Toward a parallel implementation of Concurrent ML

John Reppy and Yingqi Xiao

University of Chicago

Abstract. Concurrent ML (CML) is a high-level message-passing language that supports the construction of first-class synchronous abstractions called events. This mechanism has proven quite effective over the years and has been incorporated in a number of other languages. While CML provides a concurrent programming model, its implementation has always been limited to uniprocessors. This limitation is exploited in the implementation of the synchronization protocol that underlies the event mechanism, but with the advent of cheap parallel processing on the desktop (and laptop), it is time for Parallel CML.
We are pursuing such an implementation as part of the Manticore project. In this paper, we describe a parallel implementation of Asymmetric CML (ACML), which is a subset of CML that does not support output guards. We describe an optimistic concurrency protocol for implementing CML synchronization. This protocol has been implemented as part of the Manticore system.

1 Introduction

Concurrent ML (CML) [1, 2] is a statically-typed higher-order concurrent language that is embedded in Standard ML [3]. CML extends SML with synchronous message passing over typed channels and a powerful abstraction mechanism, called *first-class synchronous operations*, for building synchronization and communication abstractions. This mechanism allows programmers to encapsulate complicated communication and synchronization protocols as first-class abstractions, which encourages a modular style of programming where the actual underlying channels used to communicate with a given thread are hidden behind data and type abstraction. CML has been used successfully in a number of systems, including a multithreaded GUI toolkit [4], a distributed tuple-space implementation [2], and a system for implementing partitioned applications in a distributed setting [5]. The design of CML has inspired many implementations of CML-style concurrency primitives in other languages. These include other implementations of SML [6], other dialects of ML [7], other functional languages, such as HASKELL [8], SCHEME [9], and our own MOBY language [10], and other high-level languages, such as JAVA [11].

One major limitation of CML is that its implementation is *single-threaded* and cannot take advantage of multicore or multiprocessor systems.[1] We are incorporating the CML concurrency primitives into the functional parallel-programming language *Manticore* [12, 13], so this limitation must be addressed. In this paper, we take a major step in that direction by describing a parallel implementation of a subset of CML, which

[1] In fact, almost all of the existing implementations of events have this limitation.

```
type 'a event

val choose : ('a event * 'a event) -> 'a event
val wrap : 'a event * ('a -> 'b) -> 'b event
val guard : (unit -> 'a event) -> 'a event
val withNack : (unit event -> 'a event) -> 'a event

val sync : 'a event -> 'a

val never : 'a event
val always : 'a -> 'a event

type 'a chan
val recvEvt : 'a chan -> 'a event
val sendEvt : ('a chan * 'a) -> unit event
```

Fig. 1. The core features of CML

we call *Asymmetric Concurrent ML* (ACML). This subset of CML includes the full set of CML combinators, but does not support *output guards* (*i.e.*, send operations in a choice). We try to provide both an intuitive explanation of the synchronization protocol that underlies ACML, as well as enough of the nitty-gritty details to help other implementors. Because of space constraints, much of the implementation is omitted, but an extended version of this paper will be available as technical report [14].

2 A CML overview

Concurrent ML is a higher-order concurrent language that is embedded into Standard ML [1, 2]. It supports a rich set of concurrency mechanisms, but for purposes of this paper we focus on the core mechanisms of communication and events, which are shown in Figure 1. Communication in CML is based on synchronous message passing on typed channels. Because channels are synchronous, both the send and receive operations are blocking.

To support more complicated interactions, CML provides *event values*, which are first-class synchronous abstractions. Base events constructed by `sendEvt` and `recvEvt` describe simple communications on channels. There are also two special base-events: `never`, which is never enabled and `always`, which is always enabled for synchronization. These events can be combined into more complicated event values using the event combinators:

- Event wrappers (`wrap`) for post-synchronization actions.
- Event generators (`guard` and `withNack`) for pre-synchronization actions and cancellation (`withNack`).
- Choice (`choose`) for managing multiple communications. In CML, this combinator takes a list of events as its argument, but we restrict it to be a binary operator here. Choice of a list of events can be constructed using `choose` as a "cons" operator and `never` as "nil."

```
type 'a queue

val queue : unit -> 'a queue
val isEmptyQ : 'a queue -> bool
val enqueue : ('a queue * 'a) -> unit
val dequeue : 'a queue -> 'a option
```

Fig. 2. Specification of queue operations

To use an event value for synchronization, we apply the `sync` operator to it.

Event values are pure values similar to function values. When the `sync` operation is applied to an event value, a dynamic instance of the event is created, which we call a *synchronization event*. A single event value can be synchronized on many times, but each time involves a unique synchronization event.

In this paper, we describe an implementation ACML, which differs from the interface in Figure 1 in that it does not have the `sendEvt` event constructor. Instead, sending a message is supported using the function

```
val send : ('a chan * 'a) -> unit
```

This function is still blocking, but does not support sending a message in a choice context.

3 Preliminaries

We present our implementation using SML syntax with a few extensions. To streamline the presentation, we elide several aspects of the actual implementation, such as thread IDs and processor affinity.

3.1 Queues

Our implementation uses queues to track pending messages and waiting threads in channels. We omit the implementation details here, but give the interface to the queue operations that we use in Figure 2. These operations have the expected semantics.

3.2 Threads and thread scheduling

As in the uniprocessor implementation of CML, we use first-class continuations to implement threads and thread-scheduling. The continuation operations have the following specification:

```
type 'a cont
val callcc : ('a cont -> 'a) -> 'a
val throw  : 'a cont -> 'a -> 'b
```

We represent the state of a suspended thread as a `unit` continuation

```
type thread = unit cont
```

The interface to the scheduling system is represented by two atomic operations:

```
val enqueueRdy : thread -> unit
val dispatch : unit -> 'a
```

The first enqueues a ready thread in the scheduling queue and the second transfers control to the next ready thread in the scheduler queue.

3.3 Compare and swap

Our implementation also relies on the atomic *compare-and-swap* instruction. We use the following SML specification for this operation:

```
val cas  : ('a ref * 'a * 'a) -> 'a
```

Note that `cas` does not follow the SML equality semantics in that it performs pointer equality. With this operation, we build spinlocks:

```
val spinLock   : bool ref -> unit
val spinUnlock : bool ref -> unit
```

For purposes of this paper, we assume that threads are not preempted, so spinlocks are a reasonable locking mechanism.

4 A parallel implementation of PCML

Our parallel implementation is based on a core subset of the CML event operations, called *Primitive CML* (PCML). This subset has an event type with a minimal set of combinators, a condition-variable type used for signaling, and support for channels with input events. The signature of PCML is given in Figure 3. Note that unlike full CML (see Figure 1), there are no `guard` or `withNack` combinators. As we discuss in Section 5, these can be implemented on top of PCML.

4.1 The synchronization protocol

The heart of the implementation is the protocol for synchronization on a choice of events. This protocol is split between the `sync` operator and the base-event constructors (*e.g.*, `waitEvt` and `recvEvt`). Each base event is represented by a record of three functions: `pollFn`, which tests to see if the base-event is enabled (*e.g.*, there is a message waiting); `doFn`, which is used to synchronize on an enabled event; and `blockFn`, which is used to block the calling thread on the base event. In the single-threaded implementation of CML [15, 2], we rely heavily on the fact that `sync` is executed as an atomic operation. The single-threaded protocol is as follows:

1. Poll the base events in the choice to see if any of them are enabled. This phase is called the *polling phase*.

```
signature PRIM_CML =
  sig

  (* events *)
    type 'a evt
    val never : 'a evt
    val always : 'a -> 'a evt
    val choose : ('a evt * 'a evt) -> 'a evt
    val wrap : 'a evt * ('a -> 'b) -> 'b evt
    val sync : 'a evt -> 'a

  (* condition variables *)
    type  cvar
    val new : unit -> cvar
    val set : cvar -> unit
    val waitEvt : cvar -> unit evt

  (* channels *)
    type 'a chan
    val channel : unit -> 'a chan
    val recvEvt : 'a chan -> 'a evt
    val send : ('a chan * 'a) -> unit

  end
```

Fig. 3. Primitive CML

2. If one or more base events are enabled, pick one and synchronize on it using its `doFn`. This phase is called the *commit phase*.
3. If no base events are enabled we execute the *blocking phase*, which has the following steps:
   (a) Enqueue a continuation for the calling thread on each of the base events using its `blockFn`.
   (b) Switch to some other thread.
   (c) Eventually, some other thread will complete the synchronization.

We use the term *synchronization setup* for steps 1, 2, and 3(a) of this protocol.

Because the implementation of `sync` is atomic, the single-threaded implementation does not have to worry about the state of a base event changing between when we poll it and when we invoke the `doFn` or `blockFn` on it. In a parallel implementation, however, the global lock would be a bottleneck, so we must design a more complicated protocol. This design is further constrained by the fact that a given event may involve multiple occurrences of the same event. For example, the following code nondeterministicly tags the message received from `ch` with either `1` or `2`:

```
sync (choose (
  wrap (recvEvt ch, fn x => (1, x)),
  wrap (recvEvt ch, fn y => (2, y))
))
```

We must also avoid deadlock when multiple threads are simultaneously attempting communication on the same channel. For example, if thread P is executing

```
sync (choose (recvEvt ch1, recvEvt ch2))
```

at the same time that thread Q is executing

```
sync (choose (recvEvt ch2, recvEvt ch1))
```

we have a potential deadlock if the implementation of `sync` attempts to hold a lock on both channels simultaneously (*i.e.*, where P holds the lock on `ch1` and attempts to lock `ch2`, while Q holds the lock on `ch2` and attempts to lock `ch1`).

Our approach to avoiding these pitfalls is to use an *optimistic* protocol that does not hold a lock on more than one channel at a time and avoids locking whenever possible. The basic protocol has a similar structure to the sequential one described above, but it must deal with the fact that the state of a base event can change before the synchronization setup is complete. This fact means that the commit phase may fail and that the blocking phase may commit. The parallel synchronization protocol is as follows:

- The protocol starts with the polling phase, which is done in a *lock-free* way.
- The If one or more base events are enabled, pick one and *attempt* to synchronize on it using its `doFn`. This attempt may fail because of changes in the base-event state since the polling was done.
- If there are no enabled base events (or all attempts to synchronize failed), we enqueue a continuation for the calling thread on each of the base events using its `blockFn`. When blocking the thread on a particular base event, we may discover that synchronization is now possible, in which case we can synchronize immediately.

This design is guided by the goal of minimizing synchronization overhead and maximizing concurrency.

4.2 The PCML event type

A primitive-event value is represented as a binary tree, where the internal nodes represent choice and the leaves represent single synchronous operations. This canonical representation of events relies on the following equivalences:

$$\texttt{wrap}(\texttt{wrap}(ev, g), f) = \texttt{wrap}(ev, f \circ g)$$
$$\texttt{wrap}(\texttt{choose}(ev_1, ev_2), f_1) = \texttt{choose}(\texttt{wrap}(ev_1, f), \texttt{wrap}(ev_2, f))$$

We use this equivalence to maintain a canonical representation of events as trees in which the leaves are wrapped base-event values and the interior nodes are choice operators. Figure 4 illustrates the mapping from a nesting of `wrap` and `choose` combinators to its canonical representation.

Another issue that we must deal with is that another thread may attempt to complete the synchronization before setup is finished. We solve this problem by piggybacking on the mechanism used in the single-threaded implementation to do "garbage collection" of completed events. For each synchronization event, we allocate an event-state reference to hold the state of the synchronization.

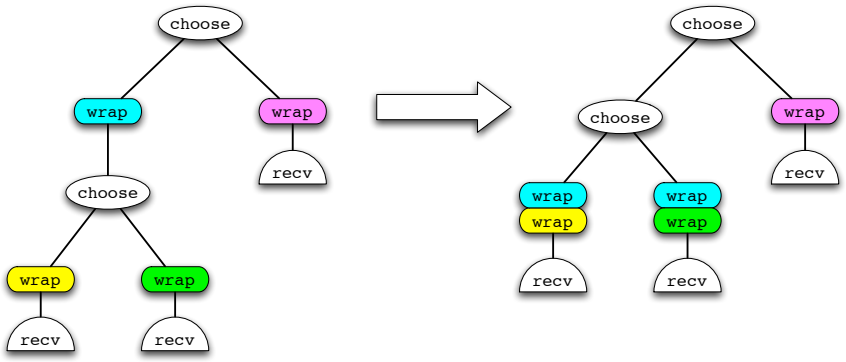

Fig. 4. The canonical-event transformation

```
datatype event_status = INIT | WAITING | SYNCHED
type event_state = event_status ref
```

The `INIT` state denotes that event setup is in progress, `WAITING` denotes that setup is complete and the event is available for synchronization, and `SYNCHED` denotes that the event has been synchronized on.

The canonical-event representation is implemented by the following datatype:

```
datatype 'a evt
  = BEVT of {
        pollFn : unit -> bool,
        doFn : 'a cont -> unit,
        blockFn : (event_state * 'a cont) -> unit
      }
  | CHOOSE of 'a evt * 'a evt
```

In this type, wrapped base events are represented by three functions: the `pollFn` is used to poll an event to test for its availability, the `doFn` is used to synchronize on an enabled event, and the `blockFn` is used to enqueue a suspended thread on the event. Both `doFn` and `blockFn` take resumption continuations as arguments. These continuations are used to return from the invoking `sync` operation. Note also that the `blockFn` takes a state flag as an argument. This flag is enqueued along with the resume continuation in the waiting queue maintained by the underlying communication object.

4.3 The PCML `sync` operation

The implementation of the `sync` operation is given in Figure 5. It is structured as three functions that correspond to the items in the protocol description above. Each of these functions does a walk over the tree representation of the event value to apply its operation to the base events at the leaves. The `poll` function polls each base event and returns a list of `doFn` functions for the base events that were enabled. The `doEvt` function, which is applied to this list, attempts to complete the synchronization

```
fun sync ev = callcc (fn resumeK => let
    (* optimistically poll the base events *)
      fun poll (BEVT{pollFn, doFn, blockFn}, enabled) =
            if pollFn()
              then doFn::enabled
              else enabled
        | poll (CHOOSE(ev1, ev2), enabled) =
            poll(ev2, poll(ev1, enabled))
    (* attempt to complete an enabled communication *)
      fun doEvt [] = blockThd()
        | doEvt (doFn::r) = (
            doFn resumeK;
          (* if we get here, that means that the
           * attempt failed, so try the next one
           *)
            doEvt r)
    (* record the calling thread's continuation in the
     * event waiting queues
     *)
      and blockThd () = let
            val flg = ref INIT
            fun block (BEVT{blockFn, ...}) =
                  blockFn(flg, resumeK)
              | block (CHOOSE(ev1, ev2)) = (
                  block ev1; block ev2)
            in
              block ev;
            (* if we get here, then setup is complete *)
              flg := WAITING;
              dispatch()
            end
      in
        doEvt (poll (ev, []))
      end)
```

Fig. 5. The primitive `sync` operation

on one of the base event's using its `doFn` function. Since the state of the base event might have changed since it was polled, it possible for the `doFn` to fail, in which case it returns. Otherwise, it will transfer control to the resume continuation. If `doEvt` is unable to complete the synchronization of any of the enabled events (or there were no enabled events), then it calls `blockThd`. This function allocates the state flag and then calls the `blockFn` of each of the base events to enqueue the resumption continuation. If the state of the base event has changed since polling (*i.e.*, it has become enabled), then the `blockFn` will complete the synchronization, otherwise it returns. If all of the `blockFn`s return, then the event's state is changed to `WAITING` and some other thread is dispatched.

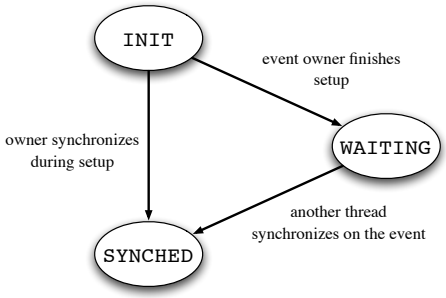

Fig. 6. The state-transitions of a synchronization event.

Because `sync` does not hold locks on the underlying communication objects, it is possible that some other thread may attempt to synchronize on one of the base events before `blockThd` has completed its work. Our policy is to only allow the owner thread of a synchronization event (*i.e.*, the caller of the `sync` operation) to change its state from `INIT`, as is shown in Figure 6. To implement this policy, non-owners use the following utility function to change the state:

```
fun claimEvent flg = (case cas(flg, WAITING, SYNCHED)
       of WAITING => true
        | INIT => claimEvent flg
        | SYNCHED => false
      (* end case *))
```

This function forces its caller to wait until setup is complete before being allowed to synchronize on the event. If the state is already `SYNCHED`, then it returns false.

An obvious simplification of this design would be to combine `pollFn` and `doFn` into a single function. There is a disadvantage of merging these two functions, however, which is that by polling all of the base events first, it is possible to impose an ordering on enabled events, such as priorities or to support fairness [2].

4.4 The PCML event combinators

The implementation of the primitive-event combinators is largely straightfoward, with the exception of `wrap`, which involves both the continuation hacking needed to hook in the wrapper function and event canonicalization. The implementation of `wrap` is given in Figure 7. When applied to a base-event value, we need to arrange for the wrapper function (`f`) to be applied to the values thrown to the resumption continuation by `doFn` and `pollFn`. When applied to a `CHOOSE` value, it pushes the wrapper down into both branches as described by the equivalence in Section 4.2.

4.5 PCML channels

The other half of the synchronization protocol is implemented in the base-event values for the communication objects. The representation of a channel consists of a spinlock, a

```
fun wrap (BEVT{pollFn, doFn, blockFn}, f) = BEVT{
        pollFn = pollFn,
        doFn = fn k => callcc (fn retK =>
          throw k (f (callcc (fn k' => (doFn k';
                                        throw retK ()))))),
        blockFn = fn (flg, k) => callcc (fn retK =>
          throw k (f (callcc (fn k' => (blockFn(flg, k');
                                        throw retK ())))))
      }
  | wrap (CHOOSE(ev1, ev2), f) =
      CHOOSE(wrap(ev1, f), wrap(ev2, f))
```

Fig. 7. The primitive `wrap` combinator

queue of blocked senders (with messages), and a queue of blocked receivers (with their owner's event state).

```
datatype 'a chan = Ch of {
    lock : bool ref,
    sendq : ('a * unit cont) queue,
    recvq : (event_state * 'a cont) queue
  }
```

The code for `recvEvt` is a non-trivial example of a base-event implementation and is given in Figure 8. The `pollFn` checks to see if the channel's `sendq` is empty. Since this operation only involves reading the state of the queue, it can be done without locking. Even if the results are erroneous because of conflicts with other threads, the fallback code in the `doFn` and `blockFn` will ensure correct behavior. The `doFn` is called when the `sendq` is expected to be nonempty. It locks the channel, removes an item from the `sendq` and then releases the lock. If the queue was empty (*i.e.*, `NONE` was returned), then the `doFn` returns. Otherwise, it enqueues the blocked sender and throws the message to the resume continuation of the `sync` operation. The `blockFn` is called when the `sendq` is expected to be empty. It also locks the channel and then checks the `sendq` in case its state has changed since polling. If there is an item available, then it is used to complete the synchronization. Otherwise, the resume continuation and event-state flag are enqueued in the channel's `recvq`.

The `send` operation on channels is given in Figure 9. The body of this function is a loop that examines the `recvq` for waiting events. If it finds one, then it completes the synchronization, otherwise it enqueues its resume continuation and message on the `sendq`.

5 Implementing full CML

In this section, we sketch how to build an implementation of the full set of CML event combinators from the `PRIM_CML` interface that we implemented in the previous section. The basic idea, which was suggested by Matthew Fluet [16], is to move the bookkeeping used to track negative acknowledgments out of the implementation of sync and

```
fun recvEvt (Ch{lock, sendq, recvq}) = let
      fun pollFn () = not(isEmptyQ(sendq))
      fun doFn k = let
            val _ = spinLock lock
            val item = dequeue sendq
            in
              spinUnlock lock;
              case item
               of NONE => ()
                | SOME(msg, sendK) => (
                    enqueueRdy sendK;
                    throw k msg)
              (* end case *)
            end
      fun blockFn (flg : event_state, k) = (
            spinLock lock;
          (* if we are lucky, a sender may have arrived
           * on the channel since we polled it.
           *)
            case dequeue sendq
             of SOME(msg, sendK) => (
                (* there is a matching send *)
                  spinUnlock lock;
                  flg := SYNCHED;
                  enqueueRdy sendK;
                  throw k msg)
              | NONE => (
                  enqueue (recvq, (flg, k));
                  spinUnlock lock)
            (* end case *))
      in
        BEVT{pollFn = pollFn, doFn = doFn, blockFn = blockFn}
      end
```

Fig. 8. The `recvEvt` event constructor

into guards and wrappers. Space does not permit a complete description of this layer, but we cover the highlights.

In this implementation, negative acknowledgments are signaled using the condition variables (cvars) provided by PCML. Since we must create these variables at synchronization time, we represent events as suspended computations (or *thunks*). The event type has the following definition:

```
datatype 'a event
  = E of (cvar list * (cvar list * 'a thunk) PCML.evt) thunk
```

where the `thunk` type is

```
type 'a thunk = unit -> 'a
```

```
fun send (Ch{lock, sendq, recvq}, msg) = callcc (fn sendK => let
      val _ = spinLock lock
      fun tryLp () = (case dequeue recvq
             of SOME(flg, recvK) =>
                (* there is a matching recv, but we must
                 * check to make sure that some other
                 * thread has not already claimed the event.
                 *)
                  if claimEvent flg
                    then ( (* we got it *)
                      spinUnlock lock;
                      enqueueRdy sendK;
                      throw recvK msg)
                    else (* someone else got the event *)
                      tryLp ()
              | NONE => (
                  enqueue (sendq, (msg, sendK));
                  spinUnlock lock;
                  dispatch ())
            (* end case *))
      in
        tryLp ()
      end)
```

Fig. 9. The `send` operation

The outermost thunk is a suspension used to delay the evaluation of guards until synchronization time. When evaluated, it produces a list of cvars and a primitive event. The cvars are used to signal the negative acknowledgments for the event. The primitive event, when synchronized, will yield a list of those cvars that need to be signaled and a thunk that is the suspended wrapper action for the event. With this representation, the `sync` operation is straightforward.

```
fun sync (E thunk) = let
      val (_, ev) = thunk()
      val (cvs, act) = PCML.sync ev
      in
        List.app PCML.set cvs;
        act()
      end
```

We start by evaluating the top-level thunk to get the primitive event value, which we then synchronize on. The result of synchronization will be a list of cvars that need to be signaled and the wrapper thunk. We signal the nacks by setting the cvars and then evaluate the wrapper thunk.

The two combinators that are at the heart of the bookkeeping for negative acknowledgments are `withNack` and `choose`. The former creates a new cvar when its thunk is evaluated. This cvar is passed as an argument to `withNack`'s argument and is added to the list of cvars for its result.

```
fun withNack f = let
      fun thunk () = let
            val nack = PCML.new()
            val E thunk' = f (baseEvt (PCML.waitEvt nack))
            val (cvs, ev) = thunk' ()
            in
              (nack::cvs, ev)
            end
      in
        E thunk
      end
```

The purpose of negative acknowledgments is to signal that some other event in a choice was chosen, which means that the `choose` combinator must associate the cvars of its left side with the synchronization result of its right side (and vice versa).

```
fun choose (E thunk1, E thunk2) = let
      fun thunk () = let
            val (cvs1, ev1) = thunk1()
            val (cvs2, ev2) = thunk2()
            in (
              cvs1 @ cvs2,
              PCML.choose (
                PCML.wrap(ev1, fn (cvs, th) => (cvs @ cvs2, th)),
                PCML.wrap(ev2, fn (cvs, th) => (cvs @ cvs1, th)))
            ) end
      in
        E thunk
      end
```

Space does not permit a description of the other mechanisms, but they can be found in a forthcoming technical report [14].

6 Related work

Various authors have described implementations of choice protocols using message passing as the underlying mechanism [17–20]. While these protocols could, in principle, be mapped to a shared-memory implementation, we believe that our approach is both simpler and more efficient.

Russell described a monadic implementation of CML-style events on top of Concurrent Haskell [8]. His implementation uses Concurrent Haskell's M-vars for concurrency control and he uses an ordered two-phase locking scheme to commit to communications. A key difference in his implementation is that choice is biased to the left, which means that he can commit immediately to an enabled event during the polling phase. This feature greatly simplifies the implementation, since it does not have to handle changes in event status between the polling phase and the commit phase. Russell's implementation did not support multiprocessors (because Concurrent Haskell did not support them at the time), but presumably would work on a parallel implementation of

Concurrent Haskell. Donnelly and Fluet have implemented a version of events that support transactions on top of Haskell's STM mechanism [16]. Their mechanism is quite powerful and, thus, their implementation is quite complicated.

In earlier work, we reported on specialized implementations of CML's channel operations that can be used when program analysis determines that it is safe [21]. Those specialized implementations fit into our framework and can be regarded as complementary.

7 Conclusion

We have described a new protocol for implementing Asymmetric CML on multiprocessors. This implementation consists of a primitive layer that provides basic synchronous operations, non-deterministic choice, and post-synchronization wrappers. This layer is implemented using a new optimistic-concurrency protocol. The full set of CML event combinators is then constructed on top of this primitive layer. One advantage of this architecture is that the more complicated upper layer does not directly use locks or thread scheduling operations.

We have implemented the primitive layer in the Manticore system using the Manticore compiler's BOM intermediate representation [13]. This implementation must also deal with preemption, which we do by locally masking preemption. Unfortunately, Manticore is not yet stable enough to be able to run meaningful performance tests, although we have been able to test the correctness of the implementation on an 8-way parallel system. We expect that the basic performance of the primitives will be good when channels are used to implement point-to-point communications (as is common), but the interesting question will be how they perform in a situation with many senders or receivers sharing a single channel. We plan to provide preliminary performance results in a forthcoming technical report [14].

In the longer term, we want to extend the PCML layer to support output guards (*i.e.*, `sendEvt`). In our protocol, adding this event constructor complicates the implementation in a couple of significant ways. First, it becomes possible to write code that has matching communications in a single choice, as in the following example:

```
sync (choose (
  recvEvt ch,
  wrap (sendEvt(ch, 1), fn () => 2)))
```

The implementation must detect such cases and avoid having a thread communicate with itself. The second problem is that committing to a synchronization will require atomically updating the states of two different synchronization events. Two-phase locking is one possible solution, but it requires introducing a linear order on synchronization events to avoid deadlock. Instead, we are exploring the use of implementation techniques from STM [22], but we have not worked out the details.

References

1. Reppy, J.H.: CML: A higher-order concurrent language. In: PLDI '91, New York, NY, ACM (June 1991) 293–305

2. Reppy, J.H.: Concurrent Programming in ML. Cambridge University Press, Cambridge, England (1999)
3. Milner, R., Tofte, M., Harper, R., MacQueen, D.: The Definition of Standard ML (Revised). The MIT Press, Cambridge, MA (1997)
4. Gansner, E.R., Reppy, J.H. In: A Multi-threaded Higher-order User Interface Toolkit. Volume 1 of Software Trends. John Wiley & Sons (1993) 61–80
5. Young, C., YN, L., Szymanski, T., Reppy, J., Pike, R., Narlikar, G., Mullender, S., Grosse, E.: Protium, an infrastructure for partitioned applications. In: HotOS-X. (January 2001) 41–46
6. MLton: Concurrent ML Available at http://mlton.org/ConcurrentML.
7. Leroy, X.: The Objective Caml System (release 3.00). (April 2000) Available from `http://caml.inria.fr`.
8. Russell, G.: Events in Haskell, and how to implement them. In: ICFP '01. (September 2001) 157–168
9. Flatt, M., Findler, R.B.: Kill-safe synchronization abstractions. In: PLDI '04. (June 2004) 47–58
10. Fisher, K., Reppy, J.: The design of a class mechanism for Moby. In: PLDI '99. (May 1999) 37–49
11. Demaine, E.D.: Higher-order concurrency in Java. In: WoTUG20. (April 1997) 34–47 Available from http://theory.csail.mit.edu/ edemaine/papers/WoTUG20/.
12. Fluet, M., Rainey, M., Reppy, J., Shaw, A., Xiao, Y.: Manticore: A heterogeneous parallel language. In: DAMP '07, New York, NY, ACM (January 2007) 37–44
13. Fluet, M., Ford, N., Rainey, M., Reppy, J., Shaw, A., Xiao, Y.: Status report: The Manticore project. In: ML '07, New York, NY, ACM (October 2007) 15–24
14. Reppy, J., Xiao, Y.: Toward parallel CML (extended version). Technical report, Department of Computer Science, University of Chicago *Forthcoming*.
15. Reppy, J.H.: First-class synchronous operations in Standard ML. Technical Report TR 89-1068, Dept. of CS, Cornell University (December 1989)
16. Donnelly, K., Fluet, M.: Transactional events. In: ICFP '06, New York, NY, ACM (2006) 124–135
17. Buckley, G.N., Silberschatz, A.: An effective implementation for the generalized input-output construct of CSP. ACM TOPLAS **5**(2) (April 1983) 223–235
18. Bornat, R.: A protocol for generalized occam. SP&E **16**(9) (September 1986) 783–799
19. Knabe, F.: A distributed protocol for channel-based communication with choice. Technical Report ECRC-92-16, European Computer-industry Research Center (October 1992)
20. Demaine, E.D.: Protocols for non-deterministic communication over synchronous channels. In: Proceedings of the 12th International Parallel Processing Symposium and 9th Symposium on Parallel and Distributed Processing (IPPS/SPDP'98). (March 1998) 24–30 Available from http://theory.csail.mit.edu/ edemaine/papers/IPPS98/.
21. Reppy, J., Xiao, Y.: Specialization of CML message-passing primitives. In: POPL '07, New York, NY, ACM (January 2007) 315–326
22. Shavit, N., Touitou, D.: Software transactional memory. In: PODC '95, New York, NY, ACM (1995) 204–213