the newly connected host. The serve threads operate in parallel, themselves running a loop that listens for new requests from their host and serves them.

In the background, the server monitors the request log for unusual activity in order to discover "bad" hosts that may be attempting a denial-of-service attack (in a similar fashion to a case study

[4]The previous version of the web server also included a background computation, but this was a simpler computation that just collected statistics about the log, and ran at only one priority.

by Sultana et al. [34]). Addresses of bad hosts are added to a hash table,[5] and when the accept loop receives a new connection, it checks for the host in the hash table.

```
1 priority low_class_p
2 priority accept_p
3 priority high_class_p
4 priority serve_p
5
6 order low_class_p < accept_p
7 order accept_p < high_class_p
8 order high_class_p < serve_p
```

Fig. 9. Priority and order declarations.

The priorities of threads matches the relative urgency of their tasks. We prioritize serving existing hosts over accepting new ones, and so the serve threads run at a priority (`serve_p`) that is higher than that of the accept threads (`accept_p`). Most of the time, we want the host-classification thread to be a background process that runs when there are spare processors, and so it generally runs at a priority `low_class_p` that is lower than the other two priorities. However, if the system is heavily loaded (possibly because a denial-of-service attack is in progress), we prioritize classifying hosts over accepting new ones—at such times, we dynamically increase the priority of the classifier to `high_class_p`, which is between `accept_p` and `serve_p`. The priorities and their orderings are declared using the code in Figure 9.

In the discussion below, we focus on the interesting uses of first-class priorities, which are mostly in the classification threads. The appendix contains a more thorough discussion of the case study, including the accept and serve threads, the use of shared state, and interfacing with OCaml libraries.

Classification Threads. The classification process runs periodically in a loop, as shown below. We omit some code for space reasons; the full code for this function is in the appendix.

```
1 fun trackloop _ =
2   let val p = if Atomic.get conns > conns_cutoff then high_class_p else low_class_p
3   in cmd {
4     change[p];
5     ...
6     hosts <- cmd { ... }; (* Add new requests to hash table *)
7     (* Process new hosts in parallel. *)
8     bad_hosts <- process_hosts p hosts;
9     ... (* Add new bad hosts to hash table *)
10    trackloop ()
11  } end
```

It first (lines 2–4) checks if the number of active connections is greater than a certain threshold, and changes its priority if necessary. Next, it acquires a mutex, and retrieves and removes the new additions to the request log buffer. The code then iterates over the new requests, adding them to a persistent log. On line 8, it calls `process_hosts`, which spawns a fixed number of threads at the current priority `p`, to process the current request log, dividing the list of hosts among the threads. This returns a list of bad hosts, which it adds to a hash table shared with the accept threads.

Discussion on the Use of Priorities. Clearly, the design of the classification subsystem relies on the ability to change priorities: without this, we would not be able to dynamically increase the priority of the classification thread when necessary (without some inelegant solution such as maintaining two classification threads at different priorities which we activate and deactivate). The code relies on first-class priorities as well. On the surface, it is a convenience: without this, the `change` operation on line 4 would need to be refactored into an `if` statement that returns one of two commands with the desired hard-coded priority, but there is a deeper problem than this inconvenience.

[5]We, however, use the OCaml `Hashtbl` library rather than the hash tables Sultana et al. designed for this purpose.

During the implementation of the web server case study, PriML's type system prevented a priority inversion, which otherwise might have been easy to introduce (and which we avoided with an additional use of first-class priorities). The `process_hosts` functions spawns threads at the priority at which the classification thread is running, and later synchronizes them, also at this priority. The original thread-spawning code in `process_hosts` was:

```
1 t <- spawn[if Atomic.get conns > conns_cutoff then high_class_p else low_class_p]
2           { ret (List.flatten (List.map process_one_host l)) };
```

This code, however, has the potential for a priority inversion: if the number of connections changed between the `change` operation on line 4 of `trackloop` and the above line in `process_hosts`, the additional processing threads might be spawned at a different, and possibly lower, priority than the main classification thread, which will later `sync` on these threads. Our extension of PriML's type system is able to catch this bug: if we conflate priority refinements with sets of priorities, the refinement on the priority of the classification thread at the time of the `sync` is $\{\texttt{low\_class\_p, high\_class\_p}\}$, and the type of `t` is $\texttt{host list thread}[\{\texttt{low\_class\_p, high\_class\_p}\}]$. It is not the case that, as required by rule Sync, $\{\texttt{low\_class\_p}, \texttt{high\_class\_p}\} \preceq \{\texttt{low\_class\_p}, \texttt{high\_class\_p}\}$. To fix this bug, we take advantage of the fact that priorities are first class in order to bind the new priority for this iteration of the classification loop as `p` and pass this as an argument to `process_hosts`, which it uses as the priority at which to spawn the threads. The type system can reason that the priority of the main classification thread and the spawned thread `t` are both `p` and thus the synchronization is safe, even though it is not known statically what priority `p` will be.

6 Related Work

Cost Models for Parallelism. Directed graphs have long been used to model parallel computations (e.g., [16, 28]). The use of work and span to bound execution time goes back to Brent [5], who gave a bound for a simple "level-by-level" scheduler; this bound was later extended to all *greedy* schedules [10]. The approach and notation we use in this paper for using cost semantics to produce a graph are inspired by Blelloch and Greiner [2, 3]. *Prompt* schedules, the formal definition of response time, and the notions of competitor work and $\vec{u}$-span come from the cost model of λ^4 [23, 24]. In our work, because priorities can change while a thread is running, priorities are assigned to *vertices* rather than to threads. In addition to the changes to the DAG notation, this requires a finer-grained definition of competitor work and, in turn, a slightly different proof technique for Theorem 1.

Competitive Multithreading. The use of multithreading for reducing latency and structuring programs is far too long-standing and widespread to summarize here, so we focus on thread priorities and their challenges. The problem of priority inversion has been observed since at least the 1980s [8, 18]. In the classic case, a priority inversion occurs when a high-priority thread is waiting for a low-priority thread to release a resource, such as a mutex. Priority inversions involving only mutexes can be solved using dynamic protocols such as *priority ceiling* [30]—our implementation assumes a runtime that behaves this way and accounts for the resulting changes in priorities when tracking the possible priorities of a thread. However, dynamic techniques are not suitable for more complicated priority inversions such as those involving condition variables [9].

Cooperative Multithreading. Abstractions for cooperative multithreading date to at least the 1970s, with early work on Id [1] and Multilisp [12, 13]. Since then, cooperative multithreading in the form of various mechanisms such as fork-join parallelism, async-finish, and futures, has appeared many times in the research literature, including various extensions of C/C++ [11], Java [7, 14, 19], and Haskell [6, 17], as well as frameworks such as Threading Building Blocks [15] and TPL [20]. In these languages and frameworks, programmers express parallelism at a high level by simply

annotating code with indications of where parallelism *might* be beneficial. Seeing the benefits of this model in terms of programmer productivity and runtime performance, many mainstream languages have in recent years added some variation of these features in production.

Responsive Parallelism. Recent work on *responsive parallelism* has aimed to unify cooperative and competitive multithreading. Muller et al. [23, 24] extended the cost models and scheduling results of cooperative multithreading to account for thread priorities. The scheduling results, like those of this paper, assume a suitable notion of well-formedness for cost graphs. They also introduced the λ^4 calculus, with a type system that guarantees the cost semantics will produce well-formed cost graphs that can be scheduled efficiently, as well as the PriML language, which implemented the ideas of the type system. Follow-up work showed that, with some extensions of the cost graph framework, similar ideas (and largely unchanged typing restrictions) could guarantee well-formedness in the presence of mutable state [25], and that an ownership-based type system for condition variables together with dynamic mechanisms like priority inheritance can allow responsive parallelism in the presence of low-level synchronization mechanisms [26]. The latter was the first work to consider threads that change priority during execution, but only in the limited context of mutex critical sections. None of this work was implemented in a sound static type system, and this paper is the first to consider first-class priorities as well as threads that can change priority in a general way.

The original implementation of PriML used a work stealing scheduler with various heuristics that attempt to approximate prompt scheduling (Section 3.1), but no formal results were shown about the scheduler. A separate line of work [31, 32] has developed work stealing algorithms for prioritized threads that approximate prompt scheduling in theory and perform well empirically. The scheduler in Priodomainslib, which this paper's implementation uses as a runtime, is based on the scheduling algorithm for Prompt I-Cilk, one of the systems from that line of work.

Dependent and Refinement Types. The use of constraints to refine the type of priorities in λ^4_ψ fits into a long line of work on introducing limited forms of dependent types into general-purpose programming languages. The work on DML(C) [37] shows that type checking for a dependently-typed extension of ML can be decidable if constraints on types are drawn from a decidable domain C. Liquid types [29] further restrict refinements in order to guarantee decidable type *inference*: at key points in the typing derivation, types are required to be logically quantified ("liquid") by a conjunction of quantifiers drawn from a decidable logic; this particular restriction ensures that there are a finite number of potential refinements and guarantees the correctness of the solving and weakening procedures. Our priority refinements are of the same form, but use new forms of constraint not present in previous liquid types literature. Our type inference algorithm, as well as the notation for concepts like pending substitutions, are drawn from the liquid types literature.

7 Conclusion

This work represents an important step toward a language that combines low-level concurrent code and fine-grained parallelism. We extend the state of the art in responsive parallelism by developing a core calculus λ^4_ψ whose threads are dynamically assigned priorities, which are first-class values of the language. A refinement type system tracks priorities throughout the code and prevents priority inversions which would hinder efficient execution. We have implemented these ideas in the PriML language for responsive parallelism, and implemented an operational web server as a case study.

Acknowledgments

The authors would like to thank the anonymous reviewers for their helpful feedback. This paper was based on work partially supported by the National Science Foundation under award CCF-2603531.

Data Availability Statement

The artifact for this paper consists of the PriML compiler, extended with type inference for dynamic and first-class priorities as described in Section 4, and the new backend to generate Multicore OCaml code. In addition, the artifact contains the full PriML code for the web server case study described in Section 5. The artifact is available on Zenodo. [21]

The appendix referred to in the paper containing full versions of figures, proofs, and other details, is available as supplementary material in the ACM Digital Library.

References

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

[2] Guy Blelloch and John Greiner. 1995. Parallelism in sequential functional languages. In *Proceedings of the 7th International Conference on Functional Programming Languages and Computer Architecture (FPCA '95)*. ACM, 226–237. doi:10.1145/224164.224210

[3] Guy E. Blelloch and John Greiner. 1996. A provable time and space efficient implementation of NESL. In *Proceedings of the 1st ACM SIGPLAN International Conference on Functional Programming*. ACM, 213–225. doi:10.1145/232627.232650

[4] William J. Bowman. 2024. A Low-Level Look at A-Normal Form. *Proc. ACM Program. Lang.* 8, OOPSLA2, Article 277 (Oct. 2024), 27 pages. doi:10.1145/3689717

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

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

[7] Philippe Charles, Christian Grothoff, Vijay Saraswat, Christopher Donawa, Allan Kielstra, Kemal Ebcioglu, Christoph von Praun, and Vivek Sarkar. 2005. X10: an object-oriented approach to non-uniform cluster computing. In *Proceedings of the 20th annual ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications* (San Diego, CA, USA) *(OOPSLA '05)*. ACM, 519–538. doi:10.1145/1103845.1094852

[8] Dennis Cornhill and Lui Sha. 1987. Priority Inversion in Ada. *Ada Letters* VII, 7 (Nov. 1987), 30–32. doi:10.1145/36072.36073

[9] Tommaso Cucinotta. 2013. Priority Inheritance on Condition Variables. *Proceedings of the 9th International Workshop on Operating Systems Platforms for Embedded Real-Time applications (OSPERT 2013)*.

[10] Derek L. Eager, John Zahorjan, and Edward D. Lazowska. 1989. Speedup versus efficiency in parallel systems. *IEEE Transactions on Computing* 38, 3 (1989), 408–423. doi:10.1109/12.21127

[11] Matteo Frigo, Charles E. Leiserson, and Keith H. Randall. 1998. The Implementation of the Cilk-5 Multithreaded Language. In *PLDI*. 212–223. doi:10.1145/277650.277725

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

[13] Robert H. Halstead, Jr. 1984. Implementation of Multilisp: Lisp on a Multiprocessor. In *Proceedings of the 1984 ACM Symposium on LISP and functional programming* (Austin, Texas, United States) *(LFP '84)*. ACM, 9–17. doi:10.1145/800055.802017

[14] Shams Mahmood Imam and Vivek Sarkar. 2014. Habanero-Java library: a Java 8 framework for multicore programming. In *2014 International Conference on Principles and Practices of Programming on the Java Platform Virtual Machines, Languages and Tools, PPPJ '14*. 75–86. doi:10.1145/2647508.2647514

[15] Intel. 2011. Intel Threading Building Blocks. https://www.threadingbuildingblocks.org/.

[16] Richard M. Karp and Rayamond E. Miller. 1966. Properties of a Model for Parallel Computations: Determinacy, Termination, Queueing. *SIAM J. Appl. Math.* 14, 6 (1966), 1390–1411. doi:10.1137/0114108

[17] Gabriele Keller, Manuel M.T. Chakravarty, Roman Leshchinskiy, Simon Peyton Jones, and Ben Lippmeier. 2010. Regular, shape-polymorphic, parallel arrays in Haskell. In *Proceedings of the 15th ACM SIGPLAN international conference on Functional programming* (Baltimore, Maryland, USA) *(ICFP '10)*. 261–272. doi:10.1145/1863543.1863582

[18] Butler W. Lampson and David D. Redell. 1980. Experience with Processes and Monitors in Mesa. *Commun. ACM* 23, 2 (1980), 105–117. doi:10.1145/358818.358824

[19] Doug Lea. 2000. A Java fork/join framework. In *Proceedings of the ACM 2000 conference on Java Grande* (San Francisco, California, USA) *(JAVA '00)*. 36–43. doi:10.1145/337449.337465

[20] Daan Leijen, Wolfram Schulte, and Sebastian Burckhardt. 2009. The design of a task parallel library. In *Proceedings of the 24th ACM SIGPLAN conference on Object Oriented Programming Systems Languages and Applications* (Orlando, Florida, USA) *(OOPSLA '09)*. 227–242. doi:10.1145/1640089.1640106

[21] Marelle León, My Dinh, and Stefan K Muller. 2026. *Responsive Parallelism with Dynamic and First- class Priorities.* doi:10.5281/zenodo.19074795

[22] Stefan K. Muller. 2024. Priodomainslib: Prioritized Fine-grained Parallelism for Multicore OCaml. Talk. OCaml Users and Developers Workshop.

[23] Stefan K. Muller, Umut A. Acar, and Robert Harper. 2017. Responsive Parallel Computation: Bridging Competitive and Cooperative Threading. In *Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation* (Barcelona, Spain) *(PLDI 2017)*. ACM, New York, NY, USA, 677–692. doi:10.1145/3062341.3062370

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

[25] Stefan K. Muller, Kyle Singer, Noah Goldstein, Umut A. Acar, Kunal Agrawal, and I-Ting Angelina Lee. 2020. Responsive parallelism with futures and state. In *Proceedings of the 41st ACM SIGPLAN Conference on Programming Language Design and Implementation* (London, UK) *(PLDI 2020)*. Association for Computing Machinery, New York, NY, USA, 577–591. doi:10.1145/3385412.3386013

[26] Stefan K. Muller, Kyle Singer, Devyn Terra Keeney, Andrew Neth, Kunal Agrawal, I-Ting Angelina Lee, and Umut A. Acar. 2023. Responsive Parallelism with Synchronization. *Proc. ACM Program. Lang.* 7, PLDI, Article 135 (June 2023), 24 pages. doi:10.1145/3591249

[27] John H. Reppy. 1999. *Concurrent Programming in ML.* Cambridge University Press, New York, NY, USA.

[28] Jorge E Rodriguez Bezos. 1969. *A Graph Model for Parallel Computations.* Ph. D. Dissertation. Massachusetts Institute of Technology, Cambridge, Massachusetts.

[29] Patrick M. Rondon, Ming Kawaguci, and Ranjit Jhala. 2008. Liquid types. In *Proceedings of the 29th ACM SIGPLAN Conference on Programming Language Design and Implementation* (Tucson, AZ, USA) *(PLDI '08)*. Association for Computing Machinery, New York, NY, USA, 159–169. doi:10.1145/1375581.1375602

[30] Lui Sha, Ragunathan Rajkumar, and John P Lehoczky. 1990. Priority inheritance protocols: An approach to real-time synchronization. *IEEE Transactions on computers* 39, 9 (1990), 1175–1185. doi:10.1109/12.57058

[31] Kyle Singer, Kunal Agrawal, and I-Ting Angelina Lee. 2023. An Efficient Scheduler for Task-Parallel Interactive Applications. In *Proceedings of the 35th ACM Symposium on Parallelism in Algorithms and Architectures* (Orlando, FL, USA) *(SPAA '23)*. Association for Computing Machinery, New York, NY, USA, 27–38. doi:10.1145/3558481.3591092

[32] Kyle Singer, Noah Goldstein, Stefan K. Muller, Kunal Agrawal, I-Ting Angelina Lee, and Umut A. Acar. 2020. Priority Scheduling for Interactive Applications. In *SPAA '20: 32nd ACM Symposium on Parallelism in Algorithms and Architectures, Virtual Event, USA, July 15-17, 2020*, Christian Scheideler and Michael Spear (Eds.). 465–477. doi:10.1145/3350755.3400236

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

[34] Nik Sultana, Pardis Pashakhanloo, Zihao Jin, Achala Rao, and Boon Thau Loo. 2019. Hashtray: Turning the tables on Scalable Client Classification. In *2019 IFIP/IEEE Symposium on Integrated Network and Service Management (IM)*. 67–70.

[35] Kun Suo, Yong Zhao, Jia Rao, Luwei Cheng, Xiaobo Zhou, and Francis C. M. Lau. 2017. Preserving I/O prioritization in virtualized OSes. In *Proceedings of the 2017 Symposium on Cloud Computing, SoCC 2017, Santa Clara, CA, USA, September 24-27, 2017*. ACM, 269–281. doi:10.1145/3127479.3127484

[36] J.D. Ullman. 1975. NP-complete scheduling problems. *J. Comput. System Sci.* 10, 3 (1975), 384 – 393. doi:10.1016/S0022-0000(75)80008-0

[37] Hongwei Xi and Frank Pfenning. 1999. Dependent types in practical programming. In *Proceedings of the 26th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages* (San Antonio, Texas, USA) *(POPL '99)*. Association for Computing Machinery, New York, NY, USA, 214–227. doi:10.1145/292540.292560

Received 2025-11-13; accepted 2026-04-03