℞$^{\text{CML}}$: Migrating MultiMLton to the Cloud

KC Sivaramakrishnan
Purdue University
chandras@cs.purdue.edu

Lukasz Ziarek
SUNY Buffalo
lziarek@buffalo.edu

Suresh Jagannathan
Purdue University
suresh@cs.purdue.edu

Abstract

A functional programming discipline, combined with abstractions like Concurrent ML's first-class synchronous events, offers an attractive programming model for shared-memory concurrency. In particular, synchronous message-passing eases the burden of coordinating and reasoning about concurrent threads by having a communication action serve double-duty as a both a data transfer mechanism (sending or receiving data on a typed channel) as well as a synchronization point (senders and receivers block until a matching action is available). In high-latency distributed environments, like the cloud, however, synchronous communication comes with a high price in performance, given the significant cost of communicating data from one node to another. While switching to an explicitly asynchronous communication model may reclaim some of the performance lost in a synchronous world, by helping to mask communication overheads, 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 suitable that *implements* CML communication asynchronously, but guarantees that the resulting execution is *faithful* to the synchronous semantics of CML. We exploit MultiMLton's support for lightweight checkpointing and rollback to integrate a notion of speculation that allows ill-formed executions to be re-executed, replacing offending asynchronous executions with safe synchronous ones. Several realistic case studies deployed on the Amazon EC2 demonstrate the utility of our approach.

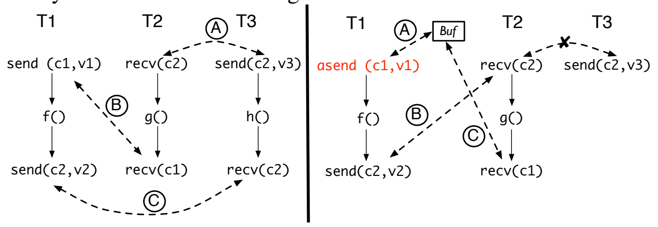
1. Introduction

Cloud computing offers a low-cost scalable execution environment for parallel programs that naturally make use of message-passing. However, the cost models that drive program construction aimed at a tightly-coupled multicore machine are vastly different from the ones that inform the construction of distributed programs written for loosely-coupled environments such as the cloud. In particular, communication latencies between nodes in a cloud ensemble are orders of magnitude greater than those between different cores in a shared-memory multicore machine. While rewriting concurrent applications from scratch to exploit the cloud's computing resources is certainly a possibility, albeit an unpleasant one, we consider a more palatable alternative that exploits new compiler and runtime techniques to mitigate this migration burden.

Our investigation takes place in the context of MultiMLton [2], an extension of the MLton Standard ML compiler that targets shared memory multicore architectures. Programs are modelled as a collection of threads that primarily communicate using CML's first-class synchronous events [3]. In MultiMLton, concurrent programs enjoy strong guarantees on the ordering and visibility of communicated data, simplifying program reasoning. However, while arguably easier to reason about, synchronous execution is an inefficient programming model for a high-latency cloud environment. To bridge the gap between our desire for a simple easy-to-understand programming model and the reality of a high-latency communication model imposed by a geo-distributed cloud environment, we have extended MultiMLton to implement synchronous operations asynchronously, with a transparent lightweight monitoring and rollback mechanism intended to ensure that any speculative behavior is *observably equivalent* to synchronous CML semantics.

2. Observable Equivalence

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: $[\![\texttt{send}\,(c, v)]\!]k \equiv [\![\texttt{asend}\,(c, v)]\!]k$ where k is a continuation, `send` is a 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 on a matching receiver.



Unfortunately, naïvely replacing synchronous communication with an asynchronous one is not usually meaning-preserving as the example given above illustrates. The dashed edges reflect communication and synchronization dependencies among threads, while solid edges capture thread-local control-flow. The communication edges are bi-directional to capture the synchronization flow in either direction between the end points. Assume that `f`, `g`, and `h` do not perform any communications. Under a synchronous evaluation protocol (shown on the left), thread T2 would necessarily communicate first with thread T3, receiving `v3` on channel `c2`. It is then able to receive `v1` from thread T1; finally, T1 can communicate `v2` to T3.

If the `send(c1,v1)` operation by T1 were replaced by `asend (c1,v1)` (shown on the right), `v1` is simply added to the buffer attached to `c1`, and T1 can continue. Hence, the first receive on T2 has, in addition to the first send on T3, a *new potential matching opportunity* – the send of `v2` on channel `c2`. If the receive by T2 matches with the send on T1, it is impossible to satisfy the send on T3. This behavior could not be realized by a synchronous execution since the receive on T2 would never match with send on T1. Thus, the resulting behavior is not observably equivalent to the original synchronous program.

3. Axiomatic Semantics

To understand the conditions under which our desired equivalence holds, we have developed a *relaxed execution model* for Concurrent ML (CML) [3] programs. In order to precisely specify the correct-

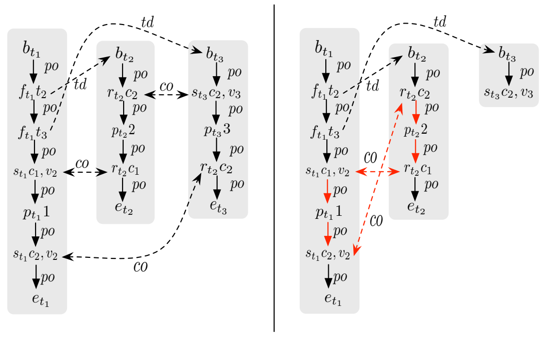


Figure 1: Axiomatic executions.

ness conditions, we have devised an axiomatic formulation of CML executions that express equivalences in terms of causal dependencies captured by a *happens-before* relation that relate communication actions performed by different threads. Informally, a happens-before edge exists between statements (a) within a thread related by program order (i.e. , sequential dependencies); (b) α and β where α is a communication action performed by thread T with another statement α' found in thread T' and α' precedes β in program order; (c) α and β where β found in thread T precedes statement α' and α' performs a communication action that is paired with α.

More formally, the *happens-before* order of an execution is the transitive closure of the union of program (*po*) order, thread dependence (*td*) order (an order between a thread creation action and the first statement executed by a thread), and actions related by communication (*co*) and program order:

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

To illustrate these definitions, Figure 1 shows the axiomatic executions of the example presented above. We assume thread `T1` spawns `T2` and `T3`, and we replace calls to `f`, `g`, and `h` found in the original program, with an observable action, like a print statement. The execution on the left imposes no causal dependence between the observable actions in `T2` or `T3`; 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.

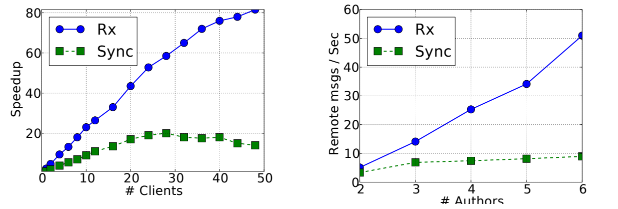
In contrast, the execution depicted on the right-hand side of the figure, which corresponds to the erroneous execution discussed in Section 2, exhibits a happens-before cycle between `T1` and `T2`. Such cyclic dependences *never* manifest in any correct CML execution. Thus, *any axiomatic execution that contains a happens-before cycle is considered to be ill-formed with respect to CML behavior.*

4. Implementation

Our implementation dynamically tracks happens-before dependences by building a distributed dependence graph. 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 R^{CML}(RELAXED CML)[1].

There are several challenges in transplanting MultiMLton (and more specifically, CML) to a distributed environment. (1) **Absence of coherent shared memory:** In a shared-memory environment, CML channels can be implemented as a lock-protected queues. This enables communicating threads to atomically poll the channels for availability of a matching communication, and block on the channel if none is available. In a distributed setting, it is necessary to reason about multiple replicated, yet globally consistent, versions of CML channels. (2) **Serialization:** CML channels allow typesafe communication of polymorphic values. Since CML imposes no restriction on these values, which may be arbitrarily complex data structures, a serialization mechanism that can communicate these values across different machines is necessary. (3) **Transport layer:** CML channels allow multiple producers and consumers to operate over the same channel. Supporting this functionality in a distributed setting requires an *intelligent* transport layer that supports efficient broadcast as a primitive operation. (4) **Speculative Execution:** Central to R^{CML}'s design is the speculative execution of message sends that allows a synchronous send to be transparently executed asynchronously. Because speculations can be wrong, the implementation must provide a low-cost mechanism to save application state, detect errors, and rollback to a globally consistent state [5] *without* requiring a global barrier for error detection or rollback.

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



(a) Scalability on OLTP benchmark. (b) Throughput on collaborative editing benchmark.

Figure 2: Performance comparison of R^{CML}vs CML (synchronous).

5. Evaluation

We have evaluated the performance of R^{CML}on a number of parallel and distributed applications executed on an Amazon EC2 cloud infrastructure. We present the results of two of these applications with respect to scalability and throughput in Fig. 2: the first is a distributed online transaction processing (OLTP) benchmark derived from the `vacation` benchmark found in STAMP [1] benchmark suite; the second is a collaborative editing benchmark that simulates concurrent editing of a document by multiple authors [4]. These programs are written in CML, designed assuming a synchronous communication protocol; synchronous actions are transparently turned into asynchronous ones by the runtime, with monitoring and rollback functionality injected to ensure safe behavior. We believe these results indicate that the transparent relaxation of synchrony can lead to substantial performance improvement, while preserving the ease of synchronous programming.

References

[1] C. Cao Minh, J. Chung, C. Kozyrakis, and K. Olukotun. STAMP: Stanford transactional applications for multi-processing. In *IISWC*, 2008.

[2] MultiMLton. MLton for Scalable Multicore Architectures, 2013. URL `http://multimlton.cs.purdue.edu`.

[3] J. Reppy. *Concurrent Programming in ML*. Cambridge University Press, 2007.

[4] M. Suleiman, M. Cart, and J. Ferrié. Serialization of Concurrent Operations in a Distributed Collaborative Environment. In *GROUP*, pages 435–445, 1997.

[5] L. Ziarek and S. Jagannathan. Lightweight Checkpointing for Concurrent ML. *Journal of Functional Programming*, 20(2):137–173, 2010.