Purdue University
Purdue e-Pubs

Department of Computer Science Technical Reports | Department of Computer Science

2010

Composable Asynchronous Events

Lukasz Ziarek
Purdue University, lziarek@cs.purdue.edu

KC Sivaramakrishnan
Purdue University, chandras@cs.purdue.edu

Suresh Jagannathan
Purdue University, suresh@cs.purdue.edu

Report Number:
10-008

Ziarek, Lukasz; Sivaramakrishnan, KC; and Jagannathan, Suresh, "Composable Asynchronous Events" (2010). *Department of Computer Science Technical Reports.* Paper 1730.
https://docs.lib.purdue.edu/cstech/1730

This document has been made available through Purdue e-Pubs, a service of the Purdue University Libraries. Please contact epubs@purdue.edu for additional information.

Composable Asynchronous Events

Lukasz Ziarek, KC Sivaramakrishnan, and Suresh Jagannathan
Purdue University
{lziarek, chandras, suresh}@cs.purdue.edu

Abstract

Asynchronous communication is an important feature of many concurrent systems. Central to any asynchronous protocol is the ability to split the creation of a communication action from its consumption. An important challenge to building expressive asynchronous communication abstractions is defining mechanisms that allow programmers to express *composable* post-creation and post-consumption behavior.

Abstractions like CML's synchronous events enable composable construction of synchronous communication protocols. Supporting similar functionality in an asynchronous setting leads to additional challenges because different threads of control are responsible for handling the action's creation and consumption.

In this paper, we present the design and rationale for *asynchronous events*, an abstraction that adapts CML's synchronous events to support composable asynchronous protocols. Asynchronous events enable seamless composition of asynchronous protocols and interoperate cleanly with existing CML primitives. We discuss the definition of a number of useful asynchronous abstractions that can be built on top of asynchronous events (e.g., composable callbacks) and provide a detailed case study of how asynchronous events can be used to substantially improve the definition and performance of I/O intensive server applications.

1. Introduction

Asynchronous communication is used extensively in concurrent programming. In message-passing functional languages like Erlang (1), F# (25; 19), JoCaml (16; 13), or CML (21), asynchrony is typically expressed through the use of asynchronous messaging primitives (such as asynchronous channels in JoCaml, mailboxes in CML, or Erlang's asynchronous `send` operation), or through the use of threads which perform the desired asynchronous actions synchronously.

Regardless of the technique chosen, an asynchronous communication protocol effectively splits the creation of the communication action from its consumption. Thus, an asynchronous action is *created* when a computation places a message on a channel (e.g., in the case of a message send), and is *consumed* when it is matched with its corresponding action (e.g., the send is paired with a receive); the creating thread may perform arbitrarily many actions before the message is consumed. In contrast, synchronous message-passing obligates the act of placing a message on a channel and the act of consuming it to take place as a single atomic action.

The challenge to building expressive asynchronous communication abstractions is defining mechanisms that allow programmers to express *composable post-creation* and *post-consumption* behavior. Even with synchronous message-passing, which conflates notions of creation and consumption, important functionality like selective communication confounds the use of simple λ-abstraction as a means of composability. This has led to the development of expressive abstractions like CML's first-class synchronous events (21) that enable construction of composable synchronous protocols.

Supporting composable post-creation and post-consumption in an asynchronous setting leads to substantial additional challenges because achieving such composability necessarily entails the involvement of two distinct threads of control – the thread that creates the action, and the thread that discharges it. In other words, just as λ-abstraction was found to be inadequate for supporting composable synchronous actions, threads offer a poor foundation upon which to build composable asynchronous ones.

To address this shortcoming, we introduce a new family of event combinators and primitives that explicitly deal with asynchronous communication. Our extensions enable seamless composition of asynchronous protocols, and interoperate with existing CML primitives. There are two key differences between an asynchronous event primitive and its synchronous counterpart: (1) the base asynchronous primitives we define (`aSendEvt` and `aRecvEvt`) expose both creation and consumption actions – this means that internal asynchrony (e.g., threads created by a synchronous event) is never hidden within events; (2) the asynchronous counterparts to combinators like `wrap` and `guard` allow composition of post-consumption and post-creation actions of an asynchronous event.

Contributions. In this paper, we explore the design and implementation of composable asynchronous events. Our motivation stems from the observation that just as synchronous events provide a solution to composable, synchronous message-passing that could not be easily accommodated using λ-abstraction and application, asynchronous events offer a solution for composable asynchronous message-passing that is not easily expressible using synchronous communication and explicit threading abstractions. We make the following contributions:

1. We present a comprehensive design for asynchronous events, and describe a family of combinators analogous to their synchronous variants available in CML. To the best of our knowledge, this is the first treatment to consider the meaningful integration of composable CML-style event abstractions with asynchronous functionality.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.

Copyright © ACM [to be supplied]. . . $5.00.

2. We provide implementations of useful asynchronous abstractions such as callbacks and mailboxes, along with a number of case studies extracted from realistic concurrent applications (e.g., a concurrent I/O library, concurrent file processing, etc.). Our abstractions operate over ordinary CML channels, enabling interoperability between synchronous and asynchronous protocols.
3. We discuss an implementation of asynchronous events that has been incorporated into MLton (18), and present a detailed benchmark study that shows how asynchronous events can help improve the expression and performance of a highly concurrent web-server. Our modifications, which mostly involved mechanical conversion of synchronous event abstractions to asynchronous ones, required changing less than 100 lines of code (the entire application itself is roughly 16KLOC), but resulted in over a 3.5X improvement in overall throughput and scalability.

The paper is organized as follows. In the next section, we provide motivation for asynchronous events. Design considerations are given in Sec. 3. In Sec. 4 we provide some background and describe asynchronous events as well as associated combinators and abstractions. Sec. 5 gives a formal operational semantics for asynchronous events. Implementation details are given in Sec. 6. We provide additional details through case studies given in Sec. 7. Performance results are discussed Sec. 8. Related work and conclusion are provided in Sec. 9 and Sec. 10, respectively.

2. Motivation

To illustrate the issues that asynchrony raises with respect to composability, consider a simple asynchronous message send abstraction written in CML that realizes asynchronous behavior by using a thread to encapsulate a synchronous action:

```
fun async-send-thread(f,g,c,v) =
  let val _ = spawn(fn() => sync (wrap(sendEvt(c,v),f)))
  in g()
  end
```

The function `async-send-thread` takes two functions, `f` and `g`, as well as a channel `c` and a value `v`, and creates a thread to place `v` on `c`. Post-creation actions are encapsulated by the continuation of the thread creating the asynchronous action (in this case `g`). A post-consumption action is defined using CML's `wrap` combinator; the complex event created from `wrap` when synchronized upon by `sync`, synchronously sends `v` on `c` and then evaluates `f`.

Suppose we want to leverage this simple abstraction to build more complex asynchronous protocols. Our motivating example is a simple resource manager (like a web-server) that consists of three modules: (1) a *connection manager* that listens for and accepts new connections, (2) a *processor* that processes and sends data based on input requests, and (3) an *orchestrator* that enables different kinds of interactions and protocols among: the clients, the processor, and the connection manager. If each established connection represents an independent group of actions (e.g., requests), we can leverage asynchrony to allow multiple requests to be serviced concurrently. Such a design would be particularly attractive on a multi-core system, for example.

One simple definition of an orchestrator is a procedure that chooses between the communication event defined by the processor (call it `processorEvt`) and connection manager (call it `managerEvt`). Even if the details of the communication protocol used by the connection manager and processor were visible to the orchestrator, the orchestrator would not be able to change their internal functionality since they are encapsulated with events. However, the orchestrator is free to augment the protocols via combinators.

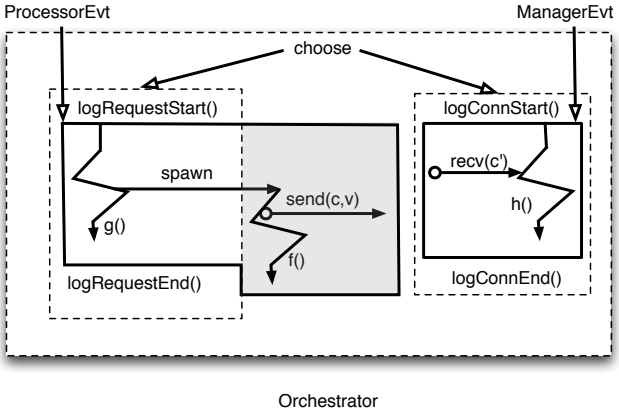

Figure 1. The figure shows the events built up by the three modules composing our abstract server. The events created by the processor and connection manager, depicted with thick squares, are opaque to the orchestrator and cannot be deconstructed. Notably, this means the orchestrator does not have the ability to extend the behavior of the thread created by `processorEvt` (shown in grey).

```
fun orchestrate(processorEvt, managerEvt) =
  sync(choose([processorEvt, managerEvt]))
```

The `choose` combinator non-deterministically selects whichever event is available to be synchronized against. If `processorEvt` is meant to perform its actions asynchronously, however, we need to modify `async-send-thread` so that it operates over events:

```
fun async-send-event(f,g,c,v) =
     wrap(alwaysEvt(),
          fn() => spawn(fn() =>
                        sync(wrap(sendEvt(c,v), f)));
                  g())
```

To create an event, we must first start with a base event and build the complex event (containing the spawn) through combinators. Since CML does not provide an event to create a thread, we opt to use `alwaysEvt`, which as its name suggests is always available for synchronization. The `alwaysEvt` does not have any communication side-effects, so any wrapped functions are immediately executed when the event is synchronized on. The function `async-send-event` is similar to `async-send-thread`, except that it returns an event that when synchronized upon will create a thread to perform the `send` asynchronously, executing `g` as a post creation action and `f` as a post consumption action (see Fig. 1).

We would hope to leverage `async-send-event` to have the processor module define useful asynchronous protocols. For example, it could supply as the post-consumption action (i.e., the argument bound to `f`), a procedure that waits for an acknowledgment indicating the sent data was received with no errors, attempting to resend otherwise. Similarly, it might supply as a post-creation action (i.e., the argument bound to `g`), clean-up or finalization code that closes a file after the data has been read.

Similarly, the connection manager can be written using synchronous receive actions. We can wrap a function `h` around the receipt of a connection to deal with post-processing actions required after the connection has been accepted, and to wait until the client closes the connection to clean up any resources the connection utilized (see Fig. 1).

```
fun h() = post processing; close connection;
fun managerEvent(c', h) =  wrap(recvEvt(c'), h)
```

Unfortunately, our design, while easy to express, does not facilitate the construction of *composable* asynchronous protocols. To see why, consider a modification to the orchestrator that incorporates logging information. We can wrap our processor protocol with a function that logs requests and our connection manager with one that logs connection details. The `guard` event combinator can be used to specify pre-synchronization actions. In this example these actions would be logging functions that record the start of a connection or request. [1]

```
fun orchestrate(processorEvt, managerEvt) =
  sync(choose([guard(fn () => logRequestStart();
                    wrap(processorEvt, logRequestEnd)),
               guard(fn () => logConnStart();
                    wrap(managerEvt, logConnEnd))]))
```

The logging functions can be simple counters that let the server keep track of concurrent connections, connection and request rates, as well as the number of outstanding requests.

This code, unfortunately, does not provide the functionality we desire. Since the processor handles its internal protocols asynchronously, wrapping the `logRequestEnd` around the `processorEvt` specifies an action that will occur in the main thread of control after the execution of `g`, the post-creation action (see Fig. 1). However, the request is only completed after the post-consumption action `f`, which is executed by the thread created internally by the event. Since this thread is *hidden* by the event, there is no way to extend it. Significantly, there is no guarantee that the request has been successfully serviced even after `g` completes. We could recover composability by either not spawning an internal thread in `async-send-event`, or weaving a protocol that required `g` to *wait* for the completion of `f`, effectively yielding synchronous behavior. Either approach would force `f` and `g` to be executed prior to `logRequestEnd`. Unfortunately, any such synchronous solution would not allow multiple requests to be processed concurrently.

The heart of the problem is a dichotomy in language abstractions; asynchrony is fundamentally expressed using distinct threads of control, yet composablity is achieved through event abstractions that are thread-unaware. The result is that CML events cannot be directly applied to build post-consumption actions for realizing composable asynchronous communication protocols.

Fig. 2 diagramatically shows how extensible post-creation and post-consumption actions can be leveraged to achieve our desired behavior. In Section 4.2, we show a modified implementation of the orchestrator using asynchronous event primitives and combinators that captures the behavior shown in the figure.

3. Design Considerations

There are three overarching goals of our design:

1. Asynchronous event combinators should permit uniform composition of pre/post creation and consumption actions. This means that protocols should be allowed to extend the behavior of an asynchronous event both with respect to the actions performed before and after the asynchronous event is created and consumed respectively.

[1] In CML, we could also use the `withNack` combinator to avoid firing both the `logRequestStart` and `logConnStart` during the choice.

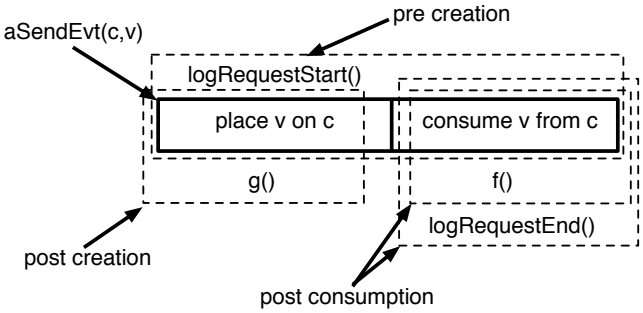

Figure 2. The figure shows how asynchronous events can be constructed from combinators on creation and consumption actions to alleviate the problems illustrated in Fig. 1. By making creation and consumption actions explicit in an asynchronous event's definition, we are able to specify `logRequestEnd` as a post consumption action and retain the ability to service multiple requests asynchronously.

2. Asynchronous events should provide sensible visibility and ordering guarantees. A post-creation action should execute with the guarantee that the asynchronous event it follows has been created (i.e., the action has been deposited on a channel), and the effects of consumed asynchronous events should be consistent with the order in which they were created.
3. Communication channels should be agnostic with respect to the kinds of events they handle. Thus, both synchronous and asynchronous events should be permitted to operate over the same sets of channels.

We elaborate on these points below.

Composable Pre/Post Creation and Consumption Actions: As we have seen, using synchronous events to encapsulate thread creation for the purpose of defining asynchronous protocols prevents composition of post-consumption actions. Once a thread is encapsulated within an event, as in the definition of `async-send-event`, we can no longer extend the asynchronous functionality of the event.

Visibility Guarantees for Post Creation Actions: The use of explicit threads for initiating asynchronous computation also fails to provide any post-creation guarantees. In our example encoding of `async-send-event`, `g` can make no guarantees at the point it commences evaluation that `v` has actually been deposited on `c`. This is because the encoding defines the creation point of the asynchronous action to be the point at which the thread is created, not the point where the `sendEvt` is synchronized on.

Ordering Guarantees on Asynchronous Actions: Using explicit threads to express asynchronous communication poses another complexity. Because threads are inherently unordered and agnostic to their payload, there is no transparent mechanism to enforce a sensible ordering relationship on the events they encapsulate. In the example below, a post-creation action creates a new thread to evaluate `h`, which in turn operates over `c`.

```
fun h () = ... ; sync(sendEvt(c, v'))

wrap(async-send-event(f,g,c,v),
     fn () => spawn h)
```

To have a rational semantics for `wrap`, we need to ensure that the original send of `v` in `async-send-event` evaluates before the send in `h`; unfortunately, using threads prevents us from

```
spawn     : (unit -> 'a) -> threadID
sendEvt   : 'a chan * 'a -> unit Event
recvEvt   : 'a chan -> 'a Event
never     : 'a Event
alwaysEvt : 'a -> 'a Event
sync      : 'a Event -> 'a
wrap      : 'a Event * ('a -> 'b) -> 'b Event
guard     : (unit -> 'a Event) -> 'a Event
choose    : 'a Event list -> 'a Event
```

Figure 3. CML event operators.

making such guarantees. Although one could envision using private channels to provide ordering between the thread created in `async-send-event` and the one created by the `wrap`, this would provide ordering between *all* actions executed within the threads instead of just the sends themselves, effectively serializing the threads. Additionally, it is not generally known if two threads will utilize the same channels, requiring programmers to be conservative in their use of private channels, fundamentally limiting asynchrony.

Specialized and Buffered Channels: Buffered channels (also called mailboxes) exist in CML. These abstractions do indeed provide asynchronous FIFO ordering of the operations they consume, but are not interchangeable with synchronous channels, necessitating careful delineation of asynchronous and synchronous behavior.

Putting it All Together. Although synchronous message passing alleviates the complexity of reasoning about arbitrary thread interleavings, and enables composable synchronous communication protocols, using threads to encode asynchrony unfortunately re-introduces these complexities. Our design equips asynchronous events with the following properties: *(*i) they are extensible both with respect to pre- and post-creation as well as pre- and post-consumption actions; *(*ii) they can operate over the same channels that synchronous events operate over, allowing both kinds of events to seamlessly co-exist; and, *(*iii) their visibility, ordering, and semantics is independent of the underlying runtime and scheduling infrastructure.

4. Asynchronous Events

Background: Concurrent ML (CML) (21) is a concurrent extension of Standard ML that utilizes synchronous message passing to enable the construction of synchronous communication protocols. Threads perform `send` and `recv` operations on typed channels; these operations block until a matching action on the same channel is performed by another thread.

CML also provides first-class synchronous *events* that abstract synchronous message-passing operations. An event value of type `'a event` when synchronized on yields a value of type `'a`. An event value represents a potential computation, with latent effect until a thread synchronizes upon it by calling `sync`. The following equivalences thus therefore hold: `send(c, v)` $\equiv$ `sync(sendEvt(c,v))` and `recv(c)` $\equiv$ `sync(recvEvt(c))`.

Besides `sendEvt` and `recvEvt`, there are other base events provided by CML: `alwaysEvt` which contains a value and is always available for synchronization and `never` which is an event that is never available for synchronization. Notably, thread creation is *not* encoded as an event – the thread `spawn` primitive simply takes a thunk to evaluate as a separate thread, and returns a thread identifier that allows access to the newly created thread's state.

Much of CML's expressive power derives from event combinators that construct complex event values from other events. We list some of these combinators in Fig. 3. The expression `wrap (ev, f)` creates an event that when synchronized on applies the result of synchronizing on event `ev` to function `f`. Conversely, `guard(f)` creates an event which when synchronized on evaluates `f()` to yield event `ev` and then synchronizes on `ev`. The `choose` event combinator takes a list of events and constructs an event value that represents the non-deterministic choice of the events in the list; for example, `choose[recvEvt(a),sendEvt(b,v)]` when synchronized on will either receive a unit value from channel `a`, or send value `v` on channel `b`.

Selective communication provided by choice motivates the need for first-class events. Composition of first-class functions prevents the expression of choice because function abstraction does not allow operators like `choose` from synchronizing on events that may be embedded within function.

4.1 Primitives

In order to provide an asynchronous protocol mechanism that adheres to the properties outlined above, we extend CML with the following two base events: `aSendEvt` and `aRecvEvt`, for creating an asynchronous send event and an asynchronous receive event respectively. The differences in their type signature from their synchronous counterparts reflect the fact that they split the creation and consumption of the communication action they define:

```
 sendEvt : 'a chan * 'a -> unit Event
aSendEvt : 'a chan * 'a -> (unit, unit) AEvent

 recvEvt : 'a chan -> 'a Event
aRecvEvt : 'a chan -> (unit, 'a) AEvent
```

An `AEvent` value is parametrized with respect to the type of the event's post-creation and post-consumption actions. In the case of `aSendEvt`, both actions are of type `unit`: when synchronized on, the event immediately returns a `unit` value, and the asynchronous action responsible for actually performing the send also yields `unit`. When synchronized on, an `aRecvEvt` also returns `unit`; the type of its post-consumption action is `'a` reflecting the type of value read from the channel.

As an example, consider Fig 2, where an event upon synchronization initiates an asynchronous action to send the value `v` on channel `c` and then executes `g`. When the send completes, `f` is first executed, and then `logRequestEnd`. In this case, the type of the computation would be:

(*return-type-of* `g`, *return-type-of* `logRequestEnd`) `AEvent`

We also introduce a new synchronization primitive, `aSync`, to synchronize asynchronous events. The `aSync` operation fires the computation encapsulated by the asynchronous event of type `('a, 'b) AEvent` and returns a value of type `'a`, corresponding to the return type of the event's post-*creation* action (see Fig. 4).

```
 sync    : 'a Event -> 'a
aSync    : ('a, 'b) AEvent -> 'a
```

Unlike their synchronous variants, asynchronous events do *not* block if no matching communication is present. For example, executing an asynchronous send event on an empty channel places the value being sent on the channel and then returns control to the executing thread (see Fig. 4(a)). In order to allow this non-blocking behavior, an *implicit* thread of control is created for the asynchronous event when the event is paired, or *consumed* as shown in Fig. 4(b). If a receiver is present on the channel, the asynchronous send event

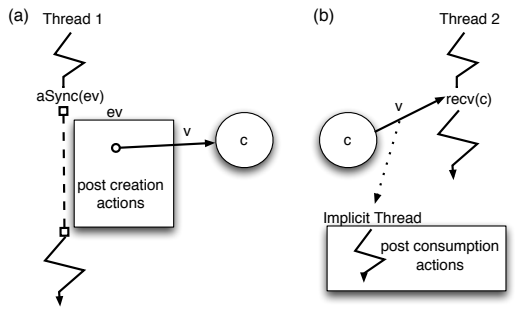

Figure 4. The figure shows a complex asynchronous event `ev`, built from a base `aSendEvt`, being executed by Thread 1. (a) When the event is synchronized via. `aSync`, the value `v` is placed on channel `c` and post-creation actions are executed. Afterwards, control returns to Thread 1. (b) When Thread 2 consumes the value `v` from channel `c`, an implicit thread of control is created to execute any post-consumption actions.

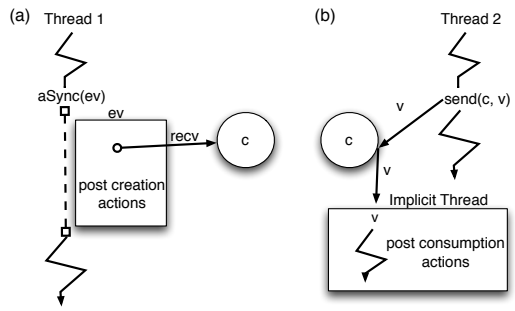

Figure 5. The figure shows a complex asynchronous event `ev`, built from a base `aRecvEvt`, being executed by Thread 1. (a) When the event is synchronized via `aSync`, the receive action is placed on channel `c` and post-creation actions are executed. Afterwards, control returns to Thread 1. (b) When Thread 2 sends the value `v` to channel `c`, an implicit thread of control is created to execute any post-consumption actions passing `v` as the argument.

behaves similarly to a synchronous event; it passes the value to the receiver. However, it still creates a new implicit thread of control if there are any post-consumption actions to be executed.

Similarly, the synchronization of an asynchronous receive event does not yield the value received; instead, it simply enqueues the receiving action on the channel. Therefore, the thread which synchronizes on an asynchronous receive always gets the value `unit`, even if a matching send exists (see Fig. 5(a)). The actual value consumed by the asynchronous receive can be passed back to the thread which synchronized on the event through the use of combinators that process post-consumption actions (see Fig. 5(b)). To illustrate, consider the two functions `f` and `af` shown below:

```
fun  f () =
     (spawn (fn () => sync (sendEvt(c, v)));
      sync (sendEvt(c, v'));
      sync (recvEvt(c)))

fun af () =
     (spawn (fn () => sync (sendEvt(c, v)));
      aSync (aSendEvt(c, v'));
      sync (recvEvt(c)))
```

The function `f`, if executed in a system with no other threads will always block because there is no recipient available for the send of `v'` on channel `c`. On the other hand, suppose there was another thread willing to accept communication on channel `c`. In this case, the only possible value that `f` could receive from `c` is `v`. This is because the receive will only occur after the value `v'` is consumed from the channel. Notice that if the spawned thread enqueues `v` on the channel before `v'`, the function `f` will block even if another thread is willing to receive a value from the channel, since a function cannot synchronize with itself.

The function `af`, on the other hand will never block. The receive may see either the value `v` or `v'` since the asynchronous send event *only* asserts that the value `v'` has been placed on the channel and *not* that it has been consumed. Consider the following refinement of `af`:

```
fun af' () =
     (aSync (aSendEvt(c, v'));
      spawn (fn () => sync (sendEvt(c, v)));
      sync (recvEvt(c)))
```

Assuming no other threads exist that read from `c`, the receive in `af'` can only witness the value `v'`. Although the spawn occurs before the synchronous receive, the channel `c` is guaranteed to contain the value `v'` prior to `v`. While asynchronous events do not block, they still enforce *ordering* constraints that reflect the order in which they were created, based on their channels. This distinguishes their behavior from our initial definition of asynchronous events that explicitly encoded asynchronous behavior in terms of threads.

4.2 Combinators

In CML, the wrap combinator allows for the specification of a post-synchronization action. Once the event is completed the function wrapping the event is evaluated. For asynchronous events, this means the wrapped function is executed after the action the event encodes is *placed* on the channel and not necessarily after that action is consumed.

```
sWrap : ('a, 'b) AEvent * ('a -> 'c) ->  ('c, 'b) AEvent
aWrap : ('a, 'b) AEvent * ('b -> 'c) ->  ('a, 'c) AEvent
```

To allow for the specification of both post-creation and post-consumption actions for asynchronous events, we introduce two new combinators: `sWrap` and `aWrap`. `sWrap` is used to specify post-creation actions. The combinator `aWrap`, on the other hand, is used to express post-consumption actions. We can apply `sWrap` and `aWrap` to an asynchronous event in any order.

```
sWrap(aWrap(e, f) g) ≡ aWrap(sWrap(e, g), f)
```

We can use `sWrap` and `aWrap` to encode a composable variant of `async-send-event`. We create a base asynchronous send event to send `v` on `c` and use `sWrap` and `aWrap` to specify `g` and `f` as a post-creation action and a post-consumption action respectively:

```
a-send-event(f,g,c,v) = aWrap(sWrap(aSendEvt(c,v),g), f)
```

Since post-creation actions have been studied in CML extensively (they act as post-synchronization actions in a synchronous context), we focus our discussion on `aWrap` and the specification of post-consumption actions. Consider the following program fragment:

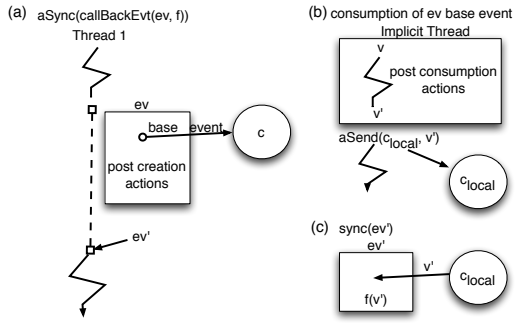

Figure 6. The figure shows a callback event constructed from a complex asynchronous event `ev` and a callback function `f`. (a) When the callback event is synchronized via `aSync`, the action associated with the event `ev` is placed on channel `c` and post-creation actions are executed. A new event `ev'` is created and passed to Thread 1. (b) An implicit thread of control is created after the base event of `ev` is consumed. Post-consumption actions are executed passing `v`, the result of consuming the base event for `ev`, as an argument. The result of the post-consumption actions, `v'` is sent on c_{local}. (c) When `ev'` is synchronized upon, `f` is called with `v'`.

```
fun f() =
  let val c_local = channel()
  in aSync (aWrap(aSendEvt(c, v),fn () => send(c_local, ())));
     g();
     recv(c_local);
     h()
  end
```

The function `f` first allocates a local channel c_{local} and then executes an asynchronous send `aWrap`-ed with a function that sends on the local channel. The function `f` then proceeds to execute functions `g` and `h` with a receive on the local channel between the two function calls. We use the `aWrap` primitive to encode a simple *barrier* based on the consumption of `v`. We are guaranteed that `h` executes in a context in which `v` has been consumed. The function `g`, on the other hand, can make no assumptions on the consumption of `v`. However, `g` is guaranteed that `v` is on the channel. Therefore, if `g` consumes values from `c`, it can witness `v` and, similarly, if it places values on the channel, it is guaranteed that `v` will be consumed *prior* to the values `g` produces. Note that `v` could have been consumed *prior* to `g`'s evaluation.

If the same code was written with a synchronous wrap, we would have no guarantee about the consumption of `v` from the channel. In fact, the code would block, as the send encapsulated by the wrap would be executed by the *same* thread of control executing `f`. Thus, the asynchronous event implicitly creates a new evaluation context and a new thread of control; the wrapping function is evaluated in this context, not the thread which performed the synchronization.

We can now encode a very basic call-back mechanism using `aWrap`. The code shown below performs an asynchronous receive and passes the result of the receive to its wrapped function. The value received asynchronously is passed as an argument to `h` by sending on the channel c_{local}.

```
let val c_local = channel()
in aSync (aWrap(aRecvEvt(c), fn x => send(c_local, x)));
   ...
   h(recv(c_local))
end
```

Although this implementation suffices as a basic call-back, it is not particularly abstract, and cannot be composed with other asynchronous events. We can create an abstract callback mechanism using both `sWrap` and `aWrap` around a base event.

```
callbackEvt : ('a, 'c) AEvent * ('c -> 'b) ->
              ('b Event, 'c) AEvent

fun callbackEvt(ev, f) =
  let c_local = channel()
  in sWrap(aWrap(ev,
                 fn x => aSync(aSendEvt(c_local, x)); x),
           fn _ => wrap(recvEvt(c_local), f))
  end
```

If `ev` contains post-creation actions when the callback event is synchronized on, they are executed, followed by execution of the `sWrap` as shown in Fig. 6(a). It returns a new event (call it `ev'`), which when synchronized on will first receive on the local channel and then apply the function `f` to the value it receives from the local channel. Synchronizing on this event will block until the event `ev` is consumed. Once `ev` is consumed, its post-consumption actions are executed in a new thread of control since `ev` is asynchronous (see Fig. 6(b)). The body of the `aWrap`-ed function simply sends the result of synchronizing on `ev` (call it `v'`) on the local channel and then passes the value `v'` to any further post-consumption actions. This is done asynchronously because the complex event returned by `callbackEvt` can be further extended with additional post consumption actions. Those actions should not be blocked if there is no thread willing to synchronize on `ev'`. Synchronizing on a callback event, thus, executes the base event associated with `ev` *and* creates a new event as a post-creation action, which when synchronized on, executes the callback function synchronously, returning the result of the callback.

We can think of the difference between a callback and an `aWrap` of an asynchronous event in terms of the thread of control which executes them. Both specify a *post*-consumption action for the asynchronous event, but the callback, when synchronized upon, is executed potentially by an *arbitrary thread* whereas the `aWrap` is *always* executed in the implicit thread created when the asynchronous event is consumed. Another difference is that the callback can be *postponed* and only executes when two conditions are satisfied: *(*i) the asynchronous event has completed and *(*ii) the callback is synchronized on. An `aWrap` returns once it has been synchronized on, and does not need to wait for other asynchronous events or post-consumption actions it encapsulates to complete.

A guard of an asynchronous event behaves much the same as a guard of a synchronous event does; it specifies pre-synchronization actions:

```
aGuard : (unit -> ('a, 'b) AEvent) -> ('a, 'b) AEvent
```

To see how we might use asynchronous guards, notice that our definition of `callbackEvt` has the unfortunate drawback that it allocates a new local channel regardless of whether or not the event is ever synchronized upon. The code below uses an `aGuard` to specify the allocation of the local channel only when the event is synchronized on:

```
fun callbackEvt(ev, f) =
  aGuard(fn () =>
    let c_local = channel()
    in sWrap(aWrap(ev,
                   fn x => aSync(aSendEvt(c_local, x));x),
             fn _ => wrap(recvEvt(c_local), f))
    end)
```

One of the most powerful combinators provided by CML is a non-deterministic choice over events. The combinator `choose` picks

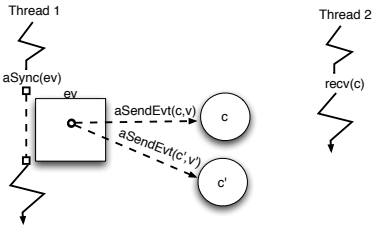

Figure 7. The figure shows Thread 1 synchronizing on a complex asynchronous event `ev`, built from a choice between two base asynchronous send events; one sending `v` on channel `c` and the other `v'` on `c'`. Thread 2 is willing to receive from channel `c`.

an *active* event from a list of events. If no events are active, it waits until one becomes active. An active event is an event which is available for synchronization. We define an asynchronous version of the choice combinator, `aChoose`, that operates over asynchronous events. Since asynchronous events are non-blocking, all events in the list are considered *active*. Therefore, the asynchronous choice always non-deterministically chooses between the list of available asynchronous events. We also provide a synchronous version of the asynchronous choice, `sChoose`, which blocks until one of the asynchronous base events has been consumed. Post-creation actions are not executed, until the choice has been made. [2]

```
 choose : 'a Event list -> 'a Event
aChoose : ('a, 'b) AEvent list -> ('a, 'b) AEvent
sChoose : ('a, 'b) AEvent list -> ('a, 'b) AEvent
```

To illustrate the difference between `aChoose` and `sChoose`, consider a complex event `ev` defined as follows:

```
val ev = aChoose[aSendEvt(c, v), aSendEvt(c',v')]
```

If there exists a thread only willing to receive from channel `c` as shown in Fig. 7, `aChoose` will nonetheless, with equal probability execute the asynchronous send on `c` and `c'`. Therefore, `aChoose` behaves in much the same way as `choose` would selecting between a list of events encoded by `async-send-event` from Sec. 1. This occurs because the base event for `async-send-event` is an `alwaysEvt` and as such is always available for synchronization. However, if we redefined `ev` to utilize `sChoose` instead, the behavior of the choice changes:

```
val ev = sChoose[aSendEvt(c, v), aSendEvt(c',v')]
```

Since `sChoose` blocks until one of the base asynchronous events is satisfiable, if there is only a thread willing to accept communication on `c` (as in Fig. 7), the choice will only select the event encoding the asynchronous send on `c`.

We have thus far provided a mechanism to choose between sets of synchronous events and sets of asynchronous events. However, we would like to allow programmers to choose between both synchronous and asynchronous events. Currently, their different type structure would prevent such a formulation. Notice, however, that an asynchronous event with type `('a, 'b) AEvent` and a synchronous event with type `'a Event` both yield `'a` in the thread which synchronizes on them. Therefore, it is sensible to allow choice to operate over *both* asynchronous and synchronous events provided the type of the asynchronous event's post-creation action is the same as the type encapsulated by the synchronous event. To facilitate this interoperability, we provide combinators to transform asynchronous events types to synchronous events types and *vice-versa*:

```
aTrans : ('a, 'b) AEvent -> 'a Event
sTrans : 'a Event -> (unit, 'a) AEvent
```

The `aTrans` combinator takes an asynchronous event and creates a synchronous version by *dropping* the asynchronous portion of the event. As a result, we can no longer specify post-consumption actions for the event. However, we can still apply `wrap` to specify post-creation actions to the resulting synchronous portion exposed by the `'a Event`. Asynchronous events that have been transformed and are part of a larger `choose` event are only selected if their base event is satisfiable. Therefore, the following equivalence holds for two asynchronous events, `aEvt1` and `aEvt2`:

```
choose[aTrans(aEvt1), aTrans(aEvt2)] ≡ sChoose[aEvt1, aEvt2]
```

The `sTrans` combinator takes a synchronous event and changes it into an asynchronous event with no post-creation actions. The wrapped computation of the original event occurs now as a post-consumption action. We can utilize `sTrans` to encode asynchronous versions of `alwaysEvt` and `never` from their synchronous counterparts.

```
aAlwaysEvt : 'a -> (unit, 'a) AEvent
aNever  : (unit, 'a) AEvent

aAlwaysEvt(v) =  sTrans alwaysEvt(v)
aNever = sTrans never
```

Armed with asynchronous events and the combinators discussed above, we can now implement a composable orchestrator module from our simple abstract server example given in Sec. 1. We use `aGuard` to specify pre-creation actions, `aWrap` for asynchronous post-consumption actions, and `aTrans` to hide the post-consumption actions. This allows us to freely choose between the asynchronous `processorEvt` and the synchronous `managerEvt`.

```
fun orchestrate(processorEvt, managerEvt) =
  sync(choose([aTrans
               aGuard(fn () => logRequestStart();
                      aWrap(processorEvt, logRequestEnd)),
               guard(fn () => logConnStart();
                     wrap(managerEvt, logConnEnd))]))
```

4.3 Extending Mailboxes

Mailboxes, or buffered asynchronous channels, are provided by the core CML implementation. Mailboxes are a specialized channel that supports asynchronous sends and synchronous receives. However, mailboxes are not built directly on top of CML channels, requiring a specialized structure, on the order of a 140 lines of CML code, to support asynchronous sends.

Asynchronous events provide the necessary components from which a mailbox structure can be defined, allowing the construction of mailboxes from regular CML channels, and providing a facility to define *asynchronous* send events on the mailbox. Having an asynchronous send event operation defined for mailboxes allows for their use in selective communication. Additionally, asynchronous events now provide the ability for programmers to specify post-creation *and* post-consumption actions. The asynchronous send operator and asynchronous send event can be defined as follows:

```
fun send(mailbox, value) =
  CML.aSync(CML.aSendEvt(mailbox, value))

fun sendEvt(mailbox, value) =
  CML.aSendEvt(mailbox, value)
```

[2] This behavior is equivalent to a scheduler not executing the thread which created the asynchronous action until it has been consumed.

The synchronous receive and receive event are expressed in terms of regular CML primitives. This highlights the interoperability of asynchronous events with their synchronous counterparts and provides programmers with a rich interface of combinators to utilizes with mailboxes.

5. Semantics

Our semantics (see Fig 8 and Fig. 9) is defined in terms of a core call-by-value functional language with threading and communication primitives. Communication between threads is achieved using synchronous channels and events. Our language extends a synchronous-event core language with asynchronous constructs. For perspicuity, the language omits many useful event combinators such as `chooseEvt` and `withNack`, since they raise no interesting semantic issues with respect to asynchronous communication. References are also omitted for this reason. We focus in this section on core synchronous and asynchronous events and their associated `wrap` and `guard` combinators. The semantics for other combinators (such as choice) are given in the appendix.[3]

In our syntax (see Fig. 8), v ranges over values, c over channel references, γ over constants, e over expressions, and `t` over thread identifiers. We use an event context ($\varepsilon[]$) to demarcate event expressions that are built from combinators such as guard and wrap as well as their asynchronous counterparts. The semantics also includes a new expression form, $\{e_1, e_2\}$ to denote asynchronous communication actions; the expression e_1 corresponds to the creation (and post-creation) of an asynchronous event , while e_2 corresponds to the consumption (and post-consumption) of an asynchronous event.

A program state consists of a set of threads ($\overline{T}$), a communication map (Δ), and a channel map ($\mathcal{C}$). The communication map is used to track the state of an asynchronous action, while the channel map records the state of channels with respect to waiting (blocked) actions. Evaluation is specified via a relation ($\rightarrow$) that maps one program state to another program state. Evaluation rules are applied up to commutativity of parallel composition ($\|$).

Encoding Communication: A communication action is split into two message parts: one corresponding to a sender and the other to a receiver. A send message part (m) is, in turn, composed of two actions: a *send act* ($\texttt{sendAct}_m(c,v)$) and a *send wait* ($\texttt{sendWait}_m$). The *send act* places the value (v) on the channel (c), while the *send wait* blocks until the value has been consumed off of the channel. The message identifier m is used to correctly pair the "act" and "wait" pieces. Similarly, a receive message part is composed of a *receive act* ($\texttt{recvAct}_m(c)$) and a *receive wait* ($\texttt{recvWait}_m$) action. A *receive wait* action behaves as its send counterpart. A *receive act* removes a value from the channel. We can think of computations occurring after an act as post-creation actions and those occurring after a wait as post-consumption actions.

Splitting a communication message part into an "act" and a "wait" allows for the expression of many types of message passing primitives. For instance, a traditional synchronous send is simply the sequencing a *send act* followed by a *send wait*: $\texttt{sendAct}_m(c,v)$; $\texttt{sendWait}_m$. This encoding immediately causes the thread executing the operation to block after the value has been deposited on a channel, unless there is a matching *receive act* currently available. A synchronous receive is encoded in much the same manner.

[3] Defining choice requires bookkeeping to track which event from a list of possible event is satisfied.

We use the global communication map (Δ) to track act and wait actions for a given message. When a new message is created (Rules SENDEVENT, RECVEVENT, ASENDEVENT, and ARECVEVENT), the communication map is augmented to reflect this fact. If $\Delta(m) = \perp$, it means a message m has been created by a base event, but a corresponding *send act* or *receive act* has not yet occurred on m. Once such an act occurs, the map is updated to reflect the value yielded by the act (see Rule MESSAGE) through an auxiliary relation ($\diamond$). Notice, when a send act occurs the communication map will hold a binding to unit for the corresponding message, but when a receive act occurs the communication map binds the corresponding message to the value received. The values stored in the communication map are passed to the wait actions corresponding to the message (Rules SEND WAIT and RECV WAIT).

Events and Combinators: There are four rules for creating base events, (SENDEVENT) and (RECVEVENT) for synchronous events, and (ASENDEVENT) and (ARECVEVENT) for their asynchronous counterparts. From base act and wait actions, we define asynchronous events (*e.g.*, $\varepsilon[\{\texttt{sendAct}_m(c,v), \texttt{sendWait}_m\}]$). The first component of an asynchronous event is executed in the thread in which the expression evaluates, and is the target of synchronization (`sync`), while the second component defines the actual asynchronous computation.

Complex events are built from combinators such as those defined in Sec. 4.2. The rule (WRAP) takes a synchronous event and provides a post-consumption action. It binds the result of the synchronous event to the argument of the wrapped function. The rule (SWRAP) takes an asynchronous event and provides a post-creation action. It binds the result of the first component of the asynchronous event (e) to the argument of the wrapped function. The rule (AWRAP), however, specifies a post-consumption action by wrapping the second component of the event (e') to the argument of the wrapped function. The rules (GUARD) and (AGUARD) provide pre-creation actions; both rules take a function as an argument which returns an event – (GUARD) returns a synchronous event and (AGUARD) an asynchronous one.

We write ($\texttt{wrap}(\varepsilon[e], \lambda x.e')$) and ($\texttt{guard}(\lambda x.e')$) to wrap the event ($\varepsilon[e]$) and create a guard function respectively. Similarly, we denote wrapping post creation actions by ($\texttt{sWrap}(\varepsilon[\{e_1, e_2\}], \lambda x.e')$) and post consumption actions by ($\texttt{aWrap}(\varepsilon[\{e_1, e_2\}], \lambda x.e')$) in rules (SWRAP and AWRAP).

Event Evaluation: Events are deconstructed by the `sync` operator in rule (SYNC). It strips the event context ($\varepsilon[]$) and triggers the evaluation of the internal expression. The expression can be an arbitrary one, built from base actions (such as act and wait over message sends and receives) and combinators.

The rule (ASYNC EVAL) defines the evaluation of an asynchronous event. The asynchronous portion of the event is wrapped in a new thread of control and placed in the regular pool of threads. The newly created thread, however, will not be able to be evaluated further as the first expression to be evaluated will always be a "wait" for the corresponding act for the base event comprising the complex asynchronous event (as defined by the base event and combinator rules).

Communication and Ordering: There are five rules for communicating over channels (SEND MATCH, SEND BLOCK, RECV MATCH, RECV BLOCK, and EMPTY BLOCK). The channel map ($\mathcal{C}$) encodes abstract channel states mapping a channel to a sequence of actions. This sequence encodes a FIFO queue and provides ordering between actions on the channel. The channel will

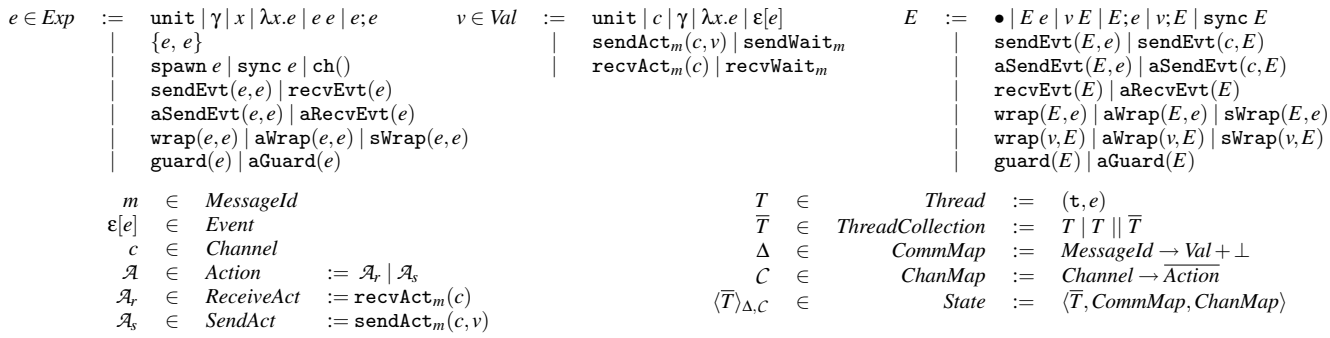

Figure 8. A core language for asynchronous events.

have either a sequence of send acts ($\mathcal{A}_s$) or receive acts ($\mathcal{A}_r$), but never both at the same time. This is because if there are, for example, send acts enqueued on it, a receive action will immediately match the send, instead of needing to be enqueued and *vice versa* (rules SEND MATCH and RECV MATCH). If the channel is empty, either a send act or a recv act will be enqueued (see rule EMPTY BLOCK). If a channel already has send acts enqueued on it, any thread wishing to send on the channel will enqueue its act and *vice versa*. Notice after enqueueing its act, a thread can proceed with its evaluation. Rules (SEND BLOCK) and (RECV BLOCK) enqueue the act on the channel.

Ordering for asynchronous acts and their post consumption actions as well as blocking of synchronous events is achieved by rules (SEND WAIT) and (RECV WAIT). Both rules block the evaluation of a thread until the corresponding act has been evaluated. In the case of synchronous events, this thread is the one that initiated the act; in the case of an asynchronous event, the thread that creates the act is different from the one that waits on it, and the blocking rules only block the implicitly created thread.

6. Implementation

We have implemented asynchronous events in a multi-core aware implementation of MLton (18), a whole-program optimizing compiler for Standard ML (SML) (17). MLton has the ability to compile SML programs to both native code as well as C; the results described in this paper are based on code compiled to C and then compiled to native by gcc version 4.1.2. Our implementation closely follows the semantics given in Section 5. To allow synchronous and asynchronous events to seamlessly co-exist involved implementing a unified framework for events, that is agnostic to the underlying channel, scheduler and runtime implementation. The implementation of asynchronous events is thus composed of four parts: *(i)* the definition of base event values, *(ii)* the internal synchronization protocols for base events, *(iii)* synchronization protocols for choice, and *(iv)* the definition of various combinators. The implementation is roughly 4KLOC lines of ML code.

7. Case Study: A Parallel Web-server

`Swerve` (18) is an open-source third-party web-server wholly written in CML and is roughly 16KLOC of CML code. The server is composed of five separate interacting modules. Communication protocols that govern the interactions between the modules that comprise Swerve makes extensive use of CML events and message-passing.

To motivate the use of asynchronous events, we consider the interactions of four of `Swerve`'s modules: the `Listener`, the `File Processor`, the `Network Processor` and the `Timeout Manager`. The `Listener` module receives incoming `HTTP` requests and delegates file serving requirements to concurrently executing processing threads. For each new connection, a new listener is spawned; thus, each connection has one main governing entity. The `File Processor` module handles access to the underlying file system. Each file that will be hosted is read by a file processor thread that chunks the file and sends it via message-passing to the `Network Processor`. The `Network Processor`, like the `File Processor` handles access to the network. The `File Processor` and `Network Processor` execute in lock-step, requiring the `Network Processor` to have completed sending a chunk before the next one is read from disk. Timeouts are processed by the `Timeout Manager` through the use of timed events.

We briefly touch upon three aspects of Swerve's design that were amenable to using asynchronous events. In Section 8, we show how these changes lead to substantially improvement in throughput and scalability.

Lock-step File and Network I/O. Swerve was engineered assuming lock-step file and network I/O. While adequate when under low request loads, this design has poor scalability characteristics. This is because (a) file descriptors, a bounded resource, can remain open for potentially long periods of time, as many different requests are multiplexed among a set of compute threads, and (b) for a given request, a file chunk is read only after the network processor has sent the previous chunk. Asynchronous events can be used to alleviate both bottlenecks.

To solve the problem of lockstep transfer of file chunks, we might consider using simple asynchronous sends. However, Swerve was engineered such that the file processor was responsible for detecting timeouts. If a timeout occurs, the file processor sends a notification to the network processor on the same channel used to send file chunks. Therefore, if asynchrony was used to simply buffer the file chunks, a timeout would not be detected by the network processor until *all* the chunks were processed. Changing the communication structure to send timeout notifications on a separate channel would entail substantial structural modifications to the code base.

The code shown below is a simplified version of the file processing module modified to use asynchronous events. It uses an arbitrator defined within the file processor to manage the file chunks produced by the `fileReader`. Now, the `fileReader` sends file chunks asynchronously to the arbitrator on the channel `arIn` (line 12). Note that each such asynchronous send acts as an arbitrator for the next asynchronous send. The `arbitrator` accepts file chunks

APP

$$\frac{}{\langle(\mathtt{t}, E[(\lambda x.e)\ v])\ ||\ \overline{T}\rangle_{\Delta,\mathcal{C}} \rightarrow \langle(\mathtt{t}, E[e[v/x]])\ ||\ \overline{T}\rangle_{\Delta,\mathcal{C}}}$$

CHANNEL

$$\frac{c\ \mathit{fresh}}{\langle(\mathtt{t}, E[\mathtt{ch}()])\ ||\ \overline{T}\rangle_{\Delta,\mathcal{C}} \rightarrow \langle(\mathtt{t}, E[c])\ ||\ \overline{T}\rangle_{\Delta,\mathcal{C}[c\mapsto\perp]}}$$

SPAWN

$$\frac{\mathtt{t}'\ \mathit{fresh}}{\langle(\mathtt{t}, E[\mathtt{spawn}\ e])\ ||\ \overline{T}\rangle_{\Delta,\mathcal{C}} \rightarrow \langle(\mathtt{t}', [e])\ ||\ (\mathtt{t}, E[\mathtt{unit}])\ ||\ \overline{T}\rangle_{\Delta,\mathcal{C}}}$$

SYNC

$$\frac{}{\langle(\mathtt{t}, E[\mathtt{sync}\ \varepsilon[e]])\ ||\ \overline{T}\rangle_{\Delta,\mathcal{C}} \rightarrow \langle(\mathtt{t}, E[e])\ ||\ \overline{T}\rangle_{\Delta,\mathcal{C}}}$$

ASYNC EVAL

$$\frac{\mathtt{t}'\ \mathit{fresh}}{\langle(\mathtt{t}, E[\{e,\ e'\}])\ ||\ \overline{T}\rangle_{\Delta,\mathcal{C}} \rightarrow \langle(\mathtt{t}, E[e])\ ||\ (\mathtt{t}', e')\ ||\ \overline{T}\rangle_{\Delta,\mathcal{C}}}$$

SENDEVENT

$$\frac{m\ \mathit{fresh}}{\langle(\mathtt{t}, E[\mathtt{sendEvt}(c,v)])\ ||\ \overline{T}\rangle_{\Delta,\mathcal{C}} \rightarrow \langle(\mathtt{t}, E[\varepsilon[\mathtt{sendAct}_m(c,v); \mathtt{sendWait}_m]])\ ||\ \overline{T}\rangle_{\Delta[m\mapsto\perp],\mathcal{C}}}$$

ASENDEVENT

$$\frac{m\ \mathit{fresh}}{\langle(\mathtt{t}, E[\mathtt{aSendEvt}(c,v)])\ ||\ \overline{T}\rangle_{\Delta} \rightarrow \langle(\mathtt{t}, E[\varepsilon[\{\mathtt{sendAct}_m(c,v),\ \mathtt{sendWait}_m\}]])\ ||\ \overline{T}\rangle_{\Delta[m\mapsto\perp],\mathcal{C}}}$$

RECVEVENT

$$\frac{m\ \mathit{fresh}}{\langle(\mathtt{t}, E[\mathtt{recvEvt}(c)])\ ||\ \overline{T}\rangle_{\Delta,\mathcal{C}} \rightarrow \langle(\mathtt{t}, E[\varepsilon[\mathtt{recvAct}_m(c); \mathtt{recvWait}_m]])\ ||\ \overline{T}\rangle_{\Delta[m\mapsto\perp],\mathcal{C}}}$$

ARECVEVENT

$$\frac{m\ \mathit{fresh}}{\langle(\mathtt{t}, E[\mathtt{aRecvEvt}(c)])\ ||\ \overline{T}\rangle_{\Delta,\mathcal{C}} \rightarrow \langle(\mathtt{t}, E[\varepsilon[\{\mathtt{recvAct}_m(c),\ \mathtt{recvWait}_m\}]])\ ||\ \overline{T}\rangle_{\Delta[m\mapsto\perp],\mathcal{C}}}$$

SWRAP

$$\frac{}{\langle(\mathtt{t}, E[\mathtt{sWrap}(\varepsilon[\{e,\ e'\}], \lambda x.e'')])\ ||\ \overline{T}\rangle_{\Delta,\mathcal{C}} \rightarrow \langle(\mathtt{t}, E[\varepsilon[\{(\lambda x.e'')\ e,\ e'\}]])\ ||\ \overline{T}\rangle_{\Delta,\mathcal{C}}}$$

WRAP

$$\frac{}{\langle(\mathtt{t}, E[\mathtt{wrap}(\varepsilon[e], \lambda x.e')])\ ||\ \overline{T}\rangle_{\Delta,\mathcal{C}} \rightarrow \langle(\mathtt{t}, E[\varepsilon[(\lambda x.e')\ e]])\ ||\ \overline{T}\rangle_{\Delta,\mathcal{C}}}$$

AWRAP

$$\frac{}{\langle(\mathtt{t}, E[\mathtt{aWrap}(\varepsilon[\{e,\ e'\}], \lambda x.e'')])\ ||\ \overline{T}\rangle_{\Delta,\mathcal{C}} \rightarrow \langle(\mathtt{t}, E[\varepsilon[\{e,\ (\lambda x.e'')\ e'\}]])\ ||\ \overline{T}\rangle_{\Delta,\mathcal{C}}}$$

GUARD

$$\frac{}{\langle(\mathtt{t}, E[\mathtt{guard}(\lambda x.e)])\ ||\ \overline{T}\rangle_{\Delta,\mathcal{C}} \rightarrow \langle(\mathtt{t}, E[\varepsilon[(\lambda x.e)\ \mathtt{unit}]])\ ||\ \overline{T}\rangle_{\Delta,\mathcal{C}}}$$

AGUARD

$$\frac{}{\langle(\mathtt{t}, E[\mathtt{aGuard}(\lambda x.e)])\ ||\ \overline{T}\rangle_{\Delta,\mathcal{C}} \rightarrow \langle(\mathtt{t}, E[\varepsilon[(\lambda x.e)\ \mathtt{unit}]])\ ||\ \overline{T}\rangle_{\Delta,\mathcal{C}}}$$

SEND WAIT

$$\frac{\Delta(m) = \mathtt{unit}}{\langle(\mathtt{t}, E[\mathtt{sendWait}_m])\ ||\ \overline{T}\rangle_{\Delta,\mathcal{C}} \rightarrow \langle(\mathtt{t}, E[\mathtt{unit}])\ ||\ \overline{T}\rangle_{\Delta,\mathcal{C}}}$$

RECEIVE WAIT

$$\frac{\Delta(m) = v}{\langle(\mathtt{t}, E[\mathtt{recvWait}_m])\ ||\ \overline{T}\rangle_{\Delta,\mathcal{C}} \rightarrow \langle(\mathtt{t}, E[v])\ ||\ \overline{T}\rangle_{\Delta,\mathcal{C}}}$$

SEND MATCH

$$\frac{\begin{array}{c}\mathcal{C}(c) = \mathtt{recvAct}_{m'}(c) : \overline{\mathcal{A}_r} \\ \Delta, \mathtt{sendAct}_m(c,v) \diamond \Delta' \quad \Delta', \mathtt{recvAct}_{m'}(c), v \diamond \Delta''\end{array}}{\langle(\mathtt{t}, E[\mathtt{sendAct}_m(c,v)])\ ||\ \overline{T}\rangle_{\Delta,\mathcal{C}} \rightarrow \langle(\mathtt{t}, E[\mathtt{unit}])\ ||\ \overline{T}\rangle_{\Delta'',\mathcal{C}[c\mapsto\overline{\mathcal{A}_r}]}}$$

SEND BLOCK

$$\frac{\mathcal{C}(c) = \overline{\mathcal{A}_s} \quad \mathcal{C}' = \mathcal{C}[c \mapsto \overline{\mathcal{A}_s} : \mathtt{sendAct}_m(c,v)]}{\langle(\mathtt{t}, E[\mathtt{sendAct}_m(c,v)])\ ||\ \overline{T}\rangle_{\Delta,\mathcal{C}} \rightarrow \langle(\mathtt{t}, E[\mathtt{unit}])\ ||\ \overline{T}\rangle_{\Delta,\mathcal{C}'}}$$

RECV MATCH

$$\frac{\begin{array}{c}\mathcal{C}(c) = \mathtt{sendAct}_{m'}(c,v) : \overline{\mathcal{A}_s} \\ \Delta, \mathtt{sendAct}_{m'}(c,v) \diamond \Delta' \quad \Delta', \mathtt{recvAct}_m(c), v \diamond \Delta''\end{array}}{\langle(\mathtt{t}, E[\mathtt{recvAct}_m(c)])\ ||\ \overline{T}\rangle_{\Delta,\mathcal{C}} \rightarrow \langle(\mathtt{t}, E[\mathtt{unit}])\ ||\ \overline{T}\rangle_{\Delta'',\mathcal{C}[c\mapsto\overline{\mathcal{A}_s}]}}$$

RECV BLOCK

$$\frac{\mathcal{C}(c) = \overline{\mathcal{A}_r} \quad \mathcal{C}' = \mathcal{C}[c \mapsto \overline{\mathcal{A}_r} : \mathtt{recvAct}_m(c)]}{\langle(\mathtt{t}, E[\mathtt{recvAct}_m(c)])\ ||\ \overline{T}\rangle_{\Delta,\mathcal{C}} \rightarrow \langle(\mathtt{t}, E[\mathtt{unit}])\ ||\ \overline{T}\rangle_{\Delta,\mathcal{C}'}}$$

EMPTY BLOCK

$$\frac{\mathcal{C}(c) = \perp \quad \mathcal{C}' = \mathcal{C}[c \mapsto \mathcal{A}]}{\langle(\mathtt{t}, E[\mathcal{A}])\ ||\ \overline{T}\rangle_{\Delta,\mathcal{C}} \rightarrow \langle(\mathtt{t}, E[\mathtt{unit}])\ ||\ \overline{T}\rangle_{\Delta,\mathcal{C}'}}$$

MESSAGE

$$\frac{\Delta(m) = \perp}{\Delta, \mathtt{sendAct}_m(c,v) \diamond \Delta[m \mapsto \mathtt{unit}]} \qquad \frac{\Delta(m) = \perp}{\Delta, \mathtt{recvAct}_m(c), v \diamond \Delta[m \mapsto v]}$$

Figure 9. Language semantics.

from the `fileReader` on this channel and synchronously sends the file chunks to the consumer as long as a timeout has not been detected. This is accomplished by choosing between an `abortEvt` (used by the `Timeout` manager to signal a timeout) and receiving a chunk from file processing loop (lines 13-20). When a timeout is detected, an asynchronous message is sent on channel `arOut` to notify the file processing loop of this fact (line 9); subsequent file processing then stops. This loop synchronously chooses between accepting a timeout notification (line 17), or asynchronously processing the next chunk (lines 11 - 12). The arbitrator executes as a post-consumption action.

```
datatype Xfr = TIMEOUT | DONE | X of chunk
1. fun fileReader name abortEvt consumer =
2. let
3.  val (arIn, arOut) = (channel(), channel())
4.  fun arbitrator() = sync
5.      (choose [
6.        wrap (recvEvt arIn,
7.              fn chunk => send (consumer, chunk)),
8.        wrap (abortEvt, fn () =>
9.              (aSync(aSendEvt(arOut, ()));
10.             send(consumer, TIMEOUT)))])
11. fun sendChunk(chunk) =
12.      aSync(aWrap(aSendEvt(arIn, X(chunk)),arbitrator))
13. fun loop strm =
14.      case BinIO.read (strm, size)
15.      of SOME chunk => sync
16.              (choose [
17.                recvEvt arOut,
18.                wrap(alwaysEvt,
19.                      fn () => (sendChunk(chunk);
20.                                loop strm))])
21.       | NONE => aSync(aSendEvt(arIn, DONE))
22. val _ = aSync(aWrap(aRecvEvt(arIn),
23.                fn chunk => send(consumer, chunk)))
24. in
25. case BinIO.openIt name of
26.      NONE => ()
27.    | SOME strm => (loop strm; BinIO.closeIt strm)
28. end
```

Since asynchronous events operate over regular CML channels, we were able to modify the the file producer to utilize asynchrony without having to change any of the other modules or the communication patterns and protocols they expect. Being able to choose between synchronous and asynchronous events in the `fileReader` function also allowed us to create a buffer of file chunks, but stop file processing file if a timeout was detected by the `arbitrator`. Recall, each asynchronous send acts as an arbitrator for the next asynchronous send.

Underlying I/O and Logging. To improve scalability and responsiveness, we also implemented a non-blocking I/O library composed of a language-level interface and associated runtime support. The library implements all MLton I/O interfaces, but internally utilizes asynchronous events. The library is structured around callback events as defined in Sec. 4.2 operating over I/O resource servers. Internally, all I/O requests are translated into a potential series of callback events.

Web-servers utilize logging for administrative purposes. For long running servers, logs tend to grow quickly. Some web-servers (like Apache) solve this problem by using a *rolling log*, which automatically opens a new log file after a set time period (usually a day). In Swerve, all logging functions were done synchronously. Using asynchronous events, we were able to easily change the logging infrastructure to use rolling logs. Because asynchronous events preserve ordering guarantees, log entries reflect actual thread action order. Post consumption actions were utilized to implement the rolling log functionality, by closing old logs and opening new logs after the appropriate time quantum.

8. Results

To measure the effectiveness of asynchronous events, we compared the performance of Swerve, modified to use asynchronous events (see Sec. 7), with the original (that uses only synchronous communication), and two other widely-used web-servers – Apache and Mongrel. Apache is written in C, while Mongrel is implemented using Ruby on Rails.

The benchmarks were run on an AMD Opteron 865 server with 8 processors, each containing two symmetric cores, and 32 GB of total memory, with each CPU having its own local memory of 4 GB. Access to non-local memory is mediated by a hyper-transport layer that coordinates memory requests between processors. *httperf*, an automated web-server performance analysis tool was used to conduct our experiments.

Fig. 10 shows detailed comparisons. In Fig. 10(a) we show throughput numbers as a function of increasing connections. Fig. 10(b) shows latency overheads, not including network latency, since clients and server execute on the same machine. Fig. 10(c) shows overall speedup between the original implementation and the modified one as a function of the number of processors. Translating the implementation to use asynchronous events leads to a roughly 3.5X performance improvement over the original, allowing the modified version to significantly outperform Mongrel, and greatly increase its competitiveness with Apache.

Not surprisingly, the results show that asynchronous communication, when carefully applied, can yield substantial performance gains. More significantly, however, is that these gains were achieved without having to perform large scale surgery on the application: there were roughly a total of 100 lines of Swerve code that needed to be modified to achieve these results. These changes were almost always mechanical, often just involving the replacement of a synchronous event combinator with an asynchronous one. The most complex changes were on the order of the modifications shown in Sec. 7. There were no changes required to module interfaces or the program's overall logical structure. We believe these experiments provide useful validation of the design goals given in Section 3.

9. Related Work

Many functional programming languages such as Erlang (1), F# (25; 19) and JoCaml (10) provide intrinsic support for asynchronous programming. In Erlang, message sends are inherently asynchronous. In JoCaml, complex asynchronous protocols are defined using join patterns (2; 11) that define synchronization protocols over asynchronous and synchronous channels. In F#, asynchronous behavior is defined using asynchronous work flows that permit asynchronous objects to be created and synchronized. Convenient monadic-style `let!`-syntax permits callbacks (i.e., continuations) to be created within an asynchronous computation. The callback defines the post-computation function for an asynchronous operation. While these abstractions and paradigms provide expressive ways to define asynchronous computations, they do not provide a convenient mechanism to specify *composable* asynchronous abstractions, especially with respect to asynchronous post-consumption actions. It is the investigation of this important aspect of asynchronous programming, and its incorporation within

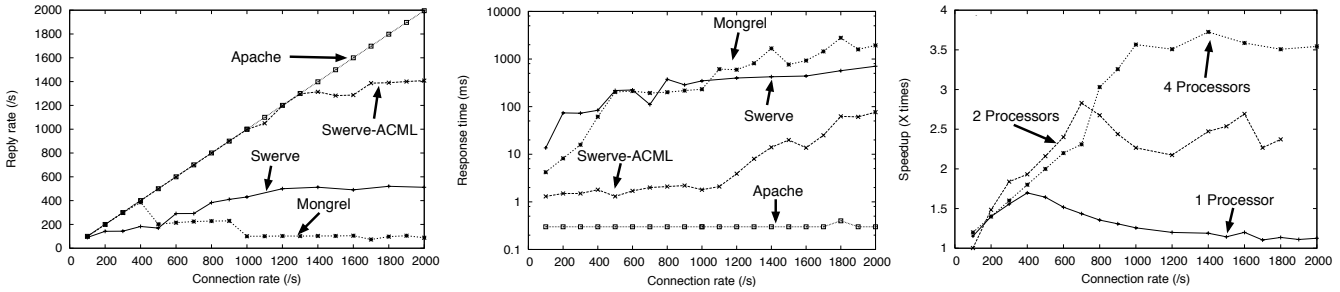

Figure 10. Performance of Swerve, using synchronous and asynchronous events, compared to Apache and Mongrel.

a CML-style event framework, that distinguishes the contributions of this paper from these other efforts.

Reactive programming (15) is an important programing style often found in systems programming that uses event loops to react to outside events (typically related to I/O). In this context, events do not define abstract communication protocols (as they do in CML), but typically represent I/O actions delivered asynchronously by the underlying operating system. While understanding how reactive events and threads can co-exist is an important one, it is orthogonal to the focus of this work which is concerned with the efficient expression of composable asynchronous protocols.

There have been incarnations of CML in languages and systems other than ML (e.g., Haskell (4; 22), Scheme (8), and MPI (5)) There has also been much recent interest in extending CML with transactional support (6; 7) and other flavors of parallelism (9). We believe transactional events (6; 7) provide an interesting platform upon which to implement non-blocking versions of `sChoose` that retains the same semantics. Additionally, we expect that previous work on specialization of CML primitives (20) can be applied to improve the performance of asynchronous primitives.

10. Conclusions

This paper presents the design, rationale, and implementation for asynchronous events, a concurrency abstraction that generalize the behavior of CML-based synchronous events to enable composable construction of asynchronous computations. Our experiments indicate that asynchronous events can seamlessly co-exist with other CML primitives, and can be effectively leveraged to improve performance of realistic highly-concurrent applications.

References

[1] Joe Armstrong, Robert Virding, Claes Wikstrom, and Mike Williams. *Concurrent Programming in Erlang*. Prentice-Hall, 2nd edition, 1996.

[2] Jean-Pierre Banâtre and Daniel Le Métayer. Programming by Multiset Transformation. *Commun. ACM*, 36(1), 1993.

[3] Nick Benton, Luca Cardelli, and Cédric Fournet. Modern Concurrency Abstractions for C#. *ACM Trans. Program. Lang. Syst.*, 26(5):769–804, 2004.

[4] Avik Chaudhuri. A Concurrent Ml Library in Concurrent Haskell. In *ICFP*, pages 269–280, 2009.

[5] Erik Demaine. First-Class Communication in MPI. In *MPIDC '96: Proceedings of the Second MPI Developers Conference*, 1996.

[6] Kevin Donnelly and Matthew Fluet. Transactional Events. *The Journal of Functional Programming*, pages 649–706, 2008.

[7] Laura Effinger-Dean, Matthew Kehrt, and Dan Grossman. Transactional Events for ML. In *ICFP*, pages 103–114, 2008.

[8] Matthew Flatt and Robert Bruse Findler. Kill-safe Synchronization Abstractions. In *PLDI*, pages 47–58, 2004.

[9] Matthew Fluet, Mike Rainey, John Reppy, and Adam Shaw. Implicitly-Threaded Parallelism in Manticore. In *ICFP '08*, pages 119–130, 2008.

[10] Cédric Fournet, Fabrice Le Fessant, Luc Maranget, and Alan Schmidt. JoCaml: A Langauge for Concurrent Distributed and Mobile Programming. In *Advanced Functional Programming*, pages 129–158. 2002.

[11] Cédric Fournet and Georges Gonthier. The reflexive cham and the join-calculus. In *POPL*, pages 372–385, 1996.

[12] G. Stewart Itzstein and Mark Jasiunas. On Implementing High-Level Concurrency in Java. In *Advances in Computer Systems Architecture*, pages 151–165, 2003. LNCS 2823.

[13] Jocaml. http://jocaml.inria.fr/.

[14] Guodong Li, Michael Delisi, Ganesh Gopalakrishnan, and Robert M. Kirby. Formal Specification of the MPI-2.0 Standard in TLA+. In *PPoPP*, pages 283–284, 2008.

[15] Peng Li and Steve Zdancewic. Combining Events and Threads for Scalable Network Services, and Evaluation of Monadic, Application-Level Concurrency Primitives. In *PLDI*, pages 189–199, 2007.

[16] Qin Ma and Luc Maranget. Compiling Pattern Matching in Join-Patterns. In *CONCUR*, pages 417–431, 2004.

[17] Robin Milner, Mads Tofte, and David Macqueen. *The Definition of Standard ML*. MIT Press, Cambridge, MA, USA, 1997.

[18] MLton. http://www.mlton.org.

[19] Robert Pickering. *Foundations of F#*. Apress, 2007.

[20] John Reppy and Yingqi Xiao. Specialization of CML Message-Passing Primitives. In *POPL*, pages 315–326, 2007.

[21] John H. Reppy. *Concurrent Programming in ML*. Cambridge University Press, 1999.

[22] George Russell. Events in Haskell, and How to Implement Them. In *ICFP*, pages 157–168, 2001.

[23] http://www.scala-lang.org.

[24] Satnam Singh. Higher-Order Combinators for Join Patterns Using STM. In *ACM Transact Workshop*, 2006.

[25] Don Syme, Adam Granicz, and Antonio Cisternino. *Expert F#*. Apress, 2007.