℞-CML
: A Prescription for Safely Relaxing Synchrony

KC Sivaramakrishnan

Lukasz Ziarek  
SUNY Buffalo

Suresh Jagannathan
Purdue University
Introduction

Two often competing goals when designing and implementing concurrency abstractions
Introduction

Two often competing goals when *designing* and *implementing* concurrency abstractions

- Simplicity
- Safety
- Performance
- Functionality
Introduction

Two often competing goals when *designing* and *implementing* concurrency abstractions

Always desirable to marry the two whenever possible
Big Picture

• Functional language + Synchronous message passing
  ★ Communication = Data transfer + Synchronization
Big Picture

- Functional language + Synchronous message passing
  - Communication = Data transfer + Synchronization
- However, in the cloud,
Big Picture

- Functional language + Synchronous message passing
  - Communication = Data transfer + Synchronization
- However, in the cloud,

  ★ Explicit asynchrony complicates reasoning

Synchrony

latency
Big Picture

- Functional language + Synchronous message passing
  ★ Communication = Data transfer + Synchronization
- However, in the cloud,
  ★ Explicit asynchrony complicates reasoning

Can we discharge synchronous communications asynchronously while ensuring observable equivalence?
Goal
1. Formalize the conditions under which the following equivalence holds:

\[ \left[ \text{send} \ (c, v) \right]_k \equiv \left[ \text{asend} \ (c, v) \right]_k \]
Goal

1. Formalize the conditions under which the following equivalence holds:

\[ \left[ \text{send}(c, v) \right]^k \equiv \left[ \text{asend}(c, v) \right]^k \]

2. A cloud infrastructure + speculative execution framework

   a. discharges synchronous sends asynchronously
   b. detects when the equivalence fails, and
   c. repairs failed executions
Context

- A distributed extension of MultiMLton - MLton for scalable architectures
Context

- A distributed extension of MultiMLton - MLton for scalable architectures
- Parallel extension of Concurrent ML
  ★ Dynamic lightweight threads
  ★ Synchronous message passing
  ★ First-class events
    ✦ Composable synchronous protocols
Context

- A distributed extension of MultiMLton - MLton for scalable architectures
- Parallel extension of Concurrent ML
  - Dynamic lightweight threads
  - Synchronous message passing
  - First-class events
    - Composable synchronous protocols

```ml
val channel : unit -> 'a chan
val spawn : (unit -> unit) -> thread_id
val send : 'a chan * 'a -> unit
val recv : 'a chan -> 'a
val sendEvt : 'a chan * 'a -> unit event
val recvEvt : 'a chan -> 'a event
val sync : 'a event -> 'a
val never : 'a event
val alwaysEvt : 'a -> 'a event
val wrap : 'a event -> ('a -> 'b) -> 'b event
val guard : (unit -> 'a event) -> 'a event
val choose : 'a event list -> 'a event
...
```
Basic Idea (1)

Synchronous Execution

T1
send(c1, v1)
f()
send(c2, v2)

T2
recv(c2)
g()
recv(c1)

T3
send(c2, v3)
h()
recv(c2)

T1
send(c1, v1)
\downarrow
f()
\downarrow
send(c2, v2)

T2
recv(c2)
\downarrow
g()
\downarrow
recv(c1)

T3
send(c2, v3)
\downarrow
h()
\downarrow
recv(c2)
Basic Idea (1)

T1
send(c1,v1) f()
send(c2,v2)

T2
recv(c2) g()
recv(c1)

T3
send(c2,v3) h()
recv(c2)

Synchronous Execution

T1
send(c1,v1) → f()
send(c2,v2) → g()

T2
recv(c2) ←→ send(c2,v3)

T3
h()
recv(c2)
been successfully transmitted to a receiver. The cost of synchrony evaluation of a message-send is guaranteed that the data being sent has been safely converted into an asynchronous action in which the re-
v3
ıvely replacing synchronous communication asend(c1,v1), and send(c2,v2) with an asynchronous one is not usually meaning-preserving as the resulting execution nonetheless exhibits behavior observably equiva-
ment to one in which all communication is performed synchronously. Several realistic case studies deployed in a cloud environment demonstrate the utility of our approach.

Our goal is to retain the expressivity and simplicity of CML's syn-
chronous operations in writing concurrent programs and reasoning about the underlying runtime the freedom to allow a sender to communi-
tations to the core set of event combinators CML supports, is to give extensions can be used to gain performance, they sacrifice the sim-
plexity provided by synchronous communication in favor of a more complex and sophisticated set of primitives.

One way to enhance performance without requiring new addi-

The use of asynchrony can help reclaim performance, it also

Unfortunately, na-

Asynchronous Execution

Synchronous Execution

T1
send(c1,v1)
recv(c2)
send(c2,v2)
T2
recv(c2)
g(v2)
f()
h(v3)
T3
send(c2,v3)
recv(c2)
recv(c1)

Figure 1: Performing the first addition to the first send on T3, a

\[ send \text{(c1,v1)} \]

\[ f() \]

\[ send \text{(c2,v2)} \]

\[ \text{T1} \]

\[ \text{T2} \]

\[ \text{T3} \]

\[ \text{recv} \text{(c2)} \]

\[ g() \]

\[ \text{recv} \text{(c1)} \]

\[ h() \]

\[ \text{send} \text{(c2,v3)} \]

\[ \text{recv} \text{(c2)} \]
Synchronous Execution

**Basic Idea (1)**

T1
send(c1,v1)
\[f()\]
send(c2,v2)

T2
recv(c2)
\[g()\]
recv(c1)

T3
send(c2,v3)
\[h()\]
recv(c2)

T1
\[send(c1,v1)\]  T2 \[recv(c2)\]  T3 \[send(c2,v3)\]

\[f()\]  \[g()\]  \[h()\]

\[send(c2,v2)\]  \[recv(c1)\]  \[recv(c2)\]

\[a\]  \[b\]  \[c\]
Basic Idea (2)

Asynchronous Execution

T1
send(c1,v1)  
   f()  
   send(c2,v2)

T2
recv(c2)  
   g()  
   recv(c1)

T3
send(c2,v3)  
   h()  
   recv(c2)

T1
\textbf{asend}(c1,v1)

T2
recv(c2)  
   g()  
   recv(c1)

T3
send(c2,v3)
Basic Idea (2)

Asynchronous Execution

<table>
<thead>
<tr>
<th>T1</th>
<th>T2</th>
<th>T3</th>
</tr>
</thead>
<tbody>
<tr>
<td>send(c1,v1)</td>
<td>recv(c2)</td>
<td>send(c2,v3)</td>
</tr>
<tr>
<td>f()</td>
<td>g()</td>
<td>h()</td>
</tr>
<tr>
<td>send(c2,v2)</td>
<td>recv(c1)</td>
<td>recv(c2)</td>
</tr>
</tbody>
</table>

T1 performs a synchronous send on channel c1, v1, followed by f().

Asynchronous variant:

- send(c1,v1) becomes asend(c1,v1)
- f() becomes a continuation

This allows T1 to perform another synchronous operation send(c2,v2) and f() to be evaluated without waiting for a matching receiver.

After asend is completed, T2 can receive from channel c1 and evaluate g().

T3 sends c2,v3 without waiting for a matching receiver, allowing execution to proceed asynchronously.
Basic Idea (2)

Asynchronous Execution

- send(c1,v1)
- f()
- send(c2,v2)

T1

<table>
<thead>
<tr>
<th>recv(c2)</th>
</tr>
</thead>
<tbody>
<tr>
<td>g()</td>
</tr>
<tr>
<td>recv(c1)</td>
</tr>
</tbody>
</table>

T2

| send(c2,v3) |
| h()         |
| recv(c2)    |

T3

$asend(c1,v1)$

$send(c2,v2)$

$f()$

$g()$

$recv(c1)$

$recv(c2)$

$send(c2,v3)$

Unfortunately, na...
Asynchronous Execution

Basic Idea (2)
Synchronous evaluation never results in cyclic dependence

★ Cyclic dependence => divergent behavior w.r.t synchronous evaluation
Example: Distributed Group Chat
Example: Distributed Group Chat

- No central server + Preserve causal dependence
Example: Distributed Group Chat

- No central server + Preserve causal dependence
  - **Causal broadcast primitive**: If message A is generated as a response to message B, then A is delivered after B at every site
Example: Distributed Group Chat

- No central server + Preserve causal dependence
  - **Causal broadcast primitive**: If message A is generated as a response to message B, then A is delivered after B at every site
  - Asynchronous messaging => explicitly manage vector clocks and buffering on receiver side
Example: Distributed Group Chat

- No central server + Preserve causal dependence
  - **Causal broadcast primitive**: If message A is generated as a response to message B, then A is delivered after B at every site
  - Asynchronous messaging => explicitly manage vector clocks and buffering on receiver side
  - Synchronous messaging => directly using point-to-point messaging
Example: Distributed Group Chat

- No central server + Preserve causal dependence
  - **Causal broadcast primitive**: If message A is generated as a response to message B, then A is delivered after B at every site
  - Asynchronous messaging => explicitly manage vector clocks and buffering on receiver side
  - Synchronous messaging => directly using point-to-point messaging

```
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
```

- synchronously send values
  - prevent receivers from proceeding until all members have received the value

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.
Example: Distributed Group Chat

- No central server + Preserve causal dependence
  - **Causal broadcast primitive**: If message A is generated as a response to message B, then A is delivered after B at every site
  - Asynchronous messaging => explicitly manage vector clocks and buffering on receiver side
  - Synchronous messaging => directly using point-to-point messaging

```ml
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
```

- Simple but likely to be inefficient - phase 2 is a global barrier!
Example: Distributed Group Chat

- No central server + Preserve causal dependence
  - **Causal broadcast primitive**: If message A is generated as a response to message B, then A is delivered after B at every site
  - Asynchronous messaging => explicitly manage vector clocks and buffering on receiver side
  - Synchronous messaging => directly using point-to-point messaging

```ml
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
```

- Simple but likely to be inefficient - **phase 2 is a global barrier!**
  - Discharging asynchronously breaks causal ordering
Example: Distributed Group Chat

- No central server + Preserve causal dependence
  - **Causal broadcast primitive**: If message A is generated as a response to message B, then A is delivered after B at every site
  - Asynchronous messaging => explicitly manage vector clocks and buffering on receiver side
  - Synchronous messaging => directly using point-to-point messaging

```
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
```

- Simple but likely to be inefficient - phase 2 is a global barrier!
  - Discharging asynchronously breaks causal ordering
  - **Our idea**: program synchronously, discharge asynchronously, detect and remediate causal ordering violations
Example: Distributed Group Chat

- A distributed group chat program = \{Node\}
- Node = MultiMLton process = \{CML threads\}
<table>
<thead>
<tr>
<th>Display</th>
<th>IP</th>
<th>Daemon</th>
</tr>
</thead>
<tbody>
<tr>
<td>Display</td>
<td>IP</td>
<td>Daemon</td>
</tr>
</tbody>
</table>

Distributed Group Chat - Run 1
### Distributed Group Chat - Run 1

<table>
<thead>
<tr>
<th>Display</th>
<th>IP</th>
<th>Daemon</th>
</tr>
</thead>
<tbody>
<tr>
<td>show (X)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>bcast (X)</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Display</th>
<th>IP</th>
<th>Daemon</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Distributed Group Chat - Run 1

Display | IP | Daemon
---|---|---

show (X)

X

bcast (X)

Display | IP | Daemon
---|---|---

show (Y)

Y

bcast (Y)
Distributed Group Chat - Run 1

Display | IP | Daemon
---|---|---
X | | show (X)
| bcast (X) | |

Display | IP | Daemon
---|---|---
Y | | show (Y)
| brecv (X) | |
| bcast (Y) | |

Tuesday, January 21, 14
Distributed Group Chat - Run 1

<table>
<thead>
<tr>
<th>Display</th>
<th>IP</th>
<th>Daemon</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

- `show (X)`
- `bcast (X)`
- `brecv (Y)`

<table>
<thead>
<tr>
<th>Display</th>
<th>IP</th>
<th>Daemon</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

- `show (Y)`
- `bcast (Y)`
- `brecv (X)`
Distributed Group Chat - Run 1

Display | IP | Daemon
---|---|---

show (X) → bcast (X) → X

Display | IP | Daemon
---|---|---

show (Y) → brecv (Y) → Y

X → bcast (Y) → Y

Y → show (X) → X
Distributed Group Chat - Run 1

```
show (X) brecv (Y)
bcast (X) show (Y)
```

```
show (Y) brecv (X)
bcast (Y) show (X)
```
• Observations

★ X and Y independently generated => No causal dependence between bcast (X) and bcast (Y)

• No Cycles => Correct execution!
Distributed Group Chat - Run 2

<table>
<thead>
<tr>
<th>Display</th>
<th>IP</th>
<th>Daemon</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Distributed Group Chat - Run 2

<table>
<thead>
<tr>
<th>Display</th>
<th>IP</th>
<th>Daemon</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Display</th>
<th>IP</th>
<th>Daemon</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Display</th>
<th>IP</th>
<th>Daemon</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Show (X)

Bcast (X)
Distributed Group Chat - Run 2

Display | IP | Daemon
--- | --- | ---
| | | show (X)

Presume causal dependence $X \rightarrow Y$

Display | IP | Daemon
--- | --- | ---
| | | brecv (X)

Display | IP | Daemon
--- | --- | ---
| | | show (X)
### Distributed Group Chat - Run 2

#### Table:

<table>
<thead>
<tr>
<th>Display</th>
<th>IP</th>
<th>Daemon</th>
</tr>
</thead>
<tbody>
<tr>
<td>X</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

- `show (X)`
- `bcast (X)`

<table>
<thead>
<tr>
<th>Display</th>
<th>IP</th>
<th>Daemon</th>
</tr>
</thead>
<tbody>
<tr>
<td>Y</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

- `show (Y)`
- `bcast (Y)`

<table>
<thead>
<tr>
<th>Display</th>
<th>IP</th>
<th>Daemon</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

- `brecv (X)`
- `show (X)`

**Example Events:**

- `X` displays `show (X)` and sends `bcast (X)`.
- `Y` receives `brecv (X)` and displays `show (Y)`.
- `Y` sends `bcast (Y)`.
Distributed Group Chat - Run 2

<table>
<thead>
<tr>
<th>Display</th>
<th>IP</th>
<th>Daemon</th>
</tr>
</thead>
<tbody>
<tr>
<td>X</td>
<td></td>
<td></td>
</tr>
<tr>
<td>show (X)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>bcast (X)</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Display</th>
<th>IP</th>
<th>Daemon</th>
</tr>
</thead>
<tbody>
<tr>
<td>X</td>
<td></td>
<td></td>
</tr>
<tr>
<td>show (X)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>brecv (X)</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Display</th>
<th>IP</th>
<th>Daemon</th>
</tr>
</thead>
<tbody>
<tr>
<td>Y</td>
<td></td>
<td></td>
</tr>
<tr>
<td>show (Y)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>bcast (Y)</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Display</th>
<th>IP</th>
<th>Daemon</th>
</tr>
</thead>
<tbody>
<tr>
<td>Y</td>
<td></td>
<td></td>
</tr>
<tr>
<td>show (Y)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>brecv (Y)</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Tuesday, January 21, 14
Distributed Group Chat - Run 2

<table>
<thead>
<tr>
<th>Display</th>
<th>IP</th>
<th>Daemon</th>
</tr>
</thead>
<tbody>
<tr>
<td>X</td>
<td></td>
<td></td>
</tr>
<tr>
<td>show (X)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>bcast (X)</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Display</th>
<th>IP</th>
<th>Daemon</th>
</tr>
</thead>
<tbody>
<tr>
<td>X</td>
<td></td>
<td></td>
</tr>
<tr>
<td>show (X)</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Display</th>
<th>IP</th>
<th>Daemon</th>
</tr>
</thead>
<tbody>
<tr>
<td>Y</td>
<td></td>
<td></td>
</tr>
<tr>
<td>show (Y)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>bcast (Y)</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Display</th>
<th>IP</th>
<th>Daemon</th>
</tr>
</thead>
<tbody>
<tr>
<td>X</td>
<td></td>
<td></td>
</tr>
<tr>
<td>show (X)</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Tuesday, January 21, 14
Distributed Group Chat - Run 2

Causal dependence violated!
Distributed Group Chat - Run 2

Cycle Detected!
Distributed Group Chat - Results

- Simulation on 3 geo-distributed Amazon EC2 instances
- Measure time between message initiation and receipt by all parties over 1000 iterations

<table>
<thead>
<tr>
<th>Execution</th>
<th>Avg. time (ms)</th>
<th>Errors</th>
</tr>
</thead>
<tbody>
<tr>
<td>Sync</td>
<td>1540</td>
<td>0</td>
</tr>
<tr>
<td>Unsafe Async</td>
<td>520</td>
<td>7</td>
</tr>
<tr>
<td>Safe Async (R&lt;sup&gt;CML&lt;/sup&gt;)</td>
<td>533</td>
<td>0</td>
</tr>
</tbody>
</table>
Formalization Overview
Formalization Overview

- Reason *axiomatically* about executions (relaxed or otherwise)
  - Similar to formalizations used in relaxed memory models
  - Declarative characterization of (relaxed) CML behavior
Formalization Overview

• Reason *axiomatically* about executions (relaxed or otherwise)
  ★ Similar to formalizations used in relaxed memory models
  ★ Declarative characterization of (relaxed) CML behavior

• *Actions + happens-before relation*
  ★ Captures visibility and dependence properties
Formalization Overview

- Reason *axiomatically* about executions (relaxed or otherwise)
  - Similar to formalizations used in relaxed memory models
  - Declarative characterization of (relaxed) CML behavior
- **Actions + happens-before relation**
  - Captures visibility and dependence properties
- Happens-before is intentionally *relaxed*: may define more behaviors than possible in CML
  - Strengthen the relation with *well-formedness* conditions
Actions and Execution

- **Actions:**

\[
\mathcal{A} := \begin{align*}
b_t & \quad \text{(thread t starts)} \\
e_t & \quad \text{(thread t ends)} \\
\ell_t^m & \quad \text{(thread t detects thread t’ has terminated)} \\
\ell_t^m & \quad \text{(thread t creates a new thread t’)} \\
s_t^m & \quad \text{(thread t sends value v on channel c)} \\
r_t^m & \quad \text{(thread t receives a value on channel c)} \\
p_t^m & \quad \text{(thread t outputs an observable value v)}
\end{align*}
\]

- **Execution:**

\[
E := \langle P, A, \rightarrow_{po}, \rightarrow_{co} \rangle
\]

- **Diagram Notes:**
  - **Program:** a set of actions
  - **Program order:** sequential actions of a thread
  - **Communication order:** relates matching communication actions
Communication and Thread Dependence

- Synchronous communication $\rightarrow$ communication order is symmetric:

  $$ a \rightarrow_{co} b \implies b \rightarrow_{co} a $$

- Thread dependence order:

  $$ \alpha \rightarrow_{td} \beta \text{ if:} $$

  (1) $\alpha = f_{t'}^m t'$ and $\beta = b_{t'}$ or

  (2) $\alpha = e_t$ and $\beta = j_{t'}^m t$
Happens-before Relation

- Establishes both intra- and inter-thread dependences:

\[ \rightarrow_{hb} = (\rightarrow_{po} \cup \rightarrow_{td} \cup \{(\alpha, \beta) \mid \alpha \rightarrow_{co} \alpha' \land \alpha' \rightarrow_{po} \beta\} \cup \{(\beta, \alpha) \mid \beta \rightarrow_{po} \alpha' \land \alpha' \rightarrow_{co} \alpha\})^+ \]
Happens-before Relation

- Establishes both intra- and inter-thread dependences:

\[ \rightarrow_{hb} = (\rightarrow_{po} \cup \rightarrow_{td} \cup \{(\alpha, \beta) | \alpha \rightarrow_{co} \alpha' \land \alpha' \rightarrow_{po} \beta\}) \cup \{(\beta, \alpha) | \beta \rightarrow_{po} \alpha' \land \alpha' \rightarrow_{co} \alpha\} \]

- Two actions not related by happens-before relation are said to be concurrent

★ A send action and its matching receive action are concurrent!
Happens-before Relation

- Establishes both intra- and inter-thread dependences:

\[ \rightarrow_{hb} = (\rightarrow_{po} \cup \rightarrow_{td} \cup \{(\alpha, \beta) \mid \alpha \rightarrow_{co} \alpha' \land \alpha' \rightarrow_{po} \beta\} \cup \{(\beta, \alpha) \mid \beta \rightarrow_{po} \alpha' \land \alpha' \rightarrow_{co} \alpha\})^+ \]

- Two actions not related by happens-before relation are said to be concurrent

★ A send action and its matching receive action are concurrent!

\[ \xrightarrow{s^m_t c, v} \xleftarrow{r^{m'}_t c} \xrightarrow{co} \xleftarrow{po} \xrightarrow{hb} \beta \]
Happens-before Relation

- Establishes both intra- and inter-thread dependences:

\[ \rightarrow_{hb} = (\rightarrow_{po} \cup \rightarrow_{td} \cup \{(\alpha, \beta) | \alpha \rightarrow_{co} \alpha' \land \alpha' \rightarrow_{po} \beta\} \cup \{(\beta, \alpha) | \beta \rightarrow_{po} \alpha' \land \alpha' \rightarrow_{co} \alpha\})^+ \]

- Two actions not related by happens-before relation are said to be concurrent

★ A send action and its matching receive action are concurrent!
Example

- Assume T1 spawns T2 and T3
- Let f, g, h = print 1, print 2, print 3
• Assume T1 spawns T2 and T3
• Let f, g, h = print 1, print 2, print 3
Example

- Assume T1 spawns T2 and T3
- Let f, g, h = print 1, print 2, print 3
Obs (Well-formed Execution of P) ∈ \{Obs (CML Execution of P)\}
Well-formed Executions

Obs (Well-formed Execution of P) $\in \{\text{Obs (CML Execution of P)}\}$

1. Sensible intra-thread semantics
2. Acyclic happens-before relation
3. No outstanding communication action preceding an observable action
Track executions to see if they become ill-formed (rollback) or turn into CML executions (commit)
Track executions to see if they become *ill-formed (rollback)* or turn into CML executions (*commit*)
Track executions to see if they become ill-formed (rollback) or turn into CML executions (commit)
Well-formedness -> Speculation

Track executions to see if they become ill-formed (rollback) or turn into CML executions (commit)

A well-formed execution that can lead to a CML execution
Well-formedness -> Speculation

Track executions to see if they become ill-formed (rollback) or turn into CML executions (commit)
Well-formedness -> Speculation

Track executions to see if they become ill-formed (rollback) or turn into CML executions (commit)

Not a well-formed execution
Implementation: Overview

- **Rx-CML: RelaXed CML**
  - MultiMLton with distribution support
  - Rx-CML application = {Instances}
  - Supports full CML
  - Built-in serialization (immutable values and function closures)
  - Transport layer is ZeroMQ
Implementation: Overview

- **Rx-CML: RelaXed CML**
  - MultiMLton with distribution support
  - Rx-CML application = {Instances}
  - Supports full CML
  - Built-in serialization (immutable values and function closures)
  - Transport layer is ZeroMQ
- Check the integrity of the speculative actions on-the-fly
Implementation: Overview

- **Rx-CML: RelaXed CML**
  - MultiMLton with distribution support
  - Rx-CML application = \{Instances\}
  - Supports full CML
  - Built-in serialization (immutable values and function closures)
  - Transport layer is ZeroMQ

- Check the integrity of the speculative actions on-the-fly
  - Build a dependence graph that captures happens-before relation
Implementation: Overview

- **Rx-CML: RelaXed CML**
  - MultiMLton with distribution support
  - Rx-CML application = \{Instances\}
  - Supports full CML
  - Built-in serialization (immutable values and function closures)
  - Transport layer is ZeroMQ

- Check the integrity of the speculative actions on-the-fly
  - Build a dependence graph that captures happens-before relation
    - Same structure as an axiomatic execution
check on-the-fly; in other words, we need to build the relations before relation. However, in practice it is necessary to perform this operation with a well-formed, i.e., there are no cycles in the constructed happens-subsumes the behavior of that trace, we can check if that execution would result in a match of the appropriate dependence graph.

Theorem 17.

Definition 15

Intra-thread semantics, we first provide a translation that produces a well-formed antecedent of the CML channel communication. This translation is characterized with the intra-thread semantics as a normal synchronous operation.

Proof Sketch.

For perspicuity, in the definition of operational action is a set of non-silent, non-commit actions in the trace, otherwise, if then

Details are provided in the supplementary material.

State AX, where

Absence of coherent shared memory: Absence can be prevented by using a global barrier for error detection or rollback. Communication between instances is performed through the ZeMLton program. These instances might run on heterogeneous hardware.

A schematic diagram of the application stack is presented in Figure 6. Central to the application consists of multiple instances encapsulated within the definition of well-formedness found in the MulCML complement CML channel communication. User-level threads, Communication Manager, Cycle Detector, Serialization Support, ZeroMQ Pub/Sub, Cloud.

• Rx-CML: RelaXed CML
  • MultiMLton with distribution support
  • Rx-CML application = {Instances}
  • Supports full CML
  • Built-in serialization (immutable values and function closures)
  • Transport layer is ZeroMQ

• Check the integrity of the speculative actions on-the-fly
  • Build a dependence graph that captures happens-before relation
    ✷ Same structure as an axiomatic execution
  • Automatically check dependence graph integrity before an observable action (ref cell accesses, system calls, FFI, etc)
Implementation: Overview

- \textbf{Rx-CML: RelaXed CML}
  - MultiMLton with distribution support
  - Rx-CML application = \{Instances\}
  - Supports full CML
  - Built-in serialization (immutable values and function closures)
  - Transport layer is ZeroMQ

- Check the integrity of the speculative actions on-the-fly
  - Build a \textit{dependence graph} that captures happens-before relation
    - Same structure as an axiomatic execution
  - Automatically check dependence graph integrity before an observable action (ref cell accesses, system calls, FFI, etc)
  - Roll-back ill-formed executions, re-execute non-speculatively

\textit{PURDUE UNIVERSITY}
Channel Consistency (1)

- Single consistent channel image across all instances without coherence?
Channel Consistency (1)

- Single consistent channel image across all instances without coherence?
  - Communication manager thread @ every instance
Channel Consistency (1)

- Single consistent channel image across all instances without coherence?
  - Communication manager thread @ every instance
  - Maintains a replica of CML channel
Channel Consistency (1)

- Single consistent channel image across all instances without coherence?
  - Communication manager thread @ every instance
  - Maintains a replica of CML channel
  - Utilize speculative execution to recover from inconsistent state
Channel Consistency (1)

- Single consistent channel image across all instances without coherence?
  - Communication manager thread @ every instance
  - Maintains a replica of CML channel
  - Utilize speculative execution to recover from inconsistent state

<table>
<thead>
<tr>
<th>Instance 1</th>
<th>Instance 2</th>
<th>Instance 3</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>send(c1,0);</code></td>
<td><code>recv(c1);</code></td>
<td></td>
</tr>
</tbody>
</table>

\{c1:[], c2:[]\}
Channel Consistency (1)

- Single consistent channel image across all instances without coherence?
  - Communication manager thread @ every instance
  - Maintains a replica of CML channel
  - Utilize speculative execution to recover from inconsistent state

```
send(c1,0);
recv(c1);
s(c1,0)
```

Instance 1
```
\{c1: [], c2: []\}
```

Instance 2
```
\{c1: [s(0)], c2: []\}
```

Instance 3
```
\{c1: [s(0)], c2: []\}
```

Tuesday, January 21, 14
Channel Consistency (1)

- Single consistent channel image across all instances without coherence?
  - Communication manager thread @ every instance
  - Maintains a replica of CML channel
  - Utilize speculative execution to recover from inconsistent state
Channel Consistency (1)

- Single consistent channel image across all instances without coherence?
  - Communication manager thread @ every instance
  - Maintains a replica of CML channel
  - Utilize speculative execution to recover from inconsistent state

Instance 1

send(c1,0);

Instance 2

recv(c1);

Instance 3

\{c1:\[s(0)\],c2:\[]\}

- Communication manager thread @ every instance
- Maintains a replica of CML channel
- Utilize speculative execution to recover from inconsistent state
Channel Consistency (1)

- Single consistent channel image across all instances without coherence?
  - Communication manager thread @ every instance
  - Maintains a replica of CML channel
  - Utilize speculative execution to recover from inconsistent state

![Diagram showing channel consistency across instances]

- Instance 1
  - `send(c1,0);`

- Instance 2
  - `recv(c1);`

- Instance 3
  - `match (s(c1,0) + r(c1))`

- Success
Channel Consistency (1)

- Single consistent channel image across all instances without coherence?
  - Communication manager thread @ every instance
  - Maintains a replica of CML channel
  - Utilize speculative execution to recover from inconsistent state

![Diagram of channel consistency process]

- **Instance 1**
  - `send(c1, 0);`
- **Instance 2**
  - `recv(c1);`
- **Instance 3**
  - `request_to_match(c1)`
  - `match (s(c1, 0) + r(c1))`
  - `Success`
  - `Flush unconsumed send!`

- Channel Consistency Image Across All Instances Without Coherence?
  - Communication Manager Thread @ Every Instance
  - Maintains a Replica of CML Channel
  - Utilize Speculative Execution to Recover from Inconsistent State
Channel Consistency (2)

Instance 1

\texttt{send(c2,1);}

Instance 2

\texttt{recv(c2);}

Instance 3

\texttt{recv(c2);}
Channel Consistency (2)

Instance 1

\textbf{send}(c2, 1);

\{c1: [], c2: [s(1)]\}

\textbf{s}(c2, 1)

Instance 2

\textbf{recv}(c2);

\{c1: [], c2: [s(1)]\}

\textbf{s}(c2, 1)

Instance 3

\textbf{recv}(c2);

\{c1: [], c2: [s(1)]\}

\textbf{s}(c2, 1)
Channel Consistency (2)

Instance 1

send(c2,1);

Instance 2
recv(c2);

Instance 3
recv(c2);

Concurrent execution!

{c1: [], c2: [s(1)]}

s(c2,1)

{c1: [], c2: [s(1)]}

s(c2,1)

{c1: [], c2: [s(1)]}

s(c2,1)

{c1: [], c2: [s(1)]}
Channel Consistency (2)

Instance 1

\[\text{send(c2,1);} \]

\[\{c1:[], c2:[s(1)]\}\]

\[\{c1:[], c2:[r2]\}\]

\[\{c1:[], c2:[r2,r3]\}\]

\[\text{request_to_match(c2)}\]

\[\text{s(c2,1)}\]

Instance 2

\[\text{recv(c2);} \]

\[\{c1:[], c2:[s(1)]\}\]

\[\{c1:[], c2:[r2]\}\]

\[\{c1:[], c2:[r2,r3]\}\]

\[\text{request_to_match(c2)}\]

\[\text{s(c2,1)}\]

Instance 3

\[\text{recv(c2);} \]

\[\{c1:[], c2:[s(1)]\}\]

\[\{c1:[], c2:[r2]\}\]

\[\{c1:[], c2:[r2,r3]\}\]

\[\text{request_to_match(c2)}\]

\[\text{s(c2,1)}\]
Channel Consistency (2)

{c1: [], c2: [s(1)]}

{c1: [], c2: [r2]}

{c1: [], c2: [r2, r3]}

send(c2, 1);

recv(c2);

recv(c2);

{c1: [], c2: [s(1)]}

request_to_match(c2)

s(c2, 1)

{c1: [], c2: [s(1)]}

Instance 1

Instance 2

Instance 3

Concurrent execution!

Lost exactly once semantics
Channel Consistency (2)

Instance 1
\[ \text{send}(c2,1); \]
\[ \{c1:[], c2:[s(1)]\} \]
\[ \{c1:[], c2:[r2]\} \]
\[ \{c1:[], c2:[r2,r3]\} \]

Instance 2
\[ \text{recv}(c2); \]
\[ \text{recv}(c2); \]
\[ \{c1:[], c2:[s(1)]\} \]
\[ \text{request_to_match}(c2) \]
\[ \{c1:[], c2:[r2,r3]\} \]

Instance 3
\[ \text{recv}(c2); \]
\[ \{c1:[], c2:[s(1)]\} \]
\[ \text{request_to_match}(c2) \]
\[ \{c1:[], c2:[r2,r3]\} \]

Concurrent execution!
Lost exactly once semantics

First-come first-match
Channel Consistency (2)

Instance 1

send(c2, 1);

{c1: [], c2: [s(1)]}

{c1: [], c2: [r2]}

{c1: [], c2: [r2, r3]}

First-come first-match

Request_to_match(c2)

match (s(c2, 1) + r2(c2))

Instance 2

recv(c2);

{c1: [], c2: [s(1)]}

request_to_match(c2)

Instance 3

recv(c2);

{c1: [], c2: [s(1)]}

Concurrent execution!

recv(c2);

{c1: [], c2: []}

s(c2, 1)

{c1: [], c2: []}

s(c2, 1)

{c1: [], c2: []}

{c1: [], c2: [s(1)]}

match (s(c2, 1) + r2(c2))

Success

{c1: [], c2: []}

{c1: [], c2: [s(1)]}

Mis-speculation!
Speculative Execution
Speculative Execution

- Consistent, replicated dependence graph @ each instance
  - *Snoop on match messages* from communication manager
  - Broadcast thread spawn and thread join messages
Speculative Execution

- Consistent, replicated dependence graph @ each instance
  - *Snoop on match messages* from communication manager
  - Broadcast thread spawn and thread join messages
- Well-formedness check *is local to the instance!*
  - GC dependence graph on successful well-formedness check
Speculative Execution

- Consistent, replicated dependence graph @ each instance
  - *Snoop on match messages* from communication manager
  - Broadcast thread spawn and thread join messages
- Well-formedness check *is local to the instance!*
  - GC dependence graph on successful well-formedness check
- Automatic checkpointing
  - 1 continuation per thread
  - *Uncoordinated!* - thread local - does not require barriers
Speculative Execution

- Consistent, replicated dependence graph @ each instance
  - *Snoop on match messages* from communication manager
  - Broadcast thread spawn and thread join messages
- Well-formedness check *is local to the instance!*
  - GC dependence graph on successful well-formedness check
- Automatic checkpointing
  - 1 continuation per thread
  - *Uncoordinated*! - thread local - does not require barriers
- Remediation
  - *Uncoordinated*! - Transitivity inform each mis-speculated thread to rollback
  - *Check-point (Continuation) + Log-based (Dependence graph)* recovery
  - Rollback to last checkpoint, replay correct speculative actions
  - Continues non-speculatively until next observable action = Progress
Results

- Optimistic OLTP
  - Distributed version of STAMP Vacation benchmark
  - Database split into 64 shards, with concurrent transaction requests from geo-distributed clients
  - Uses explicit lock servers -> Rx-CML executes transactions optimistically
Results

- Optimistic OLTP
  - Distributed version of STAMP Vacation benchmark
  - Database split into 64 shards, with concurrent transaction requests from geo-distributed clients
  - Uses explicit lock servers -> Rx-CML executes transactions optimistically

- P2P Collaborative editing
  - Simulates concurrent document editing (operational transformation)
  - Total order broadcast - built out of synchronous events + choice combinator.
Results

- Optimistic OLTP
  - Distributed version of STAMP Vacation benchmark
  - Database split into 64 shards, with concurrent transaction requests from geo-distributed clients
  - Uses explicit lock servers -> Rx-CML executes transactions optimistically

- P2P Collaborative editing
  - Simulates concurrent document editing (operational transformation)
  - Total order broadcast - built out of synchronous events + choice combinator.

- Rx-CML was 5.8X to 7.6X faster than the synchronous version
  - 9-17% of communications were mis-speculated.

![Graph of OLTP and Collaborative Editing](image)
Conclusion

- Composable synchronous events vs. High latency compute cloud
Conclusion

- Composable synchronous events vs. High latency compute cloud
- Rx-CML (Relaxed CML)
  - optimistic concurrency control for CML
  - reason synchronously, but implement asynchronously
  - retain simplicity and composability, but gain performance
Conclusion

- Composable synchronous events vs. High latency compute cloud
- Rx-CML (Relaxed CML)
  - optimistic concurrency control for CML
  - reason synchronously, but implement asynchronously
  - retain simplicity and composability, but gain performance
- Distributed implementation of MultiMLton
  - Case studies demonstrate effectiveness of the approach
Conclusion

- Composable synchronous events vs. High latency compute cloud
- Rx-CML (Relaxed CML)
  - optimistic concurrency control for CML
  - reason synchronously, but implement asynchronously
  - retain simplicity and composability, but gain performance
- Distributed implementation of MultiMLton
  - Case studies demonstrate effectiveness of the approach
- Future Work - Fault tolerance
  - Make checkpoints and dependence graph resilient
  - Treat failures as mis-speculations -> rollback to last saved checkpoint
Questions?

http://multimlton.cs.purdue.edu