Synchronous Message-Passing with Priority

Cheng-En Chuang(✉), Grant Iraci, and Lukasz Ziarek

University at Buffalo, Buffalo, NY 14260, USA
{chengenc,grantira,lziarek}@buffalo.edu

Abstract. In this paper we introduce a tiered-priority mechanism for a synchronous message-passing language with support for selective communication and first-class communication protocols. Crucially our mechanism allows higher priority threads to communicate with lower priority threads, providing the ability to express programs that would be rejected by classic priority mechanisms that disallow any (potentially) blocking interactions between threads of differing priorities. We provide a prototype implementation of our tiered-priority mechanism capable of expressing Concurrent ML and built in the MLton SML compiler and runtime. We evaluate the viability of our implementation by implementing a safe and predictable shutdown mechanisms in the Swerve webserver and eXene windowing toolkit. Our experiments show that priority can be easily added to existing CML programs without degrading performance. Our system exhibits negligible overheads on more modest workloads.

Keywords: Priority · Synchronous message passing · Concurrent ML

1 Introduction

Message-passing is a common communication model for developing concurrent and distributed systems where concurrent computations communicate through the passing of messages via *send* and *recv* operations. With growing demand for robust concurrent programming support at the language level, many programming languages or frameworks, including Scala [12], Erlang [13], Go [11], Rust [14], Racket [1], Android [2], and Concurrent ML [19] have adopted this model, providing support for writing expressive (sometimes first-class) communication protocols.

In many applications, the desire to express priority over communication arises. The traditional approach to this is to give priority to threads [17]. In a *shared memory* model, where concurrent access is regulated by locks, this approach works well. The trivial application of priority to *message passing* languages, however, fails when messages are not just simple primitive types but communication protocols themselves (i.e. first-class representations of communication primitives and combinators). These first-class entities allow threads to perform communication protocols on behalf of their communication partners –

J. F. Morales and D. Orchard (Eds.): PADL 2021, LNCS 12548, pp. 37–53, 2021.
https://doi.org/10.1007/978-3-030-67438-0_3

a common paradigm in Android applications. For example, consider a thread receiving a message carrying a protocol from another thread. It is unclear with what priority that passed protocol should be executed - should it be the priority of the sending thread, the priority of receiving thread, or a user specified priority?

In *message-passing* models such as Concurrent ML (CML), threads communicate synchronously through protocols constructed from send and receive primitives and combinators. In CML synchronizing on the communication protocol triggers the execution of the protocol. Importantly, CML provides *selective communication*, allowing for computations to pick non-deterministically between a set of available messages or block until a message arrives. As a result of non-deterministic selection, the programmer is unable to impose preference over communications. If the programmer wants to encode preference, more complicated protocols must be introduced. Whereas adding priority to selective communication gives the programmer to ability to specify the order in which messages should be picked.

Adding priority to such a model is challenging. Consider a *selective communication*, where multiple potential messages are available and one must be chosen. If the *selective communication* only looks at messages and not their blocked senders, a choosing thread may inadvertently pick a low priority thread to communicate with, when there is a thread with higher priority waiting to be unblocked. Such a situation would lead to priority inversion. Since these communication primitives must therefore be priority-aware, a need arises for clear rules about how priorities should compose and be compared. Such rules should not put undue burden on the programmer or complicate the expression of already complex communication protocols.

In this paper, we propose a *tiered-priority scheme* that defines prioritized messages as first-class citizens in a CML-like *message-passing* language. Our scheme introduces the core computation within a message, an *action*, as the prioritized entity. We provide a concrete realization of our priority scheme called *PrioCML*, as a modification to Concurrent ML. To demonstrate the practicality of *PrioCML*, we evaluate its performance by extending an existing web server and X-windowing toolkit. The main contributions of this paper are:

1. We define a meaning of priority in a *message-passing* model with a *tiered-priority scheme*. To our knowledge, this is the first definition of priority in a *message-passing* context. Crucially we allow the ability for threads of differing priorities to communicate and provide the ability to prioritize first-class communication protocols.
2. We present a new language *PrioCML*, which provides this *tiered-priority scheme*. *PrioCML* can express the semantics of polling, which cannot be modeled correctly in CML due to non-deterministic communication.
3. We implement the language *PrioCML* and evaluate on the Swerve web server and the eXene windowing toolkit.

2 Background

We realize our priority-scheme in the context of Concurrent ML (CML), a language extension of Standard ML [16]. CML enables programmers to express first-class synchronous message-passing protocols with the primitives shown in Fig. 1. The core building blocks of protocols in CML are events and event combinators. The two base events are `sendEvt` and `recvEvt`. Both are defined over a channel, a conduit through which a message can be passed. Here `sendEvt` specifies putting a value into the channel, and `recvEvt` specifies extracting a value from the channel. It is important to note both `sendEvt` and `recvEvt` are first-class protocols, and do not perform their specified *actions* until synchronized on using the `sync` primitive. Thus the meaning of sending or receiving a value is the composition of synchronization and an event – `sync (sendEvt(c, v))` will place the value `v` on channel `c` and, `sync (recvEvt(c))` will remove a value `v` from channel `c`. In CML, both sending and receiving are synchronous, and therefore the execution of the protocol will block unless there is a matching action.

```
sendEvt:'a chan * 'a -> unit event   guard:(unit -> 'a event) -> 'a event
recvEvt:'a chan -> 'a event          wrap:'a event * ('a -> 'b) -> 'b event
sync   :'a event -> 'a               choose:'a event list -> 'a event
```

Fig. 1. CML Primitives

The expressive power of CML is derived from the ability to compose events using event combinators to construct first-class communication protocols. We consider three such event combinators: `wrap`, `guard`, and `choose`. The `wrap` combinator takes an event `e1` and a post-synchronization function `f` and creates a new event `e2`. When the event `e2` is synchronized on, the actions specified in the original event `e1` are executed, then the function `f` is applied to the result. Thus the result of synchronizing on the event `e2` is the result of the function `f`. Much like `wrap` provides the ability to specify post-synchronization actions, `guard` provides the ability to specify pre-synchronization actions.

To allow the expression of complex communication protocols, CML supports selective communication. The event combinator `choose` takes a list of events and picks an event from this list to be synchronized on. For example `sync (choose([recvEvt(c1), sendEvt(c2, v2)]))` will pick between `recvEvt(c1)` and `sendEvt(c2, v2)` and based on which event is chosen, will execute the action specified by that event. The semantics of choice depends on whether any of the events in the input event list have a matching communication partner available. Simply put, `choose` picks an available event if only one is available, or nondeterministically picks an event from the subset of available events out of the input list. For example, if some other thread in our system performed `sync (sendEvt(c1, v1)`, then `choose` will pick `recvEvt(c1)`. However, if a third

thread has executed `recvEvt(c2)`, then `choose` will pick nondeterministically between `recvEvt(c1)` and `sendEvt(c2, v2)`.

3 Motivation

To illustrate the desire for priority in communication, consider a server written in CML. For such a server, it is important to handle external events gracefully and without causing errors for clients. One such external event is a shutdown request. We want the server to terminate, but only once it has reached a consistent state and without prematurely breaking connections to clients. Conceptually, each component needs to be notified of the shutdown request and act accordingly.

Leveraging the first-class events of CML, we can elegantly accomplish this. If a server is encoded to accept new work via communication in its main processing loop, we can add in shutdown behavior by using selective communication. Specifically, we can pick between a shutdown notification and accepting new work. The component can either continue or begin the termination process. However, by introducing selective communication, we also introduce non-determinism into our system. The consequence is we have no guarantee that the server will process the shutdown event if it consistently has the option to accept new work. The solution is to constrain the non-deterministic behavior through the introduction of priority. If we attach a higher priority to the shutdown event, we express our desire that given the option between accepting new work and termination, we would prefer termination. Here priority allows the programmer to express intent and guide the resolution of the non-deterministic behavior.

Where to added priority in the language, however, is not immediately clear. In a message-passing system, we have two entities to consider: computations, as represented by threads, and communications as represented by first-class events. In our shutdown example, the prioritized element is a communication, not a computation. If we directly applied a thread-based model of priority to the system, the priority of that communication would be tied to the thread that created it. We could isolate a communication into a dedicated thread to separate its priority. While simple, this approach has a few major disadvantages. It requires an extra thread to be spawned and scheduled. This approach also is not easily composed, with a change of priority requiring the spawning of yet another thread. A bigger issue is that the introduction of the new thread breaks the synchronous behavior that the CML communication primitives provide. When communication is the only method ordering computations between threads, this is a major limitation on what can be expressed.

Instead, consider what happens if we attach priority directly to communication. In the case of CML, since communications are first-class entities, this would mean prioritizing events. By giving a higher priority to the shutdown event (or a user interaction event), the programmer can express the intent for those to be handled as soon as is possible. If the time between communications is bounded, this provides a guarantee of responsiveness to the application. As soon as we hit the communication point, any available shutdown messages will be processed, even if new computations are available.

While event priority allows us to express communication priority, we still desire a way to express the priority of the computations. In the case of our server, we may want to give a higher priority to the act of serving clients over background tasks like logging. The issue here is not driven by communications between threads but rather competing for computation. As such, we arrive at a system with both event and thread priority.

In a system with message passing, however, this gives rise to priority inversion caused by communication. This happens when communication patterns result in a low priority thread getting scheduled in place of a high priority thread due to a communication choosing the low priority thread over the high priority one. We have no guarantee that the communication priorities agree with the thread priorities. To see this effect, consider the CML program shown in Fig. 2.

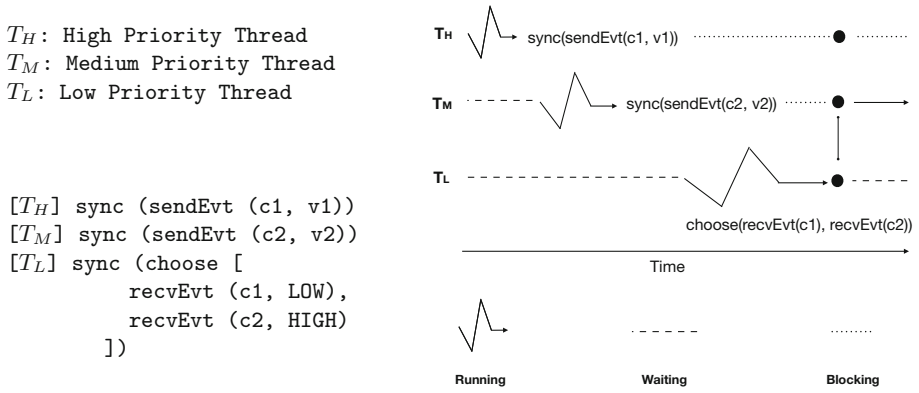

Fig. 2. Priority Inversion by Choice

The programmer is free to specify event priorities that contradict the priorities of threads. Therefore, to avoid priority inversion, we must make `choose` aware of thread priority. A naive approach is to force the thread priority onto events. That is, an event would have the priority equal to that of the thread that created it. At first glance, it seems to solve the problem that shows up in the example above. The choice in T_L now can pick `recvEvt c1` as the matching `sendEvt (c1, v1)` comes from T_H. This approach effectively eliminates event priorities, reviving all of the above issues with a purely thread-based model.

The solution is to combine the priorities of the thread and the event. In order to avoid priority inversion, the thread priority must take precedence. This resolves the problem illustrated in Fig. 2. To resolve choices between threads of the same priority, we allow the programmer to specify an event priority. This priority is considered after the priority of all threads involved. This allows the message in our shutdown example to properly take precedence over other messages from high priority threads.

This scheme is nearly complete but is complicated by CML's exposure to events as first-class entities. Specifically, events can be created within one thread and sent over a channel to another thread for synchronization. When that happens, applying the priority of the thread that created the event brings back the possibility of priority inversion. To see why, consider the example in Fig. 3.

```
[T_H] sync(sendEvt(c3, sync(sendEvt(c2, v2)))); sync(sendEvt(c1, v1))
[T_M] sync(recvEvt(c3))
[T_L] choose(recvEvt(c1), recvEvt(c2))
```

Fig. 3. Priority Inversion Due to Passing of Events

In this example, T_H sends a `sendEvt` over the channel `c2` which will be received and synchronized on by T_M. It is to be noted that this `sendEvt` will be at the highest priority (which was inherited from its creator T_H) even though it is synchronized on by T_M. T_H then sends out a value `v1` on channel `c1`. T_L has to choose between receiving the value on channel `c1` or on channel `c2`. Since T_H and T_M are both of higher priority than T_L, they will both execute their communications before T_L does. Thus T_L will have to make a choice between either unblocking T_M or T_H (by receiving on channel `c2` or `c1` respectively). Recall in the current scenario, the priority is determined by the thread that created the event and not by the thread that synchronizes it. Therefore this choice will be non-deterministic; both communications are of the same priority as those created by the same thread. T_L might choose to receive on channel `c2` and thus allow the medium priority thread T_M to run while the high priority thread T_H is still blocked - a priority inversion.

The important observation to be made from this example is that priority, when inherited from a thread, should be from the thread that synchronizes on an event instead of the thread that creates the event. This matches our intuition about the root of priority inversion, as the synchronizing thread is the one that blocks, and priority inversion happens when the wrong threads remain blocked.

We have now reconciled the competing goals of user-defined event priority and inversion-preventing thread priority. In doing so, we arrive at a tiered-priority scheme. The priority given to threads takes precedence, as is necessary to prevent priority inversion. A communication's thread priority inherits from the thread that synchronizes on the event, as was shown to be required. When there is a tie between thread priorities, the event priority is used to break it. We note that high priority communications tend to come from higher priority computations. Thus, this approach is flexible enough to allow the expression of priority in real-world systems. In Sect. 5, we show this in the context of a web server and a GUI application in CML.

4 Implementation

To demonstrate our priority mechanism, we have implemented it as an extension to the CML implementation in MLton, an open source compiler for Standard ML. Our implementation consists of approximately 1400 LOC, wholly in ML.

4.1 Priority atop CML

To understand why priority at the CML language level is needed, we first consider a prioritized communication channel built from existing CML primitives.[1] Implementing communication using a prioritized channel requires a two step communication. We need one step to convey the event priority and another to effect the event's communication. The prioritized channel itself is encoded as a server that accepts communications and figures out the appropriate pairings of sends and receives (in this case based on priority).

The sender blocks, waiting to receive a notification from the server that is acting as the priority queue, while it waits for its message to be delivered by the priority queue to a matching receiver. Once the priority queue successfully sends the value to a receiver, it unblocks the sender by sending a message. The mechanism is nearly identical for a receiver, but since we need to return a value, we pass an event generating function to the channel. While the per-communication overhead is undesirable, this encoding captures the behavior of event priority for send and receive. On selective communication, however, this encoding becomes significantly more complicated. A two stage communication pattern makes encoding the clean up of events that are not selected during the choice challenging. We also still lack the ability to extract the priority information from threads. Recall that preventing priority inversions requires reasoning about the priority of both threads and events. Instead, we opted to realize our priority mechanism as a series of small modifications to the existing CML runtime.

4.2 Extensions to CML

The major changes made to CML are to the thread scheduler and channel structure. These changes are exposed through a set of new prioritized primitives, shown in Fig. 4.

```
spawnp      : (unit -> unit) -> threadPriority -> thread_id
sendEvtP    : 'a chan * 'a * eventPrio -> unit event
recvEvtP    : 'a chan * eventPrio -> 'a event
changePrio  : ('a event * eventPrio) -> 'a event
```

Fig. 4. *PrioCML* Primitives

[1] Available at: https://gist.github.com/Cheng-EnC/ea317edb62f01f55b85a9406f6093217.

We extend the thread scheduler to be a prioritized round-robin scheduler with three fixed thread priorities. While other work has explored finer-grained approaches to priority [18], for simplicity, we use a small, fixed number of priority levels. We chose three priority levels as that is enough to encode complex protocols such as the earliest deadline first scheduling [3]. Our implementation could be extended to more priority levels if desired. The new primitive `spawnp` spawns a new thread with a user-specified thread priority: `LOW`, `MED`, or `HIGH`. Threads within the highest priority level are executed in a round-robin fashion until all are unable to make further progress. This happens when all are blocking on communication. If all high priority threads are blocked, then the medium priority threads are run until either a high priority thread is unblocked or all medium threads block. This process continues with low priority threads. This scheme guarantees that a thread will never be chosen to run unless no thread of higher priority is able to make progress.

Event priority is managed by following primitives: `sendEvtP`, `recvEvtP`, and `changePrio`. The `eventPrio` is an integer where a larger number implies higher priority. The two base event primitives `sendEvt` and `recvEvt` are replaced by their prioritized versions. These functions take in an event priority and tie that priority to the created events. The `changePrio` function allows the priority of an existing event to be changed. We also note that all CML primitives continue to exist in *PrioCML*. The primitive `spawn` creates a thread with `LOW` priority. The base event constructors are given default priority levels and reduce calls to the new prioritized primitives. The combinators continue to work unchanged. In this way, our system is fully backward compatible with existing CML programs.

4.3 Preventing Priority Inversion

To make the local selection, we leverage the channel structure. To see how this is done, first consider the action pairing mechanism in unmodified CML [20]. When an event is synchronized, the corresponding action is placed in a queue over the channel it uses. If there is a match already in the channel queue, the actions are paired and removed. In the case of choice, all potential actions are enqueued. Each carries a reference to a shared flag that indicates if the choice is still valid. Once the first action in a given choice is paired, the flag is set to invalid. If the action has its flag set to invalid upon attempting a match, it is removed, and the next action in the queue is considered. This lazy cleaning of the channel queues amortizes the cost of removal.

Figure 5 shows how a synchronized event is paired. We split the channel queue into three queues in our prioritized implementation: one for each thread priority level. Keeping those three priority queues separate is what allows us to realize our tiered-priority mechanism efficiently. By looking first at the higher thread priority queues, we give precedence to thread priority over the event priority that orders each queue.

Choice is handled similarly to how it was handled before priority. Again, lists are cleared lazily to amortize the costs of removal. The major overhead our scheme introduces is that inserting an action into a channel now requires

```
val send_pqueue = array of three priority queues for pending send events
val recv_pqueue = array of three priority queues for pending recv events
if is_send_event(current_evt) then
   case deque(recv_pqueue[H]) of
     SOME recv_evt => pair with recv_evt
     NONE => case deque(recv_pqueue[M]) of ...
               case deque(recv_pqueue[L]) of ...
                 NONE => enque(send_pqueue[thread_priority(current_evt)],
                               current_evt)
else  (* Same structure as the if branch
         but switch the recv_pqueue to send_pqueue *)
```

Fig. 5. Pairing a synchronized event

additional effort to keep the queues in order. For a choice, this overhead must be dealt with for each possible communication path. The impacts of this are measurable, but minor, as discussed in Sect. 5.1.

4.4 Polling

Polling, a common paradigm in concurrent programming, is fundamentally the ability to do a non-blocking query on an event. The primitives of CML (Fig. 1 from Sect. 2) do not provide the ability to express non-blocking synchronization. The only available synchronization operation is `select`, which is blocking.

This problem is illustrated by Reppy in *Concurrent Programming in ML* [20]. At first glance, the `always` event primitive could provide a non-blocking construction. This event is constructed with a value, and when synchronized on, it immediately yields the wrapped value. By selecting between `always` and `recv` events, the synchronization is guaranteed not to block. This flawed approach, as explained by Reppy, would look as follows:

```
fun pollCh ch = sync (choose [alwaysEvt NONE, wrap (recvEvt ch, SOME)])
```

While it is true that this construction will never block, it may also ignore available communications on the channel. The choose operation in CML is nondeterministic, and could choose the `alwaysEvt` branch, even if the `recvEvt` would not block. This problem led to the introduction of a dedicated polling primitive `recvPoll` in CML. While its use is generally discouraged, it serves as an important optimization in some communications protocols outlined by Reppy.

In our discussion of *PrioCML* thus far, we have omitted discussion of the `always` event for simplicity. The `always` event is different than the communication events as there is no blocked communication partner. It represents a one sided communication. Therefore it is not immediately clear how it fits into our tiered-priority which looks at the thread priorities of two threads. By introducing priority to the `always` event primitive we could capture the polling behavior that would otherwise require a dedicated primitive. We do this by giving

`always` events a fixed priority lower than any in the tiered priority system, e.g. `changePrio(alwaysEvt, -1)` while 0 is the lowest priority in event priority. We choose to give it the lowest priority because that expresses our desire to allow another thread to proceed if at all possible. This means during a choice, we will only pick the `always` event if no other events are available. Because of our guarantee that an event is always picked if one is available though, it will still never block. Therefore, under our prioritized implementation, the above example actually works with the intended behavior.

5 Evaluation

To demonstrate that our implementation is practical we have conducted a series of microbenchmarks to measure overheads as well as a case study in a real-world webserver and GUI framework written wholly in CML. The benchmarks and case study were run on our implementation and on MLton 20180207. The benchmarking system had an Intel i7-6820HQ quad-core processor with 16 GB of RAM. We note that MLton is a single core implementation, so although it supports multiple threads these are multiplex over a single OS thread.

5.1 Microbenchmarks

We create microbenchmarks that exercise spawn, send-receive, and choice. In spawn and send-receive, we see constant overheads as shown in Fig. 6 and Fig. 7. We note that the send-receive benchmark performs n communications where n is the number of iterations, so the constant overhead leads to a steeper slope to the line. To benchmark choice, we build a lattice of selective communication. It has a grid of choice cells where a single message is sent at the top and bounces around non-deterministically until it falls out the bottom. To show the growth behavior of this benchmark, we scaled both the height and width, so for a run parameterized by n, there were n^2 choice cells, of which the message would pass through n. From the results shown in Fig. 8, we observe that the runtimes of both CML and *PrioCML* appear quadratic. Our implementation shows a cost higher by a constant factor, and thus a steeper curve. From a static analysis of the open-source Swerve web-server implementation, we believe deeply nested choice operations to be rare in real-world applications. Thus, while our implementation does exhibit noticeable slowdown on the synthetic benchmarks, we expect real-world performance to be unaffected.

5.2 Case Study: Termination in Swerve

To demonstrate that the problem of timely graceful termination is prevalent in message passing programs, we take a look at a large CML project: the Swerve web server. Swerve is full featured, modular web server written using CML with approximately 30,000 lines of code [22]. As noted by [22], Swerve lacks a graceful shutdown mechanism. Currently, shutdown of the webserver is accomplished

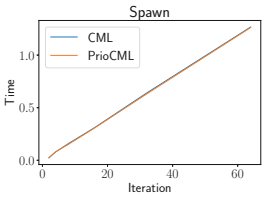

Fig. 6. Spawn

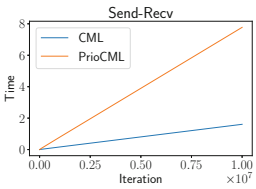

Fig. 7. Send-Receive

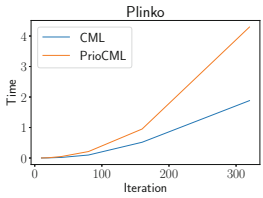

Fig. 8. Plinko (Choice)

by sending a UNIX signal to terminate the process. This approach has several drawbacks. As the process is killed immediately, it does not have the opportunity to flush the asynchronous logging channel. This can lead to incomplete logs near server shutdown. Additionally, clients being served at the time of server shutdown have their connections closed abruptly, without a chance for the server to finish a reply. This can lead to an error on the client side, or in the case that the request was not idempotent, inconsistent or partially updated state server-side. Thus to cleanly exit the server, it is important to allow all currently running tasks to complete, including both flushing the log and handling connected clients. As [22] explains, this can be handled by rejecting all new clients and waiting for existing ones to finish before flushing the logs and exiting the process. We implement such a system in Swerve, the core of which is seen in Fig. 9.

```
select [wrap (recvEvt acceptChan, new_connect),
        wrap (recvEvt lchan, handle_msg),
        wrap (shutdownEvt, fn () => shutdown num_connects)]
```

Fig. 9. Graceful Shutdown in Swerve

Here we select between the three possible actions in the main connection handling loop. We can accept an incoming connection over the channel `acceptChan` by invoking the function `new_connect`. Alternatively, we can handle a client disconnect event, sent as a message on the channel `lchan` via `handle_msg`. Lastly, we can receive a shutdown signal via the event `shutdownEvt`. This event is a receive event on a channel shared with the signal handler registered to the UNIX interrupt signal. Upon receipt of such a signal, the handler will send a message on that channel to indicate the server should begin shutdown. We leverage CML's first class events to encapsulate this mechanism and hide the implementation from the main loop. When the event `shutdownEvt` is chosen, we invoke the shutdown function which stops accepting new connections, waits for all existing connections to close, flushes the log, then removes a lock file and exits.

While this change successfully resolves the possibility of broken connections and inconsistent server state, it still has a notable limitation. We have no guarantee of a timely shutdown. The original approach of killing the process via a

```
select [wrap (recvEvt acceptChan, new_connect),
        wrap (recvEvt lchan, handle_msg),
        wrap (changePrio (shutdownEvt, 0), fn () => shutdown num_connects)]
```

Fig. 10. Prioritized Shutdown in Swerve

signal is effectively instantaneous. However, because we want to complete the currently running server tasks, the server can't shutdown immediately. We do however, want to be sure that the server does not accept additional work after being told to shutdown. Under the existing CML semantics, the server is free to continue to accept new connections indefinitely after the shutdown event has become ready, provided a steady stream of new connections is presented. This is because there is no guarantee as to which event in a choice list is selected, only that it does not unnecessarily block. Since CML only allows safe interactions between threads via message passing, we have no other way for the signal handler to alert the main loop that it should cease accepting new connections. Thus, under heavy load, the server could take on arbitrarily more work than needed to ensure a safe shutdown. We note that the MLton implementation of CML features an anti-starvation heuristic which in our testing was effective at preventing shutdown delays. This approach however is not a semantic guarantee. By adding priority, as shown in Fig. 10, we obtain certainty that our shutdown will be effected in a timely manner.

We verify the operation of this mechanism by measuring the number of clients that report broken connections at shutdown. With a proper shutdown mechanism we would see no broken connections as the server would allow all to complete before termination. As seen in Fig. 13, without the shutdown mechanism in place clients can experience broken connections. When there are very few clients, the chances that any client is connected when the process terminates are low. As the number of clients increases however, the odds of a broken connection do as well. By adding our shutdown mechanism, we prevent these broken connections. We emphasize that the introduction of priority means achieving a guarantee that the shutdown is correct is simple. The implementing code is short and concise because our mechanism integrates nicely with CML and retains its full composability. We note that event priorities are crucial to ensuring this timely shutdown. For example, consider the case where the signal handler was extended to pass on an additional type of signal such as configuration reload. We would still want to ensure that the shutdown event takes precedence. Thus we need to assign more granular priorities than those available based solely on the priority of the communicating thread.

5.3 Case Study: A GUI Shutdown Protocol

To demonstrate that priority can benefit the response time of graceful shutdown, in this section, we present an evaluation of response time measurement with a shutdown protocol in the context of eXene[10], a GUI toolkit in CML. A typical

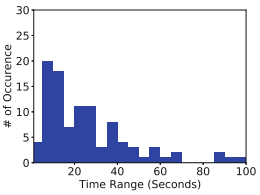

Fig. 11. CML

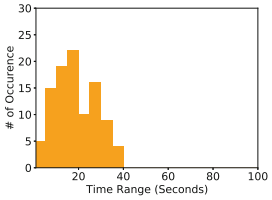

Fig. 12. *PrioCML*

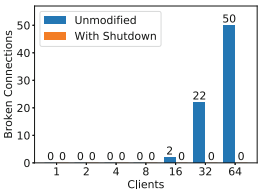

Fig. 13. Swerve

eXene program contains widgets. To realize a graceful shutdown protocol, our eXene program needs to wait for all widgets to close upon receiving a shutdown request. As a result, busy widgets tend to slow down the shutdown protocol. Moreover, the choice's nondeterministic selection degenerates the response time as widgets may overlook a shutdown request. We improve the response time with both shortening and stabilize it by proper encoding of priority in the communication protocol.

We build a widget network in eXene to compute the Fibonacci number. Each widget has a number with the corresponding position in the Fibonacci sequence. Upon a user click, the widget will calculate the corresponding Fibonacci number. By the definition of Fibonacci sequence, a widget of $fib(n)$, except $fib(0)$ and $fib(1)$, needs to communicate with other widgets, which is responsible for computing $fib(n-1)$ and $fib(n-2)$, In the meanwhile, we need to encode the shutdown event so that widget has a chance to receive shutdown request. A widget can be implemented with CML code in Fig. 14

```
select[wrap(recvEvt out_ch_req,  (* Outermost select *)
          fn p => (select[ (* Middle select *)
             wrap(sendEvt(fib_pre1_req, ()),
                  fn () => (let val v1 = recv (fib_pre1_com)
                                val _  = send (fib_pre2_req, ())
                                val v2 = recv (fib_pre2_com)
                            in select [ (* Innermost select *)
                                  wrap(sendEvt(out_ch_com, v1+v2), ...),
                                          shutdownEvt]
                            end)), ...,
              shutdownEvt])),
          shutdownEvt]
```

Fig. 14. Communication Protocol of Fibonacci Widget

Note that in above code we omit the case on the `sendEvt(fib_pre2_req, ())` for brevity. On the outermost `select`, the widget is waiting for either a compute request from `out_ch_req` or a shutdown request. Once receive a compute request, it goes to middle select. The middle select picks between the widgets

it needs to communicate and the shutdown event. The code above shows the case the widget of $fib(n-1)$ is available. After we compute the result from $fib(n-1)$, it moves to $fib(n-2)$. Finally, it adds the result and sends it to the output channel in the innermost select, which picks with another shutdown event. As for the `shutdownEvt`, every widget propagates the shutdown request to the widget of $fib(n-1)$. Hence, the shutdown protocol in the Fibonacci network is a linear chain from the largest Fibonacci widget.

We encode priority in two places. First, the priority of the shutdown event is higher than other events. The use of priority in shutdown events ensures that the shutdown request will be chosen whenever it is available during a selection. Second, we give the priority on `send` and `recv` on requesting and receiving the computation of the Fibonacci number. The message priority is higher as the number of Fibonacci is larger in the network. As a result, the widget with a larger number has the priority to request or receive computation. By giving these widgets preference, we boost the shutdown protocol as the linear chain is from largest to smallest widget.

The histogram of CML and *PrioCML* is shown as Fig. 11 and 12 respectively. We run each setting for 100 times and record the time needed to finish the shutdown protocol. We compute a large Fibonacci number to fill the network computation requests so that every widget is saturated with Fibonacci computation before requesting the shutdown protocol. The result shows that the average time spends on shutdown is improved by 26%, from 25.5 s to 18.8 s. Also, it stabilizes the response time by reducing the standard deviation from 20.7 to 9.2. This experiment shows that a shutdown protocol can be improved and become more predictable by properly encoding the priority.

6 Related Work

Priority in Multithreading: Exploration into prioritized computation extends far back into research on multithreaded systems. Early work at Xerox on the Mesa [15] programming language, and its successor project Cedar [23], illustrated the utility of multiple priority levels in a multithreaded system. These systems exposed a fork-join model of concurrency, wherein the programmer would specify that any procedure shall be called by forking a new process in which to run it. The join operation then provides a synchronization point between the two threads and allows the result of the computation to be obtained. This was implemented atop monitors, a form of mutual exclusion primitive. These systems did not consider communication as a first-class entity and only allowed it through the use of monitored objects.

First-Class Communication: Concurrent ML introduced first-class synchronous communication as a language primitive [19]. Since then, there have been multiple incarnations of these primitives, both in languages other than ML (including Haskell [4,21], Scheme [8], Go [11], and MPI [5]). Others adopted CML primitives as the base for the parallel programming language Manticore [9]. Other work has

considered extending Concurrent ML with support for first-class asynchrony [24]. We believe our approach to priority would be useful in this context. It would, however, raise some questions regarding the relative priority of synchronous and asynchronous events, analogous to the aforementioned issues with always events. Another extension of interest would be transactional events [6,7]. The introduction of priority would be a natural fit as it provides a precise expression of how multiple concurrently executing transactions should be resolved.

Internal Use of Priority in CML Implementations: As mentioned by [20] in describing the SML/NJ implementation of CML, a concept of prioritization has been previously considered in selective communication [20]. There, the principal goal is to maintain fairness and responsiveness. To achieve this goal, [20] proposes internally prioritizing events that have been frequently passed over in previous selective communications. We note that these priorities are never exposed to the programmer, and exist only as a performance optimization in the runtime. Even if exposed to user, this limited notion of priority only encompasses selective communication and ignores any consideration of the pairing communication. Our realization of priority, and the associated tiered priority scheme is significantly more powerful. This is both due to the exposure of priority to the programmer and the ability of our realization of priority to encompass information from both parties in a communication when considering the priority of an event.

Priority in ML: Recent work has looked at the introduction of priority to Standard ML [18].

To accomplish this, the system [18] propose, PriML, "rejects programs in which a high-priority may synchronize with a lower-priority one." Since all communication in CML is synchronous, in order for a high priority thread to communicate with a lower priority thread, they must synchronize. This is exactly the interaction that is explicitly disallowed by PriML.

7 Conclusion

This paper presents the design and implementation of *PrioCML*, an extension to Concurrent ML that introduces priority to synchronous messages passing. By leveraging a tiered-priority mechanism that considers both thread priority and event priority, *PrioCML* avoids potential priority inversions. Our evaluation shows that this mechanism can be realized to enable the adoption of priority with little effort and minimal performance penalties. The further work is to formalize the priority inversion and provide semantics for *PrioCML*.

Acknowledgment. This work is supported in part by National Science Foundation grants: CRI:1823230 and SHF:1749539. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the author(s) and do not necessarily reflect the views of the National Science Foundation.

References

1. The racket reference (2019). https://docs.racket-lang.org/reference/channel.html
2. Using binder IPC (2020). https://source.android.com/devices/architecture/hidl/binder-ipc
3. Buttazzo, G.: Hard Real-Time Computing Systems: Predictable Scheduling Algorithms and Applications. Springer, New York (2011). https://doi.org/10.1007/978-1-4614-0676-1
4. Chaudhuri, A.: A concurrent ML library in concurrent Haskell. SIGPLAN Not. **44**(9), 269–280 (2009). https://doi.org/10.1145/1631687.1596589
5. Demaine, E.: First class communication in MPI. In: Proceedings of the Second MPI Developers Conference, MPIDC 1996, p. 189. IEEE Computer Society, USA (1996)
6. Donnelly, K., Fluet, M.: Transactional events. J. Funct. Program. **18**(5–6), 649–706 (2008). https://doi.org/10.1017/S0956796808006916
7. Effinger-Dean, L., Kehrt, M., Grossman, D.: Transactional events for ML. In: Proceedings of the 13th ACM SIGPLAN International Conference on Functional Programming, ICFP 2008, pp. 103–114. Association for Computing Machinery, New York (2008). https://doi.org/10.1145/1411204.1411222
8. Flatt, M., Findler, R.B.: Kill-safe synchronization abstractions. SIGPLAN Not. **39**(6), 47–58 (2004). https://doi.org/10.1145/996893.996849
9. Fluet, M., Rainey, M., Reppy, J., Shaw, A.: Implicitly threaded parallelism in manticore. J. Funct. Program. **20**(5–6), 537–576 (2010). https://doi.org/10.1017/S0956796810000201
10. Gansner, E.R., Reppy, J.H.: A Multi-Threaded Higher-Order User Interface Toolkit (1993)
11. Gerrand, A.: Share memory by communicating (2010). https://blog.golang.org/share-memory-by-communicating
12. Haller, P., Odersky, M.: Scala actors: unifying thread-based and event-based programming. Theor. Comput. Sci. **410**, 202–220 (2009)
13. Armstrong, J., Virding, R., Wikström, C., Williams, M.: Concurrent Programming in Erlang, 2nd edn. Prentice-Hall (1996)
14. Klabnik, S., Nichols, C.: The rust programming language (2020). https://doc.rust-lang.org/book/ch16-02-message-passing.html
15. Lampson, B.W., Redell, D.D.: Experience with processes and monitors in Mesa. Commun. ACM **23**(2), 105–117 (1980). https://doi.org/10.1145/358818.358824
16. Milner, R., Tofte, M., Macqueen, D.: The Definition of Standard ML. MIT Press, Cambridge (1997)
17. Mueller, F.: A library implementation of posix threads under Unix. In: USENIX Winter (1993)
18. Muller, S.K., Acar, U.A., Harper, R.: Competitive parallelism: getting your priorities right. Proc. ACM Program. Lang. **2**(ICFP), 1–30 (2018). https://doi.org/10.1145/3236790
19. Reppy, J.H.: CML: a higher concurrent language. In: Proceedings of the ACM SIGPLAN 1991 Conference on Programming Language Design and Implementation, PLDI 1991, pp. 293–305. ACM, New York (1991). https://doi.org/10.1145/113445.113470
20. Reppy, J.H.: Concurrent Programming in ML, 1st edn. Cambridge University Press, New York (2007)

21. Russell, G.: Events in Haskell, and how to implement them. SIGPLAN Not. **36**(10), 157–168 (2001). https://doi.org/10.1145/507669.507655
22. Shipman, A.L.: System Programming with Standard ML (2002)
23. Swinehart, D.C., Zellweger, P.T., Hagmann, R.B.: The structure of cedar. In: Proceedings of the ACM SIGPLAN 85 Symposium on Language Issues in Programming Environments, SLIPE 1985, pp. 230–244. Association for Computing Machinery, New York (1985). https://doi.org/10.1145/800225.806844
24. Ziarek, L., Sivaramakrishnan, K., Jagannathan, S.: Composable asynchronous events. In: Proceedings of the 32nd ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2011, pp. 628–639. Association for Computing Machinery, New York (2011). https://doi.org/10.1145/1993498.1993572