$\mathbb{R}^{\text{CML}}$: A Prescription for Safely Relaxing Synchrony

KC Sivaramakrishnan[1], Lukasz Ziarek[2], and Suresh Jagannathan[1]

[1] Purdue University ({chandras,suresh}@cs.purdue.edu)
[2] SUNY Buffalo (lziarek@buffalo.edu)

Abstract. A functional programming discipline, combined with abstractions like Concurrent ML (CML)'s first-class synchronous events, offers an attractive programming model for concurrency. In high-latency distributed environments, like the cloud, however, the high communication latencies incurred by synchronous communication can compromise performance. While switching to an explicitly asynchronous communication model may reclaim some of these costs, program structure and understanding also becomes more complex. To ease the challenge of migrating concurrent applications to distributed cloud environments, we have built an extension of the MultiMLton compiler and runtime that *implements* CML communication asynchronously, but guarantees that the resulting execution is *faithful* to the synchronous semantics of CML. We formalize the conditions under which this equivalence holds, and present an implementation that builds a decentralized dependence graph whose structure can be used to check the integrity of an execution with respect to this equivalence. We integrate a notion of speculation to allow ill-formed executions to be rolled-back and re-executed, replacing offending asynchronous actions with safe synchronous ones. Several realistic case studies deployed on the Amazon EC2 cloud infrastructure demonstrate the utility of our approach.

Keywords: Message-passing, Speculative Execution, Axiomatic Semantics, Cloud Computing

1 Introduction

Concurrent ML [18] (CML) provides an expressive concurrency mechanism through its use of first-class composable synchronous events. When synchronized, events allow threads to communicate data via message-passing over first-class channels. Synchronous communication simplifies program reasoning because every communication action is also a synchronization point; thus, the continuation of a message-send is guaranteed that the data being sent has been successfully transmitted to a receiver. The cost of synchrony comes at a high price in performance, however; recent proposals therefore suggest the use of asynchronous variants of CML's synchronous events [28] to overcome this cost. While asynchronous extensions can be used to gain performance, they sacrifice the simplicity provided by synchronous communication in favor of a more complex and sophisticated set of primitives.

One way to enhance performance without requiring new additions to CML's core set of event combinators is to give the underlying runtime the freedom to allow a sender to communicate data asynchronously. In this way, the cost of synchronous communication can be masked by allowing the sender's continuation to begin execution even if a matching receiver is not yet available. Because asynchrony is introduced only by the runtime, applications do not have to be restructured to explicitly account for new behaviors introduced by this additional concurrency. Thus, we wish to have the runtime enforce the equivalence: $[\![$ `send` $(c, v)]\!]k \equiv [\![$ `asend` $(c, v)]\!]k$ where k is a continuation, `send` is CML's synchronous send operation that communicates value v on channel c, and `asend` is an asynchronous variant that buffers v on c and does not synchronize with matching receiver.

Motivation. To motivate the utility of safe relaxation of synchronous behavior, consider the problem of building a distributed chat application. The application consists of a number of participants, each of whom can broadcast a message to every other member in the group. The invariant that must be observed is that any two messages sent by a participant must appear in the same order to all members. Moreover, any message `Y` broadcast in response to a previously received message `X` must always appear after message `X` to every member. Here, message `Y` is said to be *causally dependent* on message `X`.

```
datatype 'a bchan = BCHAN of ('a chan list (*val*) * unit chan list (*ack*))

fun newBChan (n: int) (* number of participants *) =
  BCHAN(tabulate(n,fn _ => channel()), tabulate(n,fn _ => channel()))

fun bsend (BCHAN (vcList, acList), v: 'a, id: int) : unit =
let
  val _ = map (fn vc => if (vc = nth (vcList, id)) then () else send (vc, v))
            vcList (* phase 1 -- Value distribution *)
  val _ = map (fn ac => if (ac = nth (acList, id)) then () else recv ac)
            acList (* phase 2 -- Acknowledgments *)
in ()
end

fun brecv (BCHAN (vcList, acList), id: int) : 'a=
let val v = recv (nth (vcList, id))
    val _ = send (nth (acList, id), ())
in v
end
```

Fig. 1: Synchronous broadcast channel

Building such an application using a centralized server is straightforward, but hinders scalability. In the absence of central mediation, a causal broadcast protocol [2] is required. One possible encoding of causal broadcast using CML primitives is shown in Figure 1. A broadcast operation involves two phases. In the first phase, values (i.e., messages) are synchronously communicated to all receivers (except to the sender). In the second phase, the sender simulates a barrier by synchronously receiving acknowledgments from all recipients.

The synchronous nature of the broadcast protocol along with the fact that the acknowledgment phase occurs only after message distribution ensure that no member can proceed immediately after receiving a message until all other members have also received the message. This achieves the desired causal ordering between broadcast messages since every member would have received a message before the subsequent causally ordered message is generated. We can build a distributed group chat server using the broadcast channel as shown below.

```
(* bc is broadcast chan, daemon is spawn as a separate thread *)
fun daemon id = display (brecv (bc, id)); daemon id
fun newMessage (m, id) = display m; bsend (bc, m, id)
```

Assume that there are n participants in the group, each with a unique identifier *id* between 0 and $n-1$. Each participant runs a local *daemon* thread that waits for incoming messages on the broadcast channel `bc`. On a reception of a message, the daemon displays the message and continues waiting. The clients broadcast a message using `newMessage` after displaying the message locally. Observe that remote messages are only displayed after all other participants have also received the message. In a geo-distributed environment, where the communication latency is very high, this protocol results in a poor user experience that degrades as the number of participants increases.

Without making wholesale (ideally, zero!) changes to this relatively simple protocol implementation, we would like to improve responsiveness, while preserving correctness. One obvious way of reducing latency overheads is to convert the synchronous sends in `bsend` to an asynchronous variant that buffers the message, but does not synchronize with a matching receiver. There are two opportunities where asynchrony could be introduced, either during value distribution or during acknowledgment reception. Unfortunately, injecting asynchrony at either point is not guaranteed to preserve causal ordering on the semantics of the program.

Consider the case where the value is distributed asynchronously. Assume that there are three participants: p_1, p_2, and p_3. Participant p_1 first types message `X`, which is seen by p_2, who in turn types the message `Y` after sending an acknowledgment. Since there is a causal order between the message `X` and `Y`, p_3 must see `X` followed by `Y`. The key observation is that, due to asynchrony, message `X` sent by the p_1 to p_3 might be *in-flight*, while the causally dependent message `Y` sent by p_2 reaches p_3 out-of-order. This leads to a violation of the protocol's invariants. Similarly, it is easy to see that sending acknowledgments message asynchronously is also incorrect. This would allow a participant that receives a message to asynchronously send an acknowledgment, and proceed before all other participants have received the same message. As a result, causal dependence between messages is lost.

To quantify these issues in a realistic setting, we implemented a group chat simulator application using a distributed extension of the MultiMLton Standard ML compiler. We launched three Amazon EC2 instances, each simulating a participant in the group chat application, with the same communication pattern described in the discussion above. In order to capture the geo-distributed nature of the application, participants were placed in three different availability zones – EU West (Ireland), US West (Oregon), and Asia Pacific (Tokyo), resp.

During each run, p_1 broadcasts a message `X`, followed by p_2 broadcasting `Y`. We consider the run to be successful if the participant p_3 sees the messages `X`,`Y`, in that order. The experiment was repeated for 1K iterations. We record the time between protocol initiation and the time at which each participant gets the message `Y`. We consider the largest of the times across the participants to be the running time. The results are presented below.

Execution	*Avg.time (ms)*	*Errors*
Sync	1540	0
Unsafe Async	520	7
Safe Async ($\mathrm{R}^{\mathrm{CML}}$)	533	0

The *Unsafe Async* row describes the variant where both value and acknowledgment distribution is performed asynchronously; it is three times as fast as the synchronous variant. However, over the total set of 1K runs, it produced seven erroneous executions. The *Safe Async* row illustrates our implementation, $\mathrm{R}^{\mathrm{CML}}$, that detects erroneous executions on-the-fly and remediates them. The results indicate that the cost of ensuring safe asynchronous executions is quite low for this application, incurring only roughly 2.5% overhead above the unsafe version. Thus, in this application, we can gain the performance benefits and responsiveness of the asynchronous version, while retaining the simplicity of reasoning about program behavior synchronously.

Contributions. The formalization of *well-formed executions*, those that are the result of asynchronous evaluation of CML send operations, but which nonetheless are observably equivalent to a synchronous execution, and the means by which erroneous executions can be detected and repaired, form the focus of this paper. Specifically, we make the following contributions:

– We present the rationale for a *relaxed execution model* for CML that specifies the conditions under which a synchronous operation can be safely executed asynchronously. Our model allows applications to program with the simplicity and composability of CML synchronous events, but reap the performance benefits of implementing communication asynchronously.
– We develop an axiomatic formulation of the model that can be used to reason about correctness in terms of causal dependencies captured by a *happens-before* relation.
– A distributed implementation, $\mathrm{R}^{\mathrm{CML}}$, that treats asynchronous communication as a form of *speculation* is described. A mis-speculation, namely one that produces an execution that could not have been realized using only synchronous communication can be detected and rolled back, with the offending operation re-executed as a synchronous action, known to be safe.
– Several case studies on a realistic cloud deployment demonstrate the utility of the model in improving the performance of CML programs in distributed environments without requiring *any* restructuring of application logic to deal with asynchrony.

The paper is organized as follows. In the next section, we present an axiomatic formalization of our intuition behind a relaxed synchronous execution. Section 3 details the $\mathrm{R}^{\mathrm{CML}}$ design and implementation, and discusses the role of speculation to implement asynchronous communication transparently; a central part of the $\mathrm{R}^{\mathrm{CML}}$architecture is a *distributed dependence graph* that is used to check the validity of speculative actions, and facilitate rollback when a mis-speculation occurs. Section 4 presents case studies of several benchmarks to illustrate the benefit of the model. Related work and conclusions are given in Sections 5 and 6, respectively.

2 Axiomatic Semantics

We introduce an axiomatic formalization for reasoning about the relaxed behaviors of a concurrent message-passing programs with dynamic thread creation. Not surprisingly, our formulation is similar in structure to axiomatic formalizations used to describe, for example, relaxed memory models [7, 19, 21].

An *axiomatic execution* is captured by a set of *actions* performed by each thread and the relationship between them. These actions abstract the relevant behaviors possible in a CML execution, relaxed or otherwise. Relation between the actions as a result of sequential execution, communication, thread creation and thread joins define the dependencies that any sensible execution must respect. A relaxed execution, as a result of speculation, admits more behaviors than observable under synchronous CML execution. Therefore, to understand the validity of executions, we define a *well-formedness* condition that imposes additional constraints on executions to ensure their observable effects correspond to correct CML behavior.

We assume a set of $\mathbb{T}$ threads, $\mathbb{C}$ channels, and $\mathbb{V}$ values. The set of actions is provided below. Superscripts m and n denote a unique identifier for the action.

Actions $\mathbb{A} :=$	b_t	(t starts)	$\mid e_t$	(t ends)
	$\mid j_t^m t'$	(t detects t' has terminated)	$\mid f_t^m t'$	(t forks a new t')
	$\mid s_t^m c, v$	(t sends value v on c)	$\mid r_t^m c$	(t receives a value v on c)
	$\mid p_t^m v$	(t outputs an observable value v)		

$c \in \mathbb{C}$ (Channels) $\quad t, t' \in \mathbb{T}$ (Threads) $\quad v \in \mathbb{V}$ (Values) $\quad m, n \in \mathbb{N}$ (Numbers)

Action b_t signals the initiation of a new thread with identifier t; action e_t indicates that thread t has terminated. A join action, $j_t^m t'$, defines an action that recognizes the point where thread t detects that another thread t' has completed. A thread creation action, where thread t spawns a thread t', is given by $f_t^m t'$. Action $s_t^m c, v$ denotes the communication of data v on channel c by thread t, and $r_t^m c$ denotes the receipt of data from channel c. An external action (e.g., printing) that emits value v is denoted as $p_t^m v$. We can generalize these individuals actions into a family of related actions:

$\mathbb{A}_r = \{r_t^m c \mid t \in \mathbb{T}\}$	(Receives)	$\mathbb{A}_s = \{s_t^m c, v \mid t \in \mathbb{T}, v \in \mathbb{V}\}$	(Sends)
$\mathbb{A}_c = \mathbb{A}_s \cup \mathbb{A}_r$	(Communication)	$\mathbb{A}_o = \{p_t^m v \mid t \in \mathbb{T}, v \in \mathbb{V}\}$	(Observables)

Notation. We write $T(\alpha)$ to indicate the thread in which action α occurs, and write $V(s_t^m c, v)$ to extract the value v communicated by a send action. Given a set of actions $\mathsf{A} \in 2^{\mathbb{A}}$, $\mathsf{A}_x = \mathsf{A} \cap \mathbb{A}_x$, where $\mathbb{A}_x$ represents one of the action classes defined above.

Definition 1 (Axiomatic Execution). *An axiomatic execution is defined by the tuple* $\mathsf{E} := \langle \mathsf{P}, \mathsf{A}, \rightarrow_{po}, \mathsf{M} \rangle$ *where:*

- P *is a program.*
- A *is a set of actions.*
- $\rightarrow_{po} \subseteq \mathsf{A} \times \mathsf{A}$ *is the program order, a disjoint union of the sequential actions of each thread (which is a total order).*
- $\mathsf{M} \in (\mathbb{A}_s \rightharpoonup \mathbb{A}_r) \cup (\mathbb{A}_r \rightharpoonup \mathbb{A}_s)$ *is a communication-match function that maps each send and receive to its matching communication action (i.e, if* $\alpha = \mathsf{M}(\alpha')$ *then* $\alpha' = \mathsf{M}(\alpha)$*). Moreover, a send and its matching receive must operate on the same channel and operate in different threads (i.e., if* $\mathsf{M}(s_t^m c, v) = r_{t'}^n c'$ *or* $\mathsf{M}(r_{t'}^n c') = s_t^m c, v$ *then* $t \neq t'$ *and* $c = c'$*).*

Definition 2 (Communication Order). *A communication order is established between matching communication actions. If $\beta = \mathsf{M}(\alpha)$, then $\alpha \rightarrow_{co} \beta$ and $\beta \rightarrow_{co} \alpha$.*

There is also an obvious ordering on thread creation and execution, as well as the visibility of thread termination by other threads:

Definition 3 (Thread Dependence). *If $\alpha = f_t^m t'$ and $\beta = b_{t'}$ or $\alpha = e_t$ and $\beta = j_{t'}^m t$ then $\alpha \rightarrow_{td} \beta$ holds.*

Definition 4 (Happens-before relation). *The happens-before order of an execution is the transitive closure of the union of program order, thread dependence order, and actions related by communication and program order:*

$$\begin{aligned} \rightarrow_{hb} = (&\rightarrow_{po} \cup \rightarrow_{td} \cup \\ &\{(\alpha,\beta) \mid \alpha \rightarrow_{co} \alpha' \wedge \alpha' \rightarrow_{po} \beta\} \cup \\ &\{(\beta,\alpha) \mid \beta \rightarrow_{po} \alpha' \wedge \alpha' \rightarrow_{co} \alpha\})^+ \end{aligned}$$

For any two actions $\alpha, \beta \in \mathsf{A}$, if $\alpha \not\leftrightarrow_{hb} \beta$, then α and β are said to be *concurrent* actions. Importantly, our happens-before relation defines a preorder. A preorder is a reflexive transitive binary relation. Unlike partial orders, preorders are not necessarily anti-symmetric, i.e. they may contain cycles.

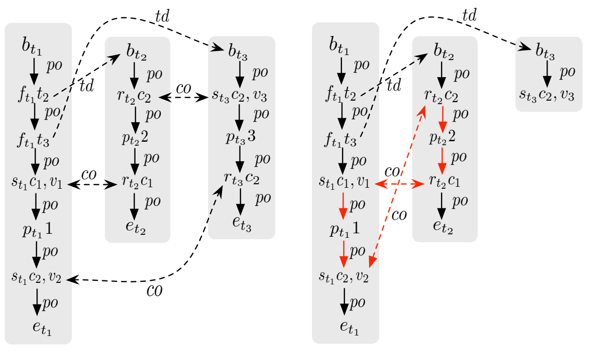
Definition 5 (Happens-before Cycle). *A cycle exists in a happens-before relation if for any two actions α, β and $\alpha \rightarrow_{hb} \beta \rightarrow_{hb} \alpha$.*

```
(* current thread is t1 *)
val t2 = spawn (fn () =>
           recv c2;
           print "2";
           recv c1)

val t3 = spawn (fn () =>
           send(c2,v2);
           print "3";
           recv c2)

val _ = send(c1,v1)
val _ = print "1"
val _ = send(c2,v2)
```

(a) A CML program



(b) Well-formed execution (c) Ill-formed execution

Fig. 2: A CML program and its potential axiomatic executions.

We provide an example to illustrate these definitions and to gain an insight into erroneous executions that manifest as a result of speculative communication. Consider the example presented in Figure 2 which shows a simple CML program and two possible executions. The execution in Figure 2b imposes no causal dependence between the observable actions (i.e., print statements) in t_2 or t_3; thus, an interleaving derived from this execution may permute the order in which these statements execute. All interleavings derivable from this execution correspond to valid CML behavior.

In contrast, the execution depicted in Figure 2c, exhibits a happens-before cycle between t_1 and t_2, through a combination of program and communication order edges. *Such cyclic dependences never manifest in any correct CML execution.* Cyclic dependences may however manifest when synchronous sends are speculatively discharged asynchronously. We must therefore strengthen our notion of correct executions to discard those that contain such cycles.

To do so, we first note that the semantics as currently presented is concerned only with actions that introduce some form of causal dependence either within a thread (via program order) or across threads (via thread dependence or communication order). However, a real program also does computation, and reasoning about an execution's correctness will require us to specify these actions as well. To facilitate this reasoning, we abstract the intra-thread semantics, and parameterize our definition of an axiomatic execution accordingly.

Intra-thread semantics. The intra-thread semantics is abstracted in our formulation via a labeled transition system. Let $\mathsf{State}_{\mathsf{intra}}$ denote the intra-thread state of a thread; its specific structure is not interesting for the purposes of the axiomatic definition[3]. A labeled transition between intra-thread states is captured by the relation

$$\xrightarrow{\cdot}\!\!\!\!\rightarrow \subseteq \mathsf{State}_{\mathsf{intra}} \times \mathsf{Label}_{\mathsf{intra}} \times \mathsf{State}_{\mathsf{intra}}$$

given to each thread $t \in \mathbb{T}$. The transition labels are in the set $\mathsf{Label}_{\mathsf{intra}} = (\mathbb{A} \setminus \mathbb{A}_r) \cup (\mathbb{A}_r \times \mathbb{V}) \cup \{\tau\}$. Thus, a thread can either take a global action step (e.g., creating another thread, performing a send action, ending a thread, etc.), execute a *silent* thread-local computation (denoted by label τ), or execute a receive action that receives the value associated with the label. The requirements on the intra-thread semantics are:

- $\xrightarrow{\cdot}\!\!\!\!\rightarrow$ can only relate states belonging to the same thread.
- there is an initial state READY: no transition leads to it, and a thread t steps from it if and only if it emits a begin action b_t.
- there is a final state DONE: a thread leads to it if and only if it emits an end action e_t and no transition leads from it.

Definition 6 (Intra-trace). *Let $tr = \overline{\alpha}$ be a sequence of actions in set A, and M be a communication match function on A. Given a thread $t \in \mathbb{T}$ in a program P, tr is a valid intra-trace for t if there exists a set of states $\{\delta_0, \delta_1, \ldots\}$, and a set of labels $\overline{l} = \{l_0, l_1, \ldots\}$ such that:*

- *for all $\alpha_i \in \overline{\alpha}, T(a) = t$*
- *δ_0 is the initial state* READY
- *for all $0 \leq i, \delta_i \xrightarrow{l_i}\!\!\!\!\rightarrow \delta_{i+1}$*
- *the projection $\overline{\beta}$ of $\overline{l}$ to non-silent labels is such that $\beta_i = (\alpha_i, V(\mathsf{M}(\alpha_i)))$ if $\alpha_i \in \mathsf{A}_r$, or $\beta_i = \alpha_i$ otherwise.*

We write $\mathsf{InTr}^{\mathsf{P}}[\mathsf{t}]$ set of such pairs (tr, M) for P.

[3] The concrete instantiation of the intra-thread state, and an operational semantics for the language are given in a technical report, available from: `http://multimlton.cs.purdue.edu/mML/rx-cml.html`

Definition 7 (Well-formed Execution). *An execution $E := \langle \mathsf{P}, \mathsf{A}, \rightarrow_{po}, \mathsf{M}\rangle$ is well-formed if the following conditions hold:*

1. *Intra-thread consistency: for all threads $t \in \mathbb{T}$, $([\rightarrow_{po}]_t, \mathsf{M}) \in \mathsf{InTr}^{\mathsf{P}}[\mathsf{t}]$*
2. *Happens-before correctness: The happens-before relation $\rightarrow_{hb}$ constructed from E has no cycles.*
3. *Observable correctness: Given $\alpha \in \mathsf{A}_o$ and $\beta \in \mathsf{A}_c$ if $\beta \rightarrow_{hb} \alpha$ then there exists $\beta' \in \mathsf{A}_c$ s.t. $\mathsf{M}(\beta) = \beta'$.*

For an axiomatic execution $\mathsf{E} := \langle \mathsf{P}, \mathsf{A}, \rightarrow_{po}, \mathsf{M}\rangle$ to be well-formed, the actions, program order relation, and the communication match function must have been obtained from a valid execution of the program P as given by the intra-thread semantics defined above (1). As we noted in our discussion of Figure 2, no valid execution of a CML program may involve a cyclic dependence between actions; such dependencies can only occur because of *speculatively* performing what is presumed to be a synchronous send operation (2).

Finally, although the relaxed execution might speculate, i.e., have a send operation transparently execute asynchronously, the observable behavior of such an execution should mirror some valid non-speculative execution, i.e., an execution in which the send action was, in fact, performed synchronously. We limit the scope of speculative actions by requiring that they complete (i.e., have a matching recipient) before an observable action is performed (3). Conversely, this allows communication actions not preceding an observable action to be speculated upon. Concretely, a send not preceding an externally visibile action can be discharged asynchronously. The match and validity of the send needs to be checked only before discharging the next such action. This is the key idea behind our speculative execution framework.

Safety. An axiomatic execution represents a set of interleavings, each interleaving defining a specific total order that is consistent with the partial order defined by the execution[4]. The well-formedness conditions of an axiomatic execution implies that any observable behavior of an interleaving induced from it must correspond to a synchronous CML execution. The following two definitions formalize this intuition.

Definition 8 (Observable dependencies). *In a well-formed axiomatic execution $\mathsf{E} := \langle \mathsf{P}, \mathsf{A}, \rightarrow_{po}, \mathsf{M}\rangle$, the observable dependencies A_{od} is the set of actions that precedes (under $\rightarrow_{hb}$) some observable action, i.e., $\mathsf{A}_{od} = \{\alpha \mid \alpha \in \mathsf{A}, \beta \in \mathsf{A}_o, \alpha \rightarrow_{hb} \beta\}$.*

Definition 9 (CML Execution). *Given a well-formed axiomatic execution*

$$\mathsf{E} := \langle \mathsf{P}, \mathsf{A}, \rightarrow_{po}, \mathsf{M}\rangle$$

, the pair $(\mathsf{E}, \rightarrow_{to})$ is said to be in CML(P) if $\rightarrow_{to}$ is a total order on A_{od} and $\rightarrow_{to}$ is consistent with $\rightarrow_{hb}$.

In the above definition, an interleaving represented by $\rightarrow_{to}$ is only possible since the axiomatic execution is well-formed, and thereby does not contain a happens-before cycle.

[4] Two ordering relations P and Q are said to be *consistent* if $\forall x, y, \neg(xPy \ \wedge \ yQx)$.

Lemma 1. *If a total order $\rightarrow_{to}$ is consistent with $\rightarrow_{hb}$, then $\rightarrow_{hb}$ does not contain a cycle involving actions in A_{od}.*

Next, we show that a well-formed axiomatic execution respects the safety property of non-speculative execution of a CML program. When a CML program evaluates non-speculatively, a thread performing a communication action is blocked until a matching communication action is available. Hence, if $(\langle \mathsf{P}, \mathsf{A}, \rightarrow_{po}, \mathsf{M}\rangle, \rightarrow_{to}) \in \textsc{Cml}(\mathsf{P})$, and a communication action α on a thread t is followed by an action β on the same thread, then it must be the case that there is a matching action $\alpha' = \mathsf{M}(\alpha)$ that happened before β in $\rightarrow_{to}$. This is captured in the following theorem.

Theorem 1. *Given a CML execution $(\mathsf{E}, \rightarrow_{to}) \in \textsc{Cml}(\mathsf{P})$, $\forall \alpha, \beta$ such that $\alpha \in \mathbb{A}_c, T(\alpha) = T(\beta), \alpha \rightarrow_{to} \beta$, there exists an action $\alpha' = \mathsf{M}(\alpha)$ such that $\alpha' \rightarrow_{to} \beta$.*

Proof. Let $\mathsf{E} := \langle \mathsf{P}, \mathsf{A}, \rightarrow_{po}, \mathsf{M}\rangle$. First, we show that $\alpha' \in \mathsf{A}$. Since $\alpha \rightarrow_{to} \beta$, $\alpha \in \mathsf{A}_{od}$, by Definition 9. By Definition 8, there exists some $\gamma \in \mathsf{A}_o$ such that $\alpha \rightarrow_{hb} \gamma$. Since E is well-formed and $\alpha \rightarrow_{hb} \gamma$, by Definition 7, $\alpha' = \mathsf{M}(\alpha) \in \mathsf{A}$. Next, we show that $\alpha' \in \mathsf{A}_{od}$. By Definitions 2 and 4, $\alpha' \rightarrow_{co} \alpha \rightarrow_{hb} \gamma$ implies $\alpha' \rightarrow_{hb} \gamma$. Hence, $\alpha' \in \mathsf{A}_{od}$, and is related by $\rightarrow_{to}$. Finally, since $T(\alpha) = T(\beta)$ and $\alpha \rightarrow_{to} \beta$, $\alpha \rightarrow_{po} \beta$. And, $\alpha' \rightarrow_{co} \alpha \rightarrow_{po} \beta$ implies $\alpha' \rightarrow_{hb} \beta$. By Lemma 1 and Definition 9, $\alpha' \rightarrow_{to} \beta$.

3 Implementation

The axiomatic semantics provides a declarative way of reasoning about correct CML executions. In particular, a well-formed execution does not have a happens-before cycle. In practice, the speculative execution framework needs to perform this check *on-the-fly*. Since we are free to discharge synchronous sends asynchronously (speculatively), we need to build the relations necessary to check the integrity of the interleaving as the program executes.

To do so, we construct a *dependence graph* that captures the dependencies described by an axiomatic execution, and ensure the graph has no cycles. If a cycle is detected, we rollback the effects induced by the offending speculative action, and re-execute it as a normal synchronous operation. The context of our investigation is a distributed implementation of CML called $\mathbb{R}^{\text{CML}}$(RELAXED CML)[5] built on top of the MultiMLton SML compiler and runtime [15]. We have extended MultiMLton with the infrastructure necessary for distributed execution.

3.1 System Architecture

An $\mathbb{R}^{\text{CML}}$ application consists of multiple *instances*, each of which runs the *same* MultiMLton executable. These instances might run on the same node, on different nodes within the same datacenter, or on nodes found in different data centers. Each instance has a scheduler which preemptively multiplexes execution of user-level CML threads over multiple cores. We use the ZeroMQ messaging library [26] as the transport layer over which the $\mathbb{R}^{\text{CML}}$ channel communication is implemented. In addition to providing reliable and efficient point-to-point communication, ZeroMQ also provides the ability to construct higher-level multicast patterns. In particular, we leverage ZeroMQ's publish/subscribe support to implement CML's first-class channel based communication.

[5] `http://multimlton.cs.purdue.edu/mML/rx-cml.html`

The fact that every instance in an R$^{\text{CML}}$ application runs the same program, in addition to the property that CML channels are strongly-typed, allows us to provide typesafe serialization of immutable values as well as functions closures. Serializing mutable references is disallowed, and an exception is raised if the value being serialized refers to a mutable object. To safely refer to the same channel object across multiple instances, channel creation is parameterized with an identity string. Channels created with the same identity string refer to the same channel object across all instances in the R$^{\text{CML}}$ application. Channels are first-class citizens and can be sent as messages over other channels to construct complex communication protocols.

3.2 Communication Manager

Each R$^{\text{CML}}$ instance runs a single communication manager thread, which maintains globally consistent replica of the CML channels utilized by its constituent CML threads. The protocol for a single CML communication is illustrated in Figure 3. Since CML channel might potentially be shared among multiple threads across different instances, communication matches are determined dynamically. In general, it is not possible to determine the matching thread and its instance while initiating the communication action. Hence, whenever a thread intends to send or receive a value on the channel, its intention (along with a value in the case of a send operation), is broadcast to every other R$^{\text{CML}}$ instance. Importantly, the application thread performing the send does not block and *speculatively* continues execution.

Subsequently, an application thread that performs a receive on this channel consumes the send action, sends a *join message* to the sender thread's instance, and proceeds immediately. In particular, receiver thread does not block to determine if the send action was concurrently consumed by a thread in another instance. This corresponds to speculating on the communication match, which will succeed in the absence of concurrent receives for the same send action. On receiving the join message, a *match message* is broadcast to every instance, sealing the match. Those instances that speculatively matched with the send, except the one indicated in the match message, treat their receive action as a mis-speculation. Other instances that have not matched with this particular send remove the send action from the corresponding local channel replica.

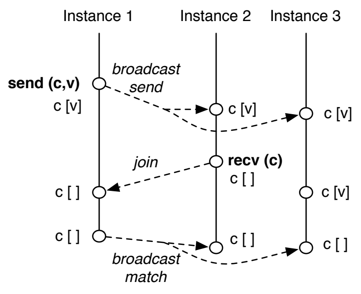


Fig. 3: Communication manager behavior during a send and its matching receive.

3.3 Speculative Execution

Aborting a mis-speculation requires restoring the computation to a previously known consistent state. Achieving this entails rolling back all threads that communicated with the offending action, transitively. In this regard, *stabilizers* [27] provide a suitable abstraction for restoring consistent checkpoints in message-passing programs. A stabilizer builds a dependence graph that takes into account intra-thread program order and inter-thread communication dependence. However, the implementation reported in [27] assumes a centralized structure, and a global barrier that stops all execution while a checkpoint is restored; neither condition is reasonable in a high-latency, distributed environment.

Replicated dependence graph. Instead, R^{CML} exploits the broadcast nature of the match message (Section 3.2) to incrementally construct a globally-consistent replica of the dependence graph at every instance. The nodes in the dependence graph correspond to the actions in the axiomatic definition. Thread spawn and join actions are broadcast to allow other instances to add necessary nodes and edges. Maintaining a replica of the dependence graph at each replica allows ill-formed executions to be detected locally and remediated.

Well-formedness check. To ensure observable behavior of an R^{CML} program to its synchronous equivalent, the compiler automatically inserts a *well-formedness check* before observable actions in the program. R^{CML} treats system calls, access to mutable references, and foreign function calls as observable actions. On reaching a well-formedness check, a *cycle-detector* is invoked which checks for cycles in the dependence graph leading up to this point. If the execution is well-formed (no cycles in the dependence graph), then the observable action is performed. Since there is no need to check for well-formedness of this fragment again, the verified dependence graph fragment is garbage collected on all instances.

Checkpoint. After a well-formedness check, the state of the current thread is consistent. Hence, right before the next (speculative) communication action, we checkpoint the current thread by saving its current continuation. This ensures that the observable actions performed after the well-formedness check are not re-executed if the thread happens to rollback. In addition, this checkpointing scheme allows multiple observable actions to be performed between a well-formedness check and the subsequent checkpoint. Unlike Stabilizers [27], every thread in an R^{CML} application has exactly one saved checkpoint continuation during the execution. Moreover, R^{CML} checkpointing is un-coordinated [10], and does not require that all the threads that transitively interacted capture their checkpoint together, which would be unreasonable in geo-distributed application.

Remediation. If the well-formedness check does report a cycle, then all threads that have *transitively* observed the mis-speculation are rolled back. The protocol roughly follows the same structure described in [27], but is asynchronous and does not involve a global barrier. The recovery process is a combination of checkpoint (saved continuation) and log-based (dependence graph) rollback and recovery [10]. Every mis-speculated thread is eventually restored to a consistent state by replacing its current continuation with its saved continuation, which was captured in a consistent state.

Recall that R^{CML} automatically captures a checkpoint, and only stores a single checkpoint per thread. As a result, rolling back to a checkpoint might entail re-executing, in addition to mis-speculated communication actions, correct speculative communications as well (i.e., communication actions that are not reachable from a cycle in the dependence graph). Thus, after the saved continuation is restored, correct speculative actions are *replayed* from the dependence graph, while mis-speculations are discharged non-speculatively (i.e., synchronously). This strategy ensures progress. Finally, we leverage ZeroMQ's guarantee on FIFO ordered delivery of messages to ensure that messages in-flight during the remediation process are properly accounted for.

3.4 Handling full CML

Our discussion so far has been limited to primitive `send` and `recv` operations. $\mathcal{R}^{\text{CML}}$ also supports base events, `wrap`, `guard`, and `choice` combinators. The `wrap` and `guard` combinators construct a complex event from a simpler event by suffixing and prefixing computations, resp. Evaluation of such a complex event is effectively the same as performing a *sequence* of actions encapsulated by the event. From the perspective of reasoning about well-formed executions, `wrap` and `guard` are purely syntactic additions.

Choices are more intriguing. The `choose` combinator operates over a list of events, which when discharged, non-deterministically picks one of the enabled events. If none of the choices are already enabled, one could imagine speculatively discharging every event in a choice, picking one of the enabled events, terminating other events and rolling back the appropriate threads. However, in practice, such a solution would lead to large number of mis-speculations. Hence, $\mathcal{R}^{\text{CML}}$ discharges choices non-speculatively. In order to avoid spurious invocations, negative acknowledgment events (`withNack`) are enabled only after the selection to which they belong is part of a successful well-formedness check.

3.5 Extensions

Our presentation so far has been restricted to speculating only on synchronous sends. Speculation on receives is, in general, not possible since the continuation might depend on the value received. However, if the receive is on a unit channel, speculation has a sensible interpretation. The well-formedness check only needs to ensure that the receive action has been paired up, along with the usual well-formedness checks. Speculating on these kinds of receive actions, which essentially serve as synchronization barriers, is useful, especially during a broadcast operation of the kind described in Figure 1 for receiving acknowledgments.

4 Case Studies

4.1 Online Transaction Processing

Our first case study considers a CML implementation of an online transaction processing (OLTP) system. Resources are modeled as actors that communicate to clients via message-passing, each protected by a lock server. A transaction can span multiple resources, and is implemented pessimistically. Hence, a transaction must all relevant locks before starting its computation. We can use our relaxed execution semantics to allow transactions to effectively execute optimistically, identifying and remediating conflicting transactions *post facto*; the key idea is to model conflicting transactions as an ill-formed execution. We implement each lock server as a single CML thread, whose kernel is:

```
fun lockServer (lockChan: unit chan) (unlockChan: unit chan) =
  (recv lockChan; recv unlockChan; lockServer lockChan unlockChan)
```

which protects a single *resource* by ensuring atomic access. It is up to the application to ensure that the lock servers are correctly used, and when obtaining multiple locks, locks are sorted to avoid deadlocks.

In the absence of contention, the involvement of the lock server adds unnecessary overhead. By communicating with `lockChan` asynchronously, we can allow the client (the thread performing the transaction), to concurrently proceed with obtaining other locks or executing the transaction. However, the transactional guarantees are lost in this case. Under ℞$^{\text{CML}}$ such serializability violation shows up as a cycle in the happens-before dependence graph. ℞$^{\text{CML}}$ rejects such executions, causing the transaction to abort, and re-execute non-speculatively.

For our evaluation, we implemented a distributed version of this program (`vacation`) taken from the STAMP benchmark suite [4]. To adapt the benchmark for a distributed environment, we partitioned resources into 16 *shards*, each protected by a lock server. The workload was setup for moderate contention, and each transaction involves 10 operations. The shards were spread across 16 EC2 M1 large instances within the same EC2 availability zone. The clients were instantiated from all of the different regions on M1 small instances to simulate the latencies involved in a real web-application. A benchmark run involved 10K transactions, spread equally across all of the available clients. Each benchmark run was repeated 5 times.

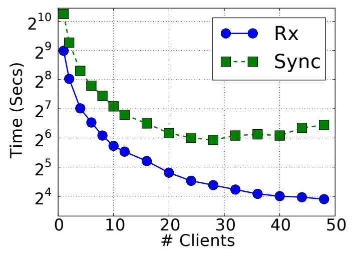


Fig. 4: Performance comparison on distributed `vacation` (OLTP) benchmark. Lower is better.

The performance results are presented in the Figure 4. The number of clients concurrently issuing transaction requests was increased from 1 to 48. ℞$^{\text{CML}}$ is the speculative version, while Sync is the synchronous, non-speculative variant. The 1-client Sync version took 1220 seconds to complete. For comparison, we extended the original C version with a similar shared distribution structure. This run was 1.3× faster than the CML baseline. The benchmark execution under ℞$^{\text{CML}}$ scales much better than the Sync version due to optimistic transactions. With 48 clients, ℞$^{\text{CML}}$ version was 5.8× faster than then Sync version. Under ℞$^{\text{CML}}$, the number of transaction conflicts does increase with the number of clients. With 48 clients, 9% of the transactions executed under ℞$^{\text{CML}}$ were tagged as conflicting and re-executed non-speculatively. This does not, however, adversely affect scalability.

4.2 Collaborative Editing

Our next case study is a real-time, decentralized collaborative editing tool. Typically, such commercial offerings such as Google Docs, Apache Wave, EtherPad, etc,utilize a centralized server to coordinate between the authors. Not only does the server eventually become a bottleneck, but service providers also need to store a copy of the document, along with other personal information, which is undesirable. We consider a fully decentralized solution, in which authors works on a local copy of the shared document for responsiveness, with updates from other authors added incrementally to the working copy. Although replicas are allowed to diverge, they are expected to converge eventually. This convergence is achieved through *operational transformation* [22]. Dealing with operational transformation in the absence of a centralized server is tricky [16], and commercial collaborative editing services like Google Wave impose additional restrictions with respect to the frequency of remote updates [24] in order to build a tractable implementation.

We simplify the design by performing *causal atomic broadcast* when sending updates to the replicas. Causal atomic broadcast ensures that the updates are applied on all replicas in the same global order, providing a semblance of a single centralized server. Implemented naïvely, i.e., performing the broadcast synchronously, however, is an expensive operation, requiring coordination among all replicas for every broadcast operation compromising responsiveness. Our relaxed execution model overcomes this inefficiency. The key advantage of our system is that the causal atomic broadcast is performed speculatively, allowing client threads to remain responsive.

We use a collaborative editing benchmark generator described in [14] to generate a random trace of operations, based on parameters such as trace length, percentage of insertions, deletions, number of replicas, local operation delay, etc. Our benchmarking trace contains 30K operations, 85%(15%) of which are insertions(deletions), and 20% of which are concurrent operations. We insert a 25 ms delay between two consecutive local operations to simulate user-interaction. Updates from each replica is causal atomically broadcasted every 250 ms. Each replica is represented by a $\mathbb{R}^{\text{CML}}$ instance placed in widely distributed Amazon EC2 availability zones chosen to capture the geo-distributed nature of collaborative editing. The average inter-instance latency was 173 ms, with a standard deviation of 71.5. Results are reported as the average of five runs.

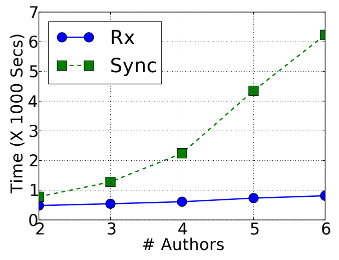


Fig. 5: Performance comparison on collaborative editing benchmark. Lower is better.

We consider the time taken by a collaborative editing session to be the time between the first operation generation and the completion of the last broadcast operation, at which point the documents at every replica would have converged. Figure 5 shows results with respect to total running time. Sync represents an ordinary CML execution, while $\mathbb{R}^{\text{CML}}$ represents our new implementation. With 2-authors, $\mathbb{R}^{\text{CML}}$ version took 485 seconds to complete, and was 37% faster than the synchronous version. As we increase the number of concurrent authors, the number of communication actions per broadcast operation increases. Hence, we expect the benchmark run to take longer to complete. The non-speculative version scales poorly due to the increasing number of synchronizations involved in the broadcast operations. Indeed, Sync is 7.6× slower than $\mathbb{R}^{\text{CML}}$ when there are six concurrent authors. Not surprisingly, $\mathbb{R}^{\text{CML}}$ also takes longer to complete a run as we increase the number of concurrent authors. This is because of increasing communication actions per broadcast as well as increase in mis-speculations. However, with six authors, it only takes 1.67× longer to complete the session when compared to having just two authors, and illustrates the utility of speculative communication.

5 Related Work

Causal-ordering of messages is considered an important building block [2] for distributed applications. Similar to our formulation, Charron-Bost *et al.* [5] develop an axiomatic formulation for causal-ordered communication primitives, although their focus is on characterizing communication behavior and verifying communication protocols, rather than latency hiding. Speculative execution has been shown to be beneficial in other circumstances under high latency environments such as distributed file systems [17], asynchronous virtual machine replication [6], state machine replication [25], deadlock detection [13] etc., although we are unaware of other attempts to use it for transparently converting synchronous operations to asynchronous ones.

Besides Erlang [1], there are also several distributed implementations of functional languages that have been proposed [23, 20]. More recently, Cloud Haskell [11] has been proposed for developing distributed Haskell programs. While all these systems deal with issues such as type-safe serialization and fault tolerance central to any distributed language, ℞$^{\text{CML}}$'s focus is on enforcing equivalence between synchronous and asynchronous evaluation. The formalization used to establish this equivalence is inspired by work in language and hardware memory models [21, 7, 3]. These efforts, however, are primarily concerned with visibility of shared-memory updates, rather than correctness of relaxed message-passing behavior. Thus, while language memory models [3, 7] are useful in reasoning about compiler optimizations, our relaxed communication model reasons about safe asynchronous manifestations of synchronous protocols.

Transactional events(TE) [8, 9] combine first-class synchronous events with an all-or-nothing semantics. They are strictly more expressive than CML, although such expressivity comes at the price of an expensive runtime search procedure to find a satisfiable schedule. Communicating memory transactions (CMT) [12] also uses speculation to allow asynchronous message-passing communication between shared-memory transactions, although CMT does not enforce any equivalence with a synchronous execution. Instead, mis-speculations only arise because of a serializability violation on memory.

6 Conclusions and Future Work

CML provides a simple, expressive, and composable set of synchronous event combinators that facilitate concurrent programming, albeit at the price of performance, especially in high-latency environments. This paper shows how to regain this performance by transparently implementing synchronous operations asynchronously, effectively treating them as speculative actions. We formalize the conditions under which such a transformation is sound, and describe a distributed implementation of CML called ℞$^{\text{CML}}$ that incorporates these ideas. Our reported case studies illustrate the benefits of our approach, and provide evidence that ℞$^{\text{CML}}$ is a basis upon which we can build clean, robust, and efficient distributed CML programs. An important area of future work is the integration of fault tolerance into the system. Note that the state of a failed instance can be recovered from the dependence graph (which includes all saved continuations), enabling the use of checkpoint restoration and replay as a feasible response mechanism to node failures.

References

[1] Armstrong, J., Virding, R., Wikström, C., Williams, M.: Concurrent Programming in Erlang (2nd ed.) (1996)

[2] Birman, K.P., Joseph, T.A.: Reliable Communication in the Presence of Failures. ACM Trans. Comput. Syst. 5(1), 47–76 (Jan 1987)

[3] Boehm, H.J., Adve, S.V.: Foundations of the C++ Concurrency Memory Model. In: PLDI. pp. 68–78 (2008)

[4] Cao Minh, C., Chung, J., Kozyrakis, C., Olukotun, K.: STAMP: Stanford transactional applications for multi-processing. In: IISWC (2008)
[5] Charron-Bost, B., Mattern, F., Tel, G.: Synchronous, Asynchronous, and Causally Ordered Communication. Distrib. Comput. 9(4), 173–191 (1996)
[6] Cully, B., Lefebvre, G., Meyer, D., Feeley, M., Hutchinson, N., Warfield, A.: Remus: High Availability via Asynchronous Virtual Machine Replication. In: NSDI. pp. 161–174 (2008)
[7] Demange, D., Laporte, V., Zhao, L., Jagannathan, S., Pichardie, D., Vitek, J.: Plan B: A Buffered Memory Model for Java. In: POPL. pp. 329–342 (2013)
[8] Donnelly, K., Fluet, M.: Transactional Events. In: ICFP. pp. 124–135 (2006)
[9] Effinger-Dean, L., Kehrt, M., Grossman, D.: Transactional Events for ML. In: ICFP. pp. 103–114 (2008)
[10] Elnozahy, E.N.M., Alvisi, L., Wang, Y.M., Johnson, D.B.: A survey of rollback-recovery protocols in message-passing systems. ACM Comput. Surv. 34(3), 375–408 (Sep 2002)
[11] Epstein, J., Black, A.P., Peyton-Jones, S.: Towards Haskell in the Cloud. In: Haskell Symposium. pp. 118–129 (2011)
[12] Lesani, M., Palsberg, J.: Communicating Memory Transactions. In: PPoPP. pp. 157–168 (2011)
[13] Li, T., Ellis, C.S., Lebeck, A.R., Sorin, D.J.: Pulse: A Dynamic Deadlock Detection Mechanism Using Speculative Execution. In: USENIX ATC. p. 3144 (2005)
[14] Martin, S., Ahmed-Nacer, M., Urso, P.: Controlled Conflict Resolution for Replicated Documents. In: CollaborateCom. pp. 471–480 (2012)
[15] MultiMLton: MLton for Scalable Multicore Architectures (2013), `http://multimlton.cs.purdue.edu`
[16] Nichols, D.A., Curtis, P., Dixon, M., Lamping, J.: High-latency, Low-bandwidth Windowing in the Jupiter Collaboration System. In: UIST. pp. 111–120 (1995)
[17] Nightingale, E.B., Chen, P.M., Flinn, J.: Speculative Execution in a Distributed File System. In: SOSP. pp. 191–205 (2005)
[18] Reppy, J.: Concurrent Programming in ML. Cambridge University Press (2007)
[19] Sarkar, S., Sewell, P., Nardelli, F.Z., Owens, S., Ridge, T., Braibant, T., Myreen, M.O., Alglave, J.: The Semantics of x86-CC Multiprocessor Machine Code. In: POPL. pp. 379–391 (2009)
[20] Sewell, P., Leifer, J.J., Wansbrough, K., Nardelli, F.Z., Allen-Williams, M., Habouzit, P., Vafeiadis, V.: Acute: High-level Programming Language Design for Distributed Computation. J. Funct. Program. 17(4-5), 547–612 (2007)
[21] Sewell, P., Sarkar, S., Owens, S., Nardelli, F.Z., Myreen, M.O.: x86-TSO: A Rigorous and Usable Programmer's Model for x86 Multiprocessors. Commun. ACM 53(7), 89–97 (Jul 2010)
[22] Suleiman, M., Cart, M., Ferrié, J.: Serialization of Concurrent Operations in a Distributed Collaborative Environment. In: GROUP. pp. 435–445 (1997)
[23] Wakita, K., Asano, T., Sassa, M.: D'Caml: Native Support for Distributed ML Programming in Heterogeneous Environment. In: Euro-Par. pp. 914–924 (1999)
[24] Wang, D., Mah, A., Lassen, S.: Operational Transformation (2010), `http://www.waveprotocol.org/whitepapers/operational-transform`
[25] Wester, B., Cowling, J.A., Nightingale, E.B., Chen, P.M., Flinn, J., Liskov, B.: Tolerating Latency in Replicated State Machines Through Client Speculation. In: NSDI. pp. 245–260 (2009)
[26] ZeroMQ: The Intelligent Transport Layer (2013), `http://www.zeromq.org`
[27] Ziarek, L., Jagannathan, S.: Lightweight Checkpointing for Concurrent ML. Journal of Functional Programming 20(2), 137–173 (2010)
[28] Ziarek, L., Sivaramakrishnan, K., Jagannathan, S.: Composable Asynchronous Events. In: PLDI. pp. 628–639 (2011)