Responsive Parallelism with Synchronization

STEFAN K. MULLER, Illinois Institute of Technology, USA
KYLE SINGER, Washington University in St. Louis, USA
DEVYN TERRA KEENEY, Illinois Institute of Technology, USA
ANDREW NETH, Illinois Institute of Technology, USA
KUNAL AGRAWAL, Washington University in St. Louis, USA
I-TING ANGELINA LEE, Washington University in St. Louis, USA
UMUT A. ACAR, Carnegie Mellon University, USA

Many concurrent programs assign priorities to threads to improve responsiveness. When used in conjunction with synchronization mechanisms such as mutexes and condition variables, however, priorities can lead to *priority inversions*, in which high-priority threads are delayed by low-priority ones. Priority inversions in the use of mutexes are easily handled using dynamic techniques such as *priority inheritance*, but priority inversions in the use of condition variables are not well-studied and dynamic techniques are not suitable.

In this work, we use a combination of static and dynamic techniques to prevent priority inversion in code that uses mutexes and condition variables. A type system ensures that condition variables are used safely, even while dynamic techniques change thread priorities at runtime to eliminate priority inversions in the use of mutexes. We prove the soundness of our system, using a model of priority inversions based on cost models for parallel programs. To show that the type system is practical to implement, we encode it within the type systems of Rust and C++, and show that the restrictions are not overly burdensome by writing sizeable case studies using these encodings, including porting the Memcached object server to use our C++ implementation.

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

Additional Key Words and Phrases: condition variables, priority inversions, type systems, cost semantics

ACM Reference Format:
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. https://doi.org/10.1145/3591249

1 INTRODUCTION

For decades, software applications have used concurrency to perform tasks simultaneously on multiple threads. In many such applications, particularly those that interact with a user or the outside environment, different tasks have different resource and latency requirements. For example, in an email client, an event loop processing user input must run frequently to ensure responsiveness of the user interface, but a background thread that uses spare cycles to compress stored emails has no such requirement. Many systems for multithreading allow programmers to associate *priorities*

Authors' addresses: Stefan K. Muller, smuller2@iit.edu, Illinois Institute of Technology, USA; Kyle Singer, kdsinger@wustl.edu, Washington University in St. Louis, USA; Devyn Terra Keeney, dkeeney2@hawk.iit.edu, Illinois Institute of Technology, USA; Andrew Neth, aneth@hawk.iit.edu, Illinois Institute of Technology, USA; Kunal Agrawal, kunal@wustl.edu, Washington University in St. Louis, USA; I-Ting Angelina Lee, angelee@wustl.edu, Washington University in St. Louis, USA; Umut A. Acar, umut@cmu.edu, Carnegie Mellon University, USA.

135

This work is licensed under a Creative Commons Attribution 4.0 International License.
© 2023 Copyright held by the owner/author(s).
2475-1421/2023/6-ART135
https://doi.org/10.1145/3591249

with threads, corresponding to these requirements: a highly latency-sensitive thread might run at high priority, while a less-sensitive background thread would run at low priority.

For almost as long as priorities have been used in concurrent programs, programmers and researchers have observed the problem of *priority inversions*. While many formulations of the problem exist with slight variations, the essence is this: a higher-priority thread finds itself waiting for a lower-priority thread to release some resource or establish some condition. In most cases, the wait will be brief and such *bounded priority inversions* are sometimes not considered harmful in themselves. However, if the lower-priority thread in this scenario is preempted by a long-running thread of intermediate priority, this could become an *unbounded* priority inversion, which can severely impact responsiveness or prevent the program from making progress entirely. As such, it is desirable to be able to detect and prevent priority inversions.

When priority inversions are caused by contention on simple synchronization primitives such as locks or mutexes, these can be fixed using dynamic mechanisms such as *priority inheritance* or *priority ceiling* [Sha et al. 1990], both of which temporarily raise the priority of the low-priority thread so it can run and release the resource. However, the problem of priority inversion is much less understood in the context of *condition variables*, a powerful and general synchronization mechanism that can be used to encode many other mechanisms such as semaphores and monitors [Hansen 1973, 1975; Hoare 1974]. Indeed, it is impossible to use dynamic techniques analogous to priority inheritance or priority ceiling to detect and prevent priority inversions caused by condition variables. To see why this is the case, consider a program consisting of a main thread A, which spawns threads B and (some time later) C. If thread B *waits* on a condition variable (an operation which blocks until another thread performs a corresponding *signal* operation), there is no way for the thread scheduler to know which thread will perform the signal operation: indeed, thread C may not even have been spawned at the time of the "wait". Thus, it is not even possible to determine on the fly whether this blocking constitutes a priority inversion.

In this paper, we consider a static approach to the problem: our solution is a type system that soundly determines if a priority inversion might occur. This information can then be used either to reject programs with potential inversions, or as a warning to the programmer that a particular portion of the code might require additional reasoning and/or debugging to ensure that an inversion cannot occur. We study the type system in the context of an imperative calculus with static and dynamic semantics that handle both condition variables and locks, preventing priority inversions using a combination of static and dynamic techniques: we use the dynamic priority ceiling protocol for mutexes and static techniques for condition variables.

The type system statically associates a priority with each mutex and condition variable. For mutexes, this is the priority ceiling—the highest priority at which it might be acquired. For condition variables, it is a priority higher than (or equal to) threads that might wait on it and lower than (or equal to) threads that might signal it. For condition variables, the fact that such a priority exists implies that any signaling thread will be higher-priority than any waiting thread, which would appear to rule out priority inversions. This is not, however, sufficient: consider the three-thread scenario above where threads B and C are high-priority, but A is low-priority: B must wait for lower-priority thread A to create thread C, which will signal and awake B. Both threads that use the condition variable are high-priority, but there is still a priority inversion. We prevent such inversions by adding a notion of ownership [Ahmed et al. 2007; Boyland 2003; Crary et al. 1999; Grossman et al. 2002; Smith et al. 2000] of condition variables and allowing code to *promote*, or raise the priority of, condition variables under certain conditions. In the above example, thread A could only promote the condition variable to high priority, allowing thread B to wait on it, *after* creating thread C, thus avoiding the priority inversion. The type system *does not* statically restrict the use of mutexes, other than to ensure that the annotated priority ceiling is correct, but does

ensure that the use of the priority ceiling protocol would not cause runtime type errors in code that mixes the use of mutexes and condition variables.

Proving that the type system is sound in detecting priority inversions requires a formal model of priority inversions that is rich enough to describe scenarios such as the three-thread priority inversion above. Formalisms such as that of Babaoğlu et al. [1993], which is based on relationships between threads at snapshots in time, cannot encode an inversion between a waiting thread and a not-yet-created signaling thread. Instead, we build on a recent line of work [Muller et al. 2017, 2018, 2020] that represents parallel programs with priorities as Directed Acyclic Graphs (DAGs) and shows that, if a DAG meets certain "well-formedness" conditions, the corresponding program can be scheduled efficiently by a simple scheduling principle that greedily observes priorities. This yields a definition of priority inversions in terms of the impact they have on performance: a priority inversion is any interaction among threads that delays high-priority threads due to low-priority ones, and we can be assured that no such delay occurs for well-formed DAGs. This prior work also considers type systems for preventing priority inversions, but for programs that only synchronize at the completion of threads; this is a much weaker model than that allowed by mutexes and condition variables. We extend the DAG model, well-formedness definitions, and scheduling proofs, to encode concurrent programs that use mutexes and condition variables, and show that any DAG arising from a well-typed program is well-formed and therefore free of priority inversions.

Despite its power, the type system we propose in this paper uses some well-established ideas, albeit in novel ways. Indeed, we are able to encode approximations of its restrictions (with some limitations) using advanced features of the C++ and Rust type systems. To show that the restrictions of the type system do not overly hinder productivity, we have used the encodings to develop sizable case study programs, including a large real-world interactive application, the Memcached object caching server [Mem 2009] (v1.5.13), which consists of about 20,100 lines of C code.

We begin with an overview of the key ideas through examples (Section 2), followed by a formal presentation of the type system (Section 3). In Section 4, we give an overview of the cost model we use to define priority inversions, as well as a dynamic semantics for the core calculus, including the priority ceiling protocol. We also describe how we use the cost model and dynamic semantics to prove that a well-typed program has no priority inversions. The full details of the cost model, semantics, and proof are available in the full version of the paper [Muller et al. 2023]. We then discuss the C++ and Rust implementations and case studies, discuss related work, and conclude.

2 OVERVIEW AND EXAMPLES

In this section, we present examples to illustrate priority inversions involving mutexes and condition variables (CVs), as well as an overview of our proposed approach.

2.1 Condition Variables

We consider CVs with three operations (in addition to the operation that constructs a new CV): `wait`, `signal`, and `promote`. The `wait` operation causes the calling thread to become blocked on the CV, and the `signal` operation resumes one thread that is currently blocked on the CV[1] The `promote` operation is discussed later in this section. The code in Figure 1 uses an asynchronous thread to run a function `f` and write its result into a reference `result`, after which it signals a condition variable `cv` to indicate the result is available.[2] The main thread then does some other work and, when the result of the asynchronous computation is needed, waits on the CV.

[1]We discuss later in the paper how a `broadcast` operation that resumes all blocked threads can be added as a straightforward extension.

[2]This may be seen as an encoding of *futures* or *promises*, a popular mechanism for expressing parallelism.

```
void <Low>f(CV cv, int *result) {
  ...
  *result = ...;
  signal(cv);
}
void <High>main() {
  CV cv = new CV();
  int result = 0;
  spawn<Low>(f(CV, &result));
  ...
  wait(cv);
}
```

Fig. 1. An example priority inversion.

```
void <Low>f(CV<High> cv, int *result) {
  ...
  *result = ...;
  signal(cv); //ill-typed
}
void <High>main() {
  CV<High> cv = new CV<High>();
  int result = 0;
  spawn<Low>(f(CV, &result));
  ...
  wait(cv);
}
```

Fig. 2. The priority inversion is now a type error.

Unfortunately, this code suffers from a priority inversion: the `High`-priority `main` thread waits on the `Low`-priority thread running `f`. It is impossible to detect this priority inversion statically using only priority annotations on threads as in prior work [Muller et al. 2020], because when `wait` is called, the type system does not know which thread will call `signal`. Instead, we assign a priority to the CV itself and restrict the use of CVs by threads. Initially, we require that:

(1) To wait on a CV with priority ρ, a thread's priority must be less than or equal to ρ, and
(2) To signal a CV with priority ρ, a thread's priority must be greater than or equal to ρ.

This transitively ensures that any signaling thread is higher-priority than any waiting thread. The two conditions above become the first two restrictions enforced by the type system we formalize in Section 3; we will add to this list as we consider more examples of priority inversions.

In Figure 2, we define the condition variable `cv` with priority `High`, because the `main` thread waits on it. Now, the priority inversion manifests as a type error, because the `Low` priority thread executing `f` cannot signal a high-priority condition variable. Note that switching `cv` to `Low` priority would not fix this, because then the `High`-priority `main` would wait on a low-priority CV.

A subtle priority inversion. Much of the complexity of the typing restrictions comes from a particular pattern of priority inversion, which we illustrate with an instance of the classic "producer-consumer" pattern. Figure 3 shows the pseudocode, where the `main` thread runs at priority `Low`, and spawns a "producer" thread that places messages into a buffer, and a "consumer" that removes messages from the buffer and processes them. Both the producer and consumer run at priority `High`. If the consumer gets ahead of the producer and finds the buffer empty, we wish for it to sleep until the buffer is non-empty.[3] To accomplish this, both threads also share a condition variable `cv`, which must have priority `High` because it is waited on by a high-priority thread. If the consumer finds the buffer empty, it waits on `cv`. The producer signals `cv` when it adds an element to the buffer, thus waking the consumer if it is blocked.

This code example satisfies conditions 1 and 2 governing the priorities of threads and condition variables: in the example, all relevant threads and condition variables are high priority. But, this code nevertheless contains a subtle priority inversion. Consider running the program by timesharing on a single processor. When the main thread spawns the consumer, it will immediately begin running (because it is high-priority), and will run until it finds the buffer empty and waits on `cv`. At this

[3] For simplicity and to highlight the novel aspects of our type system, we assume the buffer is unbounded, so the producer never needs to block. Our system can also represent the classic case where the producer blocks when the buffer is full.

```
1  void <High>prod(CV<High> cv, Buffer *buf) {
2    while(true) {
3      ...
4      append(buf, x);
5      signal(cv);
6    }
7  }
8  void <High>cons(CV<High> cv, Buffer *buf) {
9    while(true) {
10     if empty(buffer) {
11       wait(cv);
12     }
13     x = pop(buf);
14     ...
15   }
16 }
17 void <Low>main() {
18   CV<High> cv = new CV();
19   Buffer *buf;
20
21   spawn<High>(cons(cv, buf));
22   spawn<High>(prod(cv, buf)); // ill-typed
23 }
```

Fig. 3. Initial attempt at the producer-consumer program, which has a priority inversion (and a type error, in our full system).

```
1  void <High>prod(CV<Low> cv, Buffer *buf) {
2    while(true) {
3      ...
4      append(buf, x);
5      signal(cv);
6    }
7  }
8  void <Low>main() {
9    CV<Low> cv = new CV();
10   Buffer *buf;
11
12   spawn<High>(prod(cv, buf));
13   CV<High> cv2 = promote<High>(cv);
14   // Would now be ill-typed to spawn prod
15   spawn<High>(cons(cv2, buf));
16 }
```

Fig. 4. The producer-consumer example, without type errors or priority inversions.

point, the consumer, which is high-priority, is now waiting not for the high-priority producer, but for the low-priority `main` thread, which still needs to run to create the producer. In our time-shared system, this wait could become effectively unbounded if, for example, a long-running medium priority thread (not shown in the pseudocode) preempts the low-priority `main` thread.

The problem is that the low-priority `main` thread holds on to the high-priority `cv`. When this happens, it is possible for a thread waiting on the CV to be blocked via the low-priority holding thread. To prevent this, we add an additional restriction (which we will later restate more formally) governing the interactions between threads and condition variables:

(3) (Draft) If a thread has priority ρ, then it *and its descendants* may not signal any condition variable that has priority greater than ρ.

In the example of Figure 3, the spawn of the producer is ill-typed because the low-priority `main` thread attempts to pass a high-priority condition variable to a thread that intends to signal it (it will become clear later why the type error is on the spawn rather than the signal).

It would seem that we are now stuck: the priority of the condition variable must be high (because the high-priority consumer is waiting on it), but then it cannot be passed by the low-priority main thread. To make progress, we will allow threads to *promote* condition variables by assigning them a higher priority under certain conditions. Note that when a low-priority thread promotes a condition variable, it restricts its own power over that condition variable, e.g., it can no longer

signal it. Figure 4 shows how the `main` thread can promote the condition variable in the producer-consumer example. The condition variable now starts at low priority and is passed as such to the producer (this is acceptable, because the high-priority producer may signal a low-priority condition variable). After spawning the producer, the main thread promotes the condition variable to high priority. The `promote` operation returns a new handle `cv2` to the same underlying condition variable as `cv`, but which is typed at priority `High`. This handle is passed to the consumer, which requires a CV at priority `High`. Note that the ordering of the spawns is important here: after promoting the condition variable to high priority, *any* spawn of a thread that signals the CV would be ill-typed due to restriction 3. This ensures that no producer thread can be spawned at this point, which prevents the priority inversion and type-checks.

The promote operation. Before discussing the full set of typing restrictions on the use and promotion of condition variables, we give a high-level summary of the restrictions on the condition variable API and common patterns for using it.

First, threads that signal a CV must be higher-priority than threads that wait on the CV. In many uses of CVs in our case studies, the same threads signal and wait on CVs at different times (e.g., the same thread might act as a producer and as a consumer at different times in a program). In this case, the type system will enforce that all threads that use the CV are at the same priority. There is also a pattern in which the signaling threads are at a strictly higher priority (e.g., a high-priority interaction thread in a server application sends logging messages to a low-priority background thread in a producer-consumer fashion).

The second point of concern is the order and conditions in which the threads that use a CV are spawned. This must be controlled to avoid the priority inversion of Figure 3, in which a low-priority thread spawns the consumer, which then blocks waiting for the producer to be spawned. If all of the threads that use the CV are at the same priority ρ and all of these threads call both `wait` and `signal`, these threads should all be spawned by a thread at their priority or higher (a common mistake would be to spawn them all from the low-priority initial thread). One solution that doesn't involve promotion is to spawn a temporary thread at priority ρ (or higher) which spawns all of the threads that use the CV. If the users of the CV are split between high-priority "producers" and low-priority "consumers", one can, as in Figure 4, construct the CV at a lower priority, spawn the producers, then promote the CV and spawn the consumers.

Typing restriction details. We now discuss the typing restrictions of the API at a more detailed level. A full understanding of these details is not necessary for most users of the condition variable API, but this discussion serves as a lead-in to the formal type system. Before proceeding, we must still discuss one additional important feature of promotion. Suppose that, in Figure 4, the producer thread instead ran at priority `Low`. This still obeys all restrictions discussed so far: the producer would signal a condition variable at its own priority, and the main thread would still have the ability to pass the condition variable to the producer. Of course, this code would have a priority inversion, as the high-priority consumer would be waiting on the low-priority producer. To prevent this, we further fortify our restrictions and disallow promoting a condition variable to a priority ρ if any concurrently running thread may signal it at a priority lower than ρ.

To make this more formal, we equip condition variables with a notion of *ownership* (e.g., [Ahmed et al. 2007; Boyland 2003; Crary et al. 1999; Grossman et al. 2002; Smith et al. 2000]). Each thread has some level of ownership of a condition variable at every priority: conceptually, these levels are "none", "owned", and "shared." It is an invariant that if one thread owns a condition variable at priority ρ, no other thread owns or shares it at ρ (though another thread might own or share it at a different priority ρ'). Any number of threads can share a condition variable at a priority as long as no thread owns it at that priority. We can now re-state the restrictions governing condition variables and threads in terms of ownership (restriction 1 is unchanged):

```
1 Mutex mut;
2 void <Low> f() {
3   with (mut) { ... }
4 }
```

```
5 void <High> g() {
6   with (mut) { ... }
7 }
```

```
8 void <Medium> h() {
9   while (1) { ... }
10 }
```

Fig. 5. A priority inversion with three threads and a mutex.

(1) To wait on a CV with priority ρ, a thread's priority must be less than or equal to ρ, and
(2) To signal a CV with priority ρ, a thread must own or share the CV at the thread's priority.
(3) To pass *any* ownership of a CV at *any* priority to another thread, a thread must own or share the CV at its own priority.
(4) No thread may own or share a CV at a priority lower than that CV's priority.
(5) To promote a CV to a priority ρ, a thread must *own* the CV at every priority lower than ρ.

A number of important facts follow directly from these restrictions. First, these restrictions prevent the hypothetical priority inversion described above: if the main thread promotes `cv` from Low to High, it must (by restriction 5) own `cv` at Low, which is impossible (by restriction 1 and the definition of ownership) if the already-spawned low-priority producer signals it. Second, the restated restriction 2, together with 4, implies the original restriction 2: If a thread has a priority lower than that of the condition variable, it cannot own or share it at the thread priority and therefore cannot signal it. Third, it is a direct result of restriction 4 that, after promoting a condition variable to priority ρ, the promoting thread must also give up ownership at all lower priorities. Finally, the formal statement of restriction 3 implies the informal version from earlier because a thread cannot own or share a higher-priority condition variable at the thread's own (lower) priority, and therefore cannot "hold onto" it in a meaningful way, i.e., pass it to other threads that might signal it.

2.2 Mutexes

Mutexes are used to enforce mutual exclusion in critical sections of code, that is, ensure that only one thread runs a critical section at a time. The language we consider in this paper syntactically wraps critical sections with the syntax `with` v s. If v is a mutex, this construct attempts to acquire the mutex; if successful, it executes s, ensuring that no other thread can acquire v during the execution of s. When completed, it releases the mutex. If the mutex referenced by v is already held by another thread, the construct blocks until it can acquire the mutex and then proceeds to run s. [4]

The classic case of a priority inversion with mutexes is illustrated in Figure 5. Suppose the low priority thread runs first and acquires the mutex `mut`. The high priority thread then runs and attempts to acquire `mut` but blocks. The medium thread then preempts the low priority thread and prevents it from running for a long or indefinite period of time, thus delaying the high priority thread, which is blocked on it. (Note that, under the definition we use in this paper, a priority inversion would still be present even in the absence of the medium priority thread; without it, however, the impact on latency would be bounded by the length of low's critical section.)

We could, as we did with condition variables, add static restrictions on the use of mutexes to prevent priority inversions. Such a restriction would require that any thread attempting to acquire a given mutex `mut` have a single priority $\rho_{\texttt{mut}}$ specific to that mutex (to see why this is necessary, consider threads at two priorities A and B potentially contending on a mutex: if the lower-priority thread is successful, the higher-priority thread is waiting for it and there is a priority inversion;

[4] In many lower-level languages, this would be equivalent to `lock v; s; unlock v`. Our syntax ensures that locks and unlocks are paired and that the mutex is always unlocked by the thread that locked it. The syntax also allows a type checker to see the entire scope of a critical section, which will be important for our type system.

```
1  Mutex<High> mut;
2  CV<Low> cv;
3  void <High>producer () {
4    while(true) {
5      ...
6      with(mut) {
7        append(buf, x)
8        signal(cv)
9      }
10   }
11 }
```

```
1  void <Low>consumer () {
2    while(true) {
3      with(mut) {
4        while (empty(buffer))
5          wait(cv);  // Ill-typed
6        x = pop(buf);
7      }
8      ...
9    }
10 }
```

Fig. 6. A more realistic version of the producer-consumer problem with mutexes and condition variables.

absent any static guarantees on which thread will succeed, A and B must be the same priority). This is not as heavy-handed a restriction as it may seem at first glance—a programmer could determine, for each mutex, the highest priority at which a thread might try to acquire it (often called the *priority ceiling*), and any critical sections would run at the priority ceiling for the mutex.[5] This is similar to the approach suggested by Lampson and Redell [1980].

Most implementations, however, take one of a number of more efficient approaches to prevent priority inversions dynamically and we model one such approach in this paper. Like the solution proposed above, the approach we will refer to as the *priority ceiling protocol* associates with each mutex a priority ceiling, that is, the highest priority of a thread that might try to acquire it. This priority is specified at the time the mutex is created. A thread T at a lower priority may acquire the mutex, but if a higher priority thread tries to acquire the mutex during T's critical section, T is dynamically promoted to the priority ceiling for the remainder of its critical section, thus preventing any threads at intermediate priorities from delaying the high-priority thread.

Although we use dynamic techniques to handle priority inversions involving mutexes, the type system still comes into play in two ways. First, to ensure soundness, the type system enforces that the priority of the mutex is indeed its priority ceiling, that is, that every thread using the mutex is at a lower priority than the mutex's priority. The second type system modification is necessary in order to properly handle code that uses both mutexes and condition variables, and we demonstrate its necessity by way of another example. Figure 6 shows a more realistic implementation of the producer-consumer code (this time with the producer higher-priority than the consumer) which uses the mutex `mut` both to guard queue operations, which may not be atomic, and ensure that the queue is still nonempty when the consumer calls `pop` (this is necessary if there are multiple consumers).[6] Because the producer has priority `High`, the priority of `mut` must be `High`. However, this means that the consumer's critical section, which ordinarily runs at `Low`, may be dynamically raised to `High` if the producer attempts to acquire the mutex during the critical section. If this occurs, the consumer may then call `wait` on the low-priority condition variable `cv` while running at high priority; this is a violation of type safety.

[5] Our language does not allow for threads to change their own priority, but this can be effectively done by spawning a new thread to run the critical section and signal a condition variable, which is waited on by the main thread.

[6] POSIX CVs would also require that `mut` is passed to `wait`, which would release the mutex while the thread blocks on `cv` and reacquire it when awoken by a signal. This behavior causes no issues for safety—indeed, it is safe for threads to temporarily release locks at any point within `with` blocks, as long as the `with` block contains all points at which the lock may be held. Because this issue is therefore orthogonal to the soundness of the type system, we omit it from the model for simplicity.

$$
\begin{array}{llll}
& & \rho \in R = \mathit{Priorities} & a, b, c \in \mathit{Threads} \\
& & \alpha \in \mathit{ConditionVariables} & \beta \in \mathit{Mutexes} \qquad \eta ::= \alpha \mid \beta \\
\mathit{Types} & \tau & ::= & \texttt{unit} \mid \texttt{nat} \mid \tau\ \texttt{ref} \mid \texttt{cv[}\alpha;\rho\texttt{]} \mid \texttt{mutex[}\rho\texttt{]} \\
\mathit{Values} & v & ::= & x \mid \langle\rangle \mid \overline{n} \mid \texttt{cv[}\alpha\texttt{]} \mid \texttt{mutex[}\beta\texttt{]} \mid \texttt{ref[}s\texttt{]} \\
\mathit{Instructions} & i & ::= & v \mid \texttt{spawn[}\rho\texttt{]\{}s\texttt{\}} \mid \texttt{ref[}\tau\texttt{]}\ v \mid !v \mid v := v \\
& & \mid & \texttt{newcv[}\rho\texttt{]} \mid \texttt{wait}\ v \mid \texttt{signal}\ v \mid \texttt{promote}\ v\ \texttt{to}\ \rho \mid \texttt{newmutex[}\rho\texttt{]} \\
\mathit{Statements} & s & ::= & \texttt{let}\ x = i\ \texttt{in}\ b \mid \texttt{with}\ v\ b \mid \texttt{if}\ v\ \texttt{then}\ b\ \texttt{else}\ b \mid \texttt{while}\ v\ b \mid s; s \mid \texttt{skip}
\end{array}
$$

Fig. 7. Syntax of λ^4_s

In order to prevent the type safety violation described above, we require that every critical section (the s of `with` v s) type-check at both its normal priority and the priority ceiling of the mutex v, as the critical section may run at either or both of these priorities. In Figure 6, this means that the consumer code is ill-typed because the critical section cannot check at `High` due to the `wait(cv)`. (The code could be made type-correct by changing the priority of `cv` to `High`.) In summary, we add two restrictions to the list (in addition to the restrictions above for condition variables):

(6) Any critical section for a mutex at priority ρ must run at a priority less than or equal to ρ (i.e., ρ is the priority ceiling of the mutex).
(7) Any critical section for a mutex at priority ρ must type check at both its own priority and ρ.

3 TYPE SYSTEM FOR RESPONSIVENESS

In this section, we formalize the ideas of Section 2 by presenting a core calculus λ^4_s with threading, condition variables and mutexes.

3.1 Syntax

Figure 7 presents the syntax of λ^4_s. Priorities ρ are drawn from a totally ordered set R with total order $\preceq$. Both the set and the order are fixed for the duration of the program but may be arbitrary. We use metavariables a, b, c, and variants to denote threads. Each condition variable and mutex has a unique such name which we will use in the semantics to track acquisition, signaling, etc. We use α and variants as the names of condition variables, β for mutexes and η for either a condition variable or a mutex. Base types are `unit` and natural numbers. We also have the type τ `ref` of mutable references to values of type τ. There are also types for handles to condition variables and mutexes. Both types indicate the priority of the handle (for mutexes, this is the priority of the mutex; condition variables may have several handles at different priorities). The type of condition variable handles also contains the name of the condition variable with which the handle is associated.

The rest of the language is in three levels: values v, instructions i and statements s. Values consist of variables as well as other expressions that do not evaluate: the unit value; natural numbers; and handles to condition variables, mutexes, and references. Instructions are in "2/3-cps" form: they contain only values as subcomponents, simplifying the presentation of the semantics. Complex expressions can be built by let-binding intermediate results, so this causes no loss of generality. Instructions perform a single operation on values and return a new value. The instruction `spawn[`ρ`]{`s`}` spawns a new thread at priority ρ to run the statement s. Instructions for manipulating global state are `ref[`τ`]` v which creates a new reference of type τ, initialized to v; $!v$, which gets the value of the reference v; and $v_1 := v_2$ which assigns v_2 to the reference v_1. The instruction `newcv[`ρ`]` creates a new condition variable and returns a handle to it at priority ρ; this handle can then be used to wait, signal, or promote the condition variable.

$$\frac{\Gamma; \Psi \vdash^R_\Sigma s@\rho' \dashv \Psi'' \qquad \forall \alpha, \rho''. \Psi(\alpha, \rho'') \neq \texttt{None} \rightarrow \Psi(\alpha, \rho) \neq \texttt{None}}{\Gamma; \Psi', \Psi \vdash^R_\Sigma \texttt{spawn}[\rho']\{s\} \rightsquigarrow \texttt{unit}@\rho \dashv \Psi'} \text{ (Spawn)}$$

$$\frac{\Gamma \vdash^R_\Sigma v : \texttt{cv}[\alpha; \rho'] \qquad \Gamma \vdash^R \rho \leq \rho'}{\Gamma; \Psi \vdash^R_\Sigma \texttt{wait}\ v \rightsquigarrow \texttt{unit}@\rho \dashv \Psi} \text{ (Wait)} \qquad \frac{\Gamma \vdash^R_\Sigma v : \texttt{cv}[\alpha; \rho'] \qquad \Psi(\alpha, \rho) \neq \texttt{None}}{\Gamma; \Psi \vdash^R_\Sigma \texttt{signal}\ v \rightsquigarrow \texttt{unit}@\rho \dashv \Psi} \text{ (Signal)}$$

$$\frac{\begin{array}{c}\Gamma \vdash^R_\Sigma v : \texttt{cv}[\alpha; \rho_1] \qquad \Gamma \vdash^R \rho_1 \leq \rho_2 \\ \forall \rho_0. (\rho_1 \leq \rho_0 \wedge \rho_2 \not\leq \rho_0) \rightarrow \Psi(\alpha, \rho_0) = \texttt{Owned} \qquad \forall \rho_0. \rho_2 \not\leq \rho_0 \rightarrow \Psi'(\alpha, \rho_0) = \texttt{None} \\ \forall \rho_0. \rho_2 \leq \rho_0 \rightarrow \Psi'(\alpha, \rho_0) = \Psi(\alpha, \rho_0) \qquad \forall \alpha' \neq \alpha, \rho_0. \Psi'(\alpha', \rho_0) = \Psi(\alpha', \rho_0)\end{array}}{\Gamma; \Psi \vdash^R_\Sigma \texttt{promote}\ v\ \texttt{to}\ \rho_2 \rightsquigarrow \texttt{cv}[\alpha; \rho_2]@\rho \dashv \Psi'} \text{ (Promote)}$$

$$\frac{\begin{array}{c}\rho' \leq \rho \qquad \alpha\ \text{fresh} \qquad \forall \rho_0, \rho' \leq \rho_0. \Psi'(\alpha, \rho_0) = \texttt{Owned} \\ \forall \rho_0, \rho' \not\leq \rho_0. \Psi'(\alpha, \rho_0) = \texttt{None} \qquad \forall \alpha' \neq \alpha. \forall \rho_0. \Psi'(\alpha', \rho_0) = \Psi(\alpha', \rho_0)\end{array}}{\Gamma; \Psi \vdash^R_\Sigma \texttt{newcv}[\rho'] \rightsquigarrow \texttt{cv}[\alpha; \rho']@\rho \dashv \Psi'} \text{ (NewCV)}$$

Fig. 8. Instruction typing rules.

$$\frac{\Gamma; \Psi \vdash^R_\Sigma i \rightsquigarrow \tau@\rho \dashv \Psi' \qquad \Gamma, x : \tau; \Psi' \vdash^R_\Sigma s@\rho \dashv \Psi''}{\Gamma; \Psi \vdash^R_\Sigma \texttt{let}\ x = i\ \texttt{in}\ s@\rho \dashv \Psi''} \text{ (Let)} \qquad \frac{}{\Gamma; \Psi \vdash^R_\Sigma \texttt{skip}@\rho \dashv \Psi} \text{ (Skip)}$$

$$\frac{\Gamma \vdash^R_\Sigma v : \texttt{mutex}[\rho'] \qquad \Gamma; \Psi \vdash^R_\Sigma s@\rho' \dashv \Psi' \qquad \Gamma; \Psi \vdash^R_\Sigma s@\rho \dashv \Psi' \qquad \Gamma \vdash^R \rho \leq \rho'}{\Gamma; \Psi \vdash^R_\Sigma \texttt{with}\ v\ s@\rho \dashv \Psi'} \text{ (WithLock)}$$

$$\frac{\Gamma \vdash^R_\Sigma v : \texttt{nat} \qquad \Gamma; \Psi \vdash^R_\Sigma s_1@\rho \dashv \Psi' \qquad \Gamma; \Psi \vdash^R_\Sigma s_2@\rho \dashv \Psi'}{\Gamma; \Psi \vdash^R_\Sigma \texttt{if}\ v\ \texttt{then}\ s_1\ \texttt{else}\ s_2@\rho \dashv \Psi'} \text{ (If)}$$

$$\frac{\Gamma \vdash^R_\Sigma v : \texttt{nat} \qquad \Gamma; \Psi \vdash^R_\Sigma s@\rho \dashv \Psi}{\Gamma; \Psi \vdash^R_\Sigma \texttt{while}\ v\ s@\rho \dashv \Psi} \text{ (While)} \qquad \frac{\Gamma; \Psi \vdash^R_\Sigma s_1@\rho \dashv \Psi' \qquad \Gamma; \Psi' \vdash^R_\Sigma s_2@\rho \dashv \Psi''}{\Gamma; \Psi \vdash^R_\Sigma s_1; s_2@\rho \dashv \Psi''} \text{ (Seq)}$$

Fig. 9. Statement typing rules.

Statements handle control flow. The statement `let` $x = i$ `in` s binds x to the result of instruction i and proceeds with s. Otherwise, statements contain only values and substatements. The mutex-guarded critical section `with` v s is also a statement. Statements also consist of conditionals, while loops, concatenation of statements, and the empty statement `skip`.

3.2 Static Semantics

We now present the type system for λ^4_s, which, together with the dynamic priority ceiling mechanism for mutexes, statically ensures that programs are free of priority inversions. The typing judgment for values is of the form $\Gamma \vdash^R_\Sigma v : \tau$. The judgment has two parameters: R is the totally ordered set of priorities. The signature Σ maps reference cells to the types they contain (these entries will be written $s{\sim}\tau$), and mutexes and condition variables to their priorities ($\alpha@\rho$ and $\beta@\rho$). There will be at most one entry in a signature for each reference cell and mutex. Condition variables, however, can appear in a signature multiple times with multiple priorities, allowing different handles to the condition variable to have different priorities. The context Γ, as usual, maps variables to types. The rules for the judgment are straightforward and are omitted for space reasons.

$$\text{Owned} \longrightarrow \text{Owned} \otimes \text{None} \qquad \text{Owned} \longrightarrow \text{None} \otimes \text{Owned} \qquad \text{Owned} \longrightarrow \text{Shared} \otimes \text{Shared}$$
$$\text{Shared} \longrightarrow \text{None} \otimes \text{Shared} \qquad \text{Shared} \longrightarrow \text{Shared} \otimes \text{None} \qquad \text{Shared} \longrightarrow \text{Shared} \otimes \text{Shared}$$
$$\text{None} \longrightarrow \text{None} \otimes \text{None}$$

Fig. 10. Rules for splitting permission levels.

The typing judgment for instructions, $\Gamma; \Psi \vdash^{R}_{\Sigma} i \div \tau@\rho \dashv \Psi'$ is more complex. It uses R, Σ, and Γ as before. The typing of instructions and statements, unlike values, depends on the priority at which the instruction is run; this priority is indicated as ρ in the judgment. The judgment also tracks ownership of condition variables which, as motivated in Section 2, can be in one of three states: None, Shared, and Owned. The mapping Ψ maps pairs of condition variable names and priorities to a "permission" level, written $\Psi(\alpha, \rho) = \pi \in \{\text{None}, \text{Shared}, \text{Owned}\}$. Instructions may produce and consume permissions, so the judgment contains the mapping before (Ψ) and after (Ψ') the instruction. Finally, the judgment indicates that the instruction produces a value of type τ.

In the Spawn rule, Ψ', Ψ means that permissions are split between Ψ' and Ψ. To split permission levels, we define the judgment $\pi_1 \longrightarrow \pi_2 \otimes \pi_3$, defined in in Figure 10, which indicates that the permission in π_1 is split between π_2 and π_3. Formally, if $\Psi'' = \Psi', \Psi$, then for all α and ρ, $\Psi''(\alpha, \rho) \longrightarrow \Psi'(\alpha, \rho) \otimes \Psi(\alpha, \rho)$. The current thread keeps Ψ' and Ψ is passed to the new thread. The second premise requires that if Ψ shares or owns a condition variable α at any priority, the spawning thread shares or owns it at its own priority. This enforces restriction 3 of the list in Section 2.

We omit the rules for operations on references, which do not interact with priorities or permissions. The Wait operation requires that the subexpression have the type of a handle at priority ρ' to the condition variable α. As motivated earlier (restriction 1), waiting requires that the priority of the current thread is lower than the priority of the handle. Waiting does not require any ownership of the condition variable. The Signal rule requires that the current thread own or share the CV α (restriction 2). Recall that signaling also requires that the thread's priority be greater than or equal to that of the condition variable, but requiring non-none permission already ensures this, as there is no permission at lower priorities (restriction 4).

The Promote rule gets the CV name and priority from the type of the CV handle, and checks that the current priority of the handle is lower than ρ_2, the priority to which it is being promoted. As motivated earlier (restriction 5), promotion requires Owned permission at all priorities not greater than ρ_2; this is checked by the third premise. The remaining premises define Ψ', the returned permission mapping. Permission for α at all priorities not greater than ρ_2 is removed (to preserve restriction 4), otherwise permissions are preserved. Finally, NewCV creates a new condition variable name for the returned handle and returns a new context Ψ' with ownership of the new condition variable initialized to Owned or None at appropriate priorities.

The judgment for statements is $\Gamma; \Psi \vdash^{R}_{\Sigma} s@\rho \dashv \Psi'$, and is largely the same as that of instructions except that statements do not return a value. The only rule that directly interacts with priorities is WithLock. This rule ensures that the critical section is well-typed at both the thread's current priority and the priority ceiling of the mutex (because the critical section may be raised to this priority at runtime): this corresponds to restriction 7 of Section 2. Regardless of priority, the critical section must be typable with the same contexts Ψ and Ψ', which are threaded through. The final premise enforces restriction 6, that is, that the priority of the current thread is less than or equal to the mutex's priority ceiling. The remaining rules thread priority and ownership through substatements. The Let rule also adds the bound variable x to the context with its appropriate type when typing the statement s. Note that in If, both branches must have the same input and output permissions. The permissions for the body of a while loop must be invariant, as enforced

by the While rule; this is, however, fairly permissive due to our coarse-grained permissions. For example, Shared permissions can be split arbitrarily within a loop as long as they remain Shared.

3.3 Extensions

Below, we discuss two synchronization operations that are not currently modeled in λ_s^4 but could be added without difficulty. We have excluded them thus far in the interest of keeping the semantics and proofs as simple as possible and focusing on the key points.

Trylock. In many implementations of mutex-based synchronization, `trylock` is a nonblocking construct that attempts to acquire a mutex; if the mutex is already locked, `trylock` returns immediately with a return value or error code indicating that it failed to acquire the mutex. We could model a variant of `with` v s that does something similar; we can call it `trywith`. In the context of our syntax, on failure, `trywith` would run an alternative statement or set a designated variable to indicate failure. The restrictions on the use of `trywith` would be identical to those for `with`.

Broadcast. The `signal` operation wakes up one thread waiting on the CV. Many implementations of CVs also include a `broadcast` primitive that wakes up all waiting threads. This operation would be straightforward to include in λ_s^4; its typing restrictions would be identical to those of `signal`. The dynamic semantics rule (analogous to Signal1) would add all threads from $W(\alpha)$ back to the thread pool and add sync edges from the current thread to all waiting threads.

4 PROOF OUTLINE

In this section, we give an overview of the soundness proof for the type system of Section 3, that is, the proof that a well-typed program is free of priority inversions. The proof has three main parts:

(1) First, it is necessary to formalize a model and definition of priority inversions which we will use for the proof. In order to represent priority inversions with condition variables, the model must capture complex dependences between threads, and not simply immediate waits-for relations. We build on prior work [Blelloch and Greiner 1995, 1996] on *cost models* that represent programs as graphs that capture all of the dependences necessary to schedule a program, as well as recent work [Muller et al. 2017, 2018, 2020] that has used variations of these models to define priority inversions.
(2) We establish a link between λ_s^4 and the cost model described above by formalizing a *cost semantics* which produces a graph in the cost model from a λ_s^4 program.
(3) We prove, using techniques based on progress and preservation, that a well-typed λ_s^4 program yields a cost graph that does not have priority inversions.

The full proof is available in the long version of the paper [Muller et al. 2023].

4.1 A Graph Model for Priority Inversions with Mutexes and Condition Variables

In this section, we overview the formalization of priority inversions that we will use to prove the correctness of the type system. Recall from Sections 1 and 2 that the obvious definitions of a priority inversion in terms of priorities of blocking threads fail to capture some situations in which a high-priority thread can wait on a lower-priority thread. Instead, we start from a careful, foundational definition of priority inversions in terms of the impact they have on programs: intuitively, a priority inversion exists if a high-priority thread may be delayed by a low-priority thread when running under a reasonable scheduler; priority inversions are therefore inherently a problem of *cost*. To formalize the above intuition, we use a model based on prior work [Muller et al. 2017, 2018, 2020] which represents parallel programs as *cost graphs*, and establishes results about the efficient schedulability of such programs, provided the graphs are "well-formed", which corresponds to the absence of priority inversions. We extend the graph model and definition of

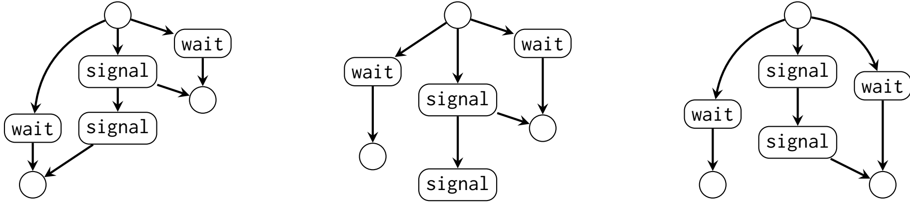

Fig. 11. Several graphs representing the same program with condition variables. Vertices are ordered vertically in order of execution.

well-formedness, and define a priority inversion in a λ_s^4 program to be any behavior that leads to an ill-formed graph. This section begins with an overview of cost graph models for parallel programs and related results and definitions, before outlining the definition of our model.

4.1.1 Preliminaries on Graph Models. We model executions of parallel programs using Directed Acyclic Graphs, or DAGs, in which vertices represent units of computation and edges represent the dependences between portions of the program. Edges can represent sequential dependences between instructions in a single thread as well as synchronizations between threads; as in the presentation of λ_s^4, we will use names like a to refer to threads: in the graph, this corresponds to the chain of vertices corresponding to instructions in thread a. We will refer to edges of the above forms as *strong edges*, and additionally use *weak edges* [Muller et al. 2020] to capture happens-before relationships between computations that occur at runtime but are not the result of explicit synchronization. As an example, weak edges were originally used to track happens-before relationships induced by global memory: if the computation u writes a value into memory and that value is read by the computation u', there may be a weak edge (u, u'). We use weak edges for a different purpose, which will be described later. We track the priorities of threads in the graph; all of the vertices of a thread are assigned the priority of that thread.

Because λ_s^4 allows arbitrary synchronization, it is possible for programs to contain *deadlocks*, where two or more threads depend on each other in a cyclic fashion; such a condition will manifest as a cycle in the dependence graph. However, because little of interest can be said about the running time of programs with deadlocks, our results will focus on non-deadlocking programs, whose graphs are acyclic. If a program is not known to be deadlock-free, the restrictions imposed by our type system are the same, and a program without priority inversions will be accepted by the type system. The guarantee we prove, however, would be a conditional one: the program can be efficiently scheduled *assuming the run of the program does not deadlock*. To be guaranteed efficient execution, our type system could be combined with static or dynamic deadlock detection, which is outside the scope of this paper. We now describe, at a high level, how we use the graph representation described above to represent programs with mutexes and condition variables.

4.1.2 Graph Models for Mutexes and Condition Variables. Because λ_s^4 programs synchronize using first-class data, rather than control flow, it is not possible to represent a given piece of code using one definite graph. Instead, a graph will represent one particular execution of a program and is necessarily dependent on scheduling decisions made at runtime.

This dynamicity can be seen clearly by considering how we would model waiting on, and signaling, a condition variable. Suppose (the instruction represented by) vertex u signals a condition variable, unblocking some thread: let u' represent the vertex in the newly-unblocked thread that becomes ready as a result of the signal. We represent the dependence between the signal and wait operations by adding the edge (u, u') to the graph. At runtime, it is easy to determine what threads

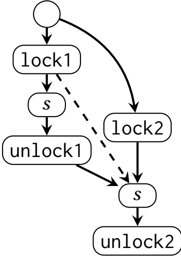

Fig. 12. Two threads contending on a lock.

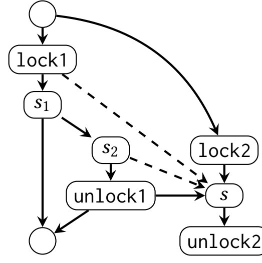

Fig. 13. Thread 1 is promoted to the priority ceiling.

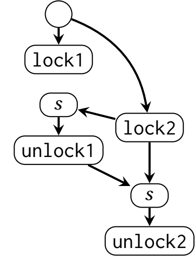

Fig. 14. The strengthening of Figure 12.

are blocked on a particular condition variable, and we do so when formalizing the operational semantics in the full proof. However, it is not possible in general to determine this statically, because which `wait` operations have run before a given `signal` operation depends on the precise runtime interleaving of the threads involved, and can be different from one run of a program to another. As an illustration, Figure 11 shows three possible DAGs that might arise from the program

```
1   CV cv; spawn {wait(cv)}; spawn {wait(cv)}; signal(cv); signal(cv);
```

The graph representations of programs with mutexes similarly depend on the order in which threads acquire mutexes. If a vertex u releases a mutex and unblocks vertex u', this induces an edge (u, u'). In addition, we add a weak edge from the vertex that successfully acquires the mutex to u', indicating the happens-before relation that the successful acquire, necessarily, ran before the unsuccessful thread. Such a DAG is illustrated in Figure 12, where `lock` indicates the beginning of a `with` v s critical section and `unlock` indicates the end of a critical section.

The model can also represent programs in which priority inversions are prevented using dynamic mechanisms such as priority inheritance or priority ceiling; we will present a graph for priority ceiling since that is the mechanism we focus on in this paper, but other mechanisms can be represented in a similar way. Figure 13 represents the same scenario as above but now supposes that the thread that acquires the mutex is running at a priority lower than that of both the second thread and the priority ceiling, so that when `lock2` runs, the first thread is promoted to the priority ceiling of the mutex. We represent this by spawning another thread (at the priority ceiling) to complete the first thread's critical section (represented by s_2 in the figure). Another weak edge is added from this thread to the (still unsuccessful) thread. When the critical section finishes, control of the first thread returns to the original thread at the original priority, represented by a sync edge from the release (`unlock1`) operation to the next operation in the original thread.

4.1.3 Well-formedness and Response Time. We now turn our attention to defining priority inversions in the extended cost model for synchronization with mutexes and condition variables.

Given a set of processors and a series of time steps, a *schedule* of a DAG is an assignment of vertices to processors at each time step respecting the dependences; this corresponds to executing the program represented by the DAG on a parallel machine. We wish to find a schedule of a DAG that minimizes the *response time* of a particular thread, which we define to be the number of steps (inclusive) from when the first vertex of the thread is spawned to when the last vertex is executed. Results such as Brent's Theorem [Brent 1974] give approximately-optimal bounds for schedules meeting certain requirements. The requirement we will use is that the schedule is *prompt* [Muller et al. 2017, 2018]. At any time step, a prompt schedule first assigns all ready nodes at the highest priority, then the next highest and so on. Processors are only left idle if no ready vertices remain.

Prompt schedules guarantee upper bounds on response times of threads for DAGs that are *well-formed* [Muller et al. 2018, 2020]. Intuitively, a DAG is well-formed if lower-priority work does not fall on the critical path of higher-priority threads. This corresponds to the absence of priority inversions, and means that programs without priority inversions can be efficiently scheduled.

In the presence of weak edges, it is possible for lower-priority work that does not actually fall on the critical path (because of happens-before relations) to *appear* to fall on the critical path. As an example, in Figure 12, the vertex `lock1` appears to be on the critical path of the right thread: it is an ancestor of `unlock2` (the last vertex of the right thread) and not an ancestor of `lock2` (the first vertex of the right thread). However, `lock2` will never have to wait for `lock1` to complete (which is, for all practical purposes, the definition of "being on the critical path") because the weak edge indicates that `lock1` has already completed when `lock2` becomes ready. We thus extend the definition of well-formedness to require that any synchronization edge that results from thread a waiting on a lock held by thread b is preceded by a weak edge indicating that thread b acquired the lock before thread a attempted to acquire it. We then use a technique called *strengthening* [Muller et al. 2020], which converts weak edges to strong edges while maintaining a conservative approximation of the same dependences with respect to a thread a. In our bound, the computations that might delay a thread a are those on the critical path of a in the strengthened graph.

The strengthening effectively encodes, using strong edges, the worst case allowed by the weak edges. As an example, in the DAG in Figure 14, the worst case is that the left thread does not run s until after the right thread attempts to lock the mutex (vertex `lock2`). We encode this with a strong edge from `lock2` to s on the left thread. Figure 14 shows the strengthening of the graph in Figure 12. Using our new definitions of well-formedness and strengthening, we prove the following:

Theorem 4.1. *Let g be a well-formed graph and a be a thread in the graph. For any prompt schedule of g that respects both weak and strong edges, the response time of a can be bounded by quantities that depend only on work at priorities higher than that of a.*

4.2 Cost Semantics

The next phase of the proof is to develop a dynamic semantics for λ_s^4 that evaluates a program and, at the same time, produces a cost graph that represents the parallel execution. The semantics operates over collections of ready threads, μ, called *thread pools*. A *configuration* $\mu; \sigma; W; L$ represents a snapshot of a running program. It includes a thread pool as well as a memory σ, which maps references to their contents. The configuration contains two additional components: W is a mapping from mutexes and condition variables to threads waiting on the mutex or CV, and L is a mapping from mutexes to either the thread currently holding the mutex or $\perp$ indicating the mutex is unlocked. The dynamic semantics judgment is $\mu; \sigma; W; L \mid g \Rightarrow \mu'; \sigma'; W'; L' \mid g'$, indicating that the configuration steps and and if the DAG is g before the step, it is g' after.

The dynamic semantics tracks information that would be available to the runtime scheduler, namely what threads are waiting on condition variables and mutexes, and what threads hold which mutexes. As the program executes, the semantics records dependency relations between threads, including both spawns and synchronizations, in the graph g. Note that all of the information in g is necessary to expose complex priority inversions involving condition variables. Consider the example of the Introduction, in which high-priority thread B is waiting for low-priority thread A to spawn thread C, which will eventually signal a condition variable. The existence of this priority inversion cannot be seen solely using information collected in W, as it depends on 1) a thread waiting on a CV, 2) an inter-thread dependence resulting from a spawn, and 3) a thread signaling a CV. All of this information is collected in the cost graph, but only 1) is collected in W.

The operational semantics models the dynamic priority ceiling protocol by raising the priority of the thread that holds a lock if a higher-priority thread attempts to acquire it. This change of priority is reflected in the graph using the mechanisms described in the previous subsection.

4.3 Correctness Proof

We prove that a graph produced by an execution of a well-typed λ_s^4 program is well-formed, meaning that it contains no priority inversions (that are not handled by dynamic priority ceiling). The essence of the proof is a type preservation proof; in addition to stating that a well-typed program remains well-typed under a step of the operational semantics, the theorem also states that a well-formed graph remains well-formed. We also ensure that a stronger set of invariants are preserved throughout execution, which are necessary to prove that well-formedness is preserved on certain transitions. Some of the key invariants are:

- Weak edges are present from threads that hold locks to threads waiting for the lock; this is required for the definition of well-formedness.
- High-priority threads do not receive low-priority vertices on their critical paths due to sync edges; we enforce this by stating that such vertices have no permission for CVs which would be required in order to signal them.
- Every thread waiting on a mutex must have a priority less than both the ceiling and the thread currently holding the lock (i.e., annotated priority ceilings are correct).
- No threads are waiting for an unlocked lock.
- Any threads waiting on a CV have a lower priority than the CV's handle.

The preservation theorem is stated roughly as follows:

Theorem 4.2. *Suppose $\mu; \sigma; W; L \mid g$ meets the invariants above, including well-typedness and well-formedness. If $\mu; \sigma; W; L \mid g \Rightarrow \mu'; \sigma'; W'; L' \mid g'$ then $\mu'; \sigma'; W'; L' \mid g'$ meets the invariants.*

We also prove a fairly standard Progress result stating that if a configuration meets the invariants, then every non-blocked thread can take a step according to the dynamic semantics. The full cost semantics and proof details are available in the full version of the paper [Muller et al. 2023].

5 IMPLEMENTATIONS AND CASE STUDIES

This section describes our Rust and C++ implementations of the type system rules and the corresponding case studies we conducted to evaluate the usability of our type system rules. Both implementations expose a library with the ability to spawn threads at various priorities and perform synchronization, and both statically enforce versions of the restrictions on synchronization primitives discussed in Section 2. We have conducted case studies using three application benchmarks, two developed from scratch and one real-world interactive application, the Memcached object caching server [Mem 2009] (v1.5.13), which we ported to use our C++ type system. We discuss the Rust implementation first, followed by the C++ implementation, focusing on where it differs from the Rust one, and then the case studies illustrating our experience working the type systems.

5.1 Rust Implementation

The Rust implementation focuses on restricting the use of CVs using Rust's rich type system, which includes notions of ownership and affinity; it does not enforce the restrictions on mutexes. Our library provides wrappers for threading and CV operations and checks the relevant restrictions on priorities and ownership before invoking equivalent operations of the standard Rust threading library. The full library consists of 551 lines of code.[7]

[7] LoC figures are measured by `tokei`.

```
1 let cv = Condvar::new(token).promote::<High>();
2 let (cv_to_p, cv_to_c) = cv.hollow_split();
3 let c_th = prio::spawn_and_pass_hollow(cv_to_c, |cv, token: &Token<High>|
4                                            {cv.dummy_wait(token);});
5 let (cv2, p_th) = prio::spawn_and_split(cv_to_p, token, |cv, token: &Token<High>|
6                                                  {cv.notify_one(token);});
```

```
error[E0277]: the trait bound `NoAccess: Partial` is not satisfied
```

Fig. 15. A skeleton of the ill-typed producer-consumer example in Rust, and the resulting error message.

To encode the seven restrictions of Section 2, our library needs to represent priorities and ownership. Priorities are represented as types that implement the trait `Priority`. Priorities are simply markers that are never instantiated; they are used only as type parameters for other generic constructs. Another trait, `Ge` (greater-or-equal), is used to establish a type-level partial ordering between these types. The priority of a thread is indicated with a `Token` type, parameterized over a priority. Each thread has a token, and some library functions require a reference to a token as proof of the current thread's priority. The `Token` type is defined such that safe code cannot access another thread's token, and a new token can only be created by spawning a thread to go with it.

The library also defines types for each ownership level (`Owned`, `Shared`, and `NoAccess`) with the `Ownership` trait. These ownership levels should not be confused with the ownership features of Rust's type system itself (which they resemble); these types make ownership explicit in the type of the CV. Each CV, of type `Condvar`, is parameterized with its priority level, as well as one of these Ownership levels for each priority. This gives rise to the limitation that the number of priorities must be fixed by the library; for our examples, three (`Low`, `Medium`, and `High`) are sufficient. We define a `split` method that, using Rust's affine-by-default type system, ensures each of the three ownership priorities follows the appropriate rules (e.g., `Owned` status implies that no other "handle" to this CV has `Owned` or `Shared` access at that same priority). Rust's affine type system also ensures that the CV handle returned by these operations is used in the future and others (e.g., old handles that have been split) are invalid. The priority and ownership(s) of a CV allow us to enforce the appropriate restrictions on uses of CVs. An important aspect of enforcing these restrictions is that sharing of `Condvar`s across threads must be restricted. We implement this using another trait, `PrioSend`, and various functions that allow for passing CVs across threads in safe ways.

As an example, Figure 15 shows simplified Rust code for the ill-typed producer-consumer example of Figure 3 (only the body of the main function is shown; the code of the producer and consumer threads have been stripped to include only the signal and wait). The main thread is initialized with a token `token` of type `token<Low>`. On line 1, we use this token to create a CV and immediately promote it to priority `High`. Rust is able to infer the type `CV<High, NoAccess, NoAccess, Owned>` for `cv`, indicating a High-priority CV owned at priority High with no ownership at Low or Medium, as required by restriction 4. On line 2, we split the CV into two handles to be passed to the two threads. The `hollow_split` method gives one of these handles, `cv_to_c`, no ownership. We then pass this CV to the consumer thread (at priority High as indicated by the type of its token), which waits on it. The other CV handle is passed (on lines 5-6) to the producer thread, also at priority `High`, which signals it using the `notify_one` method. This spawn should be ill-typed by restriction 3 because the main thread has no ownership at priority `Low`, and indeed, when the code is compiled, line 5 triggers the error shown at the bottom of the figure.

Discrepancies and Limitations. Our Rust implementation diverges in small ways from the type system discussed in Section 3. First, the Rust library is limited to a fixed number of priorities.

In addition, ownership is explicit in the type of the CV (although Rust is able to infer some of these types automatically) and the programmer must manually invoke methods to appropriately split and transfer ownership. Finally, it is possible to use `unsafe` code to sidestep the thread safety mechanisms used to enforce some of the library's restrictions.

5.2 C++ Implementation

We also implemented a C++ library (consisting of 1,252 lines of code) that approximates the features and restrictions of λ_s^4; in addition to the features of the Rust library, the C++ implementation also enforces our restrictions on mutexes. The C++ library defines wrappers around threading and synchronization features provided by I-Cilk [Muller et al. 2020; Singer et al. 2020]. The C++ library represents priorities using a strategy drawn from prior work [Muller et al. 2020]: each priority is represented as a `class` and the relationship between two priorities is captured through class hierarchy via inheritance. Every thread has a priority known at compile time, initialized by invoking a parameterized function via `spawn` with the appropriate priority type. Similarly, every CV is also initialized with its own priority type and every mutex is initialized with its ceiling priority type. When a thread operates on a mutex or CV, the type system checks for priority inversions according to the restrictions discussed in Section 2 by using static asserts and `is_base_of` on the priority types of the thread and the mutex or CV. Because every critical section needs to be type checked with both the priority of the current thread and the priority ceiling of the mutex, we require the programmer to lift any critical section into its own parameterized function so it can be explicitly instantiated with both priority levels.

Encoding the notion of ownership into the type system is more complex, as C++ has a non-affine type system and does not inherently support static ownership checking. Ownerships are represented as `enums` (`none`, `shared`, or `owned`). When a CV is created, the CV is initialized with a priority and a list of ownership states, one for each priority level used in the program. CVs are implemented with variadic templates that allow different programs to initialize CVs with varying max priorities so, unlike in the Rust library, the number of priorities is not fixed by the library (though it must still be known at compile time). This templated CV serves as a wrapper, henceforth referred to as the `wrapper_CV`, with the appropriate priority and ownership types.

As in Rust, we define explicit *ownership-changing operations* on `wrapper_CV`s such as ownership splitting, transferring, and promotion on the CV, and such operations invalidate old `wrapper_CV`s that represent previous ownership levels. Doing so is a major challenge of the C++ implementation, as it is not automatically enforced as in Rust. Change of ownership types is achieved by statically invalidating the original `wrapper_CV` variable in the current lexical scope of thread a and creating a new `wrapper_CV` variable associated with the same underlying CV to be used after the `spawn` statement. To invalidate the original `wrapper_CV` variable, we utilize a compile-time counter `counter` that can be incremented at compile time. The `counter` comes with an increment method `next` that evaluates to a integer literal at compile time, and the value is incremented by one each time `next` is encountered in the current translation unit.

We use these compile-time counters as "poison flags" to indicate the validity of a given `wrapper_CV` variable. Whenever a new `wrapper_CV` variable is introduced (whether via explicit declaration by the programmer or via ownership-changing macros), a corresponding local variable of type `counter` associated with the `wrapper_CV` is also introduced in the same lexical scope. The counter starts out with value 0, indicating a valid `wrapper_CV`. When any ownership-changing macro is invoked that invalidates the `wrapper_CV`, the macro also invokes `next` on the `counter`, bringing its value to 1, indicating an invalid `wrapper_CV`. Any operations on the `wrapper_CV`—signaling or invoking ownership-changing macros—use static assert to ensure that the `wrapper_CV` is still valid and to perform the necessary checks discussed in Section 2.

Discrepancies and Limitations. Many of the discrepancies between λ_s^4 and the Rust implementation (e.g., explicit ownership changes), are also present in the C++ implementation. In addition, in λ_s^4, `if` statements may alter the ownership map provided that both the conditional and else branches alter it the same way (IF in Section 3 Figure 9), whereas in C++ the programmer must explicitly invoke the ownership-changing macros before the `if` statement if the branches require ownership change. This is because we have no good ways to check whether the two branches alter the ownership map in the same way otherwise.

To enforce typing rules, we provide additional facilities and impose certain programming restrictions. First, to ensure a CV is passed with an ownership transfer, operations invoked on the `wrapper_CV` perform name mangling. Thus, the programmer must use the appropriate macros to transfer ownership when spawning a thread with a CV and invoke a macro upon function entry to enable the use of the CV. Without either operation, the code will fail to compile. Second, to allow for a CV to be split within a loop, the library provides an alternate macro for splitting that does not invalidate the input `wrapper_CV` and asserts that the split does not change the ownership of the input. Finally, to ensure that the if-else and looping constructs interact with CVs appropriately, the library requires the programmer to use specialized constructs provided. These specialized constructs ensure that no CVs with scope beyond that of the construct's body are invalidated within the constructs, meaning no changes to the ownership permission levels. The library macro-defines away ordinary C++ if / looping constructs so that the compiler outputs an error if ordinary C++ constructs are used in a translation unit that uses CVs. Lastly, as in prior work [Muller et al. 2020], the programmer should not use unsafe type casts to modify priorities or ownership.

5.3 Application Case Studies

To evaluate the usability of our type systems, we developed a chat server in Rust (356 lines of code) and an email client in C++ (1,252 lines of code), and we ported the Memcached server, a large interactive application (20,100 lines of C code) to use our C++ type system. We describe each application in turn and discuss our experience.

Chat Server in Rust. Our simple multi-user chat server utilizes a tiered set of threads. Threads at one priority handle each connected user's TCP stream, and accept and handle new users. Threads at a lower priority write recent messages to the channel's metadata. Each channel is associated with a CV that is signaled whenever a new message arrives from any connected client. A new thread is spawned for each connected user, which takes in the TCP stream and the CV. Each of these threads acts as both a producer and consumer, both sending messages from the connected user and routinely waiting on the CV for new messages from other connected clients.

Email Client in C++. Using our C++ implementation, we implemented a multi-user shared email server based on prior work [Muller et al. 2020]. The server utilizes 5 priority levels. At the highest priority, the server accepts new client connections (one thread per connection), listens to requests from connected clients, and spawns off other types of threads to perform requests received: `send` threads process requests to send emails (highest priority); `sort` threads handle requests for sorting emails (2nd highest); and `compress` threads compress emails and `print` threads handle requests to print emails (3rd highest). There is also a `dequeue` thread that dequeues compression tasks that arise from sending emails and generates threads to perform the compressions (4th highest). The main thread, which performs setup and tear-down tasks, is at the lowest priority.

The interesting interaction is how `compress` threads are generated. Two different kinds of threads interact in a multi-producer/single consumer model: the (multiple) `send` threads act as producers, and the `dequeue` thread acts as the consumer. Since the number of `send` threads is not known at compile time (it depends on the number of send requests), we utilize the alternative splitting macro (that does not invalidate the input) within a loop to split off CV signal permissions as we spawn off

each `send` thread. When a `send` thread pushes the number of emails in an inbox over a threshold of uncompressed messages, it enqueues information on what to compress and signals the `dequeue` thread to wake up and generate a `compress` thread.

The Memcached Object Cache Server in C. The Memcached object caching server [Mem 2009] acts as a distributed in-memory cache. It is a key-value store, with the core purpose of maintaining a hash table of objects that can be updated or retrieved by clients. Memcached is written in C, and uses pthreads and I/O multiplexing to handle many clients.

The main thread in Memcached performs most of the setup code and spawns off other long-running threads, such as a resize thread that resizes the hash table when necessary, an LRU maintenance thread that maintains the (approximately) least-recently-used (LRU) ordering of items in buckets in the hash table, an LRU crawler thread that frees up items that have gone "cold" in the cache, a slab maintainer thread for balancing free memory between different size classes of Memcached's internal memory pool, and a logger thread that aggregates and logs messages and statistics from the other threads.

We split the application into three different priority levels — whenever the actual resizing occurs, it is done as a high-priority task (spawned off by the resizing thread); on the other hand, any logging related activities are done as low-priority tasks. Activities relating to handling client requests, maintaining LRUs of the cache, and memory pool management, are done as medium-priority tasks. This is convenient, as these tasks perform signaling and waiting on condition variables within critical sections of common locks.[8] We chose to keep the single medium priority, as all the tasks at this priority are inter-dependent (e.g., a client request may run out of memory if the slab thread runs at a lower priority or if the LRU crawler thread does not free the memory in time). If we desired more fine-grained priority levels, we could have changed the priority levels of certain critical sections and potentially promoted CVs from lower to higher priorities at more places.

Porting the Memcached server required modifying about 11,400 lines of memcached code, or about 57%. The work was completed by one graduate student, who was also designing and maintaining the C++ implementation of the type system. Including time to adapt and extend the C++ library implementation, the conversion required around 80 person-hours. Many of the tasks involved in the conversion were mechanical changes, including converting code from C to C++, as well as mechanical changes required by our library (e.g. converting functions to commands, converting if statements), and could have been significantly sped up by a refactoring tool. We estimate that the conversion itself would take less than 40 person-hours without a refactoring tool, and on the order of 8 to 16 person-hours with a refactoring tool.

Discussion. Being able to type check the Memcached server using our C++ type system gives us some confidence that our typing rules are not overly restrictive. For the most part, incorporating the priority annotations as we developed the applications from scratch (i.e., the chat server and the email client) was fairly straightforward. Compilation errors that we encountered due to mishandling of types (e.g., forgetting to use the right split function at the necessary code point when transferring ownerships of CVs) are easily fixed and, indeed, helped determine correct priorities for the threads. As an example, the initial design of the Rust chat server had the threads handling connected users at a higher priority than those handling new users, but this design had a subtle priority inversion, which was caught by the static restrictions.

The process of converting the Memcached server was less straightforward, as the code base is quite mature and uses coding patterns that do not work as naturally with the way we encoded the typing rules. For instance, the lock-acquire and lock-release operations are not always well-nested (e.g., the lock is acquired in one function and released in a different one), which required us to

[8] As explained in Section 2, threads acquiring the same lock must have the same priority level to avoid priority inversions.

refactor the code so that we can lift the critical section into its own parameterized function. Another example is the use of CVs involving control constructs (i.e., if-else and loops). While transforming code to use the specialized macros for these constructs is quite mechanical and straightforward, it can be error-prone if there are multiple levels of nesting of such constructs (as occurs in Memcached). If we were to write the code from scratch, we would have rewritten the control flow to avoid complex nesting of control constructs.

The implementations and case studies also provided evidence that our system is usable and is not overly complex. None of the implementors are experts in the theory of type systems. The implementation of the Rust library was performed by an advanced undergraduate who was not involved in the design of the type system, after reading the presentation in Section 2 and one or two discussion meetings with the first author. The implementation of the Rust chat server was performed by a different advanced undergraduate, who was not involved in the design or implementation of the type system, with similar preparation.

6 RELATED WORK

Cooperative and Competitive Threading. Work on *cooperative* threading, in which threads cooperate to complete a (computational) task dates back to the late 1970s with systems such as Id [Arvind and Gostelow 1978] and Multilisp [Halstead 1985, 1984]. Since then, it has been studied in the context of many languages, including parallel dialects of ML [Arora et al. 2021; Fluet et al. 2011, 2007; Raghunathan et al. 2016; Westrick et al. 2020], Haskell [Chakravarty et al. 2007; Peyton Jones et al. 2008], and Java [Charles et al. 2005; Imam and Sarkar 2014]. *Competitive threading*, or concurrency, in which threads are used to improve a program's responsiveness, latency, or compositionality, has been the subject of research and practice for decades, as have the problems of scheduling and synchronization that result. The body of related research in this area is too large to review in detail, but we refer the interested reader to the summary by Silberschatz et al. [2005].

Prior work by some of the authors [Muller et al. 2017, 2018] observed that adding provable responsiveness guarantees to cooperative threading requires adding language constructs to prioritize interactive threads, and careful control to avoid priority inversions. This work was later extended to handle fairness [Muller et al. 2019] as well as mutable state [Muller et al. 2020].

Efficient Scheduling and Cost Semantics. Brent [1974], and later Eager et al. [1989], bounded the lengths of schedules of parallel programs in terms of *work* and *span*. Cost semantics [Rosendahl 1989; Sands 1990] extend traditional operational semantics to track resource usage. Cost semantics have proved useful for analyzing and understand parallel programs [Acar et al. 2018, 2016; Arora et al. 2021; Blelloch and Greiner 1995].

The cost semantics of this paper builds most directly on our recent model for parallel programs with mutable state [Muller et al. 2020]. That work introduced *weak edges* to represent happens-before dependencies induced by writes and reads to memory, and *strengthening* to convert weak edges to more standard dependency edges in calculating the work and span. Given these definitions, we were able to show a scheduling bound (including both throughput and response time for interactive programs) in the style of Brent, assuming graphs are *well-formed*, that is, lack priority inversions, which they enforce statically. In this paper, we use weak edges to represent happens-before relations induced by contention on mutexes, and use a combination of static and dynamic techniques to avoid priority inversions. Our soundness result builds on and generalizes these theorems in several ways. In addition, our extensions model changing priorities at runtime (to implement the priority ceiling protocol), which is not handled by prior work on responsive parallelism.

Priority Inversions. Priority inversions have been studied since the 1980s in a variety of languages and systems (e.g., [Cornhill and Sha 1987; Lampson and Redell 1980]), and a number of solutions have been proposed, including *priority inheritance* and *priority ceiling* [Sha et al. 1990]; we

implement the latter because it often results in fewer priority promotions than priority inheritance and is simpler to reason about, though it requires foreknowledge of each mutex's priority ceiling.

The problem of priority inversion control with condition variables is much less well-studied. As discussed in Sections 1 and 2, standard dynamic priority inheritance techniques are not sufficient to avoid priority inversions in programs with CVs. This problem was observed by Cucinotta [2013], who proposes allowing programmers to specify such dependencies between tasks, so that the runtime scheduler can perform priority inheritance as necessary. We discover these dependencies automatically using our type system rather than requiring programmers to specify them, and rule out resulting priority inversions at compile time. To the best of our knowledge, this paper presents the first type system for avoiding priority inversions in programs using mutexes and CVs.

Babaoğlu et al. [1993] present an elegant graph-based formalization of priority inversions and techniques for avoiding them, which is in some ways similar to the graph-based model used in this paper. However, their formulation is based on snapshots of threads blocking on each other during execution and would not suffice to represent complex priority inversions involving the spawn histories of threads, such as the one that arises in the code of Figure 3.

Ownership and Permissions. Our typing restrictions for condition variables heavily rely on a notion of ownership. Ownership has been used in a number of systems, often to prevent data races in concurrent programming or to enable safe memory management. A number of mechanisms for tracking ownership and related properties have arisen, largely derived from linear logic [Girard 1987; Reynolds 1974]. These include capabilities (e.g., [Ahmed et al. 2007; Crary et al. 1999]) and alias types [Smith et al. 2000], and have been incorporated into languages such as Cyclone [Grossman et al. 2002] and Rust (thus inspiring our Rust implementation). We are most closely inspired by fractional permissions [Boyland 2003], although we use a fairly common relaxation of fractional permissions (e.g., [Heule et al. 2011; Naden et al. 2012]) which is less restrictive.

In the language L^3, Ahmed et al. [2007] used capabilities in a type system to allow *strong updates*, which change the type of a memory cell (ownership is required for strong updates because changing the type will invalidate other references to the cell). Our `promote` operation may be seen as a kind of strong update, as it changes the priority of a condition variable handle, which in turn changes its type, and the restrictions on promotion were loosely based on those of L^3. Promotions differ from strong updates, however, in that they do not invalidate all other handles to the condition variable; other threads may still use a handle to signal at a higher priority or wait at a lower priority. We allow this using fractional permissions, which are not considered in L^3.

7 CONCLUSION

We have presented λ_s^4, a calculus for responsive parallel programming with mutexes and condition variables. The type system of λ_s^4 statically prevents priority inversions which might arise from the use of CVs, and guarantees that the graphs produced by λ_s^4's cost semantics obey strong scheduling bounds which preclude priority inversions. The type system can be approximately encoded in Rust and C++, and these encodings can be used to write substantial programs. An exciting direction of future research would be to develop a compiler to support the typing restrictions directly; such a custom compiler could track ownership of CVs with less or no programmer intervention, and could discover priorities of CVs and priority ceilings of mutexes automatically through type inference.

ACKNOWLEDGMENTS

The authors would like to thank Jonathan Aldrich and Yu David Liu as well as the anonymous reviewers. This work is partially funded by the National Science Foundation under grants CCF-2107289, CCF-1910568, CCF-1943456, CCF-2107280, CCF-1901381, CCF-2115104, CCF-2119352, CCF-2107241, CCF-2216971, and CCF-2106699.

AVAILABILITY OF SOFTWARE

The implementation of the C++ and Rust libraries, as well as our case studies, is available at https://doi.org/10.5281/zenodo.7706983.

REFERENCES

2009. Memcached. https://memcached.org/. Accessed in July 2019.

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

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

Amal Ahmed, Matthew Fluet, and Greg Morrisett. 2007. L^3: A Linear Language with Locations. *Fundam. Inform.* 77, 4 (2007), 397–449.

Jatin Arora, Sam Westrick, and Umut A. Acar. 2021. Provably Space-Efficient Parallel Functional Programming. 5, POPL, Article 18 (Jan 2021), 33 pages. https://doi.org/10.1145/3434299

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.

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

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.

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.

John Boyland. 2003. Checking interference with fractional permissions. In *International Static Analysis Symposium*. Springer, 55–72.

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

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.

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.

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

Karl Crary, David Walker, and Greg Morrisett. 1999. Typed Memory Management in a Calculus of Capabilities. In *Conference Record of the 26th ACM Symposium on Principles of Programming Languages, San Antonio, Texas*. 262–275.

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)*.

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.

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

Matthew Fluet, Mike Rainey, John Reppy, Adam Shaw, and Yingqi Xiao. 2007. Manticore: A Heterogeneous Parallel Language. In *Proceedings of the 2007 Workshop on Declarative Aspects of Multicore Programming* (Nice, France) *(DAMP '07)*. 37–44.

Jean-Yves Girard. 1987. Linear logic. *Theoretical Computer Science* 50, 1 (1987), 1–101. https://doi.org/10.1016/0304-3975(87)90045-4

Dan Grossman, Greg Morrisett, Trevor Jim, Michael Hicks, Yanling Wang, and James Cheney. 2002. Region-Based Memory Management in Cyclone. In *Proceedings of the ACM SIGPLAN 2002 Conference on Programming Language Design and Implementation* (Berlin, Germany) *(PLDI '02)*. Association for Computing Machinery, New York, NY, USA, 282–293. https://doi.org/10.1145/512529.512563

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

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.

Per Brinch Hansen. 1973. *Operating system principles.* Prentice-Hall, Inc.

Per Brinch Hansen. 1975. The programming language concurrent pascal. *IEEE Transactions on Software Engineering* 2 (1975), 199–207.

Stefan Heule, Rustan Leino, Peter Müller, and Alexander J. Summers. 2011. Fractional Permissions without the Fractions. In *FTfJP'11, July 26, 2011, Lancaster, UK.*

Charles Antony Richard Hoare. 1974. Monitors: An operating system structuring concept. In *The origin of concurrent programming*. Springer, 272–294.

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.

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

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.

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

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. https://doi.org/10.1145/3385412.3386013

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. (2023). arXiv:2304.03753 [cs.PL]

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

Karl Naden, Robert Bocchino, Jonathan Aldrich, and Kevin Bierhoff. 2012. A Type System for Borrowing Permissions. In *Proceedings of the 39th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages* (Philadelphia, PA, USA) *(POPL '12)*. Association for Computing Machinery, New York, NY, USA, 557–570. https://doi.org/10.1145/2103656.2103722

Simon L. Peyton Jones, Roman Leshchinskiy, Gabriele Keller, and Manuel M. T. Chakravarty. 2008. Harnessing the Multicores: Nested Data Parallelism in Haskell. In *FSTTCS*. 383–414.

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

John C. Reynolds. 1974. Towards a theory of type structure. In *Programming Symposium, Proceedings Colloque sur la Programmation, Paris, France, April 9-11, 1974*. 408–423.

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

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

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.

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

Kyle Singer, Noah Goldstein, Stefan K. Muller, Kunal Agrawal, I-Ting Angelina Lee, and Umut A. Acar. 2020. Priority Scheduling for Interactive Applications. In *Proceedings of the 32nd ACM Symposium on Parallelism in Algorithms and Architectures*. Association for Computing Machinery, New York, NY, USA, 465–477. https://doi.org/10.1145/3350755.3400236

Frederick Smith, David Walker, and J. Gregory Morrisett. 2000. Alias Types. In *Proceedings of the 9th European Symposium on Programming Languages and Systems (ESOP '00)*. Springer-Verlag, London, UK, UK, 366–381.

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

Received 2022-11-10; accepted 2023-03-31