Graduate School Form 30
(Updated 11/20/2014)

PURDUE UNIVERSITY
GRADUATE SCHOOL
Thesis/Dissertation Acceptance

This is to certify that the thesis/dissertation prepared

By Sivaramakrishnan Krishnamoorthy Chandrasekaran

Entitled
Functional Programming Abstractions for Weakly Consistent Systems

For the degree of Doctor of Philosophy

Is approved by the final examining committee:

Suresh Jagannathan

Patrick Eugster

Jan Vitek

Dongyan Xu

To the best of my knowledge and as understood by the student in the Thesis/Dissertation Agreement, Publication Delay, and Certification/Disclaimer (Graduate School Form 32), this thesis/dissertation adheres to the provisions of Purdue University's "Policy on Integrity in Research" and the use of copyrighted material.

Suresh Jagannathan

Approved by Major Professor(s): _____

_____

Approved by: Sunil Prabhakar 12/04/2014

Head of the Department Graduate Program Date

FUNCTIONAL PROGRAMMING ABSTRACTIONS FOR WEAKLY CONSISTENT SYSTEMS

A Dissertation

Submitted to the Faculty

of

Purdue University

by

Sivaramakrishnan Krishnamoorthy Chandrasekaran

In Partial Fulfillment of the

Requirements for the Degree

of

Doctor of Philosophy

December 2014

Purdue University

West Lafayette, Indiana

To Siva, who always believed it is better to go down trying.

ACKNOWLEDGMENTS

First and foremost, I would like to thank my advisor Suresh Jagannathan, who graciously took a naive yet enthusiastic graduate student into his research group, and taught him how to do research. I am immensely thankful for his always open office door, uncanny ability to foresee potential issues, high standards and patience. Without his guidance and support, this dissertation would not have been possible. I have been fortunate to have Patrick Eugster, Jan Vitek, and Dongyan Xu on my PhD committee. Their critiques and insightful comments helped focus and refine the dissertation.

I would like to thank my colleague, mentor and good friend Lukasz Ziarek, with whom I worked closely in my early years. He showed by example the mechanics of going from an idea to a finished paper. The early collaborations with him set the tone for the rest of the research in this dissertation. I had the good fortune to study distributed systems over two graduate courses under Patrick Eugster. His enthusiasm and industry inspired me to push for publishing the final class project. I learnt from him the tact of fleshing out abstract ideas and producing a polished product.

I have had the privilege of working with immensely talented people over my internships. The summer spent at Samsung Research was a turning point for me in no small part due to the encouragement and enthusiasm of Daniel Waddington. Samsung not only funded the internship, but also supported some of my time at graduate school through a generous gift, for which I am very grateful. The internship at Microsoft Research, Cambridge broadened my research perspectives. Tim Harris, Simon Marlow and Simon Peyton Jones taught me how to remain focused and grounded.

I am indebted to the friendship and insight of my numerous friends at Purdue University including Siddharth Narayanaswamy, Karthik Swaminathan Nagaraj, Gowtham

Kaki, K.R. Jayaram, Naresh Rapolu, Pawan Prakash, Gustavo Petri, He Zhu, Siddharth Tiwary, Karthik Kambatla, Raghavendra Prasad, Armand Navabi, and many others. I will miss you all.

I would like to thank my parents for their constant source of encouragement and offering me complete freedom to pursue my interests. Never did they doubt my completing this journey even when I doubted myself. Last but not least, my greatest debt is to my wife, Siva, whose undying support during difficult times gave me the strength to keep on fighting. No amount of gratitude can repay my debt to you.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

Figure Page

Figure Page

ABBREVIATIONS

ACML	Asynchronous Concurrent ML
CC	Causally Consistent
CML	Concurrent ML
CSH	Cached Shared Heap
DMA	Direct Memory Access
EC	Eventually Consistent
ECDS	Eventually Consistent Data Store
GC	Garbage-Collector
MAV	Monotonic Atomic View
MPB	Message-Passing Buffers
RC	Read Committed
RDT	Replicated Data Type
RR	Repeatable Read
SC	Strongly Consistent
SCC	Single-chip Cloud Computer
SMC	Software-Managed Cache-coherence
USH	Uncached Shared Heap

ABSTRACT

Krishnamoorthy Chandrasekaran, Sivaramakrishnan. Ph.D., Purdue University, December 2014. Functional Programming Abstractions for Weakly Consistent Systems. Major Professor: Suresh Jagannathan.

In recent years, there has been a wide-spread adoption of both multicore and cloud computing. Traditionally, concurrent programmers have relied on the underlying system providing *strong memory consistency*, where there is a semblance of concurrent tasks operating over a shared global address space. However, providing scalable strong consistency guarantees as the scale of the system grows is an increasingly difficult endeavor. In a multicore setting, the increasing complexity and the lack of scalability of hardware mechanisms such as cache coherence deters scalable strong consistency. In geo-distributed compute clouds, the availability concerns in the presence of partial failures prohibit strong consistency. Hence, modern multicore and cloud computing platforms eschew strong consistency in favor of weakly consistent memory, where each task's memory view is incomparable with the other tasks. As a result, programmers on these platforms must tackle the full complexity of concurrent programming for an asynchronous distributed system.

This dissertation argues that functional programming language abstractions can simplify scalable concurrent programming for weakly consistent systems. Functional programming espouses mutation-free programming, and rare mutations when present are explicit in their types. By controlling and explicitly reasoning about shared state mutations, functional abstractions simplify concurrent programming. Building upon this intuition, this dissertation presents three major contributions, each focused on addressing a particular challenge associated with weakly consistent loosely coupled systems. First, it describes Aneris, a concurrent functional programming language

and runtime for the Intel Single-chip Cloud Computer, and shows how to provide an efficient cache coherent virtual address space on top of a *non cache coherent* multicore architecture. Next, it describes R$^{\text{CML}}$, a distributed extension of MultiMLton and shows that, with the help of speculative execution, synchronous communication can be utilized as an efficient abstraction for programming asynchronous distributed systems. Finally, it presents Quelea, a programming system for *eventually consistent* distributed stores, and shows that the choice of correct consistency level for replicated data type operations and transactions can be automated with the help of high-level declarative *contracts*.

1 INTRODUCTION

In recent years, there has been a widespread adoption of both multicore and cloud computing. Multicore processors have become the norm in mobile, desktop and enterprise computing, with an increasing number of cores being fitted on a chip with every successive generation. Cloud computing has paved the way for companies to rent farms of such multicore processors on a pay-per-use basis, with the ability to dynamically scale on demand. Indeed, many real-world services for communication, governance, commerce, education, entertainment, etc., are routinely exposed as a web-service that runs in third-party cloud compute platforms such as Windows Azure [1] and Amazon's EC2 [2]. These services tend to be concurrently accessed by millions of users, increasingly through multicore-capable mobile and desktop devices.

1.1 The Problem

1.1.1 Strong Consistency

The holy grail of programming such massively parallel systems is to achieve good scalability without falling prey to the usual pitfalls of concurrency such as data races, deadlocks and atomicity violations [3]. Traditionally, programmers have relied on the underlying hardware or storage infrastructure providing a semblance of a single memory image, shared between all of the concurrent tasks. Operations from each task appear to be applied to the shared memory in the order in which they appear locally in each task, and operations from different tasks are interleaved in some total order. Such a system is said to provide *strong memory consistency*. Strong consistency is a natural extension of uniprocessor memory model to a multiprocessor setting. While

this strong consistency does not completely eliminate the possibility of concurrency bugs, it certainly simplifies reasoning about the behavior of concurrent programs.

1.1.2 Implications of the Programming Model

Our definition of strong consistency applies equally to the two popular paradigms of concurrent program design, *shared memory* and *message-passing*, differentiated by the way in which the concurrent threads interact with each other. In the shared memory paradigm, threads interact by updating and inspecting shared memory locations, whereas under the message-passing paradigm, threads interact by exchanging messages. For this discussion, let us assume that the shared memory paradigm is realized through read and write primitive to named memory locations, and message-passing paradigm is captured by asynchronous send and blocking receive primitives on named point-to-point channels. Other message-passing paradigms such as synchronous communication, Erlang-style mailboxes, thread-addressed messages can be implemented on top of point-to-point asynchronous message passing model.

Under strongly consistent shared memory, a thread performing a read will witness the latest write to the same memory location by any thread. Under strongly consistent message-passing, when a thread performs a sends a value v on an empty channel c, the sent value v is available to be consumed by every thread that subsequently performs a receive. Subsequently, when a receive operation consumes the sent value v, the act of consumption is witnessed by every thread, and no subsequent thread can consume the same value v. Indeed, semantically shared-memory and message-passing paradigms are simply two sides of the same coin [4, 5]. This is illustrated by the fact that one model can easily be implemented using the other. For example, languages like Haskell [6], ConcurrentML [7] and Manticore [8] implement message-passing paradigms over shared memory, and popular geo-distributed stores such as Dynamo [9], Cassandra [10] and Riak [11] implement shared-memory paradigm over message passing. Hence, strong memory consistency equally benefits programmers

working under either paradigms. Conversely, and more importantly, any weaker memory consistency semantics affects both paradigms.

1.1.3 Realizing Strong Consistency

Depending upon the target platform, a variety of mechanisms have been proposed to achieve strong consistency. Shared memory multicore processors designed for mainstream computing markets tend to have hierarchical memory organization, with private and shared multi-level caches, and utilize a hardware protocol for keeping the caches coherent [12]. Coherence can be viewed as a mechanism that transmits a write to a memory location to all the cached copies of the same location. Typically, each cache line has meta-data attached to it which indicate whether the local cacheline is invalid, shared or modified. When a memory location corresponding to a shared cache line is updated, coherence mechanism invalidates all other remote cache lines that also refer to the same memory location. A core accessing an invalid cacheline has to fetch the latest version, which is termed as cache miss.

In a distributed setting, techniques such as atomic broadcast [13], consensus [14], distributed transactions [15], and distributed locking services [16] are widely used in practice to provide strong consistency. These mechanisms abstract the underlying complexity of concurrent programming, and expose a simpler programming model to the developers. For example, models such as sequential consistency [17], linearizability [18] and serializability [19] are widely used in the construction of concurrent programs.

1.1.4 Cost of Strong Consistency

Despite the simplicity of strong consistency, with increasing scale, providing strong consistency guarantees is an increasingly difficult endeavor. Already, for performance reasons, modern optimizing compilers and multicore processors reorder code in ways

that are not observable by sequential code, but are very much observable under concurrent execution [20–23]. Hence, the semblance of strong consistency is broken. However, the hardware memory models do provide coherence, and to the benefit of the programmers, the language memory models ensure sequential consistency for programs that do not involve data races.

On the other hand, the complexity and power requirements for hardware support for cache coherence increases with increasing number of cores [24]. The scalability of hardware cache coherence mechanisms is mainly hindered by the scalability of coherence hardware, the storage requirements for cache meta data, and the effort to implement and verify complex coherence protocols. While there are indeed attempts to reduce the cost of cache coherence hardware on manycore systems [12], hardware vendors increasingly opt for non cache coherent architectures. Graphics processing units (GPUs) [25], the Intel Single-chip Cloud Computer (SCC) [26], the Cell BE processor [27], and the Runnemede prototype [28] are representative examples of non cache coherent architectures.

Applications that rely upon strong consistency in a distributed setting have to pay the cost of reduced *availability* in the presence of network partitions and high *latency*. In particular, Brewer's well-known CAP theorem [29–31] states that a distributed system cannot simultaneously provide strong consistency, be available to updates, and tolerate network partitions. Since network partitions are unavoidable, and web-services running on geo-distributed systems focus on providing always-on experience, application developers unfortunately have to give up the advantages offered by strong consistency. Moreover, techniques for achieving strong consistency [13–16], require *co-ordination* between the nodes in the distributed system. In a geo-distributed setting, where inter-node latencies are in the order of hundreds of milliseconds, the latency hit associated with strong consistency is unacceptable. Moreover, coordination between nodes in a geo-distributed setting while processing client requests defeats the whole purpose of geo-distribution, which is to minimize latency by serving clients from the closest data center.

1.1.5 Challenges under Weak Consistency

In response to these concerns, scalable compute platforms eschew strong consistency, and instead rely only on weaker consistency guarantees. Without strong consistency, the programmer gets to see that there is no longer a coherent shared memory abstraction, but instead a collection of *coherence domains* between which updates are lazily exchanged. The onus now falls on the programmer to ensure that the application meets is correctness requirements.

On non cache coherent multicore architectures, the programmer must explicitly perform communication actions between local address spaces through message passing or direct memory access (DMA). On architectures such as Intel SCC [26] and Runnemede [28], which provide explicit instructions to invalidate and flush caches, the programmer must ensure that the cache control instructions are correctly issued at appropriate junctures in order to maintain a coherent view of the shared memory. Any missed cache invalidations will lead to stale data being read, where as any missed cache flushes prevents a write from being exposed to other coherence domains. However, frequent invalidations and flushes lead to poor cache behavior. Understandably, this process is notoriously difficult to get right.

A geo-distributed store, where an object is replicated at multiple sites, is in essence similar to a non cache coherent architecture. Under weak consistency, programs operating over geo-distributed stores typically assume that the replicas of an object will *eventually converge* to the same state. This behavior is commonly termed as eventual consistency [32, 33]. Unlike multicore architectures, the high latency in a geo-distributed setting warrants that the application accept concurrent conflicting updates in order to remain responsive. The updates are asynchronously propagated between the sites, and a *deterministic* conflict resolution procedure ensures that the replicas eventually converge to the same state. The conflict resolution can either be automatic (such as last-writer-wins) or, in cases where the automatic resolution is non-trivial or non-existent, may involve manual intervention.

It is important to point out that eventual consistency only guarantees that the replicas will "eventually" converge to a same state, but does not provide any additional guarantees with respect to recency or causality of the operations. Hence, with two successive reads to the same object, there is no guarantee that the second read will see a "newer" version of the object. Worse still, a session might not see its own writes! These anomalies are reminiscent of the re-orderings that can occur under language and hardware memory models [20, 21], except that the anomalies in this case are due to the fact that requests from same session can be serviced by different replicas.

To address these concerns, several systems [34–38] have proposed that provide a lattice of stronger guarantees on demand. While defining new consistency guarantees and implementing them in a geo-distributed storage infrastructure is certainly a commendable endeavor, how does one match the consistency requirements at the application level with the consistency guarantees offered by the store? How does one ensure that the composition of consistency guarantees of different operations result in a sensible behavior? In short, developing correct concurrent applications under weak consistency requires large programmer effort in order to intricately reasoning about non-trivial memory interactions on top an already non-deterministic programming model.

1.2 My Thesis

In this dissertation, we argue that functional programming language abstractions can mitigate the complexity of programming weakly consistent systems. The key idea is that, since consistency issues arise out of shared state mutation, by controlling and minimizing mutation one can simplify the problem of programming under weak memory consistency.

The dissertation presents three major contributions, each focused on addressing a particular challenge associated with weakly consistent loosely coupled systems: (1) providing an efficient virtual shared memory abstraction over non cache coher-

ent architectures by exploiting mutability information, (2) utilizing composable synchronous message-passing communication as an efficient abstraction for programming asynchronous distributed systems with the help of speculative execution, and (3) a mutation-free programming model for eventual consistency that automates the choice of mapping application-level consistency requirements to consistency levels offered by the geo-distributed data store.

1.3 Contributions

In this section, we provide a brief overview of the contributions made by this dissertation.

1.3.1 Efficiently Masking the Absence of Cache Coherence

The first contribution of this thesis is a series of techniques to efficiently hide the absence of cache coherence on a non cache coherent architecture, and provide the semblance of a shared coherent global address space with strong (sequential) consistency. We demonstrate this by designing and implementing Aneris, an extension of MultiMLton [39] compiler and runtime system targeted at the 48-core memory-coupled, non cache coherent Intel SCC processor.

Providing virtual shared memory on top of distributed memory architectures is certainly not a novel endeavor. Typically, non cache coherent architectures are organized such that each core or a collection of cores share a cache coherent address space (termed as a "coherence domain"), and utilize explicit communication or DMA transfers for traffic across coherence domains. Such virtual memory systems additionally implement all the necessary inter-core communication operations for scheduling and synchronization. This model has been used on the Cell BE processor for implementing shared-memory programming models such as OpenMP [40], COMIC [41], Sequoia [42] and CellSs [43], and on the Intel SCC for X10 [44] and Shared virtual

memory model [45]. These works typically expose the distribution in the programming model, provide specialized hooks into the architectural features, or are simply agnostic of the application level consistency requirements.

Differing from these works, Aneris utilizes the key property of mostly-functional languages (in our case, Standard ML enriched with concurrent threads and synchronous message passing), that is *mutation is rare*, to efficiently realize a virtual memory abstraction using just the language runtime mechanisms. We also identify that non cache coherent architectures provide several different alternatives for inter-core communication such as on-chip high-speed message passing interconnect, scalable NoC interconnect for transferring data directly between memory banks without involving the processors, and explicit cache control instructions. We aim to allow the same programs written for cache coherent architectures to efficiently run on non cache coherent architectures, while transparently mapping the source language structures and mechanisms on to the architecture's capabilities.

Our initial system design utilizes a split-heap memory manager design [46–48], optimized for the SCC's memory hierarchy, to obtain a MultiMLton system on the SCC. This design however incorporates both read and write barriers, and we identify that the cost of read barriers under MultiMLton programming model is significant. To alleviate this, we design a novel thread local collector that utilizes ample concurrency in the programming model as a resource along with a dynamic shape analysis to eliminate the read barriers. Our final runtime design transparently utilizes SCC's support for software managed cache coherence and on-die message-passing interconnect to achieve an efficient implementation under which 99% of the memory accesses can potentially be cached. These results were published in ISMM 2012 [49] and MARC 2012 [50].

1.3.2 A Prescription for Safely Relaxing Synchrony

The second contribution of the thesis is $\mathrm{R}^{\mathrm{CML}}$, an optimistic variant of Concurrent ML [7]. $\mathrm{R}^{\mathrm{CML}}$ utilizes synchronous communication over first-class channels as an abstraction for programming asynchronous distributed systems. A mostly functional programming language combined with synchronous message passing over first-class channels offers an attractive and generic model for expressing fine-grained concurrency. In particular, an expressive language like ConcurrentML [7] composable synchronous events, the synchronous communication simplifies program reasoning by combining data transfer and synchronization into a single atomic unit. However, in a distributed setting, such a programming model becomes unviable due to two reasons:

- In a geo-distributed setting, synchronization requires coordination between nodes, which is at odds with the high inter-node latency.
- As discussed previously, the point-to-point first-class channel abstraction requires strong consistency. In particular, the channel abstraction ensures that values are consumed *exactly-once*, which requires coordination between nodes that might potentially consume a particular value on the channel.

While switching to an explicit asynchronous process-oriented communication model avoids these issues, it complicates inter-node synchronization and introduces naming issues. No longer can a programmer abstractly reason about a collection of nodes that might send or receive values on a named channel, but has to identify, communicate and coordinate with individual nodes. Additionally, the onus falls on the programmers to handle partial failures and network partitions. Thus, the loss of synchronous communication abstraction significantly burdens the programmer.

The key contribution of this work is to utilize synchronous communication as an abstraction to express programs for high-latency distributed systems, but *speculatively* discharge the communications asynchronously, and ensure that the observable behavior mirrors that of the original synchronous program. The key discovery is that

the necessary and sufficient condition for divergent behavior (mis-speculation) is the presence of happens-before cycle in the dependence relation between communication actions. We prove this theorem over an axiomatic formulation that precisely captures the semantics of speculative execution. Utilizing this idea, we build an optimistic concurrency control mechanism for concurrent ML programs, on top of MultiMLton, capable of running in compute clouds. The implementation uses a novel un-coordinated checkpoint-recovery mechanism to detect and remediate mis-speculations. Our experiments on Amazon EC2 validate our thesis that this technique is quite useful in practice. These results were published in PADL 2014 [51].

1.3.3 Declarative Programming over Replicated Data Stores

The final contribution of this thesis addresses two related challenges when programming under eventual consistency on top of geo-distributed stores:

- How do you describe practical and scalable eventually consistent data types?
- How do you map the application level consistency properties automatically to the most efficient of the consistency levels provided by the store?

Let us expand on the challenges associated with each of these goals.

Typically, commercial geo-distributed stores such as DynamoDB [52], Cassandra [10], Riak [11] provide a data model that is reminiscent distributed maps. The key-value pair is usually treated as registers, with a default last-write-wins (LWW) conflict resolution policy. Since a LWW register is not suitable for every use case, a small collection of convergent data types such as counters and sets [53] are also provided. Often the programmer has to coerce the problem at hand, which might naturally be expressed as operations over a particular abstract datatype into the ones that are supported by the store. Unlike a concurrent program written for shared memory multicore processor, the operation on low-level convergent replicated data types cannot be composed together well; with no practical consistency control mechanisms

such as fences and locks, the programmer has but to reason about the intricate weak consistency behaviors between composed operation. Often, without the necessary abstractions, it is impossible to achieve the desired semantics, and hence, the application ends up exposing the weak consistency behavior to the user.

In addition, although the store might provide stronger consistency guarantees, the lack of precise description of these guarantees, and the inherent difficulty in mapping application-level consistency requirements to store-level guarantees leads to subtle weak consistency bugs. Although there has been progress on the theoretical front to address the concern of reason about concurrent programs on eventually consistent stores [54], realization of these techniques on full-fledged commercial store implementations has not yet come by. Thus, the lack of a suitable programming model for practical replicated data types hinders software development for eventually consistent systems.

To address these issues, we present Quelea programming system for declaratively programming eventually consistent systems. Inspired by operation-based convergent replicated data types, data types in Quelea are defined in terms of its interfaces, and the effects that an operation has on a data type. Importantly, the state of an object is simply the set of all effects performed on this object. Every operation performs a *fold* over this set, and might optionally produce a new effect. The effects performed at a particular replica is asynchronously transmitted to other replicas. Since each operation witnesses all the effects, concurrent or otherwise, performed on the object so far, semantically conflicting operations can be resolved deterministically. As we will see, this particular abstraction is powerful enough to describe complex real-world scenarios including twitter-like micro-blogging service and an ebay-like auction site.

Implementing and maintaining a robust, scalable geo-distributed store is a significant undertaking. Indeed, concerns such as liveness, replication, durability and failure handling must be handled by any realistic distributed store implementation, but are orthogonal to the consistency related safety properties that we aim to address in this work. Instead or replicating the massive engineering effort and in the process

introducing subtle concurrency and scalability issues, we realize QUELEA as a shim layer on top of the industrial strength data store, Cassandra [10]. This separation of concerns allows the QUELEA programming model to be ported to other distributed stores as well.

In addition to the datatype description language, QUELEA supports a *contract* language for describing the application-level consistency properties. The contract language is used to express valid concurrent executions utilizing a particular replicated data type, over a small corpus of primitive relations, capturing properties such as visibility and session order. The executions described are similar to the the axiomatic description of relaxed memory models [20,54], declaratively capturing the well-formed behaviors in the program. Given a set of store-level consistency guarantees, also expressed in the same contract language, we *statically* map each datatype operation to one of the store-level consistency properties.

Finally, our implementation of the QUELEA programming model not only supports primitive operations, but also a series of *coordination-free transactions*. Similar to basic operations, we utilize the same contract language to map the user-defined transactions to one of the store-specific transaction isolation levels. The thesis illustrates that a mutation-free programming model for eventually consistent stores not only enables expressive declarative reasoning, but is also practically achievable on top of industrial-strength geo-distributed stores.

1.4 Road Map

The rest of the dissertation is organized as follows. Chapter 2 describes the MULTIMLTON programming model and runtime system, which serves as the exploration vehicle for ANERIS and R$^{\text{CML}}$. Chapter 3 presents ANERIS, the port of MULTIMLTON to the Intel SCC platform that provides a cache coherent shared memory abstraction for a concurrent extension of Standard ML. Chapter 4 presents R$^{\text{CML}}$, an optimistic variant of Concurrent ML [7] for distributed systems. Chapter 5 presents QUELEA, a

programming system for eventually consistent geo-distributed stores. Related work is presented at the end of each chapter. Additional related work that is relevant to the future direction of this research is given in chapter 6, along with concluding remarks.

2 MULTIMLTON

MultiMLton is an extension of the MLton [55] compiler and runtime system that targets scalable, multicore architectures. MLton is a whole-program optimizing compiler for Standard ML programming language, which a member of the ML family of programming languages that includes Objective Caml and F#. Apart from MultiMLton, another notable example in the multicore ML space is Manticore [8], and focuses on implicit parallelism under an ML-inspired language. In this chapter, we will present the programming model and the runtime system details of MultiMLton, which provides the technical background that informs the rest of the dissertation.

2.1 Programming Model

While MLton does not target multicore processors, it does include excellent support for Concurrent ML (CML) [7], a concurrent extension of Standard ML that utilizes synchronous message passing to enable to construction of synchronous communication protocols. The programming model supported by MultiMLton is heavily influenced by CML. We begin by briefly describing the CML programming model, before its extension used in MultiMLton.

2.1.1 Concurrent ML

Concurrent ML [7] is a concurrent extension of Standard ML with support for user-level thread creation, where the threads primarily interact by performing synchronous `send` and `recv` operations on typed channels; these operations block until a matching actions on the same channel is performed by a different thread.

CML also provides first-class synchronous *events* that abstract synchronous message-passing operations. An event value of type `'a Event` when synchronized on yields a value of type `'a`. An event value represents a potential computation, with latent effect until a thread synchronizes upon it by calling `sync`. The following equivalences thus therefore hold: `send(c, v)` ≡ `sync(sendEvt(c,v))` and `recv(c)` ≡ `sync(recvEvt(c))`. Notably, thread creation is *not* encoded as an event – the thread `spawn` primitive simply takes a thunk to evaluate as a separate thread, and returns a thread identifier that allows access to the newly created thread's state.

Besides `sendEvt` and `recvEvt`, there are other base events provided by CML. The `never` event, as its name suggests, is never available for synchronization; in contrast, `alwaysEvt` is always available for synchronization. These events are typically generated based on the satisfiability of conditions or invariants that can be subsequently used to influence the behavior of more complex events built from the event combinators described below. Much of CML's expressive power derives from event combinators that construct complex event values from other events. We list some of these combinators below:

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

The expression `wrap (ev, f)` creates an event that, when synchronized, applies the result of synchronizing on event `ev` to function `f`. Conversely, `guard(f)` creates an event that, when synchronized, evaluates `f()` to yield event `ev` and then synchronizes

on `ev`. The `choose` event combinator takes a list of events and constructs an event value that represents the non-deterministic choice of the events in the list; for example:

```
sync(choose[recvEvt(a),sendEvt(b,v)])
```

will either receive a unit value from channel `a`, or send value `v` on channel `b`. Selective communication provided by `choose` motivates the need for first-class events. We cannot, for example, simply build complex event combinators using function abstraction and composition because function closures do not allow inspection of the encapsulated computations, a necessary requirement for implementing combinators like `choose`.

2.1.2 Asynchronous Concurrent ML

While simple to reason about, synchronous events impose non-trivial performance penalties, requiring that both parties in a communication action be available before allowing either to proceed. To relax this condition, MultiMLton allows the expression of *asynchronous* composable events, through an asynchronous extension of concurrent ML (ACML).

An asynchronous operation initiates two temporally distinct sets of actions. The first defines *post-creation* actions – these are actions that must be executed after an asynchronous operation has been initiated, without taking into account whether the effects of the operation have been witnessed by its recipients. For example, a post-creation action of an asynchronous send on a channel might initiate another operation on that same channel; the second action should take place with the guarantee that the first has already deposited its data on the channel. The second are *post-consumption* actions – these define actions that must be executed only after the effect of an asynchronous operation has been witnessed. For example, a post-consumption action might be a callback that is triggered when the client retrieves data from a

channel sent asynchronously. These post-consumption actions take place within an implicit thread of control responsible for completing the asynchronous operation.

ACML introduces first-class *asynchronous* events with the following properties: (i) they are extensible both with respect to pre- and post-creation as well as pre- and post-consumption actions; (ii) they can operate over the same channels that synchronous events operate over, allowing both kinds of events to seamlessly co-exist; and, (iii) their visibility, ordering, and semantics is independent of the underlying runtime and scheduling infrastructure.

In order to provide primitives that adhere to the desired properties outlined above, ACML extends CML with a new asynchronous event type `('a,'b) AEvent` and the following two base events: `aSendEvt` and `aRecvEvt`, to create an asynchronous send event and an asynchronous receive event, respectively. The differences in their type signature from their synchronous counterparts reflect the split in the creation and consumption of the communication action they define:

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

The type of `AEvent` is polymorphic over the type of the return values of the event's post-creation and post-consumption actions. In the case of `aSendEvt`, both actions yield `unit`: when synchronized on, the event immediately returns a `unit` value and places its `'a` argument value on the supplied channel. The post-consumption action also yields `unit`. When synchronized on, an `aRecvEvt` returns `unit`; the type of its post-consumption action is `'a` reflecting the type of value read from the channel when it is paired with a send. The semantics of both asynchronous send and receive guarantees that successive communication operations performed by the same thread get witnessed in the order in which they were issued.

Beyond these base events, ACML also provides a number of combinators that serve as asynchronous versions of their CML counterparts. These combinators enable the extension of post-creation and post-consumption action of asynchronous events to create more complex events, and allow transformation between the synchronous and asynchronous events.

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

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

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

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

Similar to CML `wrap` combinator, `sWrap` and `aWrap` extend the post-consumption and post-creation actions of an asynchronous event, respectively. `aGuard` allows creation of a guarded asynchronous event. `sChoose` is a blocking choice operator which blocks until one of the asynchronous base events has been consumed. `aChoose` is a non-blocking variant, which has the effect of non-deterministically choosing one of the base asynchronous events if none are available for immediate consumption. Finally, `aTrans` and `sTrans` allow transformation between the synchronous and asynchronous variants.

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

We also introduce a new synchronization primitive: `aSync`, to synchronize asynchronous events. The `aSync` operation fires the computation encapsulated by the asynchronous event of type `('a, 'b) AEvent`, returns a value of type `'a`, corresponding to the return type of the event's post-*creation* action. Unlike their synchronous variants, asynchronous events do *not* block if no matching communication is present. For example, executing an asynchronous send event on an empty channel places the value being sent on the channel and then returns control to the executing thread. In order to allow this non-blocking behavior, an *implicit* thread of control is created for the asynchronous event when the event is paired, or *consumed*. If a receiver is present on the channel, the asynchronous send event behaves similarly to a synchronous event; it passes the value to the receiver. However, a new implicit thread of control is still created to execute any post-consumption actions.

Similarly, the synchronization of an asynchronous receive event does not yield the value received; instead, it simply enqueues the receiving action on the channel. Therefore, the thread that synchronizes on an asynchronous receive always gets the value `unit`, even if a matching send exists. The actual value consumed by the asynchronous receive can be passed back to the thread which synchronized on the event through the use of combinators that process post-consumption actions. This is particularly well suited to encode reactive programming idioms: the post-consumption actions encapsulate a reactive computation.

Further details about the MultiMLton programming model and ACML can be found in [56].

2.2 Compiler

Since the concurrent programming model of MultiMLton is exposed as a library on top of MLton, MultiMLton retains MLton's compiler infrastructure, and only adds a few additional compiler primitives for concurrency support. MultiMLton is a whole-program optimizing compiler for the full SML 97 language [57], including

modules and functors. During compilation, MultiMLton first transforms the source program with modules and functors into an equivalent one without by defunctorization [58]. Defunctorization duplicates each functor at every application and eliminates structures by renaming variables. Next, the program is monomorphized [59] by instantiating the polymorphic datatypes and functions at every application. The program is then defunctionalized, replacing the higher-order functions with data structures to represent them and first-order functions to apply them. The resultant intermediate language is in Static Single Assignment (SSA) form [60]. Much of the aggressive optimizations are performed in SSA passes. The SSA code is then transformed to RSSA intermediate representation. RSSA is similar SSA, but exposes data representations decisions that leads to further optimizations. The compiler can produce native code for multiple backends as well as portable C output.

2.3 Runtime System

2.3.1 Threading System

MultiMLton's runtime system is specifically optimized for efficiently handling the large number of concurrent threads, both implicit and explicit, created by the ACML programming model. MultiMLton uses an m over n threading system that leverages potentially many lightweight (language level) threads multiplexed over a collection of kernel threads. The user-level thread scheduler is in turn implemented using the `MLton.Thread` [55] library, which provides one-shot continuations. `MLton.Thread` uses a variation of Bruggeman *et al.*'s [61] strategy for implementing one-shot continuations. A `MLton.Thread` is a lightweight data structure that represents a paused computation, and encapsulates the metadata associated with the thread as well as a stack. The stack associated with the lightweight thread is allocated on the heap, and is garbage collected when the corresponding thread object is no longer reachable.

As such, `MLton.Thread` does not include a default scheduling mechanism. Instead, MultiMLton builds a preemptive, priority supported, run-queue based, multicore-capable scheduler using `MLton.Thread`. Building multicore schedulers over continuations in this way is not new, first described by Wand et al. [62], and successfully emulated by a number of modern language implementations [47,63]. Each core has a private scheduler queue, and new threads are by default spawned on the same core. The programmer can explicitly request for a thread to be spawned on a different core. However, once spawned, the threads remain pinned to their cores.

Implementing MultiMLton's threading system over one-shot continuation as opposed to full-fledged (multi-shot) continuations greatly reduces the cost of the thread and scheduler implementation. In particular, if full-fledged continuations were used to implement the scheduler, then during every thread switch, a *copy* of the current thread would have to be made to reify the continuation. This is, in our context, unnecessary since the current stack of the running thread (as opposed to the saved stack in the continuation), will never be accessed again. One-shot continuations avoid copying the stack altogether; during a thread switch, a reference to the currently running thread is returned to the programmer. The result is a very efficient baseline scheduler.

Lightweight threads are garbage collected when no longer reachable. MultiMLton's threading system multiplexes many lightweight thread on top of a few operating system threads. Each kernel thread represents a virtual processor and one kernel thread is pinned to each processor. The number of kernel threads is determined statically and is specified by the user; they are not created during program execution.

While lightweight threads provide a conceptually simple language mechanism for asynchrony, they are unsuitable for expressing the implicit asynchronous threads created by the ACML programming model. This is due to the synchronization, scheduling, and garbage collection costs associated with lightweight threads, which outweigh the benefit of running the computation concurrently. With the aim of carrying out ACML's implicit asynchronous actions, MultiMLton supports a new

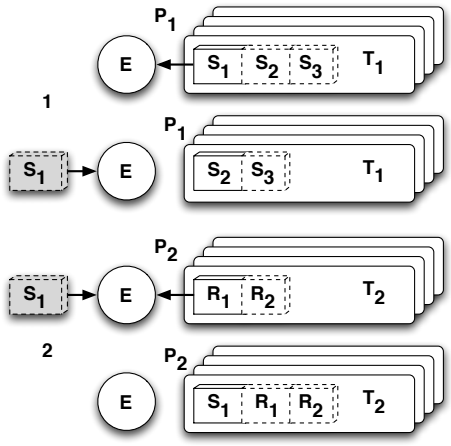


Figure 2.1. Blocking and unblocking of parasitic threads.

threading mechanism called *parasitic threads*. Thus, our runtime supports two kinds of threads: hosts and parasites. Host threads map directly to lightweight threads in the runtime. Parasitic threads can encapsulate arbitrary computation, just like host threads. However, unlike a regular thread, a parasitic thread executes using the execution context of the host that creates the parasite; it is intended primarily to serve as the execution vehicle for asynchronous actions.

Parasitic threads are implemented as raw frames living within the stack space of a given host thread. A host thread can hold an arbitrary number of parasitic threads. In this sense, a parasitic thread views its host in much the same way as a user-level thread might view a kernel-level thread that it executes on. A parasite is suspended when it performs a blocking action (e.g., a synchronous communication operation, or I/O). Such a suspended parasite is said to have been *reified*. Reified parasites are represented as stack objects on the heap. Reified parasites can resume execution once the conditions that had caused it to block no longer hold. Thus, parasitic threads are not scheduled using the language runtime; instead they *self-schedule* in a demand-driven style based on flow properties dictated by the actions they perform.

Figure 2.1 shows the steps involved in a parasitic communication, or blocking event, and we illustrate the interaction between the parasitic threads and their hosts. The

host threads are depicted as rounded rectangles, parasitic threads are represented as blocks within their hosts, and each processor as a queue of host threads. The parasite that is currently executing on a given host and its stack is represented as a block with solid edges; other parasites are represented as blocks with dotted edges. Reified parasites are represented as shaded blocks. Host threads can be viewed as a collection of parasitic threads all executing within the same stack space. When a host thread is initially created it contains one such computation, namely the expression it was given to evaluate when it was spawned.

Initially, the parasite S_1 performs a blocking action on a channel or event, abstractly depicted as a circle. Hence, S_1 blocks and is reified. The thread T_1 that hosted S_1 continues execution by switching to the next parasite S_2. S_1 becomes runnable when it is unblocked. Part 2 of the figure shows the parasite R_1 on the thread T_2 invoking an unblocking action. This unblocks S_1 and schedules it on top of R_1. Thus, the parasitic threads implicitly migrate to the point of synchronization.

Further details about parasitic threads including its operational semantics and mapping of ACML primitives to parasitic threads can be found in [39].

2.3.2 Garbage Collector

MultiMLton garbage collector (GC) is optimized for throughput. It uses a single, contiguous heap, shared among all the cores, with support for local allocation and stop-the-world collection. In order to allow local allocation, each core requests a page-sized chunk from the heap. While a single lock protects the chunk allocation, objects are allocated within chunks by bumping a core-local heap frontier.

In order to perform garbage collection, all the cores synchronize on a barrier, with one core responsible for collecting the entire heap. The garbage collection algorithm is inspired from Sansom's [64] collector, which combines Cheney's two-space copying collector and Jonker's single-space sliding compaction collector. Cheney's copying collector walks the live objects in the heap just once per collection, while Jonker's

mark-compact collector performs two walks. But Cheney's collector can only utilize half of memory allocated for the heap. Sansom's collector combines the best of both worlds. Copying collection is performed when heap requirements are less than half of the available memory. The runtime system dynamically switches to mark-compact collection if the heap utilization increases beyond half of the available space.

Since ML programs tend to have a high rate of allocation, and most objects are short-lived temporaries, it is beneficial to perform generational collection. The garbage collector supports Appel-style generational collection [65] for collecting temporaries. The generational collector has two generations, and all objects that survive a generational collection are copied to the older generation. Generational collection can work with both copying and mark-compact major collection schemes.

MultiMLton enables its generational collector only when it is profitable, which is determined by the following heuristic. At the end of a major collection, the runtime system calculates the ratio of live bytes to the total heap size. If this ratio falls below a certain (tunable) threshold, then generational collection is enabled for subsequent collections. By default, this ratio is 0.25.

3 ANERIS: A COHERENT AND MANAGED RUNTIME FOR ML ON THE SCC

In this chapter, we describe ANERIS, an extension of MULTIMLTON that provides a coherent address space on the SCC, optimizing for the SCC's memory hierarchy. We begin with a *local collector*[1] (LC) design [46–48, 66–68] that partitions the heap into local heaps on each core and a shared heap for cross-core communication. However, we observe that the cost of memory barriers utilized in preserving the heap invariants have significant costs. To eliminate theses costs, we propose a new GC design (PRC) that utilizes the ample concurrency offered by our programming model combined with a dynamic shape analysis to eliminate some of the GC overheads. This naturally leads to a GC design that focuses on *procrastination* [49], delaying writes that would necessitate establishing forwarding pointers until a GC, where there is no longer a need for such pointers. The GC leverages the mostly functional nature of ACML programs and a new object property called *cleanliness*, which enables a broad class of objects to be moved from a local to a shared heap without requiring a full traversal of the local heap to fix existing references; cleanliness enables an important optimization that achieves the effect of procrastination without actually having to initiate a thread stall. Our final design (SMC) integrates SCC's support for software-managed cache coherence (SMC) [69] into the extant memory barriers to improve the design further.

We begin by discussing in detail the architecture and programming model of the SCC, which serves as our prototype non cache coherent architecture. However, the use of SCC by no means restricts the applicability of our ideas to other scalable manycore architectures [49]. Then, we present the three GC designs. Finally, we present a comprehensive evaluation of the three designs.

[1]Other terms have been used in the literature to indicate similar heap designs, notably private nursery, local heap collector, thread-local or thread-specific heap, and on-the-fly collection.

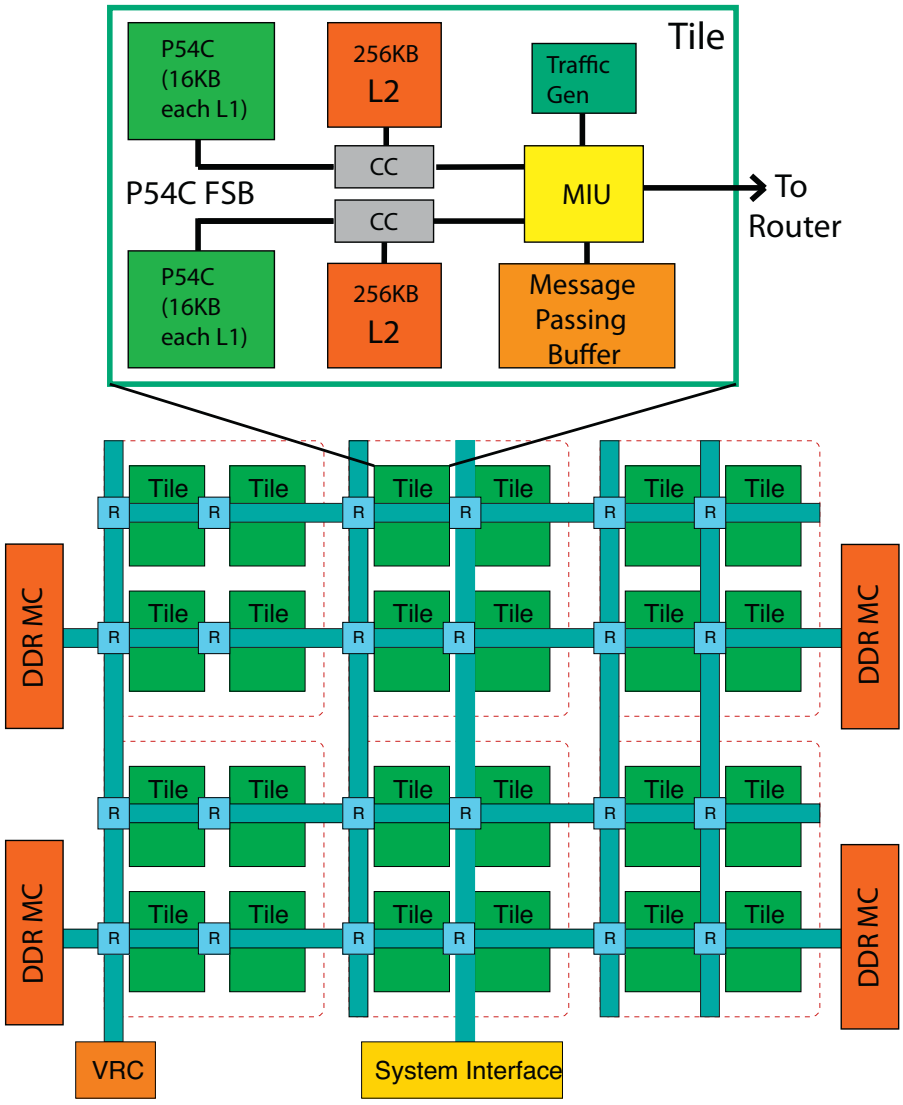


Figure 3.1. The architecture of the Intel SCC processor

3.1 The Intel Single-chip Cloud Computer

Intel SCC [26] (Figure 3.1) is a many-core processor with 48 P54C cores on a single chip, grouped as 24 tiles, organized in a 4 × 6 mesh network with a bisection bandwidth of 256 Gb/s. The most interesting aspect of the SCC architecture is the complete lack of cache coherence between the cores, and the presence of fast on-

die message-passing network interface. The 24 tiles on the chip are divided into 4 quadrants, and each quadrant is connected to a DDR3 memory controller. Each core has 16KB of private L1 instruction and data caches, and 256 KB of L2 cache shared with the other core on the same tile.

In addition, each tile has a 16KB message-passing buffer (MPB) used for message-passing between the cores. The message passing buffers are the only caches that are accessible across all of the cores. The data used in on-chip communication is read from MPB, cached in L1 cache, but bypasses the L2 cache. The cache uses no-allocate policy on writes, and L1 cache incorporates a write-combine buffer. According to the processor specifications [26], the read latencies in this architecture are:

$$\begin{aligned} \text{LocalMPB} &= 45\ k_{core} + 8\ k_{mesh} \\ \text{RemoteMPB} &= 45\ k_{core} + 4 * n * 2\ k_{mesh} \\ \text{DRAM} &= 40\ k_{core} + 4 * n * 2\ k_{mesh} + 46\ k_{ram} \end{aligned}$$

where k_{core}, k_{mesh} and k_{ram} are the cycles of core, mesh network and memory respectively. In our experimental setup, where 6 tiles share a memory controller, the number of hops n to the memory controller could be $0 < n \leq 8$. Hence, the DRAM accesses are far more expensive than the MPBs. Each core additionally has a test and set register that is accessible from all other cores. The SCC uses 32-bit Pentium cores. A programmable, software-managed Look-Up Table (LUT) provides a means for implementing hybrid private and shared address spaces in the system.

3.1.1 Software System

From the programmer's point of view, SCC resembles a cluster of nodes, with portions of memory shared between the cores. Each core runs a linux kernel image, and does not share any operating system services with the other cores. Since SCC does not provide hardware cache coherence, it provides software support for managing coherence. First, SCC provides support for tagging a specific virtual address space as

shared across all of the cores. Caching can also be selectively enabled on this address space; SCC tags this address space as having message passing buffer type (MPBT).

Data typed as MPBT bypass L2 and go directly to L1. SCC also provides a special, 1-cycle instruction called `CL1INVMB` that marks all data of type MPBT as invalid L1 lines. In addition, the usual `WBINVD` instruction can be used to flush and invalidate the L1 cache. Since the cores use write-combine buffers, a correct flushing procedure should also flush the write-combine buffers. SCC does not provide primitive support for this purpose, but write-combine buffers can easily be flushed in software by performing a series of dummy writes to distinct memory locations, which fills the buffer and flushes any previous writes.

Typically, a programmer works with release consistency in order to utilize cached shared virtual memory. The SMC [69] library provides `smcAcquire()` to fetch changes from other cores (invalidates MPBT cache lines in L1 cache) and issues `smcRelease()` to publish its updates (flushes the L1 cache, if the cache is operating in write back mode, and flushes the write-combine buffers).

SCC's software stack also includes cross-core message-passing libraries implemented over the MPBs, including RCCE [26] and RCKMPI [70]. RCCE is optimized for Single Program Multiple Data (SPMD) parallel programming model, where the program is structured in such a way that the sender and the receiver ideally arrive at the communication point at the same time. The sender writes the message to the MPB, while the receiver busy waits (invalidating its cache every iteration to fetch recent writes), waiting for a special flag value to be written along with the message. After the sender writes the flag, the receiver reads the message into its private memory, while the sender busy waits (also invalidating its cache every iteration). Finally, the receiver writes a completion flag, which concludes the message transfer.

It is worthwhile pointing out that RCCE uses just the MPBs, while RCKMPI uses MPBs for small messages (less than 8KB — the maximum message size that would fully fit in the MPB) and the DRAM for larger messages. Despite having a higher

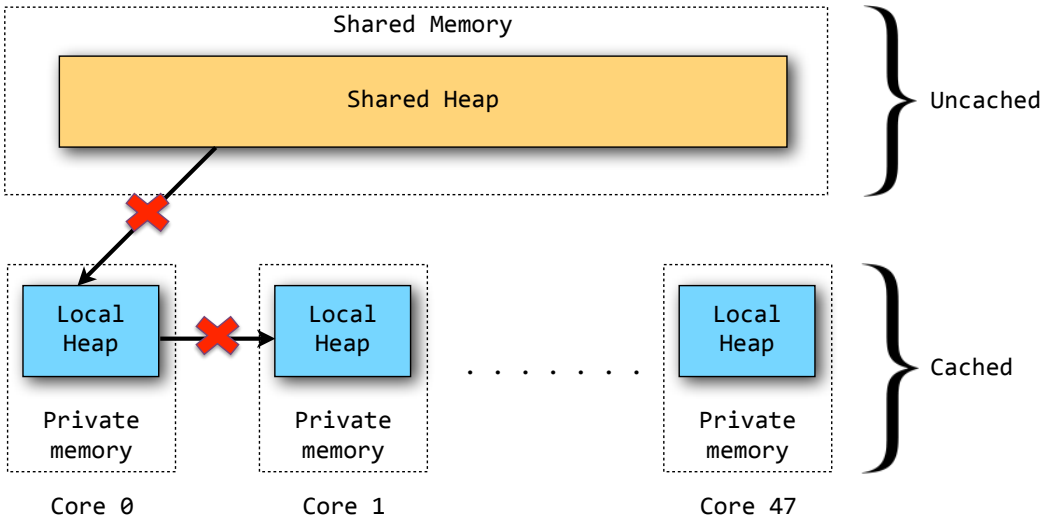


Figure 3.2. Local collector heap organization for the SCC

bandwidth and lower latency, the synchronization costs involved in transferring a large multi-part message over the MPB outweighs the benefits.

3.2 Local Collector (LC)

Splitting a program heap among a set of cores is a useful technique to exploit available parallelism on scalable multicore platforms: each core can allocate, collect, and access data locally, moving objects to a global, shared heap only when they are accessed by threads executing on different cores. This design allows local heaps to be collected independently, with coordination required only for global heap collection. In contrast, stop-the-world collectors need a global synchronization for every collection.

3.2.1 Heap Architecture

Our local collector design for the SCC is shown in Figure 3.2. The key idea here is that the local heaps are allocated in each of the cores *cached* private memory, into which new objects are allocated by default. The private heaps are allocated on the

memory banks closest to the core to optimize for the memory hierarchy and reduce mesh congestion. Since the design allows independent collection of local heaps, the design can scale to hundreds of cores [39], and benefits cache coherent architectures as well. The shared heap is allocated in the shared memory, and visible to all of the cores. In order to circumvent the coherence issues, we disable caching on the shared heap. Hence, every shared memory access goes to the DRAM. The shared heap pages are interleaved across all of the memory banks to uniformly spread the requests.

3.2.2 Heap Invariants

In order to ensure that cores cannot directly or indirectly access objects on other local heaps, which would complicate the ability to perform independent local heap collection, the following invariants need to be preserved:

- No pointers are allowed from one core's local heap to another.
- No pointers are permitted from the shared heap to the local heap.

Both invariants are necessary to perform independent local collections. The reason for the first is obvious. The second invariant prohibits a local heap from transitively accessing another local heap object via the shared heap. In order to preserve these invariants, the mutator typically executes a *write barrier* on every store operation. The write barrier ensures that before assigning a local object reference (source) to a shared heap object (target), the local object along with its transitive object closure is lifted to the shared heap. We call such writes *globalizing writes* as they export information out of local heaps. The execution of the write barrier creates *forwarding pointers* in the original location of the lifted objects in the local heap. These point to the new locations of the lifted objects in the shared heap. Since objects can be lifted to the shared heap on potentially any write, the mutator needs to execute a *read barrier* on potentially every read. The read barrier checks whether the object being read is the actual object or a forwarding pointer, and in the latter case, indirects to

the object found on the shared heap. Forwarding pointers are eventually eliminated during local collection.

3.2.3 Allocation and Collection

The allocations in the shared heap is performed similar to allocations in the stop-the-world collector, where each core allocates a page-sized chunk in the shared heap and performs object allocation by bumping its core-local shared heap frontier. Allocations in the local heaps do not require any synchronization. Garbage collection in the local heaps is similar to the baseline collector, except that it crucially does not require global synchronization.

Objects are allocated in the shared heap only if they are to be shared between two or more cores. Objects are automatically lifted to the shared heap because of globalizing writes and spawning a thread on a different core. Apart from these, all globals are allocated in the shared heap, since globals are visible to all cores by definition. Thus, for the ML programmer on this system, the absence of cache coherence is completely hidden, the SCC appears as a cache coherent multicore machine.

For a shared heap collection, all of the cores synchronize on a barrier and then a single core collects the heap. Along with globals, all the live references from local heaps to the shared heap are considered to be roots for a shared heap collection. In order to eliminate roots from dead local heap objects, before a shared heap collection, local collections are performed on each core to eliminate such references.

The shared heap is also collected using Sansom's dual-mode garbage collector. However, we do not perform generational collection on the shared heap. This is because of two reasons. First, objects in the shared heap, shared between two or more cores, are expected to live longer than a typical object collected during generational collection. Secondly, shared heap collection requires global synchronization, and it is wise to perform such collections rarely.

3.2.4 Remembered Stacks

In MultiMLton threads can synchronously or asynchronously communicate with each other over first-class message-passing communication channels. If a receiver is not available, a sender thread, or in the case of asynchronous communication the implicitly created thread, can block on a channel. If the channel resides in the shared heap, the thread object, its associated stack and the transitive closure of all objects reachable from it on the heap would be lifted to the shared heap as part of the blocking action. Since channel communication is the primary mode of thread interaction in our system, we would quickly find that most local heap objects end up being lifted to the shared heap. This would be highly undesirable.

Hence, we choose never to move stacks to the shared heap. We add an exception to our heap invariants to allow thread $\rightarrow$ stack pointers, where the thread resides on the shared heap, and references a stack object found on the local heap. Whenever a thread object is lifted to the shared heap, a reference to the corresponding stack object is added to the set of remembered stacks. This remembered set is considered as a root for a local collection to enable tracing of remembered stacks.

Before a shared heap collection, the remembered set is cleared; only those stacks that are reachable from other GC roots survive the shared heap collection. After a shared heap collection, the remembered set of each core is recalculated such that it contains only those stacks, whose corresponding thread objects reside in the shared heap, and have survived the shared heap collection.

Remembered stacks prevent thread local objects from being lifted to the shared heap, but require breaking the heap invariant to allow a thread object in the shared heap to refer to a stack object on the local heap. This relaxation of heap invariant is safe. The only object that can refer to thread-local stacks is the corresponding thread object. The thread objects are completely managed by the scheduler, and are not exposed to the programmer. As a result, while the local heap objects can point to a shared-heap thread object, whose stack might be located on a different local heap,

```
1 pointer readBarrier (pointer p) {
2   if (!isPointer(p)) return p;
3   if (getHeader(p) == FORWARDED)
4     return *(pointer*)p;
5   return p;
6 }
```

Figure 3.3. Read barrier.

the only core that can modify such a stack (by running the thread) is the core that owns the heap in which the stack is located. Thus, there is no possibility of direct references between local heaps. Hence, the remembered stack strategy is safe with respect to garbage collection.

3.2.5 Read Barrier and Overheads

In a mostly functional language like Standard ML, the number of reads are far likely to outweigh the number of mutations. Because of this fact, the aggregate cost of read barriers can be both substantial and vary dramatically based on underlying architecture characteristics [71]. To this end, we describe our read barrier design, and the cost/benefit of read barriers in our system.

Read Barrier Design

Figure 3.3 shows the pseudo-C code for our read barrier. Whenever an object is lifted to the shared heap, the original object's header is set to `FORWARDED`, and the first word of the object is overwritten with the new location of the object in the shared heap. Before an object is read, the mutator checks whether the object has been forwarded, and if it is, returns the new location of the object. Hence, our read barriers are conditional [71, 72].

MLton represents non-value carrying constructors of (sum) datatypes using non-pointer values. If such a type additionally happens to have value-carrying constructors that reference heap-allocated objects, the non-pointer value representing the empty constructor will be stored in the object pointer field. Hence, the read barrier must first check whether the presumed pointer does in fact point to a heap object. Otherwise, the original value is returned (line 2). If the given pointer points to a forwarded object, the current location of the object in the shared heap is returned. Otherwise, the original value is returned.

While our read barrier implementation is conditional [72], there exist unconditional variants [73], where all loads unconditionally forward a pointer in the object header to get to the object. For objects that are not forwarded, this pointer points to the object itself. Although an unconditional read barrier, would have avoided the cost of the second branch in our read barrier implementation, it would necessitate having an additional address length field in the object header for an indirection pointer.

Most objects in our system tend to be small. In our benchmarks, we observed that 95% of the objects allocated were less than 3 words in size, including a word-sized header. The addition of an extra word in the object header for an indirection pointer would lead to substantial memory overheads, which in turn leads to additional garbage collection costs. Moreover, trading branches with loads is not a clear optimization as modern processors allow speculation through multiple branches, especially ones that are infrequent. Hence, we choose to encode read barriers conditionally rather than unconditionally.

In addition, MultiMLton performs a series of optimizations to minimize heap allocation, thus reducing the set of read barriers actually generated. For example, references and arrays that do not escape out of a function are flattened. Combined with aggressive inlining and simplification optimizations enabled by whole-program compilation, object allocation on the heap can be substantially reduced.

The compiler and runtime system ensure that entries on thread stacks never point to a forwarded object. Whenever an object pointer is stored into a register or the stack,

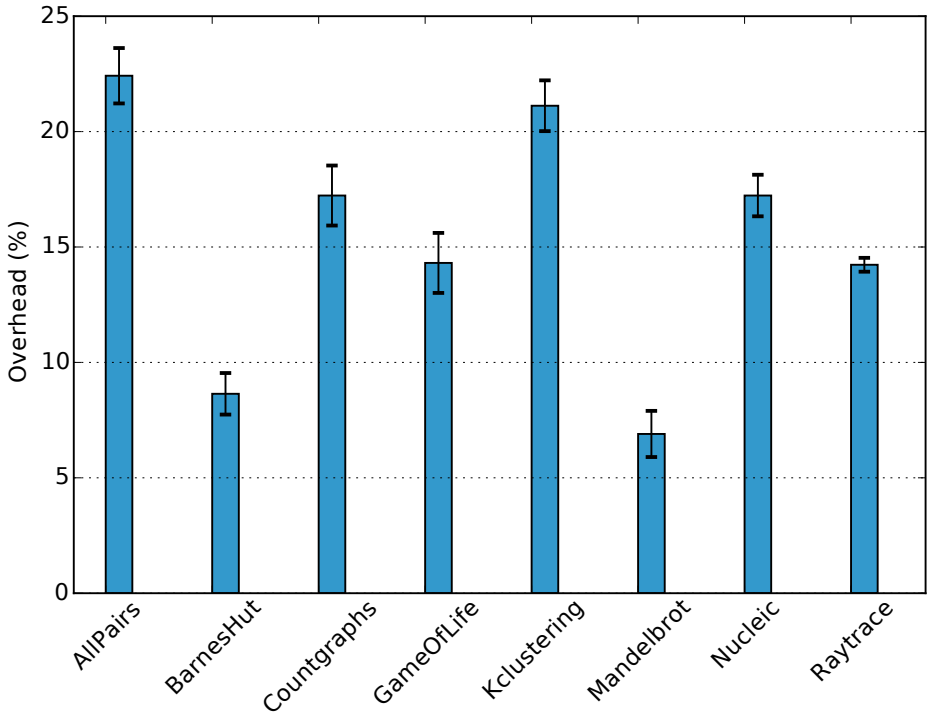


Figure 3.4. Read barrier overhead as a percentage of mutator time.

a read barrier is executed on the object pointer to get the current location of the object. Immediately after an globalizing write or a context switch, the current stack is walked and references to forwarded objects are updated to point to the new location of lifted objects in the shared heap. Additionally, before performing an globalizing write, register values are saved on the stack, and reloaded after exit. Thus, as a part of fixing references to forwarding pointers from the stack, references from registers are also fixed. This ensures that the registers never point to forwarded objects either. Hence, no read barriers are required for dereferencing object pointers from the stack or registers. This optimization is analogous to "eager" read barriers as described in [74]. Eager read barrier elimination has marked location is loaded into a register, but all further accesses can elide executing the barrier.

Table 3.1.
Effectiveness of read barrier checks: RB invocations represents the average number of read barrier invocations and forwarded represents the average number of instances when the read barrier encountered a forwarded object.

Benchmark	**RB invocations** ($\times 10^6$)	**Forwarded**
AllPairs	9,753 ±431	123 ±11
BarnesHut	2,864 ±176	52,702 ±1830
CountGraphs	2,584 ±119	0 ±0
GameOfLife	4,858 ±276	2,143 ±43
KClustering	3,780 ±265	101 ±7
Mandelbrot	2,980 ±79	23 ±3
Nucleic	2,887 ±135	328 ±21
Raytrace	2,217 ±90	0 ±0

Evaluation

We evaluated a set of 8 benchmarks (described in Section 3.5) each running on all 48 cores on the SCC to measure read barrier overheads. Figure 3.4 shows these overheads as a percentage of mutator time. Our experiments reveal that, on average, the mutator spends 15.3% of the time executing read barriers for our benchmarks.

The next question to ask is whether the utility of the read barrier justifies its cost. To answer this question, we measure the number of instances the read barrier is invoked and the number of instances the barrier finds a forwarded object (see Table 3.1). We see that read barriers find forwarded objects in less than one thousandth of a percent of the number of instances they are invoked. Thus, in our system, the cost of read barriers is substantial, but only rarely do they have to perform the task of forwarding references. These results motivate our interest in a memory management design that eliminates read barriers altogether.

3.3 Procrastinating Collector (PRC)

Eliminating read barriers, however, is non-trivial. Abstractly, one can avoid read barriers by eagerly *fixing* all references that point to forwarded objects at the time the object is lifted to the shared heap, ensuring the mutator will never encounter a forwarded object. Unfortunately, this requires being able to enumerate all the references that point to the lifted object; in general, gathering this information is very expensive as the references to an object might originate from any object in the local heap.

We consider an alternative design that completely eliminates the need for read barriers *without* requiring a full scan of the local heap whenever an object is lifted to the shared heap. The design is based on the observation that read barriers can be clearly eliminated if forwarding pointers are never introduced. One way to avoid introducing forwarding pointers is to *delay* operations that create them until a local garbage collection is triggered. In other words, rather than executing a store operation that would trigger lifting a thread local object to the shared heap, we can simply *procrastinate*, thereby stalling the thread that needs to perform the store. The garbage collector must simply be informed of the need to lift the object's closure during its next local collection. After collection is complete, the store can take place with the source object lifted, and all extant heap references properly adjusted. As long as there is sufficient concurrency to utilize existing computational resources, in the form of available runnable threads to run other computations, the cost of procrastination is just proportional to the cost of a context switch.

Moreover, it is not necessary to always stall an operation that involves lifting an object to the shared heap. We consider a new property for objects (and their transitive object closures) called *cleanliness*. A clean object is one that can be safely lifted to the shared heap without introducing forwarding pointers that might be subsequently encountered by the mutator: objects that are immutable, objects only referenced from the stack, or objects whose set of incoming heap references is known, are obvious

examples. The runtime analysis for cleanliness is combined with a specialized write barrier to amortize its cost. Thus, procrastination provides a general technique to eliminate read barriers, while cleanliness serves as an important optimization that avoids stalling threads unnecessarily.

The effectiveness of our approach depends on a programming model in which (a) most objects are clean, (b) the transitive closure of the object being lifted rarely has pointers to it from other heap allocated objects, and (c) there is a sufficient degree of concurrency in the form of runnable threads; this avoids idling available cores whenever a thread is stalled performing an globalizing write that involves an unclean object. We observe that conditions (a) and (b) are common to functional programming languages and condition (c) follows from the ACML runtime model. Our technique does not rely on programmer annotations, static analysis or compiler optimizations to eliminate read barriers, and can be completely implemented as a lightweight runtime technique.

3.3.1 Cleanliness Analysis

Although ACML provides an abundance of concurrency, with the procrastination mechanism, many of the threads in a program may end up blocked on globalizing writes, waiting for a local garbage collection to unblock them. If all of the threads on a particular core have procrastinated, then a local garbage collection is needed in order to make progress. Such *forced* local garbage collections make the program run longer, and hence subdue the benefit of eliminating read barriers. Hence, it is desirable to avoid procrastination whenever possible.

In this section, we describe our cleanliness analysis, which identifies objects on which globalizing writes do not need to be stalled. We first present auxiliary definitions that will be utilized by cleanliness checks, and then describe the analysis.

Heap Session

Objects are allocated in the local heap by bumping the local heap frontier. In addition, associated with each local heap is a pointer called `sessionStart` that always points to a location between the start of the heap and the frontier. This pointer introduces the idea of a *heap session*, to capture the notion of recently allocated objects. Every local heap has exactly two sessions: a *current session* between the `sessionStart` and the heap frontier and a *previous session* between the start of the heap and `sessionStart`. Heap sessions are used by the cleanliness analysis to limit the range of heap locations that need to be scanned to test an object closure[2] for cleanliness. Assigning the current local heap frontier to the `sessionStart` pointer starts a new session. We start a new session on a context switch, a local garbage collection and after an object has been lifted to the shared heap.

Reference Count

We introduce a limited reference counting mechanism for local heap objects that counts the number of references from other local heap objects. Importantly, we do not consider references from ML thread stacks. The reference count is meaningful only for objects reachable in the current session. For such objects, the number of references to an object can be one of four values: `ZERO`, `ONE`, `LOCAL_MANY`, and `GLOBAL`. We steal 2 bits from the object header to record this information. A reference count of `ZERO` indicates that the object only has references from registers or stacks, while an object with a count of `ONE` has exactly one pointer from the current session. A count of `LOCAL_MANY` indicates that this object has more than one reference, but that all of these references originate from the current session. `GLOBAL` indicates that the object has at least one reference that originates from outside the current session.

[2]In the following, we write *object closure* to mean the set of objects reachable from some root on the heap; to avoid confusion, we write *function closure* to mean the representation of an SML function as a pair of function code pointer and static environment.

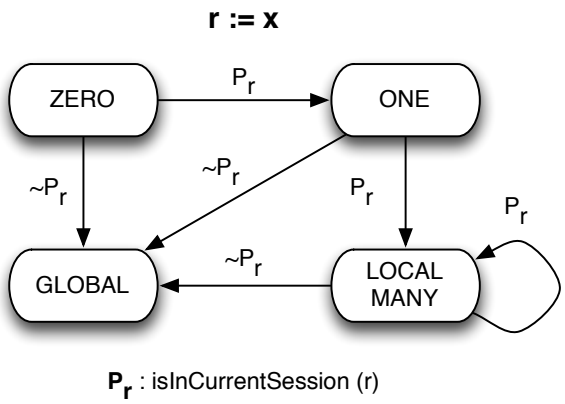


Figure 3.5. State transition diagram detailing the behavior of the reference counting mechanism with respect to object `x` involved in an assignment, $\mathsf{r} := \mathsf{x}$, where $\mathsf{P}_\mathrm{r} = \mathsf{isInCurrentSession(r)}$.

The reference counting mechanism is implemented as a part of the write barrier (Lines 13–22 in Figure 3.8). Figure 3.5 illustrates the state transition diagram for the reference counting mechanism. Observe that reference counts are non-decreasing. Hence, the reference count of any object represents the maximum number of references that pointed to the object at any point in its lifetime.

Cleanliness

An object closure is said to be clean, if for each object reachable from the root of the object closure,

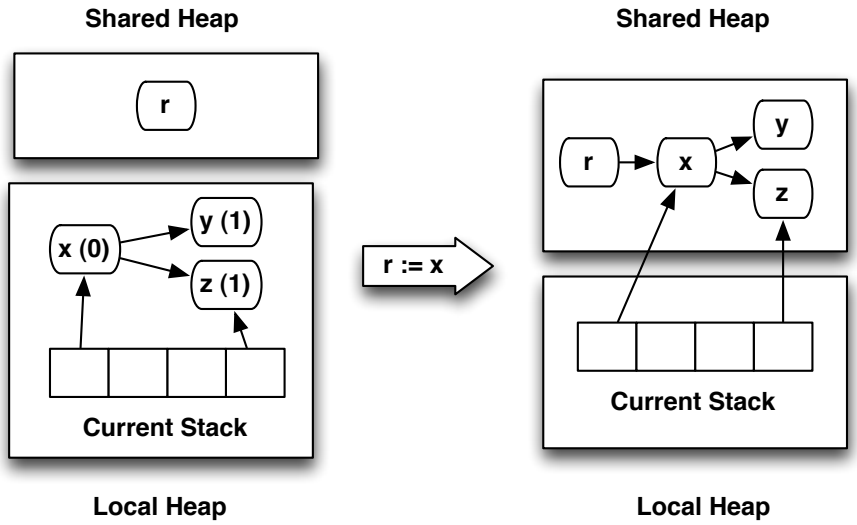
- the object is immutable or in the shared heap. Or,
- the object is the root, and has `ZERO` references. Or,
- the object is not the root, and has `ONE` reference. Or,
- the object is not the root, has `LOCAL_MANY` references, and is in the current session.

Otherwise, the object closure is not clean.

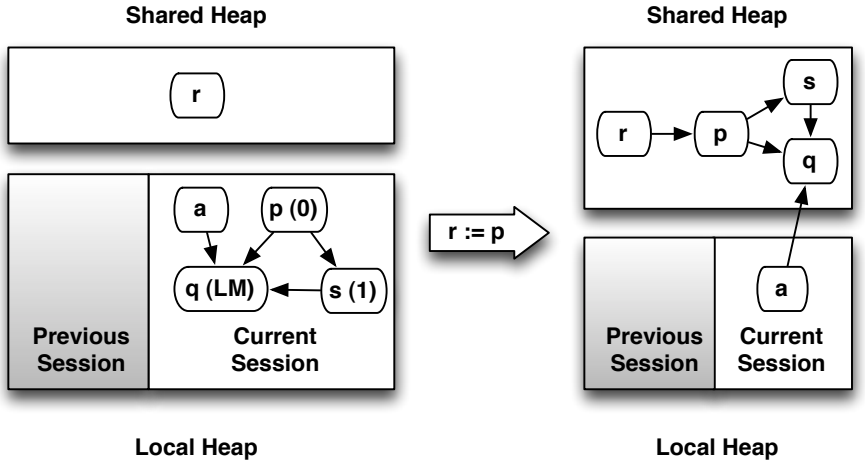
```
1  bool isClean (pointer p, bool* isLocalMany) {
2    clean = true;
3    foreach o in reachable(p) {
4      if (!isMutable(o) || isInSharedHeap(o))
5        continue;
6      nv = getRefCount(o);
7      if (nv == ZERO)
8        clean &&= true;
9      else if (nv == ONE)
10       clean &&= (o != p);
11     else if (nv == LOCAL_MANY) {
12       clean &&= (isInCurrentSession(o));
13       *isLocalMany = true;
14     }
15     else
16       clean = false;
17   }
18   return clean;
19 }
```

Figure 3.6. Cleanliness check.

Figure 3.6 shows an implementation of an object closure cleanliness check. Since the cleanliness check, memory barriers, and the garbage collector are implemented in low-level code (C, assembly and low-level intermediate language in the compiler), this code snippet, and others that follow in this section are in pseudo-C language, to better represent their implementation. If the source of an globalizing assignment is immutable, we can make a copy of the immutable object in the shared heap, and avoid introducing references to forwarded objects. Standard ML does not allow the programmer to test the referential equality of immutable objects. Equality of



(a) Tree-structured object closure



(b) Session-based cleanliness

Figure 3.7. Utilizing object closure cleanliness information for globalizing writes to avoid references to forwarded objects.

immutable objects is always computed by structure. Hence, it is safe to replicate immutable objects. If the object is already in the shared heap, there is no need to move this object.

If the object closure of the source of a globalizing write is clean, we can move the object closure to the shared heap and quickly fix all of the forwarding pointers that might be generated. For example, consider an object that defines a tree structure; such an object is clean if the root has `ZERO` references and all of its internal nodes have `ONE` reference from their parent. A root having `ZERO` references means it is accessed only via the stack; if it had a count of `ONE`, the outstanding reference may emanate from the heap. Internal nodes having a reference count of `ONE` implies they are reachable only via other nodes in the object being traced. Figure 3.7(a) shows such an object closure. In this example, we assume that all objects in the object closure are mutable. The reference count of relevant nodes is given in the brackets. Both the root and internal nodes can have pointers from the current stack not tracked by the reference count. After lifting the object closure, the references originating from the current stack are fixed by walking the stack.

Object closures need not just be trees and can be arbitrary graphs, with multiple incoming edges to a particular object in the object closure. How do we determine if the incoming edges to an object originate from the object closure or from outside the object closure (from the local heap)? We cannot answer this question without walking the local heap. Hence, we simplify the question to asking whether all the pointers to an object originate from the current session. This question is answered in the affirmative if an object has a reference count of `LOCAL_MANY` (lines 11–13 in Figure 3.6).

Figure 3.7(b) shows an example of a object closure whose objects have at most `LOCAL_MANY` references. Again, we assume that all objects in the object closure are mutable. In the transitive object closure rooted at `p`, object `q` has locally many references. These references might originate from the object closure itself (edges `p` $\rightarrow$ `q` and `s` $\rightarrow$ `q`) or from outside the object closure (edge `a` $\rightarrow$ `q`). After lifting such object closures to the shared heap, only the current session is walked to fix all of the references to forwarded objects created during the copy. In practice (Section 3.5.2),

current session sizes are much smaller than heap sizes, and hence globalizing writes can be performed quickly.

Finally, in the case of `LOCAL_MANY` references, the object closure is clean, but unlike other cases, after lifting the object closure to the shared heap, the current session must be walked to fix any references to forwarded objects. This is indicated to the caller of `isClean` function by assigning `true` to `*isLocalMany`, and is used in the implementation of lifting an object closure to the shared heap (Figure 3.9).

3.3.2 Write Barrier

In this section, we present the modifications to the write barrier to eliminate the possibility of creating references from reachable objects in the local heap to a forwarded object. The implementation of our write barrier is presented in Figure 3.8. A write barrier is invoked prior to a write and returns a new value for the source of the write. The check `isObjptr` at line 2 returns true only for heap allocated objects, and is a compile time check. Hence, for primitive valued writes, there is no write barrier. Lines 4 and 5 check whether the write is globalizing. If the source of the object is clean, we lift the transitive object closure to the shared heap and return the new location of the object in the shared heap.

Delaying Writes

If the source of an globalizing write is not clean, we suspend the current thread and switch to another thread in our scheduler. The source of the write is added to a queue of objects that are waiting to be lifted. Since the write is not performed, no forwarded pointers are created. If programs have ample amounts of concurrency, there will be other threads that are waiting to be run. However, if all threads on a given core are blocked on a write, we move all of the object closures that are waiting to be lifted to the shared heap. We then force a local garbage collection, which will,

```
1  Val writeBarrier (Ref r, Val v) {
2    if (isObjptr(v)) {
3      //Lift if clean or procrastinate
4      if (isInSharedHeap(r) &&
5          isInLocalHeap(v)) {
6        isLocalMany = false;
7        if (isClean(v, &isLocalMany))
8          v = lift(v, isLocalMany);
9        else
10         v = suspendTillGCAndLift(v);
11     }
12     //Tracking cleanliness
13     if (isInLocalHeap (r) &&
14         isInLocalHeap(v)) {
15       n = getRefCount(v);
16       if (!isInCurrentSession (r))
17         setNumRefs(v, GLOBAL);
18       else if (n == ZERO)
19         setNumRefs(v, ONE);
20       else if (n < GLOBAL)
21         setNumRefs(v, LOCAL_MANY);
22     }
23   }
24   return v;
25 }
```

Figure 3.8. Write barrier implementation.

as a part of the collection, fix all of the references to point to the new (lifted) location on the shared heap. Thus, the mutator never encounters a reference to a forwarded object.

Lifting Objects to the Shared Heap

Figure 3.9 shows the pseudo-C code for lifting object closures to the shared heap. The function `lift` takes as input the root of a clean object closure and a Boolean representing whether the object closure has any object that has `LOCAL_MANY` references. For simplicity of presentation, we assume that the shared heap has enough space reserved for the transitive object closure of the object being lifted. In practice, the lifting process requests additional shared heap chunks to be reserved for the current processor, or triggers a shared heap collection if there is no additional space in the shared heap.

Objects are transitively lifted to the shared heap, starting from the root, in the obvious way (Lines 17–18). As a part of lifting, mutable objects are lifted and a forwarding pointer is created in their original location, while immutable objects are copied and their location added to `imSet` (Lines 9–10). After lifting the transitive object closure to the shared heap, the shared heap frontier is updated to the new location.

After object lifting, the current stack is walked to fix any references to forwarding pointers (Line 21). Since we do not track references from the stack for reference counting, there might be references to forwarded objects from stacks other than the current stack. We fix such references lazily. Before a context switch, the target stack is walked to fix any references to forwarded objects. Since immutable objects are copied and mutable objects lifted, a copied immutable object might point to a forwarded object. We walk all the shared heap copies of immutable objects lifted from the local heap to fix any references to forwarded objects (Lines 22–23).

Recall that if the object closure was clean, but has `LOCAL_MANY` references, then it has at least one pointer from the current session. Hence, in this case, we walk the current session to fix the references to any forwarded objects to point to their shared heap counterparts (lines 25–27). Finally, session start is moved to the current frontier.

```
1  Set imSet;
2  void liftHelper (pointer* op, pointer* frontierP) {
3    frontier = *frontierP;
4    o = *op;
5    if (isInSharedHeap(o)) return;
6    copyObject (o, frontier);
7    *op = frontier + headerSize(o);
8    *frontierP = frontier + objectSize(o);
9    if (isMutable(o)) {setHeader(o, FORWARDED); *o = *op;}
10   else imSet += o;
11 }
12
13 pointer lift (pointer op, bool isLocalMany) {
14   start = frontier = getSharedHeapFrontier();
15   imSet = {};
16   //Lift transitive object closure
17   liftHelper (&op, &frontier);
18   foreachObjptrInRange (start, &frontier, liftHelper);
19   setSharedHeapFrontier(frontier);
20   //Fix forwarding pointers
21   foreachObjptrInObject (getCurrentStack(), fixFwdPtr);
22   foreach o in imSet
23     foreachObjptrInObject(o,fixFwdPtr);
24   frontier = getLocalHeapFrontier();
25   if (isLocalMany)
26     foreachObjptrInRange
27       (getSessionStart(), &frontier, fixFwdPtr);
28   setSessionStart(frontier);
29   return op;
30 }
```

Figure 3.9. Lifting an object closure to the shared heap.

```
1  ThreadID spawn (pointer closure, int target) {
2    ThreadID tid = newThreadID();
3    Thread t = newThread(closure, tid);
4    isLocalMany = false;
5    if (isClean(t, &isLocalMany)) {
6      t = lift(t, isLocalMany);
7      enqueThread(t, target);
8    }
9    else
10     liftAndReadyBeforeGC(t, target);
11   return tid;
12 }
```

Figure 3.10. Spawning a thread.

Remote Spawns

Apart from globalizing writes, function closures can also escape local heaps when threads are spawned on other cores. For spawning on other cores, the environment of the function closure is lifted to the shared heap and then, the function closure is added to the target core's scheduler. This might introduce references to forwarding pointers in the spawning core's heap. We utilize the techniques developed for globalizing writes to handle remote spawns in a similar fashion.

Figure 3.10 shows the implementation of thread spawn. If the function closure is clean, we lift the function closure to the shared heap, and enqueue the thread on the target scheduler. Otherwise, we add it to the list of threads that need to be lifted to the shared heap. Before the next garbage collection, these function closures are lifted to the shared heap, enqueued to target schedulers, and the references to forwarded objects are fixed as a part of the collection. When the target scheduler finds this new thread (as opposed to other preempted threads), it allocates a new stack in the local

heap. Hence, except for the environment of the remotely spawned thread, all data allocated by the thread is placed in the local heap.

Barrier Implementation

In our local collector, the code for tracking cleanliness (Lines 13–24 in Figure 3.8) is implemented as an RSSA pass. In RSSA, we are able to distinguish heap allocated objects from non-heap values such as constants, values on the stack and registers, globals, etc. This allows us to generate barriers only when necessary.

The code for avoiding creation of references to forwarded objects (Lines 4–11 in Figure 3.8) is implemented in the primitive library, where we have access to the lightweight thread scheduler. `suspendTillGCAndLift` (line 10 in Figure 3.8) is carefully implemented to not contain an globalizing write, which would cause non-terminating recursive calls to the write barrier.

3.4 Integrating Software-Managed Cache Coherence (SMC)

In our next Aneris design, we integrate the SCC specific features into the runtime system. Specifically, we describe a new GC design that integrates software-managed cache coherence (SMC) capability, and utilizing the message-passing buffers for inter-core communication. The key enabling feature in both cases is the fact that Standard ML is a mostly functional language, and MultiMLton's ability to discriminate objects at runtime based on the mutability information.

3.4.1 Heap Design

The heap design for taking advantage of SMC is given in Figure 3.11. The design is similar to the local collector design (Section 3.2) with one key difference. Instead of a single uncached shared heap, we split our shared heap into cached and uncached

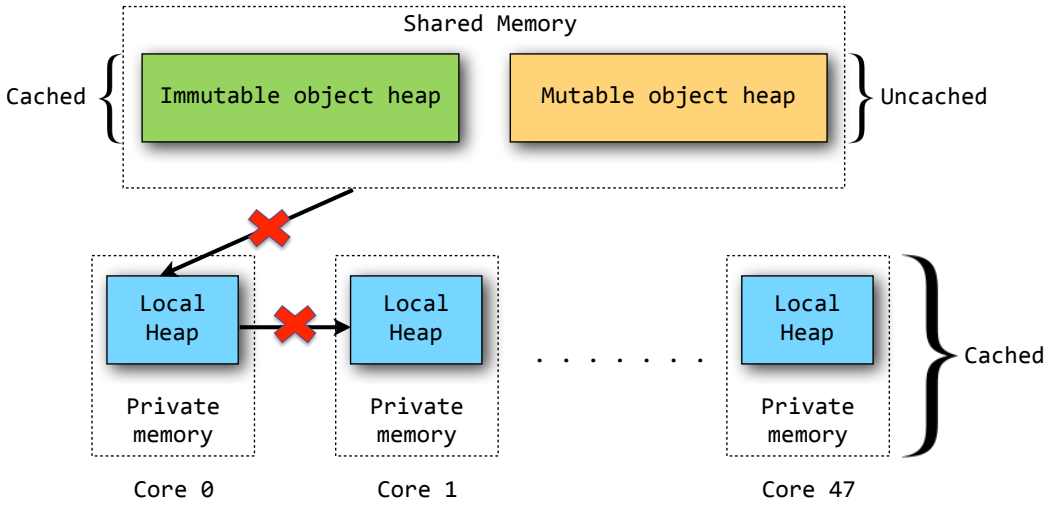


Figure 3.11. Heap design utilizing SCC's software-managed cache coherence capability.

partitions. We take advantage of the fact that standard ML can statically distinguish between mutable and immutable objects. Since immutable objects by definition will not change after initialization, we enable caching on one of the shared heaps into which only globalized immutable objects will be allocated. We call this heap a cached shared heap (CSH). Since most objects in standard ML are immutable, we gain the advantage of caching by placing these objects in CSH while not having to deal with coherence issues. CSH is implemented using Software Managed Coherence (SMC) for SCC [69], and we mark CSH as a MPB type area such that any cache line from CSH will be tagged as MPB type. The CSH cached data bypasses L2 and caching operates in a write-through mode.

Caching is disabled in the uncached shared heap (USH) into which globalized mutable objects are allocated. By disabling caching, we circumvent the coherence issues at the cost of performance. A local heap object being globalized might contain both mutable and immutable objects in its transitive object closure. Hence, globalization might involve allocating new objects in both partitions of the shared heap. For the same reason, pointers are allowed between the two partitions of the shared heap.

```
1  pointer readBarrier (pointer p) {
2    if (!isPointer(p)) return p;
3    if (getHeader(p) == FORWARDED) {
4      /* Address in shared heap */
5      p = *(pointer*)p;
6      if (p > MAX_CSH_ADDR) {
7        /* Address in cached shared heap, and has not
8         * been seen so far. Fetch the updates. */
9        smcAcquire();
10       MAX_CSH_ADDR = p;
11     }
12   }
13   return p;
14 }
```

Figure 3.12. Read barrier with software-managed cache coherence capability.

3.4.2 Memory Consistency

The key challenge now is to ensure that the updates to CSH are visible to all the cores. Since CSH is cached, and SCC does not provide hardware cache coherence, explicit cache invalidations and flushes must be implemented. Moreover, any missed flushes or invalidations will lead to incoherent caches, while frequent flushes or invalidations leads to poor performance. The key observation is that the baseline local collector design has both read and write barriers; our idea is to integrate the cache control primitives into the memory barriers.

The CSH is always mapped to an address that is greater than the starting address of the USH. Each core maintains the largest address seen in CSH in the `MAX_CSH_ADDR` variable. During an object read, if the address of the object lies in the shared heap and is greater than `MAX_CSH_ADDR`, we invalidate any cache lines that might be associated

```
1  val writeBarrier (Ref r, Val v) {
2    if (isObjptr(v) && isInSharedHeap(r)
3        && isInLocalHeap(v)) {
4        /* Move transitive object closure to shared
5         * heap, and install forwarding pointers */
6        v = globalize (v);
7        /* Publish the updates */
8        smcRelease();
9    }
10   return v;
11 }
```

Figure 3.13. Write barrier with software-managed cache coherence capability.

with CSH by invoking `smcAcquire()` (Line 9 in Figure 3.12). This ensures that the values read are not stale. We update the `MAX_CSH_ADDR` if necessary. Since the objects in CSH are immutable, there is no need to perform cache invalidation while reading an address that is less than `MAX_CSH_ADDR`. Additionally, after garbage collection, `MAX_CSH_ADDR` is set to point to the start of the CSH.

Similarly, whenever an object is globalized to the CSH, we must ensure that the updates are visible to all of the cores. After an globalizing write, we invoke `smcRelease()`, to flush any outstanding writes to the memory (Line 8 in Figure 3.13).

3.4.3 Mapping Channel Communication over Message-Passing Buffers

In this section, we describe how we map the MultiMLton communication model on top of the MPB. The main challenge here is the compatibility between the MultiMLton communication model and the capabilities of MPB. In MultiMLton, threads communicate over first-class channels, which support many-to-many communication pattern. Hence, a receiver does not know the identity of the sender and

vice versa. Moreover, if a receiver is not available, the sender thread blocks; it is descheduled, and some other thread from the scheduler queue continues execution on that core. Moreover, the values sent over the channels in MultiMLton can be mutable. The channel itself is simply a data structure implemented over shared memory. However, the communication on the SCC over libraries such as RCCE and RCKMPI is optimized for SPMD programming model, where the sender and the receiver know each other's identities, and are expected to arrive at the communication point at the same time. Hence, considering the fact that MPB memory is on 8KB, both RCCE and RCKMPI use busy waiting strategy for inter-core communication. Thus, careful design is needed to map MultiMLton communication abstraction over the MPB.

Our channel mapping implementation exploits both the cached shared heap and MPB for efficient inter-core message passing. We take advantage of our heap layout and the availability of static type information to take advantage of the fast, on-die MPB memory. We consider the following five cases:

1. Channel is located in the local heap
2. Channel is located in the shared heap, and the message being sent is an unboxed value
3. Channel is located in the shared heap, the message is in the local heap, and at least one of the objects in the transitive closure of the message being sent is mutable
4. Channel is located in the shared heap, the message is in the local heap, and all objects in the transitive closure of the message being sent are immutable
5. Channel and the message are located in the shared heap

For case 1, we observe that only channels that are located in the shared heap can be used for inter-core communication. Our heap invariants prevent pointers from one local heap to another. Thus, if a channel is located in the local heap, then no thread

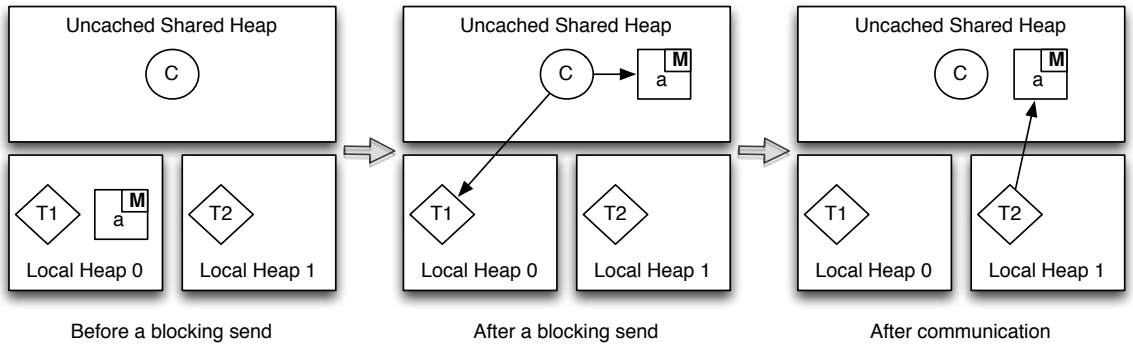


Figure 3.14. Steps involved in sending an mutable object `a` by thread `T1` on a shared heap channel `C`, which is eventually received by thread `T2`.

on the other cores have a reference to this channel. Thus, communication under case 1 only involves a value or a pointer exchange between the communicating lightweight threads.

MultiMLton supports unboxed types that represent raw values. Hence, under case 2, we add a reference to the thread along with the value being sent to the channel. In addition, we add a reference to the blocked thread to the remembered list so that the local garbage collection can trace it.

If the message being sent has a mutable object in the transitive closure, we must make this object visible to both the sender and the receiver core. Figure 3.14 shows the case where a thread `T1` sends a mutable object `a` on a shared heap channel `C`. In this case, we eagerly globalize `a` before `T1` blocks. Since the message is already in the shared heap, when the receiver thread `T2` eventually arrives, it just picks up a pointer to the message in the shared heap.

Figure 3.15 shows the case where a thread `T1` sends an immutable object `a` on a shared channel `C`. Here, we simply add to the channel `C`, a reference to the message `a` in the local heap, along with the reference to the thread `T1`. In addition, a reference

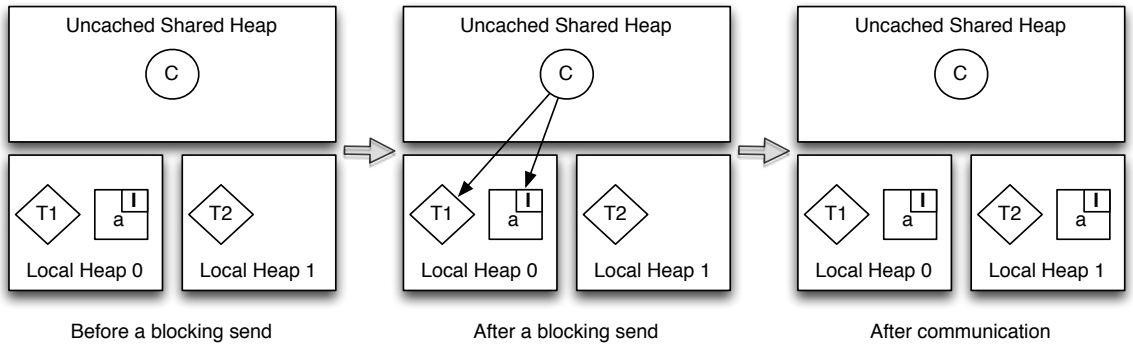


Figure 3.15. Steps involved in sending an immutable object `a` by thread `T1` on a shared heap channel `C`, which is eventually received by thread `T2`.

to the object `a` is added to the remembered list, so that a local garbage collection will be able to identify `a` as being alive.

Afterward, when the receiver thread `T2` arrives and finds the message not to be in the shared heap, it sends an inter-core interrupt to the core on which the message is located (core 0, in this case). After this message transfer is initiated over the MPB using RCCE to transfer the object `a` from core 0 to core 1. Since Standard ML immutable objects do not have identity, making a copy of the immutable object is safe under MultiMLton.

If the channel and the message are located in the shared heap, communication only involves a value or a pointer exchange. This case is similar to case 1.

3.5 Evaluation

The core, mesh controller, and memory on the SCC can be configured to run at different frequencies. For our experiments we chose 533 MHz, 800 MHz, and 800 MHz for core, mesh, and memory respectively. In our results, wherever appropriate, we present the 95% confidence intervals, obtained using Student's t-distribution.

For our experimental evaluation, we picked 8 benchmarks from the MLton benchmark suite. The benchmarks were derived from sequential standard ML implementation and were parallelized using ACML [56]. The benchmarks are:

- **AllPairs**: an implementation of Floyd-Warshall algorithm for computing all pairs shortest path.
- **BarnesHut**: an n-body simulation using Barnes-Hut algorithm.
- **CountGraphs**: computes all symmetries (automorphisms) within a set of graphs.
- **GameOfLife**: Conway's Game of Life simulator
- **Kclustering**: a k-means clustering algorithm, where each stage is spawned as a server.
- **Mandelbrot**: a Mandelbrot set generator.
- **Nucleic**: Pseudoknot [75] benchmark applied on multiple inputs.
- **Raytrace**: a ray-tracing algorithm to render a scene.

The benchmark characteristics is given in Figure 3.2. The numbers were obtained using local collector (LC) with programs running on 48 cores, and the average of the results is reported. The benchmarks were designed such that the input size and the number of threads are tunable. Out of the total bytes allocated during the program execution, on average 5.4% is allocated in the shared heap. Thus, most of the objects allocated are collected locally, without the need for stalling all of the mutators. The allocation rate on the SCC is typically much lower than comparable general purpose commercial offerings. On the SCC, not only is the processor slow (533MHz) but also the serial memory bandwidth for our experimental setup is only around 70 MB/s.

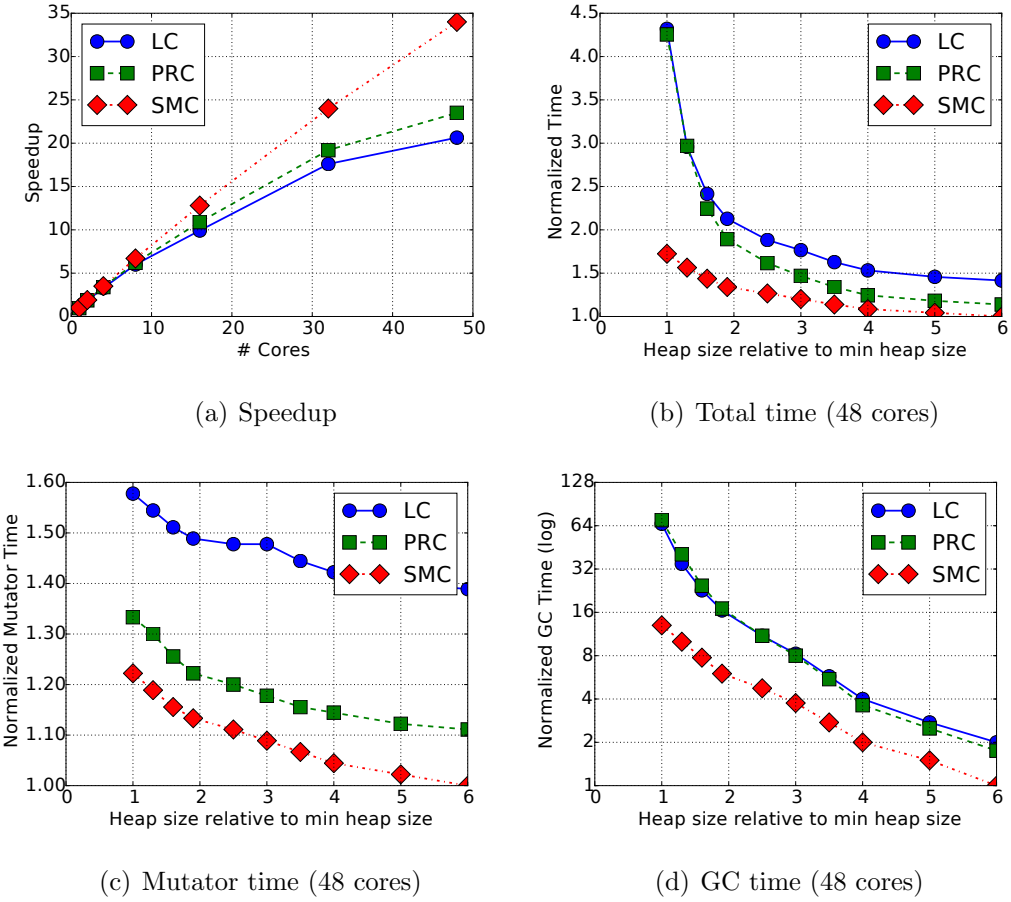
Table 3.2.
Benchmark characteristics. %Sh represents the average fraction of bytes allocated in the shared heap.

Benchmark	**Allocation Rate (MB/s)**	**Allocation**		**# Threads**
		Total (GB)	**% Sh**	
AllPairs	53 ±2.3	16 ±0.23	11 ±0.09	512
BarnesHut	70 ±2.3	20 ±0.25	2 ±0.02	1024
CountGraphs	144 ±3.8	24 ±0.32	1 ±0.01	256
GameOfLife	127 ±5.0	21 ±0.47	13 ±0.17	1024
KClustering	108 ±2.9	32 ±0.31	3 ±0.05	1024
Mandelbrot	43 ±1.7	2 ±0.02	8 ±0.03	512
Nucleic	87 ±3.4	14 ±0.17	1 ±0.00	384
Raytrace	54 ±2.6	12 ±0.14	4 ±0.03	256

3.5.1 Performance

Figure 3.16 presents the speedup results and illustrates space-time trade-offs critical for any garbage collector evaluation. Among the three variants, SMC performs the best (Figure 3.16(a)) due to the fact that most of the accesses under SMC is cached, unlike LC and PRC. We also see that the performance of LC and PRC start to flatten out due to the contention on the uncached shared memory as we increase the number of cores. Thus, with increasing number of cores, the uncached shared memory becomes the bottleneck.

As we decrease the overall heap size, we see that the programs take longer to run, due to the more frequent GCs (Figure 3.16(b)). The reduction in heap size, by definition, does not adversely affect the mutator time when compared with the GC time. At 3× the minimum heap size under which the programs would run, PRC is 17% faster than LC, and SMC is 18% faster than PRC. Overall, SMC is 32% faster than LC.



(a) Speedup

(b) Total time (48 cores)

(c) Mutator time (48 cores)

(d) GC time (48 cores)

Figure 3.16. Performance comparison of local collector with read barriers (LC), procrastinating collector without read barriers (PRC), and collector utilizing software-managed cache coherence (SMC) : Geometric mean for 8 benchmarks.

The mutator time (Figure 3.16(c)) of LC is consistently higher than PRC due to the elimination of read barrier overheads under PRC. Although SMC does have read barrier overheads, caching much of the shared memory accesses keeps the mutator time low. We instrumented our read and write barriers to classify the memory accesses. On average, across all of the benchmarks, 89% of the read or write requests were to the local heap, which is private and is cached both in L1 and L2. This is common to all three versions of the local collector. Thus, SMC derives mutator gains by caching much of the 11% of the GC requests.

Out of the shared heap memory requests, on average, 93% of all requests were to the cached shared heap. However, it should be noted that cached shared heap data bypass L2, and are only cached in the comparatively smaller L1 cache. Hence, the benefit of caching shared heap data, as far as the mutator is concerned, may not be dramatic if the cached shared heap reads are far and few between. In any case, with SMC, less than 1% of mutator accesses were to the uncached memory. Thus, SMC is able to potentially cache more than 99% of memory accesses.

There is very little difference between the GC times (Figure 3.16(d)) between LC and PRC. This is because both the variants are similar in terms of the actual GC work. However, SMC's GC time tends to be lower since part of the expensive shared heap collection itself is cached. Thus, software-managed cache coherence not only benefits the mutator but also the garbage collector.

3.5.2 Evaluating Procrastinating Collector

In this section, we will focus on the procrastinating collector (PRC) design, and analyze the impact of different optimizations.

Impact of Cleanliness

Cleanliness information allows the runtime system to avoid preempting threads on a write barrier when the source of an globalizing write is clean. In order to study the impact of cleanliness, we removed the reference counting code and cleanliness check from the write barrier; thus, every globalizing write results in a thread preemption and stall. The results presented here were taken with programs running on 48-cores.

Table 3.3 shows the number of preemptions on write barrier for different configurations. PRC represents the variant with all of the features enabled; PRC MU- shows a cleanliness optimization that does not take an object's mutability into consideration in determining cleanliness (using only recorded reference counts instead),

Table 3.3.
Average number of preemptions on write barrier.

Benchmark	**PRC**	**PRC MU-**	**PRC CL-**
AllPairs	604 ±42	616 ±43	28573 ±1429
BarnesHut	8376 ±503	82284 ±3291	23504887 ±1175244
CountGraphs	45 ±2	64 ±2	17061 ±1194
GameOfLife	11973 ±359	238462 ±16692	1936250 ±58088
KClustering	7227 ±217	15394 ±616	8107173 ±405359
Mandelbrot	44 ±2	84 ±5	5863 ±235
Nucleic	58 ±3	104594 ±4184	209840 ±14689
Raytrace	881 ±35	973 ±39	13464 ±404

and PRC CL- represents preemptions incurred when the collector does not use any cleanliness information at all. Without cleanliness, on average, the programs perform substantially more preemptions when encountering a write barrier.

Recall that if all of the threads belonging to a core get preempted on a write barrier, a local major GC is *forced*, which lifts all of the sources of globalizing writes, fixes the references to forwarding pointers and unblocks the stalled threads. Hence, an increase in the number of preemptions leads to an increase in the number of local collections.

Table 3.4 shows the percentage of local major GCs that were forced compared to the total number of local major GCs. PRC CL- shows the percentage of forced GCs if cleanliness information is not used. On average, 49% of local major collection performed is due to forced GCs if cleanliness information is not used, whereas it is less than 1% otherwise. On benchmarks like `BarnesHut`, `GameOfLife` and `Mandelbrot`, where all of the threads tend to operate on a shared global data structure, there are a large number of globalizing writes. On such benchmarks almost all local GCs are forced in the absence of cleanliness. This adversely affects the running time of programs.

Table 3.4.
Average percentage of forced GCs out of the total number of local major GCs.

Benchmark	**PRC**	**PRC MU-**	**PRC CL-**
AllPairs	0.08 ±0	0.08 ±0	38.55 ±2.31
BarnesHut	0.17 ±0.01	19.2 ±0.96	100 ±3
CountGraphs	0 ±0	0.03 ±0	0.18 ±0.01
GameOfLife	3.54 ±0.21	9.47 ±0.47	99.75 ±4.99
KClustering	0 ±0	0.02 ±0	21.64 ±1.08
Mandelbrot	1.43 ±0.1	2.86 ±0.11	86.22 ±6.04
Nucleic	0 ±0	9.37 ±0.28	19.3 ±0.58
Raytrace	1.72 ±0.1	1.72 ±0.07	24.86 ±0.99

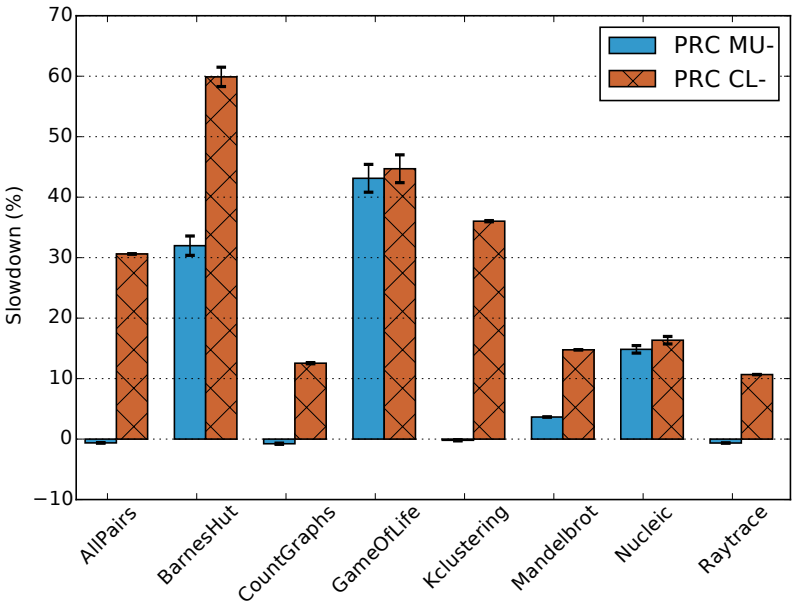


Figure 3.17. Impact of utilizing object mutability information and cleanliness analysis on the performance of PRC.

Figure 3.17 shows the running time of programs without using cleanliness. On average, programs tend to run 28.2% slower if cleanliness information is ignored. The results show that cleanliness analysis therefore plays a significant role in the PRC design.

Impact of Immutability

If the source of an globalizing write is immutable, we can make a copy of the object in the shared heap and assign a reference to the new shared heap object to the target. Hence, we can ignore the reference count of such objects. Not all languages may have the ability to distinguish between mutable and immutable objects in the compiler or in the runtime system. Hence, we study the impact of our local collector design with mutability information in mind. To do this, we ignore the test for mutability in the cleanliness check (Figure 3.6) and modify the object lifting code in Figure 3.9 to treat all objects as mutable.

PRC MU- in Table 3.3 and Table 3.4 show the number of write barrier preemptions and the percentage of forced GCs, respectively, if all objects were treated as mutable. For some programs such as `AllPairs`, `CountGraphs`, or `Kclustering`, object mutability does not play a significant factor. For benchmarks where it does, distinguishing between mutable and immutable objects helps avoid inducing preemptions on a write barrier since a copy of the immutable object can be created in the shared heap without the need to repair existing references to the local heap copy.

Figure 3.17 shows the performance impact of taking object mutability into account. While ignoring object mutability information, `BarnesHut`, `GameOfLife` and `Nucleic` are slower due to the increased number of forced GCs. Interestingly, `AllPairs`, `CountGraphs`, `Kclustering` and `Raytrace` are marginally faster if the mutability information is ignored. This is due to not having to manipulate the `imSet` (Line 15 in Figure 3.9), and walking immutable objects after the objects are lifted (Line 22 in

Table 3.5.
Impact of heap session: % LM clean represents the fraction of instances when a clean object closure has at least one object with `LOCAL_MANY` references.

Benchmark	**% LM Clean**	**Avg. Session Size (Bytes)**
AllPairs	5.35 ±0.37	2966 ±119
Barneshut	13.27 ±0.53	1596 ±96
Countgraphs	8.86 ±0.53	3648 ±73
GameOfLife	23.9 ±1.2	1384 ±55
Kclustering	18.13 ±0.54	2248 ±135
Mandelbrot	4.64 ±0.09	8549 ±598
Nucleic	13.3 ±0.27	1226 ±37
Raytrace	8.28 ±0.41	1112 ±22

Figure 3.9). On average, we see a 11.4% performance loss if mutability information is not utilized for cleanliness.

Impact of Heap Session

In order to assess the effectiveness of using heap sessions, we measured the percentage of instances where the source of an globalizing write is clean with at least one of the objects in the closure has a `LOCAL_MANY` reference. During such instances, we walk the current heap session to fix any references to forwarded objects. Without using heap sessions, we would have preempted the thread in the write barrier, reducing available concurrency. The results are presented in Table 3.5.

The first column shows the percentage of instances when an object closure is clean and has at least one object with `LOCAL_MANY` references. On average, we see that 12% of clean closures have at least one object with `LOCAL_MANY` references. We also measured the average size of heap sessions when the session is traced as a part of

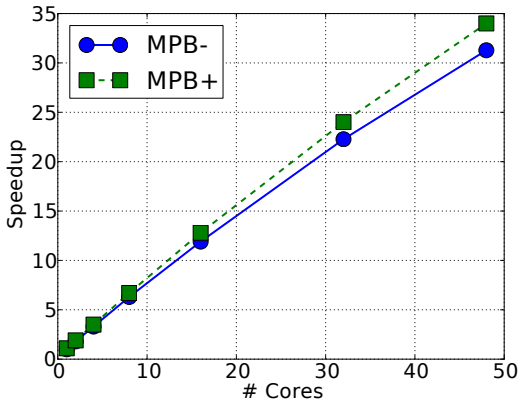


Figure 3.18. Performance comparison of first-class channel communication over the MPB `(MBP+)` vs solely over the shared memory `(MPB-)`: Geometric mean over 8 benchmarks.

lifting an object closure to the shared heap (Line 25 in Figure 3.9). The average size of a heap session when it is traced is 2859 bytes, which is less than a page size. These results show that utilizing heap sessions significantly contributes to objects being tagged as clean, and heap sessions are small enough to not introduce significant overheads during tracing.

3.5.3 MPB Mapped Channels

In this section, we study the impact of MPB mapped channels. In order to evaluate the benefit of mapping the first-class channel communication over the message passing buffer memory, we implemented a version of our communication library that does not use the message passing buffer memory. Recall that if the channel is located in the shared heap, the message in the local heap, and the message does not have a mutable object in its transitive closure, we perform the transfer over the message passing buffer (Case 4 in Section 3.4.3). Instead, we eagerly globalize the transitive closure of the message and just share the pointer with the receiving thread (similar to Case 3). We call this version MPB-, and the original version MPB+.

Figure 3.18 shows the performance comparison of MPB+ versus MPB-. On 48-cores, MPB+ is only around 9% faster than the MPB- version. We can attribute several reasons for this marginal improvement. First, we observed that, on average, around only 32% of channel communications were taking advantage of the MPB (Case 4) in the case of MPB+. The rest of the channel communications were either local or were using the shared memory to transfer the messages. Moreover, in the case of MPB-, immutable inter-core messages are transferred over the CSH which is cached.

Second, the cost of inter-core interrupts is substantial, as was observed by others [76, 77]. We measured the time it takes between a core issuing an inter-core interrupt to the time it sends or receives the first byte is around 2000 core cycles. Since majority of the immutable messages exchanged between cores are small, the overhead of setting up the message transfer outweighs the benefit of using the MPB. However, utilizing the MPB prevents immutable messages from being globalized, thus reducing the pressure on the shared memory. As a result, the number of expensive shared heap collections are reduced.

3.6 Related Work

Over the years, several local collector designs [48, 66–68] have been proposed for multi-threaded programs. Recently, variations of local collector design have been adopted for multi-threaded, functional language runtimes like GHC [46] and Manticore [47]. Doligez et al. [67] proposed a local collector design for ML with threads where all mutable objects are allocated directly on the shared heap, and immutable objects are allocated in the local heap. Similar to our technique, whenever local objects are shared between cores, a copy of the immutable object is made in the shared heap. Although this design avoids the need for read and write barriers, allocating all mutable objects, irrespective of their sharing characteristics can lead to poor performance due to increased number of shared collections, and memory access overhead due to NUMA effects and uncached shared memory as in the case of SCC. It is for

this reason we do not treat the shared memory as the oldest generation for our local generation collector unlike other designs [46, 67].

Several designs utilize static analysis to determine objects that might potentially escape to other threads [68, 78]. Objects that do not escape are allocated locally, while all others are allocated in the shared heap. The usefulness of such techniques depends greatly on the precision of the analysis, as objects that might potentially be shared are allocated on the shared heap. This is undesirable for architectures like the SCC where shared memory accesses are very expensive compared to local accesses. Compared to these techniques, our design only exports objects that are definitely shared between two or more cores. Our technique is also agnostic to the source language, does not require static analysis, and hence can be implemented as a lightweight runtime technique.

Anderson [48] describes a local collector design (TGC) that triggers a local garbage collection on every globalizing write of a mutable object, while immutable objects, that do not have any pointers, are copied to the shared heap. This scheme is a limited form of our cleanliness analysis. In our system, object cleanliness neither solely relies on mutability information, nor is it restricted to objects without pointer fields. Moreover, TGC does not exploit delaying globalizing writes to avoid local collections. However, the paper proposes several interesting optimizations that are applicable to our system. In order to avoid frequent mutator pauses on globalizing writes, TGC's local collection runs concurrently with the mutator. Though running compaction phase concurrently with the mutator would require read barriers, we can enable concurrent marking to minimize pause times. TGC also proposes watermarking scheme for minimizing stack scanning, which can be utilized in our system to reduce the stack scanning overheads during context switches and globalizing writes of clean objects.

Marlow et al. [46] propose globalizing only part of the transitive closure to the shared heap, with the idea of minimizing the objects that are globalized. The rest of the closure is exported essentially on demand during the next access from another core. This design mandates the need for a read barrier to test whether the object

being accessed resides in the local heap of another core. However, since the target language is Haskell, there is an implicit read barrier on every load, to check whether the thunk has already been evaluated to a value. Since our goal is to eliminate read barriers, we choose to export the transitive closure on an globalizing write.

Software managed cached coherence (SMC) [69] for SCC provides a coherent, shared virtual memory to the programmer. However, the distinction between private and shared memory still exists and it is the responsibility of the programmer to choose data placement. In our system, all data start out as being private, and is only shared with the other cores if necessary. The sharing is performed both through the shared memory as well as over the MPB, based on the nature of message being shared. MESH framework [79] provides a similar mechanism for flexible sharing policies on the SCC as a middle-ware layer.

In the context of mapping first-class channels to MPBs, the work by Prell et al. [80] which presents an implementation of Go's concurrency constructs on the SCC is most similar. However, unlike our channel implementation, channels are implemented directly on the MPB. Since the size of MPB is small, the number of channels that can be concurrently utilized are limited. Moreover, their implementation diverges from Go language specification in that the go-routines running on different cores run under different address spaces. Hence, the result of transferring a mutable object over the channels is undefined. Our channel communication utilizes both shared memory and the MPBs for inter-core messaging. Barrelfish on the SCC [76] uses MPBs to transfer small messages and bulk transfer is achieved through shared memory. However, Barrelfish differs from our system since it follows a shared-nothing policy for inter-core interaction.

3.7 Concluding Remarks

The Intel SCC provides an architecture that combines aspects of distributed systems (no cache coherence) with that of a shared memory machine, with support for

programmable cache coherence and fast inter-core messaging. In order to effectively utilize this architecture, it is desirable to hide the complexity behind the runtime system. To this end, the Aneris programming platform provides a cache coherent shared memory abstraction for the ML programmer. Aneris utilizes the mostly-functional and highly concurrent nature of the programming model to implement a memory management scheme that is optimized for the memory hierarchy found on the SCC. The results and experience building Aneris illustrate that functional programming language technology can mitigate the burden of developing software for highly scalable manycore systems.

4 RX-CML: A PRESCRIPTION FOR SAFELY RELAXING SYNCHRONY

Concurrent ML [81] (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 programming model of CML, however, assumes strong consistency; while the channel itself is first-class and supports many-to-many communication pattern, the communication has *exactly-once* requirement. If a receiver consumes a sent value, then no other sender can consume the same value. Thus, synchronous communication needs coordination between the communicating parties for enforcing the exactly-once requirement. Hence, while first-class channel based synchronous communication provides a good abstraction, its correctness and performance implication in a high latency, weakly consistent setting prevents its utility in a weakly consistent loosely coupled environment.

While asynchronous extensions such as ACML [56] 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. Moreover, ACML also requires the *exactly-once* requirement. Hence, even though ACML solves the problem of synchrony at the cost of increased complexity, it does not solve the problem of coherence.

One way to enhance performance without requiring new additions to the core set of event combinators CML supports, 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

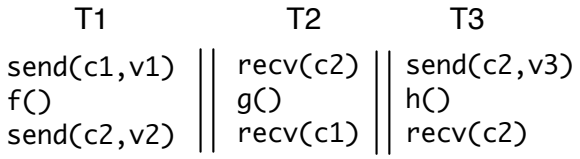
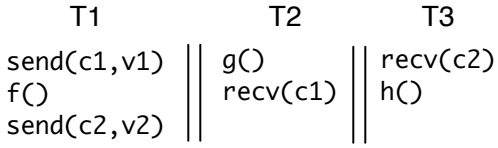


Figure 4.1. Performing the first `send` in T1 asynchronously is not meaning preserving with respect to synchronous evaluation.

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

To illustrate, consider the following simple program:



Thread T1 performs a synchronous send on channel `c1` that is received by thread T2, after it computes `g()`. After the communication is performed, T1 evaluates `f()`, and then sends `v2` on channel `c2`, which is received by thread T3. Upon receipt, T3 evaluates `h()`. Assuming `f`, `g`, and `h` perform no communication action of their own, the synchronous communication on `c1` by T1 could have been safely converted into an asynchronous action in which `v1` is buffered, and read by T2 later upon evaluation of `g()`. The observable behavior of the program in both cases (i.e., treating the initial send synchronously or asynchronously) would be the same.

Unfortunately, naïvely replacing synchronous communication with an asynchronous one is not usually meaning-preserving as the example in Figure 4.1 illustrates. Under a synchronous evaluation protocol, T2 would necessarily communicate first with T3,

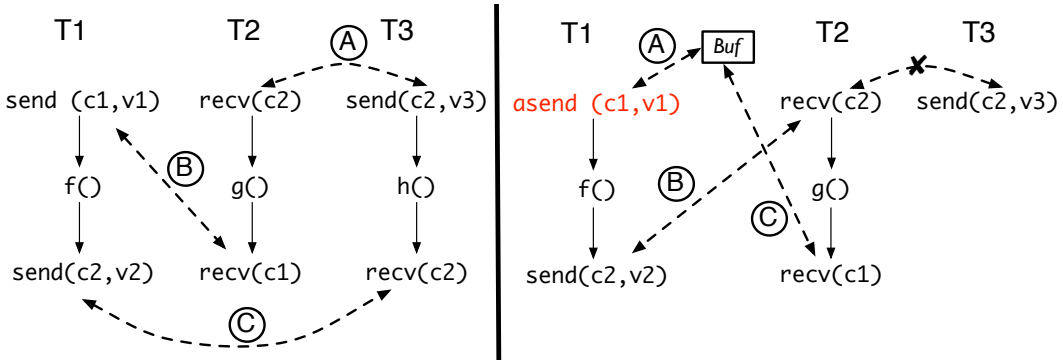


Figure 4.2. Dependence graph induced by the execution of the program presented in Figure 4.1.

receiving `v3` on channel `c2`. It is then able to receive `v1` from T1; finally, T1 can communicate `v2` to T3. If the `send(c1,v1)` operation by T1 were replaced by `asend(c1,v1)`, 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 of `v2` on channel `c2`, it is impossible to satisfy the send on T3. Thus, this asynchronous execution exhibits a new behavior not possible using just synchronous operators.

The distinction between these two executions can be explained in terms of a dependence graph that captures both intra- and inter-thread data- and control-flow. We can depict the executions by explicitly drawing these dependencies as shown in Figure 4.2.

The dashed edges reflect communication and synchronization dependencies among threads, while solid edges capture thread-local control-flow. A bi-directional edge connects a sender with either a receiver, in the case of a synchronous send, or a buffer, in the case where it is asynchronous. In both instances, there is a synchronization dependence between endpoints, and a data dependence from the sender to either the matching receiver or buffer. The left-hand side of the figure shows a possible execution in which all operations are synchronous; the right considers an execution

in which the initial send by T1 is asynchronous. The labels on the edges reflect the order in which communication actions are executed.

The synchronous execution on the left reflects the description given earlier. The asynchronous execution on the right depicts the send on thread T1 buffering its data (Ⓐ), thus allowing the synchronous communication between T1 and T2 (Ⓑ); this action prohibits communication between T2 and T3. T2 subsequently receives `v1` from the buffer associated with channel `c1` (Ⓒ). This behavior could not be realized by any synchronous execution: Ⓑ could never have been performed if the send operation on channel `c1` was not asynchronous.

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, such as the right-hand execution above, can be detected and repaired, form the focus of this chapter. 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. We relate this definition to an operational semantics that specifies relaxed execution behavior for communicating actions, and relate the set of traces admitted by the operational semantics to the safe executions defined by the axiomatic formulation.

- A distributed implementation, $\mathrm{R}^{\mathrm{CML}}$, that treats asynchronous communication as a form of *speculation* is described. A mis-speculation, namely the execution that could not have been realized using only synchronous communication, is

detected using a runtime instantiation of our axiomatic formulation. An uncoordinated, distributed checkpointing mechanism is utilized to rollback and re-execute the offending execution synchronously, which is 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.

4.1 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`.

Building such an application using a centralized server is straightforward, but hinders scalability. In the absence of central mediation, a causal broadcast protocol [82] is required. One possible encoding of causal broadcast using CML primitives is shown in Figure 4.3. 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

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

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

(* Broadcast send operation *)
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

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

Figure 4.3. Synchronous broadcast channel

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 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`. Figure 4.4 shows an execution where this is not the case. In the figure, uninteresting messages have been elided for clarity. 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

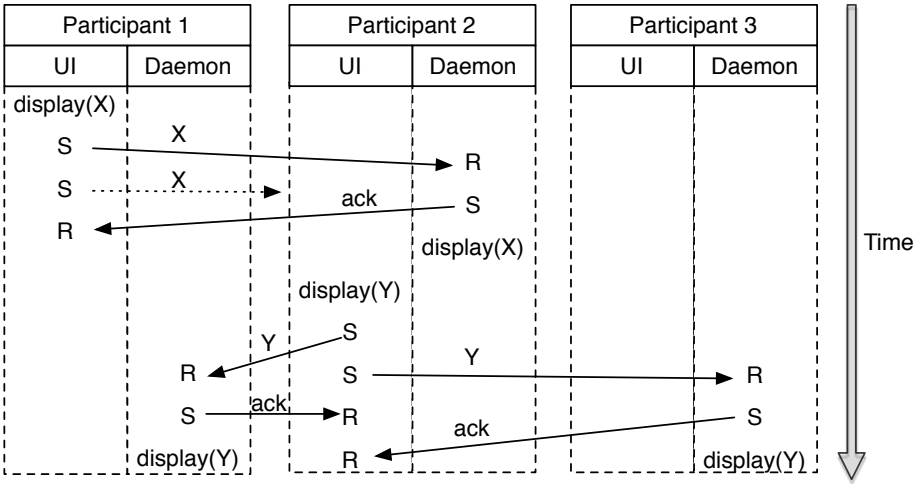


Figure 4.4. Incorrect execution due to unsafe relaxation of sends during broadcast. Dotted arrow represents in-flight message.

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

Table 4.1.
Performance comparison of causal messaging passing

Execution	*Avg.time (ms)*	*Errors*
Sync	1540 ±53	0 ±0
Unsafe Async	520 ±17	7 ±2
Safe Async ($\mathbb{R}^{\text{CML}}$)	533 ±13	0 ±0

consider the largest of the times across the participants to be the running time. The results are presented in Table 4.1.

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, $\mathbb{R}^{\text{CML}}$, 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.

4.2 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 [20–22].

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.

$$
\begin{array}{rcll}
\text{Actions } \mathbb{A} & \coloneqq & b_t & \text{(t starts)} \\
& | & e_t & \text{(t ends)} \\
& | & j_t^m t' & \text{(t detects t' has terminated)} \\
& | & f_t^m t' & \text{(t forks a new t')} \\
& | & s_t^m c, v & \text{(t sends value v on c)} \\
& | & r_t^m c & \text{(t receives a value v on c)} \\
& | & p_t^m v & \text{(t outputs an observable value v)}
\end{array}
$$

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

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:

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

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 4.2.1 (Axiomatic Execution) *An axiomatic execution is defined by the tuple* $\mathsf{E} \coloneqq \langle \mathsf{P}, \mathsf{A}, \rightarrow_{po}, \rightarrow_{co} \rangle$ *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).*
- $\rightarrow_{co} \subseteq (\mathsf{A}_s \times \mathsf{A}_r) \cup (\mathsf{A}_r \times \mathsf{A}_s)$ *is the communication order which is a symmetric relation established between matching communication actions (i.e.,* $\alpha \rightarrow_{co} \beta \implies \beta \rightarrow_{co} \alpha$*). Moreover, a send and its matching receive must operate over the same channel (i.e.,* $s_t^m c, v \rightarrow_{co} r_{t'}^n c' \implies c = c'$*).*

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

Definition 4.2.2 (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.*

```
(* 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)
```

Figure 4.5. A CML Program with potential for mis-speculation.

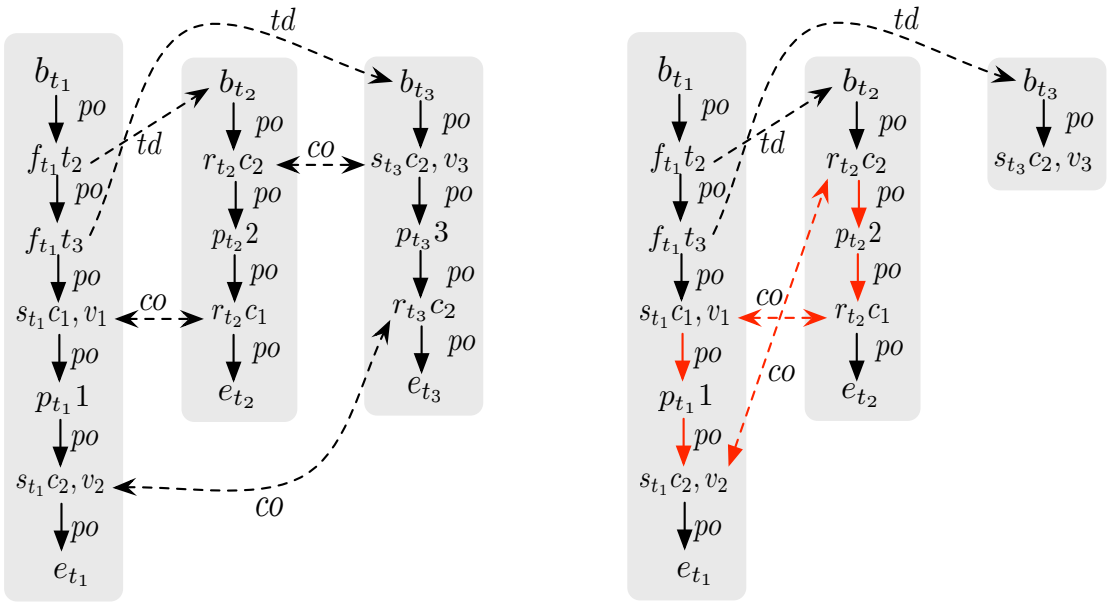
Definition 4.2.3 (Happens-before relation) *The happens-before order of an execution is the transitive closure of the union of program order, thread dependence order, and actions related by communication and program order:*

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

For any two actions $\alpha, \beta \in \mathsf{A}$, if $\alpha \nrightarrow_{hb} \beta$, 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 4.2.4 (Happens-before Cycle) *A cycle exists in a happens-before relation if for any two actions α, β and $\alpha \rightarrow_{hb} \beta \rightarrow_{hb} \alpha$.*

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 simple CML program (Figure 4.5) which shows a simple CML program and two possible executions (Figure 4.6). The execution in Figure 4.6(a) imposes no causal dependence between the observable actions (i.e., print statements) in t_2 or t_3; thus,



(a) Well-formed execution

(b) Ill-formed execution

Figure 4.6. Potential axiomatic executions of the CML program presented in Figure 4.5.

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 4.6(b), 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. A labeled transition between intra-thread states is captured by the relation, $\dot{\twoheadrightarrow} \subseteq \mathsf{State}_{\mathsf{intra}} \times \mathsf{Label}_{\mathsf{intra}} \times \mathsf{State}_{\mathsf{intra}}$, 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:

- $\dot{\twoheadrightarrow}$ 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 4.2.5 (Intra-trace) *Let $tr = \overline{\alpha}$ be a sequence of actions in set A, and $\rightarrow_{co}$ be a communication order 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* $0 \leq i, \delta_i \overset{l_i}{\twoheadrightarrow} \delta_{i+1}$
- *the projection* $\overline{\beta}$ *of* $\bar{l}$ *to non-silent labels is such that* $\beta_i = (\alpha_i, V(\gamma_i))$ *if* $\alpha_i \in \mathsf{A}_r$ *and* $\alpha_i \rightarrow_{co} \gamma_i$, *or* $\beta_i = \alpha_i$ *otherwise.*

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

Definition 4.2.6 (Well-formed Execution) *An execution* $E \coloneqq \langle \mathsf{P}, \mathsf{A}, \rightarrow_{po}, \rightarrow_{co} \rangle$ *is well-formed if the following conditions hold:*

1. *Intra-thread consistency: for all threads* $t \in \mathbb{T}$, $([\rightarrow_{po}]_t, \rightarrow_{co}) \in \mathsf{InTr}^{\mathsf{P}}[\mathsf{t}]$
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.* $\beta \rightarrow_{co} \beta'$.

For an axiomatic execution $\mathsf{E} \coloneqq \langle \mathsf{P}, \mathsf{A}, \rightarrow_{po}, \rightarrow_{co} \rangle$ to be well-formed, the actions, program order and communication order relations 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 4.6, 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 visible 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[1]. 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 4.2.7 (Observable dependencies) *In a well-formed axiomatic execution* $\mathsf{E} \coloneqq \langle \mathsf{P}, \mathsf{A}, \rightarrow_{po}, \rightarrow_{co} \rangle$*, 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 4.2.8 (CML Execution) *Given a well-formed axiomatic execution* $\mathsf{E} \coloneqq \langle \mathsf{P}, \mathsf{A}, \rightarrow_{po}, \rightarrow_{co} \rangle$*, the pair* $(\mathsf{E}, \rightarrow_{to})$ *is said to be in* $\textsc{Cml}(\mathsf{P})$ *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.

Lemma 4.2.1 *If a total order* $\rightarrow_{to}$ *is consistent with* $\rightarrow_{hb}$*, then* $\rightarrow_{hb}$ *does not contain a cycle involving actions in* A_{od}.

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

Next, we show that a well-formed axiomatic execution respects the safety property of a CML program executed non-speculatively. When a CML program evaluates non-speculatively, a thread performing a communication action is blocked until a matching communication action is available. Hence, if $(\langle \mathsf{P}, \mathsf{A}, \rightarrow_{po}, \rightarrow_{co} \rangle, \rightarrow_{to}) \in \textsc{Cml}(\mathsf{P})$, 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 $\alpha \rightarrow_{co} \alpha'$ that happened before β in $\rightarrow_{to}$. This is captured in the following theorem.

Theorem 4.2.1 *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* $\alpha \rightarrow_{co} \alpha'$ *such that* $\alpha' \rightarrow_{to} \beta$.

Proof Let $\mathsf{E} \coloneqq \langle \mathsf{P}, \mathsf{A}, \rightarrow_{po}, \rightarrow_{co} \rangle$. First, we show that $\alpha' \in \mathsf{A}$. Since $\alpha \rightarrow_{to} \beta$, $\alpha \in \mathsf{A}_{od}$, by Definition 4.2.8. By Definition 4.2.7, 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 4.2.6, there exists an $\alpha' \in \mathsf{A}$ such that $\alpha \rightarrow_{co} \alpha'$.

Next, we show that $\alpha' \in \mathsf{A}_{od}$. By Definition 4.2.3, $\alpha' \rightarrow_{co} \alpha \rightarrow_{hb} \gamma$ 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 4.2.1 and Definition 4.2.8, $\alpha' \rightarrow_{to} \beta$. ■

4.3 Operational Semantics

The axiomatic semantics provides a declarative way of reasoning about executions. However, it is unclear how to use this semantics to *execute* a CML program that performs the sends asynchronously while ensuring that the observable behaviors correspond to a synchronous execution of the program; that is, how we can we ensure that implementations produce relaxed executions that always conform to a CML execution in the sense of Definition 4.2.8? In this section, we present an operational definition of a relaxed execution as a labeled transition system that allows us to express the constraints necessary to prevent non-CML observable behaviors.

$$
\begin{array}{rcll}
e & \in & \texttt{Exp} & := \quad v \mid x \mid e\ e \mid \mathsf{ch}() \mid \mathsf{print}(e) \mid \mathsf{spawn}(e) \\
& & & \quad \mid \quad \mathsf{send}(e, e) \mid \mathsf{recv}(e) \mid \mathsf{join}(e) \\
v & \in & \texttt{Val} & := \quad \mathsf{unit} \mid c \mid \lambda x.e \mid t
\end{array}
$$

$$
\begin{array}{rcl}
E & := & \bullet \mid E\ e \mid v\ E \mid \mathsf{print}(E) \\
& \mid & \mathsf{spawn}(E) \mid \mathsf{send}(E, e) \mid \mathsf{send}(c, E) \\
& \mid & \mathsf{recv}(E) \mid \mathsf{join}(E)
\end{array}
$$

$$
\begin{array}{rcll}
c & \in & \texttt{ChannelId} & \\
\alpha, \beta & \in & \texttt{Action} & := \quad \mathbb{A} \mid (\alpha, \beta) \mid \tau_t \mid \epsilon_{(\mathsf{P}, tr)} \\
\Delta & \in & \texttt{SendSoup} & := \quad \mathbb{A}_s \\
L & \in & \texttt{LocalState} & := \quad e \mid \textsc{Ready}_{op}(e) \mid \textsc{Done}_{op} \\
t & \in & \texttt{ThreadId} & \\
T & \in & \texttt{Thread} & := \quad (t, L) \\
\overline{T} & \in & \texttt{ThreadSoup} & := \quad \emptyset \mid T \parallel \overline{T} \\
\langle \overline{T}, \Delta \rangle & \in & \textsc{ProgState} &
\end{array}
$$

Figure 4.7. Syntax and states for the relaxed execution semantics of a subset of CML.

The operational machine, Rel, takes as its input an operational execution, which is composed of a program, and a trace *tr* of *operational actions*. It evaluates the program according to the trace, by manipulating a *send soup*, an unordered set into which pending sends are added asynchronously. Informally, we can think of the trace as a history or log of actions we wish to perform; the machine either accepts the trace, if executing the actions found in the trace using the reduction rules leads to a CML execution (as defined by Definition 4.2.8), or gets stuck otherwise.

Definition 4.3.1 (Operational Action) *An operational action $\alpha \in \mathbb{A}_{op}$ is either:*

- *an action in $\mathbb{A} \setminus \mathbb{A}_r$.*

$$\langle (t, E[(\lambda\, x.e)v]) \parallel \overline{T}, \Delta\rangle \xrightarrow{\tau_t} \langle (t, E[e[v/x]]) \parallel \overline{T}, \Delta\rangle \quad [\textsc{App}]$$

$$\frac{c \;\; \mathsf{fresh}}{\langle (t, E[\mathsf{ch}()]) \parallel \overline{T}, \Delta\rangle \xrightarrow{\tau_t} \langle (t, E[c]) \parallel \overline{T}, \Delta\rangle} \quad [\textsc{Chan}]$$

$$\langle (t, \textsc{Ready}_{op}(e)) \parallel \overline{T}, \Delta\rangle \xrightarrow{b_t} \langle (t, e) \parallel \overline{T}, \Delta\rangle \quad [\textsc{Begin}]$$

$$\langle (t, v) \parallel \overline{T}, \Delta\rangle \xrightarrow{e_t} \langle (t, \textsc{Done}_{op}) \parallel \overline{T}, \Delta\rangle \quad [\textsc{End}]$$

$$\langle (t, E[\mathsf{spawn}\, e]) \parallel \overline{T}, \Delta\rangle \xrightarrow{f_t^i t'} \langle (t, E[t']) \parallel (t', \textsc{Ready}_{op}(e)) \parallel \overline{T}, \Delta\rangle \quad [\textsc{Spawn}]$$

$$\langle (t, E[\mathsf{join}\, t']) \parallel (t', \textsc{Done}_{op}) \parallel \overline{T}, \Delta\rangle \xrightarrow{j_t^i t'} \langle (t, E[\mathsf{unit}]) \parallel (t', \textsc{Done}_{op}) \parallel \overline{T}, \Delta\rangle \quad [\textsc{Join}]$$

$$\langle (t, E[\mathsf{print}\, v]) \parallel \overline{T}, \Delta\rangle \xrightarrow{p_t^i v} \langle (t, E[\mathsf{unit}]) \parallel \overline{T}, \Delta\rangle \quad [\textsc{Print}]$$

$$\frac{\alpha = s_t^i c, v}{\langle (t, E[\mathsf{send}(c, v)]) \parallel \overline{T}, \Delta\rangle \xrightarrow{\alpha} \langle (t, E[\mathsf{unit}]) \parallel \overline{T}, \Delta \cup \{\alpha\}\rangle} \quad [\textsc{Send}]$$

$$\frac{\alpha = r_t^i c \quad \beta = (s_{t'}^j c, v) \in \Delta}{\langle (t, E[\mathsf{recv}\, c]) \parallel \overline{T}, \Delta\rangle \xrightarrow{(\alpha,\beta)} \langle (t, E[v]) \parallel \overline{T}, \Delta \setminus \beta\rangle} \quad [\textsc{Recv}]$$

$$\frac{\mathsf{WF}((\mathsf{P}, tr))}{\langle \overline{T}, \Delta\rangle \xrightarrow{\epsilon_{(\mathsf{P},tr)}} \langle \overline{T}, \Delta\rangle} \quad [\textsc{Commit}]$$

Figure 4.8. A relaxed execution operational semantics for a subset of CML.

- *a pair in $\mathbb{A}_r \times \mathbb{A}_s$ where each receive action is paired up with the matching send action on the same channel,*

- *a* silent *action* τ_t *indexed by a thread identifier that indicates the evaluation of a computation step (e.g., a function call).*

- *a* commit *action* $\epsilon_{(\mathsf{P},tr)}$ *that checks whether the machine state is well-formed in the sense of Definition 4.2.6. Intuitively, a well-formed machine state is one that could have been reached if all* `send` *operations were executed synchronously.*

Definition 4.3.2 (Operational Execution) *An operational execution is a pair* (P, tr)*, where* P *is a program and* $tr \in \overline{\mathbb{A}_{op}}$*, and there are no duplicates in* tr*.*

The operational semantics of the machine used to interpret the trace is given in Figure 4.7 and Figure 4.8. The program state is composed of a tuple with $\overline{T}$ representing the pool of concurrently executing threads and a collection of unmatched send actions Δ. Each thread in $\overline{T}$ is a tuple with a thread id and a local state. This state is either an initial state $\textsc{Ready}_{op}(e)$ containing the expression that will be evaluated by this thread, or a completed state $\textsc{Done}_{op}$, or the expression being evaluated by this thread. Each state transition is affixed with an action drawn from the trace that is used by the machine to determine which rule to apply, as we describe below.

The source language contains function abstraction, application, first-class channels, a thread creation operation (spawn), message-passing primitives on these channels, and a print statement to capture an observable action. Reductions are labeled with operational actions. Rules App and Chan represent silent (thread-local) actions. The Spawn rule creates a new thread with the initial local state $\textsc{Ready}_{op}(e)$. Rule Begin initiates execution of the thread from its initial state. A thread moves to the $\textsc{Done}_{op}$ local state once the expression being evaluated is reduced to a value (Rule End). The completed thread remains in the thread pool so that other threads may also join on it. A thread can wait on another thread's completion by joining on its thread identifier. The calling thread is paused until the joined thread moves to the final local state $\textsc{Done}_{op}$ (Rule Join).

The Send rule adds the associated send action α into Δ, the collection of pending send actions, and allows execution to proceed. Thus, this rule captures asynchronous

behavior. The transition defined by the RECV has a label defined as a pair, consisting of the receive action α and an unmatched send action β, which belongs to the send pool. The receiver thread consumes the value contained in the send action, and removes the action from Δ. An observable action (rule PRINT) is evaluated only if the trace emits a print action; this rule exists primarily to allow us to reason about equivalence of observable behaviors as we describe below. Finally, we define a commit rule (rule COMMIT) that checks whether the interleaving generated thus far corresponds to a sensible CML execution, i.e., whether the interleaving exhibits a behavior that could have been observed if all `send` operations executed thus far were evaluated synchronously. It uses an operator WF defined in definition 4.3.6.

Definition 4.3.3 (REL execution) *Given a trace* $tr = \overline{\beta}.\epsilon_{(\mathsf{P},\overline{\beta})}$ *terminating in a commit action, an operational execution* (P, tr) *is a* relaxed *execution (*$\textsc{Rel}(\mathsf{P}, tr)$*) if there exists an ordered sequence* $\overline{\delta} \in \textsc{ProgState}$ *satisfying the following:*

- $\delta_0 = \langle (t_0, \textsc{Ready}_{op}(\mathsf{P})), \emptyset \rangle$ *for some unique thread identifier* t_0.
- *for all* $\alpha_i \in tr$*, there exists* δ_i, $\delta_{i+1} \in \overline{\delta}$ *such that* $\delta_i \xrightarrow{\alpha_i} \delta_{i+1}$.

Clearly, not every operational execution (P, tr) is a relaxed execution. Consider the following program P:

```
1 fun main () =
2 let val c = channel()
3 in print (send(c,v); recv c)
4 end
```

with trace

$$tr = b_t \;.\; s_t^i c, v \;.\; (r_t^j c;\; s_t^i c, v) \;.\; p_t^k v \;.\; e_t$$

where silent transitions have been elided. When suffixed with a commit action $\epsilon_{(\mathsf{P},tr)}$, the resulting behavior would not be accepted by the semantics, since no synchronous

execution of the program would result in a match of the `send` operation with a `recv` action on the same thread. The prohibition of such behavior is encapsulated within the definition of well-formedness found in the antecedent of the COMMIT rule that we formalize below.

We reason about sensible (well-formed) relaxed executions by transforming them into an equivalent axiomatic one. A well-formed REL execution is one whose corresponding axiomatic execution is well-formed. Since an axiomatic semantics is parameterized by an intra-thread semantics, we first provide a translation that produces this semantics given the transitions defined by the operational semantics.

Definition 4.3.4 (Intra-thread semantics) *Given a program* P*, let*

$$\mathsf{State}_{\mathsf{intra}} \coloneqq \textsc{Ready} \mid \textsc{Done} \mid e$$

where $e \in \mathsf{P}$*. For each labeled transition of the form* $\langle (t,s) \| \overline{T}, \Delta \rangle \xrightarrow{\alpha} \langle (t,s') \| \overline{T}', \Delta' \rangle$*, where* $\alpha \neq \epsilon_{(\mathsf{P},tr)}$*, we define* $\delta \overset{\beta}{\twoheadrightarrow} \delta'$*, where* $\delta, \delta' \in \mathsf{State}_{\mathsf{intra}}$ *such that:*

- *if* $s = \textsc{Ready}_{op}(e)$*, then* $\delta = \textsc{Ready}$*, otherwise,* $\delta = s$*.*
- *if* $s' = \textsc{Done}_{op}$*, then* $\delta' = \textsc{Done}$*, otherwise* $\delta' = s'$
- *if* $\alpha = ((r_t^i c), (s_{t'}^j c, v))$ *then* $\beta = (r_t^i c;\, v)$
- *if* $\alpha = \tau_t$ *then* $\beta = \tau$
- *otherwise,* $\alpha = \beta$

Definition 4.3.5 ($\mathcal{T}_{\text{AX}}^{\text{OP}}$ operator) *Let* $\mathsf{E}_o = (\mathsf{P}, tr)$ *be an operational execution.* $\mathcal{T}_{\text{AX}}^{\text{OP}}$ *is defined as* $\mathcal{T}_{\text{AX}}^{\text{OP}}(\mathsf{E}_o) = \langle \mathsf{P}, \mathsf{A}, \rightarrow_{po}, \rightarrow_{co} \rangle$ *parameterized with the intra-thread semantics* $\twoheadrightarrow$ *(Definition 4.3.4), where*

- A *is a set of non-silent, non-commit actions in* tr*.*
- *for all* $\alpha, \beta \in \mathsf{A}$*,* $\alpha \rightarrow_{po} \beta$ *iff* $T(\alpha) = T(\beta)$ *and* α *precedes* β *in* tr*.*

- *for all pairs* (a, b) *such that* $a = r_t^i c$ *and* $b = s_{t'}^j c, v$, $a \rightarrow_{co} b \wedge b \rightarrow_{co} a$

For perspicuity, in the definition of A and $\rightarrow_{po}$ above, a receive operational action $(a, b) \in \mathbb{A}_{op}$ is simply treated as $a \in \mathbb{A}_r$.

Definition 4.3.6 (Well-formed REL execution) *Let* $\mathsf{E}_o = (\mathsf{P}, tr) \in \textsc{Rel}(\mathsf{P})$ *be an operational execution. If the axiomatic execution* $\mathsf{E}_{ax} = \mathcal{T}_{\text{AX}}^{\text{OP}}(\mathsf{E}_o)$ *is well-formed, then* E_o *is well-formed. This is written as* $\mathsf{WF}(\mathsf{E}_o)$.

Theorem 4.3.1 *If* $E_o = (\mathsf{P}, tr)$ *and* $\mathsf{WF}(E_o)$, *then* $(\mathcal{T}_{\text{AX}}^{\text{OP}}(\mathsf{E}_o), \rightarrow_{to}) \in \textsc{Cml}(\mathsf{P})$.

Proof We provide a witness for $\mathcal{T}_{\text{AX}}^{\text{OP}}(\mathsf{E}_o)$ via a generative operator $\mathcal{G}_{\text{AX}}^{\text{OP}}$ (Definition 4.3.7) such that $\mathcal{G}_{\text{AX}}^{\text{OP}}(\mathsf{E}_o)$ is an axiomatic execution (Theorem 4.3.2) and $\mathcal{G}_{\text{AX}}^{\text{OP}}(\mathsf{E}_o) = \mathcal{T}_{\text{AX}}^{\text{OP}}(\mathsf{E}_o)$ (Theorem 4.3.3). We then prove that if E_o is well-formed, then $\mathcal{G}_{\text{AX}}^{\text{OP}}(\mathsf{E}_o)$ is well-formed (Lemma 4.3.1). Since $\mathcal{G}_{\text{AX}}^{\text{OP}}(\mathsf{E}_o)$ is well-formed, by definition 4.2.8 and theorem 4.3.3, $(\mathcal{T}_{\text{AX}}^{\text{OP}}(\mathsf{E}_o), \rightarrow_{to}) \in \textsc{Cml}(\mathsf{P})$. ■

Definition 4.3.7 ($\mathcal{G}_{\text{AX}}^{\text{OP}}$ operator) *Let* $\mathsf{E}_{op} = (\mathsf{P}, tr) \in \textsc{Rel}(\mathsf{P})$ *be a well-formed operational execution.* $\mathcal{G}_{\text{AX}}^{\text{OP}}(\mathsf{E}_{op})$ *is an axiomatic execution parameterized with the intra-thread semantics* $\twoheadrightarrow$ *(definition 4.3.4) defined as follows:*

- $\mathcal{G}_{\text{AX}}^{\text{OP}}(\mathsf{P}, \emptyset) = \langle \mathsf{P}, \emptyset, \emptyset, \emptyset \rangle$
- *If* $\mathcal{G}_{\text{AX}}^{\text{OP}}(\mathsf{P}, \overline{\alpha}) = \langle \mathsf{P}, \mathsf{A}, \rightarrow_{po}, \rightarrow_{co} \rangle$ *, then* $\mathsf{E}_{ax} = \mathcal{G}_{\text{AX}}^{\text{OP}}(\mathsf{P}, \overline{\alpha}.\beta)$ *defined as follows:*
  1. *if* $\beta = \tau_t$ *or* $\beta = \epsilon_{(\mathsf{P}, \overline{\alpha})}$, *then* $\mathsf{E}_{ax} = \langle \mathsf{P}, \mathsf{A}, \rightarrow_{po}, \rightarrow_{co} \rangle$
  2. *if* $\beta \notin (\mathbb{A}_r \times \mathbb{A}_s)$, *then* $\mathsf{E}_{ax} = \langle \mathsf{P}, \mathsf{A} \cup \{\beta\}, \rightarrow_{po} \cup \{\gamma \rightarrow_{po} \beta \mid \gamma \in \mathsf{A}, T(\gamma) = T(\beta)\}, \rightarrow_{co} \rangle$
  3. *if* $\beta = (\gamma, \zeta)$, *then* $\mathsf{E}_{ax} = \langle \mathsf{P}, \mathsf{A} \cup \{\gamma\}, \rightarrow_{po} \cup \{\eta \rightarrow_{po} \gamma \mid \eta \in \mathsf{A}, T(\eta) = T(\gamma)\}, \rightarrow_{co} \cup \{\gamma \rightarrow_{co} \zeta, \zeta \rightarrow_{co} \gamma\} \rangle$

Theorem 4.3.2 *Let* $\mathsf{E}_{op} = (\mathsf{P}, tr) \in \textsc{Rel}(\mathsf{P})$ *be an operational execution. Then,* $\mathcal{G}_{\text{AX}}^{\text{OP}}(\mathsf{E}_{op})$ *is an axiomatic execution.*

Proof We show that the quadruple generated by the $\mathcal{G}^{\text{OP}}_{\text{AX}}$ operator satisfies the conditions necessary for an axiomatic execution (Definition 4.2.1). In particular, (1) A is composed of actions from $\mathbb{A}$, (2) $\rightarrow_{po}$ is a disjoint total order on actions belonging to each thread. (3) $\rightarrow_{co}$ is a symmetric relation and relates actions belonging to the same channel. We show these conditions hold by induction on $|tr|$.

The base case is if $\mathsf{E}_{op} = (\mathsf{P}, \emptyset) \in \textsc{Rel}(\mathsf{P})$, then $\mathcal{G}^{\text{OP}}_{\text{AX}}(\mathsf{E}_{op})$ is an axiomatic execution. If the trace is empty, then $\mathcal{G}^{\text{OP}}_{\text{AX}}(\mathsf{E}_{op}) = \langle \mathsf{P}, \mathsf{A}, \rightarrow_{po}, \rightarrow_{co} \rangle$. Here, P is a valid program. A, $\rightarrow_{po}$, and $\rightarrow_{co}$ are empty. Hence, the conditions defined under Definition 4.2.1 trivially hold.

Assume $\mathcal{G}^{\text{OP}}_{\text{AX}}(\mathsf{P}, \overline{a})$ is an axiomatic execution. We show that $\mathcal{G}^{\text{OP}}_{\text{AX}}(\mathsf{P}, \overline{a}.b)$ is an axiomatic execution, where $b \in \mathbb{A}_{op}$. If b is a silent action or a commit action (Definition 4.3.7.1), then $\mathcal{G}^{\text{OP}}_{\text{AX}}(\mathsf{P}, \overline{a}.b) = \mathcal{G}^{\text{OP}}_{\text{AX}}(\mathsf{P}, \overline{a})$.

Let $b \notin (\mathbb{A}_r \times \mathbb{A}_s)$. Then $b \in \mathbb{A}$ (Definition 4.3.1). In this case (Definition 4.3.7.2), $\mathcal{G}^{\text{OP}}_{\text{AX}}$ operator adds b to A and adds $\rightarrow_{po}$ edges from all actions $\alpha \in \mathsf{A}$ such that $T(\alpha) = T(b)$ to b (preserving the total order). $\rightarrow_{co}$ is not modified. By the induction hypothesis, $\mathcal{G}^{\text{OP}}_{\text{AX}}(\mathsf{P}, \overline{a}.b)$ is an axiomatic execution.

Let $b = (\alpha, \beta)$. $\alpha \in \mathbb{A}_r, \beta \in \mathbb{A}_s$ (Definition 4.3.1). In this case, by Definition 4.3.7.3, $\mathcal{G}^{\text{OP}}_{\text{AX}}$ adds α to A, and introduces $\rightarrow_{po}$ edges from all actions in A that belong to the thread $T(\alpha)$ to α (preserving the total order). $\rightarrow_{co}$ is extended with $\{\alpha \rightarrow_{co} \beta, \beta \rightarrow_{co} \alpha\}$. By the induction hypothesis, $\rightarrow_{co}$ is a symmetric relation. By Definition 4.3.1, α and β operate on the same channel. Thus, $\mathcal{G}^{\text{OP}}_{\text{AX}}(\mathsf{P}, \overline{a}.b)$ is an axiomatic execution. ■

Theorem 4.3.3 *Given* $\mathsf{E}_o = (\mathsf{P}, tr) \in \textsc{Rel}(\mathsf{P})$, $\mathcal{T}^{\text{OP}}_{\text{AX}}(\mathsf{E}_o) = \mathcal{G}^{\text{OP}}_{\text{AX}}(\mathsf{E}_o)$.

Proof By theorem 4.3.2, $\mathcal{G}^{\text{OP}}_{\text{AX}}(\mathsf{E}_o)$ is an axiomatic execution. We need to show that each component in the axiomatic execution defined by $\mathcal{T}^{\text{OP}}_{\text{AX}}$ and generated by $\mathcal{G}^{\text{OP}}_{\text{AX}}$ are the same. Both operators use the same program P and set of actions A from E_o.

$\mathcal{T}^{\text{OP}}_{\text{AX}}$ defines a $\rightarrow_{po}$ relation between actions α and β iff $T(\alpha) = T(\beta)$ and α precedes β in the trace (Definition 4.3.5.1). Assume that the operational trace is $tr = \overline{\alpha}.\beta$.

For any γ such that $T(\gamma) = T(\beta)$, $\gamma \in \overline{\alpha}$, γ belongs to the action set A of $\mathcal{G}^{\text{OP}}_{\text{AX}}(\mathsf{P}, \overline{\alpha})$. Definition 4.3.7.2 and 4.3.7.3 add a $\rightarrow_{po}$ edge from such a γ to β.

Definition 4.3.5.3 defines $\rightarrow_{co}$ such that for all pairs (α, β) in tr, $\alpha \rightarrow_{co} \beta$ and $\beta \rightarrow_{co} \alpha$. $\mathcal{G}^{\text{OP}}_{\text{AX}}$ extends $\rightarrow_{co}$ in a similar fashion (Definition 4.3.7.3). ■

Lemma 4.3.1 *Let* $\mathsf{E}_o = (\mathsf{P}, tr) \in \textsc{Rel}(\mathsf{P})$. *If* E_o *is well-formed, then* $\mathcal{G}^{\text{OP}}_{\text{AX}}(\mathsf{E}_o)$ *is well-formed.*

Proof Since E_o is well-formed, $\mathcal{T}^{\text{OP}}_{\text{AX}}(\mathsf{E}_o)$ is a well-formed axiomatic execution (by Definition 4.3.6). By Theorem 4.3.3, $\mathcal{T}^{\text{OP}}_{\text{AX}}(\mathsf{E}_o) = \mathcal{G}^{\text{OP}}_{\text{AX}}(\mathsf{E}_o)$. Hence, $\mathcal{G}^{\text{OP}}_{\text{AX}}(\mathsf{E}_o)$ is a well-formed axiomatic execution. ■

4.4 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. However, in practice, a speculative execution framework that discharges synchronous sends asynchronously (speculatively), needs to track the relations necessary to perform the integrity check *on-the-fly*, detect and remediate any execution that has a happens-before cycle.

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. By definition, this synchronous re-execution is bound to be correct. The context of our investigation is a distributed implementation of CML called R^{CML} (RelaXed CML) built on top of the MultiMLton SML compiler and runtime [39]. We have extended MultiMLton with the infrastructure necessary for distributed execution.

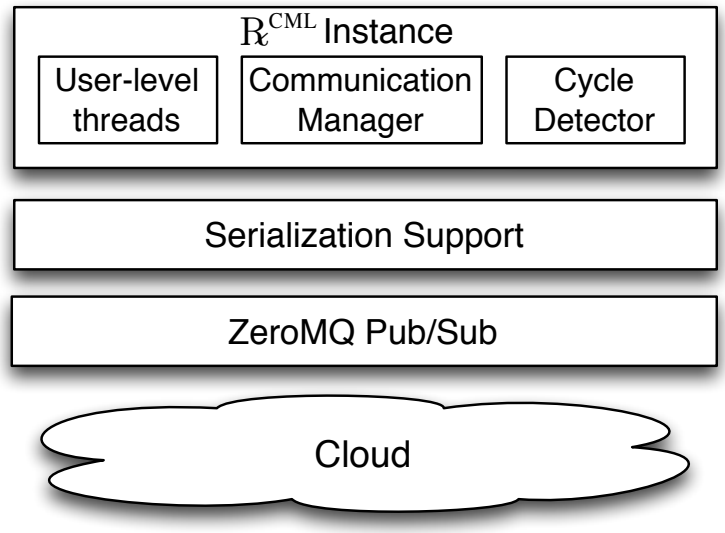


Figure 4.9. R_x^{CML} application stack.

4.4.1 System Architecture

A schematic diagram of the R_x^{CML} application stack is presented in Figure 4.9. An R_x^{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 [83] as the transport layer over which the R_x^{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.

The fact that every instance in an R_x^{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 function 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^{CML} application. Channels are first-class citizens and can be sent as messages over other channels to construct complex communication protocols.

4.4.2 Communication Manager

Each R^{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 4.10. 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^{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

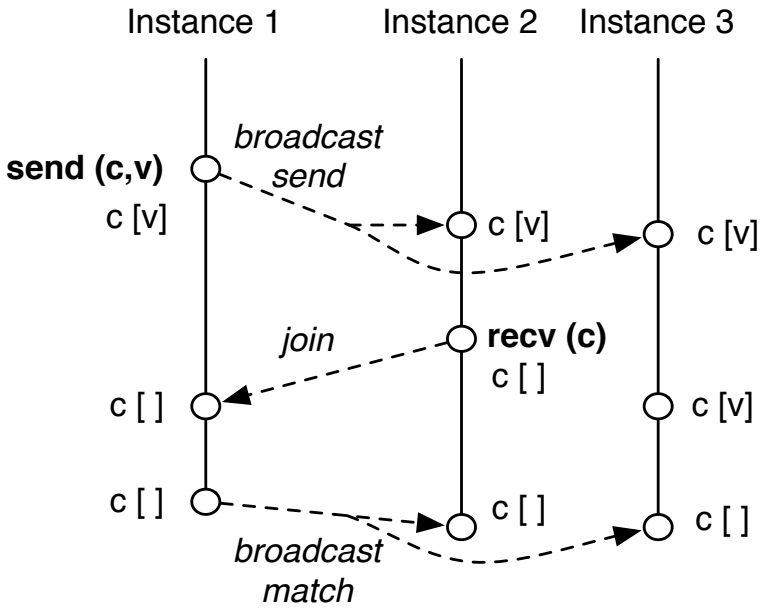


Figure 4.10. Communication manager behavior during a send and its matching receive.

not matched with this particular send remove the send action from the corresponding local channel replica.

4.4.3 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* [84] 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 [84] 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$^{\text{CML}}$ exploits the broadcast nature of the match message (Section 4.4.2) 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$^{\text{CML}}$ program to its synchronous equivalent, the compiler automatically inserts a *well-formedness check* before observable actions in the program. R$^{\text{CML}}$ 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 [84], every thread in an $\mathbb{R}^{\text{CML}}$ application has exactly one saved checkpoint continuation during the execution. Moreover, $\mathbb{R}^{\text{CML}}$ checkpointing is uncoordinated [85], 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 [84], 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 [85]. 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 $\mathbb{R}^{\text{CML}}$ 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.

4.4.4 Handling Full CML

Our discussion so far has been limited to primitive `send` and `recv` operations. $\mathrm{R}\kappa^{\mathrm{CML}}$ 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, $\mathrm{R}\kappa^{\mathrm{CML}}$ 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.

4.4.5 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 4.3 for receiving acknowledgments.

4.5 Case Studies

4.5.1 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 hold 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)
```

In order to obtain a lock, a `unit` value is *synchronously* sent to the `lockChan`. Since the lock server moves to a state where the only operation it can perform is receive a value on the `unlockChan`, we have the guarantee that no two threads can obtain the lock concurrently. After the transaction, a `unit` value is sent on the `unlockChan` to release the lock. 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. Of course, synchronous communication on the `lockChan` ensures atomicity of a transaction performed on the resource protected by the lock. However, these guarantees are lost in the presence of asynchrony.

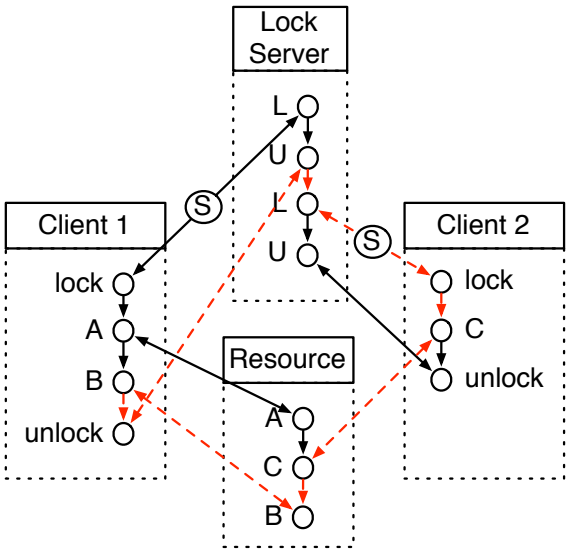


Figure 4.11. A possible serializability violation that arises because of asynchronous (speculative) communication with the lock server.

Consider the example presented in Figure 4.11, which shows a serializability violation that arises because communication with the lock server takes place asynchronously, effectively introducing speculation. Two clients transactionally perform operations A,B and C, resp. on a shared resource. This resource is protected by the lock server. Happens-before edges are represented as directed edges. During speculative execution, clients send lock requests speculatively, labeled Ⓢ on the edges, allowing them to continue with their transactions without waiting for the lock server to respond.

In this figure, serializability violations are captured as a cycle in the dependence graph, represented by the dotted edges. R$^{\text{CML}}$ rejects such executions, causing the transaction to abort, and be re-executed non-speculatively.

Results

For our evaluation, we implemented a distributed version of this program (`vacation`) taken from the STAMP benchmark suite [86]. To adapt the benchmark for a dis-

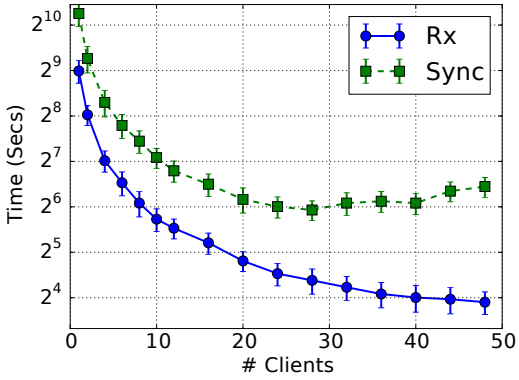


Figure 4.12. Performance comparison on distributed `vacation` (OLTP) benchmark. Lower is better.

tributed 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.

The performance results are presented in the Figure 4.12. The number of clients concurrently issuing transaction requests was increased from 1 to 48. R^{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 R^{CML} scales much better than the Sync version due to optimistic transactions. With 48 clients, R^{CML} version was 5.8× faster than then Sync version. Under R^{CML}, the number of transaction conflicts does increase with the number of clients. With 48 clients, 9% of the transactions executed under R^{CML} were tagged as conflicting and re-executed non-speculatively. This does not, however, adversely affect scalability.

4.5.2 Collaborative Editing

Our next case study is a real-time, decentralized collaborative editing tool. Typically, commercial offerings such as Google Docs, 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 work on a local copy of the shared document for responsiveness, with remote updates merged incrementally. Although replicas are allowed to diverge, they are expected to converge eventually. This convergence is achieved through *operational transformation* [87]. Dealing with operational transformation in the absence of a centralized server is tricky [88], and commercial collaborative editing services like Google Wave impose additional restrictions with respect to the frequency of remote updates [89] 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. Each client participating in the collaborative editing session runs a server daemon, whose implementation is given in Figure 4.13. The server daemon fetches updates from the user-interface thread (client) over the channel `lc`, and coordinates with other server daemons at other remote locations over the channel `rc`. `rIn` represents the list of incoming remote operations that have not yet been merged with the local document replica. `lOut` represents the list of local operations yet to be broadcast.

```
1  (* lc: client chan, sc: server chan, tc: timeOut chan *)
2  fun serverDaemon id rIn lOut =
3  let
4    (* Updates from other server daemons *)
5    val remoteRecv = wrap (brecvEvt sc, fn rOps' =>
6          serverDaemon id (rIn @ rIn') lOut)
7    (* Updates to other server daemons *)
8    val remoteSend =
9      if lOut = [] then neverEvt ()
10     else
11       let val (lOut',_) = xform (lOut, rIn)
12       in wrap (bsendEvt (sc, lOut', id),
13                fn () => serverDaemon id rIn [])
14       end
15   (* interaction with the client *)
16   val localComm =
17     wrap (recvEvt tc, fn () =>
18       let
19         val lOps = sync (choose (recvEvt lc, alwaysEvt []))
20         val (_, rIn') = xform (lOps, rIn)
21         val _ = updateDocument(rIn')
22       in serverDaemon id [] (lOut @ lOps)
23       end)
24 in
25   sync (choose (localComm,remoteRecv,remoteSend))
26 end
27
28 fun timeoutManager to =
29   sync (wrap (timeoutEvent to,
30     fn () => send (tc,()); timerThread to))
```

Figure 4.13. Server Daemon for Collaborative Editing.

At every iteration of the server daemon loop, there is a choice (line 25) between performing (a) a local receive (`localComm`), (b) a remote send (`remoteSend`), or (c) a remote receive (`remoteRecv`). We have extended the implementation of broadcast primitives presented in Figure 4.3 with events that encapsulate broadcast send and receive. Remote sends are only enabled if `lOut` is not empty. Otherwise, it is a `neverEvt()` which will never be picked in a choice. If `lOut` is not empty, then the outstanding messages are transformed against the remote messages and sent to all other daemons using causal atomic broadcast (lines 11-13).

By receiving broadcasted messages on the same thread as the one that performs the broadcast, we ensure a total order on message reception at every client. Causal atomic broadcast ensures that all daemons receive the update in the same order, ensuring convergence of all remote states. On receiving a remote update, the server daemon simply appends the update to the list of pending updates yet to be applied to the local replica (lines 5-6).

The user-interface thread sends the updates to the server daemon on the `lc` channel, making them available to other replicas. This communication is also made asynchronous through speculation, so that the UI stays responsive to the author. The daemon uses a `timeoutManager` (lines 28-30) to periodically fetch updates from the user interface thread. The daemon then receives local updates `lOps` , if any, from the channel `lc` (lines 19).

Causal atomic broadcast for inter-daemon communication ensures that operations in `rIn` at every daemon appear in the same order. In other words, every daemon is in the same abstract state. Hence, we can simply transform the unapplied remote operations `rIn` with respect to local operations `lOps` , to yield an `rIn'` (line 20) that considers the remote updates in the context of local ones. The daemon then updates the document with the remote operations, by applying further transformation to account for additional local updates that might have occurred between the time the user-interface sent a message to it and now (line 21). This operation might perform

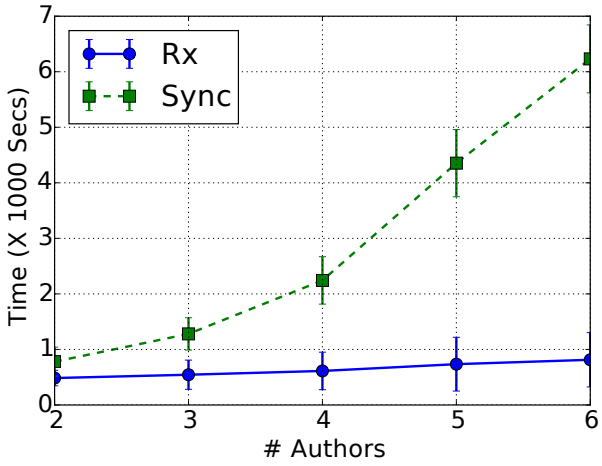


Figure 4.14. Performance comparison on collaborative editing benchmark. Lower is better.

a well-formedness check to check for consistency as updating the document is an effectful operation. This commit prevents speculation from leaking to the user.

Results

We use a collaborative editing benchmark generator described in [90] 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.

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 4.14 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.

4.6 Related Work

Causal-ordering of messages is considered an important building block [82] for distributed applications. Similar to our formulation, Charron-Bost *et al.* [91] 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 [92], asynchronous virtual machine replication [93], state machine replication [94], deadlock detection [95] etc., although we are unaware of other attempts to use it for transparently converting synchronous operations to asynchronous ones.

Besides Erlang [96], there are also several distributed implementations of functional languages that have been proposed [97, 98]. More recently, Cloud Haskell [99] 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, $\mathrm{R}^{\mathrm{CML}}$'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 [20,21,100]. 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 [20, 100] are useful in reasoning about compiler optimizations, our relaxed communication model reasons about safe asynchronous manifestations of synchronous protocols.

Transactional events(TE) [101, 102] 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) [103] 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.

4.7 Concluding Remarks

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 $\mathrm{R}_{\mathrm{x}}^{\mathrm{CML}}$ that incorporates these ideas. Our reported case studies illustrate the benefits of our approach, and provide evidence that $\mathrm{R}_{\mathrm{x}}^{\mathrm{CML}}$ is a basis upon which we can build clean, robust, and efficient distributed CML programs.

5 QUELEA: DECLARATIVE PROGRAMMING OVER EVENTUALLY CONSISTENT DATA STORES

Many real-world web services — such as those built and maintained by Amazon, Facebook, Google, Twitter, etc. — replicate application state and logic across multiple *replicas* within and across data centers. Replication is intended not only to improve application throughput and reduce user-perceived latency, but also to tolerate partial failures without compromising overall service availability. Traditionally programmers have relied on *strong consistency* guarantees such as linerarizability [18] or serializability [19] in order to build correct applications. While strong consistency is an easily stated property, it masks the reality underlying large-scale distributed systems with respect to non-uniform latency, availability and network partitions [29,31]. Indeed, modern web services, which aim to provide an "always on" experience, overwhelmingly favor availability and partition tolerance over strong consistency. To this end, several *weak consistency* models such as eventual consistency, causal consistency, session guarantees, and timeline consistency have been proposed.

Under weak consistency, the developer needs to be aware of concurrent conflicting updates, and has to pay careful attention to avoid unwanted inconsistencies (e.g., negative balances in a bank account, or having an item appear in a shopping cart after it has been removed [9]). Oftentimes, the inconsistency leaks from the application and is witnessed by the user. Ultimately, the developer must decide the consistency level appropriate for a particular operation; this is understandably an error-prone process requiring intricate knowledge of both the application as well as the semantics and implementation of the underlying data store, which typically have only informal descriptions. Nonetheless, picking the correct consistency level is critical not only for correctness but also for scalability of the application. While choosing a weaker

consistency level than required may introduce program errors and anomalies, choosing a stronger one than necessary can negatively impact program scalability.

Weak consistency also hinders compositional reasoning about programs. While an application might be naturally expressed in terms of well-understood and expressive data types such as maps, trees, queues, or graphs, geo-distributed stores typically only provide a minimal set of data types with in-built conflict resolution strategies such as last-writer-wins (LWW) registers, counters, and sets [10, 52]. Furthermore, while traditional database systems enable composability through transactions, geo-distributed stores typically lack unrestricted transactional access to the data. Working in this environment thus requires application state to be suitably coerced to function using only the capabilities of the store.

To address these issues, we describe Quelea, a declarative programming model and implementation for eventually consistent geo-distributed data stores. The key novelty of Quelea is an expressive *contract language* to declare and verify fine-grained application-level consistency properties. The programmer uses the contract language to axiomatically specify the set of legal executions allowed over the replicated data type. Contracts are constructed using primitive consistency relations such as *visibility* and *session order* along with standard logical and relational operators. A *contract enforcement system* automatically maps operations over the datatype to a particular consistency level available on the store, and provably validates the correctness of the mapping. The chapter makes the following contributions:

- We introduce Quelea, a shallow extension of Haskell that supports the description and validation of replicated data types found on eventually consistent stores. Contracts are used to specify fine-grained application-level consistency properties, and are analyzed to assign the most efficient and sound store consistency level to the corresponding operation.

- Quelea supports coordination-free transactions over arbitrary datatypes. We extend our contract language to express fine-grained transaction isolation guar-

antees, and utilize the contract enforcement system to automatically assign the correct isolation level for a transaction.

- We provide metatheory that certifies the soundness of our contract enforcement system, and ensures that an operation is only executed if the required conditions on consistency are met.

- An implementation of QUELEA as a transparent shim layer over Cassandra [10], a well-known general-purpose data store. Experimental evaluation over a set of real-world applications, including a Twitter-like micro-blogging site and an eBay-like auction site illustrates the practicality of our approach.

The rest of the chapter is organized as follows. The next section describes the system model. We describe the challenges in programming with eventually consistent data stores, and introduces QUELEA contracts as a proposed solution to overcome these issues in Section 5.2. Section 5.3 provides more details the contract language, and its mapping to the store consistency levels. Section 5.4 presents the meta-theoretic result that certifies the correctness of the QUELEA contract enforcement. Section 5.5 introduces transaction contracts and classification. Section 5.6 describes the implementation and provides details about the optimizations needed to make the system practical. Section 5.7 discusses experimental evaluation. Section 5.8 and 5.9 present related work and conclusions.

5.1 System Model

Figure 5.1 provides a schematic diagram of our system model. The distributed store is composed of a collection of *replicas*, each of which stores a set of *objects* $(x, y, \ldots)$. We assume that every object is replicated at every replica in the store. The state of an object at any replica is the set of all updates (*effects*) performed on the object. For example, the state of x at replica 1 is the set composed of effects w_1^x and w_2^x.

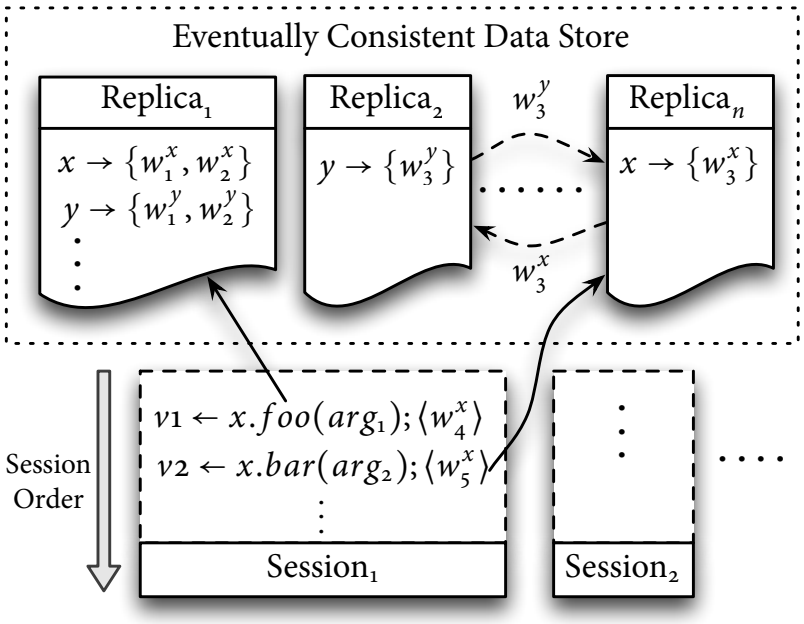


Figure 5.1. QUELEA system model.

Each object is associated with a set of *operations*. The clients interact with the store by invoking operations on objects. The sequence of operations invoked by a particular client on the store is called a *session*. The data store is typically accessed by a large number of clients (and hence sessions) concurrently. Importantly, the clients are oblivious to which replica an operation is applied to; the data store may choose to route the operation to any replica in order to minimize latency, balance load, etc. For example, the operations *foo* and *bar* invoked by the same session on the same object, might end up being applied to different replicas because replica 1 (to which *foo* was applied) might be unreachable when the client invokes *bar*.

When *foo* is invoked on a object x with arguments arg_1 at replica 1, it simply *reduces* over the current set of effects at that replica on that object (w_1^x and w_2^x), produces a result $v1$ that is sent back to the client, and emits a *single* new effect w_4^x that is appended to the state of x at replica 1. Thus, every operation is evaluated over a *snapshot* of the state of the object on which it is invoked. In this case, the effects w_1^x and w_2^x are *visible* to w_4^x, written logically as $\mathsf{vis}(w_1^x, w_4^x) \wedge \mathsf{vis}(w_2^x, w_4^x)$, where vis is the

visibility relation between effects. Visibility is an irreflexive and asymmetric relation, and only relates effects produced by operations on the same object. Executing a read-only operation is similar except that no new effects are produced.

The effect added to a particular replica is asynchronously sent to other replicas, and eventually merged into all other replicas. Two effects w_4^x and w_5^x that arise from the same session are said to be in *session order* (written logically as $\mathsf{so}(w_4^x, w_5^x)$). Session order is an irreflexive, transitive relation. The effects w_4^x and w_5^x arising from operations applied to the same object x are said to be under the *same object* relation, written $\mathsf{sameobj}(w_4^x, w_5^x)$. Finally, we can associate every effect with the operation that generated the effect with the help of a relation oper. In the current example, $\mathsf{oper}(w_4^x, foo)$ and $\mathsf{oper}(w_5^x, bar)$ hold. For simplicity, we assume all operation names across all object types are distinct.

This model admits all the inconsistencies associated with eventual consistency. The goal of this work is to identify the precise consistency level for each operation such that application-level constraints are not violated. In the next section, we will concretely describe the challenges associated with constructing a consistent bank account on top of an eventually consistent data store. Subsequently, we will illustrate how our contract and specification language, armed with the primitive relations vis, so, sameobj and oper, mitigates these challenges.

5.2 Motivation

Consider how we might implement a highly available bank account on top of an eventually consistent data store, with the *integrity* constraint that the balance must be non-negative. We begin by implementing a bank account replicated data type (RDT) in Quelea, and then describe the mechanisms to obtain the desired correctness guarantees.

5.2.1 RDT Specification

A key novelty in QUELEA is that it allows the addition of new RDTs to the store, which obviates the need for coercing the application logic to utilize the store provided data types. In addition, QUELEA treats the convergence semantics of the data type separately from its consistency properties. This separation of concerns permits *operational* reasoning for conflict resolution, and *declarative* reasoning for consistency. The combination of these techniques enhances the programmability of the store.

Let us assume that the bank account object provides three operations: `deposit`, `withdraw` and `getBalance`, with the assumption that the withdraw fails if the account has insufficient balance. Every operation in QUELEA is of the following type, written in Haskell syntax:

```
1 type Operation e a r = [e] → a → (r, Maybe e)
```

It takes a list of effects (the *context* for the operation), and an input argument, and returns a result along with an optional effect (read-only operations return `Nothing`). The new effect (if emitted) is added to the state of the object at the current replica, and asynchronously sent to other replicas. The implementation of the bank account operations in QUELEA is given in Figure 5.2:

The datatype `Acc` represents the effect type for the bank account. The context of the operations is a snapshot of the state of the object at some replica. In this sense, every operation on the RDT is atomic, and thus permitting sequential reasoning for implementing eventually consistent data types. We have implemented a large corpus of RDTs for realistic benchmarks including shopping carts, auction and microblogging sites in few tens of lines of code.

```
1  data Acc = Deposit Int | Withdraw Int | GetBalance
2
3  getBalance :: [Acc] → () → (Int, Maybe Acc)
4  getBalance ctxt _ =
5    let res = sum [x | Deposit x ← ctxt]
6              - sum [x | Withdraw x ← ctxt]
7    in (res, Nothing)
8
9  deposit :: [Acc] → Int → ((), Maybe Acc)
10 deposit _ amt = (amt, Just $ Deposit amt)
11
12 withdraw :: [Acc] → Int → (Bool, Maybe Acc)
13 withdraw ctxt v =
14   if sel1 $ getBalance ctxt () ≥ v
15   then (True, Just $ Withdraw v)
16   else (False, Nothing)
```

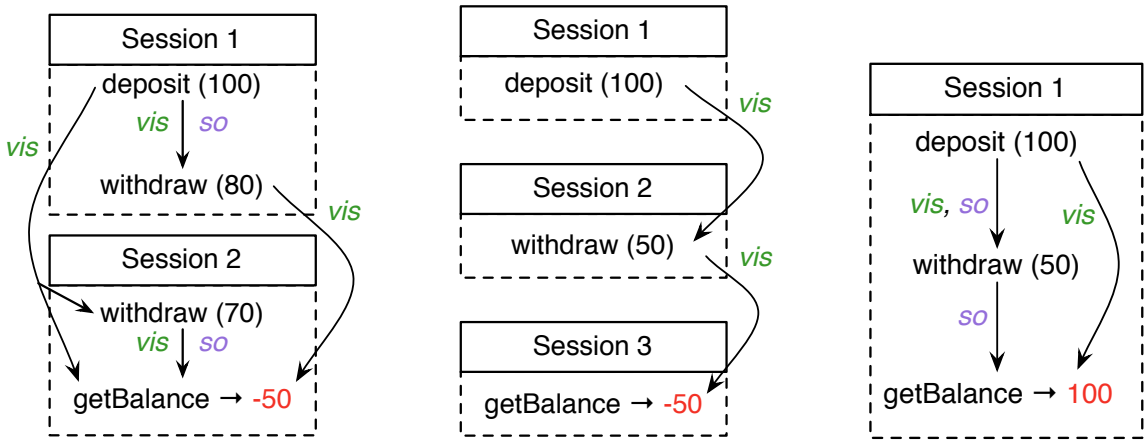
Figure 5.2. Definition of a bank account expressed in Quelea.

5.2.2 Anomalies under Eventual Consistency

Our goal is to choose the correct consistency level for each of the bank account operations such that (1) the balance remains non-negative and (2) the `getBalance` operation never incorrectly returns a negative balance. Let us first consider the anomalies that could arise under eventual consistency.

Consider the execution shown in Figure 5.3(a). Assume that all operations in the figure are on the same bank account object with the initial balance being zero. Session 1 performs a `deposit` of 100, followed by a `withdraw` of 80 in the same session. The `withdraw` operation witnesses the deposit and succeeds[1]. Subsequently, session 2 perform a `withdraw` operation, but importantly, due to eventual consistency, only

[1]Although visibility and session order relations relate effects, we have abused the notation in these examples to relate operations, with the idea that the relations relate the effect emitted by those operations



(a) Unsafe withdraw (b) Negative balance (c) Missing update

Figure 5.3. Anomalies possible under eventual consistency for the get balance operation.

witnesses the `deposit` from session 1, but not the subsequent withdraw. Hence, this `withdraw` also *incorrectly* succeeds, violating the integrity constraint. A subsequent `getBalance` operation, that happens to witness all the previous operations, would report a negative balance.

It is easy to see that preventing concurrent `withdraw` operations eliminates this anomaly. This can be done by insisting that `withdraw` be executed as a strongly consistent operation. Despite this strengthening, `getBalance` operation may incorrectly report a negative balance to the end user. Consider the execution shown in fig. 5.3(b), which consists of three concurrent sessions performing a `deposit`, a `withdraw`, and a `getBalance` operation, respectively, on the same bank account object. As the `vis` edge indicates, operation `withdraw(50)` in session 2, witnesses the effects of `deposit(100)` from session 1, concludes that there is sufficient balance, and completes successfully. However, the `getBalance` operation may only witness this successful withdraw, but not the causally preceding `deposit`, and reports the balance of negative 50 to the user.

Under eventual consistency, the users may also be exposed to other forms of inconsistencies. Figure 5.3(c) shows an execution where the `getBalance` operation in a session does not witness the effects of an earlier `withdraw` operation performed in the same session, possibly because it was served by a replica that has not yet merged the `withdraw` effect. This anomaly leads the user to incorrectly conclude that the `withdraw` operation failed to go through.

Although it is easy to understand the reasons behind the occurrence of the aforementioned anomalies, finding the appropriate fixes is not readily apparent. Making `getBalance` a strongly consistent operation is definitely sufficient to avert anomalies, but is it really necessary? Given the cost of enforcing strong consistency [52, 104], it is preferable to avoid it unless there are no viable alternatives. Exploring the space of these alternatives requires understanding the subtle differences in semantics of various kinds of weak consistency alternatives.

5.2.3 Contracts

QUELEA helps facilitate the mapping of operations to appropriate consistency levels by letting the programmer declare application-level consistency constraints as *contracts* (Figure 5.4[2]) that axiomatically specify the set of allowed executions involving this operation. In the case of the bank account, any execution that does not exhibit the anomalies described in the previous section is a *well-formed* execution on the bank account object. By specifying the set of legal executions for each data type in terms of a trace of operation invocations on that type, QUELEA ensures that all executions over that type are well-formed.

In our running example, it is clear that in order to preserve the integrity constraint, the `withdraw` operation must be strongly consistent. That is, given two `withdraw`

[2]QUELEA exposes the contract construction language as a Haskell library

operations a and b, either a is visible to b or vice versa. We express this application-level consistency requirement as a contract (ψ_w) over `withdraw`:

$$\forall(a : \texttt{withdraw}).\ \mathsf{sameobj}(a, \hat{\eta}) \Rightarrow a = \hat{\eta} \vee \mathsf{vis}(a, \hat{\eta}) \vee \mathsf{vis}(\hat{\eta}, a)$$

Here, $\hat{\eta}$ stands for the effect emitted by the `withdraw` operation. The syntax a : `withdraw` states that a is an effect emitted by a `withdraw` operation i.e., $\mathsf{oper}(a, \texttt{withdraw})$ holds. The contract specifies that if the current operation emits an effect $\hat{\eta}$, then for any operation a which was emitted by a `withdraw` operation, it is the case that $a = \hat{\eta}$ or a is visible to $\hat{\eta}$, or vice versa. Any execution on a bank account object that preserves the above contract for a `withdraw` operation is said to be derived from a correct implementation of `withdraw`.

For `getBalance`, we construct the following contract (ψ_{gb}):

$$\begin{aligned}\forall(a : \texttt{deposit}), (b : \texttt{withdraw}), (c : \texttt{deposit} \vee \texttt{withdraw}).\\ \mathsf{vis}(a, b) \wedge \mathsf{vis}(b, \hat{\eta}) \Rightarrow \mathsf{vis}(a, \hat{\eta})\\ \wedge\ (\mathsf{so} \cap \mathsf{sameobj})(c, \hat{\eta}) \Rightarrow \mathsf{vis}(c, \hat{\eta})\end{aligned}$$

The expression c : `deposit` $\vee$ `withdraw` states that c is an effect that was emitted either by a `deposit` or a `withdraw` operation. If a `withdraw` b is visible to `getBalance` $\hat{\eta}$, then all `deposit` operations a visible to b should also be visible to $\hat{\eta}$. This prevents negative balance anomalies. Our contract language provides operators to compose relations. The syntax $(R_1 \cap R_2)(a, b)$ is equivalent to $R_1(a, b) \wedge R_2(a, b)$. The last line of the above contract says that if a `deposit` or a `withdraw` operation precedes a `getBalance` operation in session order, and is applied on the same object as the `getBalance` operation, then it must be the case that the `getBalance` operation witnesses the effects of the preceding operations.

Finally, since there are no restrictions on when or how a `deposit` operation can execute, its contract is simply true.

5.2.4 From Contracts to Implementation

Notice that the contracts for `withdraw` and `getBalance` only express application-level consistency requirements, and make no reference to the semantics of the underlying store. To write contracts, a programmer only needs to reason about the semantics of the application under the QUELEA system model. The mapping of application-level consistency requirements to appropriate store-level guarantees is done automatically behind-the-scene. How might one go about ensuring that an execution adheres to a contract? The challenge is that a contract provides a declarative (axiomatic) specification of an execution, while what is required is an operational procedure for *enforcing* its implicit constraints.

One strategy would be to execute operations speculatively. Here, operations are tentatively applied as they are received from the client or other replicas. We can maintain a runtime manifestation of executions, and check well-formedness conditions at runtime, rolling back executions if they are ill-formed. However, the overhead of state maintenance and the complexity of user-defined contracts is likely to make this technique infeasible in practice.

We devise a static approach instead. Contracts are analyzed with the help of a theorem prover, and statically mapped to a particular store-level consistency property that the prover guarantees preserves contract semantics. We call this procedure *contract classification*. Given the variety and complexity of store level consistency properties, the idea is that the system implementor parameterizes the classification procedure by describing the store semantics in the *same* contract language as the one used to express the contract on the operations. In the next section, we describe the contract language in detail and describe the classification procedure for a particular store semantics.

$$
\begin{array}{rcll}
 & x, y, \hat{\eta} \in \texttt{EffVar} & & \mathsf{Op} \in \texttt{OperName} \\
\psi & \in \texttt{Contract} & ::= & \forall(x:\tau).\psi \mid \pi \\
\tau & \in \texttt{EffType} & ::= & \mathsf{Op} \mid \tau \vee \tau \\
\pi & \in \texttt{Prop} & ::= & \mathsf{true} \mid R(x,y) \mid \pi \vee \pi \\
 & & \mid & \pi \wedge \pi \mid \pi \Rightarrow \pi \\
R & \in \texttt{Relation} & ::= & \mathsf{vis} \mid \mathsf{so} \mid \mathsf{sameobj} \mid R^{+} \\
 & & \mid & R \cup R \mid R \cap R
\end{array}
$$

Figure 5.4. Contract language.

5.3 Contract Language

5.3.1 Syntax

The syntax of our core contract language is shown in Figure 5.4. The language is based on first-order logic (FOL), and admits prenex universal quantification over typed and untyped effect variables. We use a special effect variable ($\hat{\eta}$) to denote the effect of *current operation* - the operation for which a contract is being written. The type of an effect is simply the name of the operation (eg: `withdraw`) that induced the effect. We admit disjunction in types to let an effect variable range over multiple operation names. The contract $\forall(a : \tau_1 \vee \tau_2).\ \psi$ is just syntactic sugar for $\forall a.(\mathsf{oper}(a, \tau_1) \vee \mathsf{oper}(a, \tau_2)) \Rightarrow \psi$. An untyped effect variable ranges over all operation names.

Quantifier-free propositions in our contract language are conjunctions, disjunctions and implications of predicates expressing relations between pairs of effect variables. The syntactic class of relations is seeded with primitive vis, so, and sameobj rela-

$\eta \in \texttt{Effect}$ $\quad \psi \in \texttt{Contract}$ $\quad \overline{\eta} \in \texttt{Effect Set}$

$$\begin{array}{lcll} \mathsf{A} & \in & \texttt{EffSoup} & := \ \overline{\eta} \\ \mathsf{vis}, \mathsf{so}, \mathsf{sameobj} & \in & \texttt{Relations} & := \ \mathsf{A} \times \mathsf{A} \\ \mathsf{E} & \in & \texttt{ExecState} & := \ (\mathsf{A},\mathsf{vis},\mathsf{so},\mathsf{sameobj}) \end{array}$$

Figure 5.5. Axiomatic execution.

tions, and also admits derived relations that are expressible as union, intersection, or transitive closure[3] of primitive relations.

- Same object session order: $\mathsf{soo} = \mathsf{so} \ \cap \ \mathsf{sameobj}$.
- Happens-before order: $\mathsf{hb} = (\mathsf{so} \ \cup \ \mathsf{vis})^+$.
- Same object happens-before order: $\mathsf{hbo} = (\mathsf{soo} \ \cup \ \mathsf{vis})^+$.

5.3.2 Semantics

QUELEA contracts are constraints over axiomatic definitions of program executions. Figure 5.5 summarizes artifacts relevant to define an axiomatic execution. We formalize an axiomatic execution as a tuple $(\mathsf{A},\mathsf{vis},\mathsf{so},\mathsf{sameobj})$, where A, called the *effect soup*, is the set of all effects generated during the program execution, and $\mathsf{vis}, \mathsf{so}, \mathsf{sameobj} \subseteq \mathsf{A} \times \mathsf{A}$ are *visibility*, *session order*, and *same object* relations, respectively, witnessed over generated effects at run-time.

Note that the axiomatic definition of an execution (E) provides interpretations for primitive relations (eg: vis) that occur free in contract formulas, and also fixes the domain of quantification to set of all effects (A) observed during the program execution. As such, E is a potential model for any first-order formula (ψ) expressible in

[3]Strictly speaking, R^+ is not the transitive closure of R, as transitive closure is not expressible in FOL. Instead, R^+ in our language denotes *a* superset of transitive closure of R. Formally, R^+ is any relation R' such that forall x, y, and z, a) $R(x,y) \Rightarrow R'(x,y)$, and b) $R'(x,y) \ \wedge \ R'(y,z) \Rightarrow R'(x,z)$

our contract language. If E is indeed a valid model for ψ (written as $\mathsf{E} \models \psi$), we say that the execution E satisfied the contract ψ:

Definition 5.3.1 *An axiomatic execution* E *is said to satisfy a contract* ψ *if and only if* $\mathsf{E} \models \psi$.

5.3.3 Capturing Store Semantics

An important aspect of our contract language is its ability to capture store-level consistency guarantees, along with application-level consistency requirements. Similar to [54], we can rigorously define a wide variety of store semantics including those that combine any subset of session and causality guarantees, and multiple consistency levels. However, for our purposes, we identify three particular consistency levels – eventual, causal, and strong, commonly offered by many distributed stores with tunable consistency, with increasing overhead in terms of latency and availability.

- **Eventually consistency**: Eventually consistent operations can be satisfied as long as the client can reach at least one replica. In the bank account example, `deposit` is an eventually consistent operation. While eventually consistent data stores typically offer *basic* eventual consistency with all possible anomalies, we assume that our store provides stronger semantics that remain highly-available [34, 105]; the store always exposes a *causal cut* of the updates. This semantics can be formally captured in terms of the following contract definition:

$$\psi_{\mathsf{ec}} = \forall a, b.\ \mathsf{hbo}(a, b) \wedge \mathsf{vis}(b, \hat{\eta}) \Rightarrow \mathsf{vis}(a, \hat{\eta})$$

- **Causal consistency**: Causally consistent operations are required to see a causally consistent snapshot of the object state, including the actions performed on the same session. The latter requirement implies that if two operations o_1 and o_2 from the same session are applied to two different replicas r_1 and r_2, the second operation cannot be discharged until the effect of o_1 is included in

$$\frac{\psi \leq \psi_{\mathsf{sc}}}{\mathsf{WellFormed}(\psi)} \qquad \frac{\psi \leq \psi_{\mathsf{ec}}}{\mathsf{EventuallyConsistent}(\psi)}$$

$$\frac{\psi \not\leq \psi_{\mathsf{ec}} \quad \psi \leq \psi_{\mathsf{cc}}}{\mathsf{CausallyConsistent}(\psi)} \qquad \frac{\psi \not\leq \psi_{\mathsf{cc}} \quad \psi \leq \psi_{\mathsf{sc}}}{\mathsf{StronglyConsistent}(\psi)}$$

Figure 5.6. Contract classification.

r_2. The `getBalance` operation requires causal consistency, as it requires the operations from the same session to be visible, which cannot be guaranteed under eventual consistency. The corresponding store semantics is captured by the contract ψ_{cc} defined below:

$$\psi_{\mathsf{cc}} = \forall a.\ \mathsf{hbo}(a, \hat{\eta}) \Rightarrow \mathsf{vis}(a, \hat{\eta})$$

- **Strong Consistency**: Strongly consistent operations may block indefinitely under network partitions. An example is the total-order contract on `withdraw` operation. The corresponding store semantics is captured by the ψ_{sc} contract definition:

$$\psi_{\mathsf{sc}} = \forall a.\ \mathsf{sameobj}(a, \hat{\eta}) \Rightarrow \mathsf{vis}(a, \hat{\eta}) \ \vee \ \mathsf{vis}(\hat{\eta}, a) \ \vee \ a = \hat{\eta}$$

5.3.4 Contract Comparison and Classification

Our goal is to map application-level consistency constraints on operations to appropriate store-level consistency guarantees capable of satisfying these constraints. The ability to express both these kinds of constraints as contracts in our contract language lets us compare and determine if contract (ψ_{op}) of an operation (op) is weak enough to be satisfied under a store consistency level identified by the contract ψ_{st}.

Towards this end, we define a binary *weaker than* relation for our contract language as following:

Definition 5.3.2 *A contract ψ_{op} is said to be weaker than ψ_{st} (written $\psi_{op} \leq \psi_{st}$) if and only if $\Delta \vdash \forall \hat{\eta}.\psi_{st} \Rightarrow \psi_{op}$.*

The quantifier in the sequent binds $\hat{\eta}$ that occurs free in ψ_{st} and ψ_{op}. Context (Δ) of the sequent is a conjunction of assumptions about the nature of primitive relations. A *well-formed* axiomatic execution (E) is expected to satisfy these assumptions (i.e., $\mathsf{E} \models \Delta$).

Definition 5.3.3 *An axiomatic executions $\mathsf{E} = (\mathsf{A},\mathsf{vis},\mathsf{so},\mathsf{sameobj})$ is said to be well-formed if the following axioms (Δ) hold:*

- *The happens-before relation is acyclic:* $\forall a.\ \neg\mathsf{hb}(a,a)$.
- *Visibility only relates actions on the same object:* $\forall a,b.\ \mathsf{vis}(a,b) \Rightarrow \mathsf{sameobj}(a,b)$.
- *Session order is a transitive relation:* $\forall a,b,c.\ \mathsf{so}(a,b)\ \wedge\ \mathsf{so}(b,c) \Rightarrow \mathsf{so}(a,c)$.
- *Same object is an equivalence relation:*
  - $\forall a.\ \mathsf{sameobj}(a,a)$.
  - $\forall a,b.\ \mathsf{sameobj}(a,b) \Rightarrow \mathsf{sameobj}(b,a)$.
  - $\forall a,b,c.\ \mathsf{sameobj}(a,b)\ \wedge\ \mathsf{sameobj}(b,c) \Rightarrow \mathsf{sameobj}(a,c)$.

If the contract (ψ_{op}) of an operation (op) is *weaker than* a store contract (ψ_{st}), then constraints expressed by the former are implied by guarantees provided by the latter. The completeness of first-order logic allows us to assert that any well-formed execution (E) that satisfies ψ_{st} (i.e., $\mathsf{E} \models \psi_{st}$) also satisfies ψ_{op} (i.e., $\mathsf{E} \models \psi_{op}$). Consequently, it is safe to execute operation op under a store consistency level captured by ψ_{st}.

Observe that the contracts ψ_{sc}, ψ_{cc} and ψ_{ec} are themselves totally ordered with respect to the $\leq$ relation: $\psi_{\mathsf{ec}} \leq \psi_{\mathsf{cc}} \leq \psi_{\mathsf{sc}}$. This concurs with the intuition that

any contract satisfiable under ψ_{ec} or ψ_{cc} is satisfiable under ψ_{sc}, and any contract that is satisfiable under ψ_{ec} is satisfiable under ψ_{cc}. We are interested in the *weakest* guarantee (among ψ_{ec}, ψ_{cc}, and ψ_{sc}) required to satisfy the contract. We define the corresponding consistency level as the *consistency class* of the contract.

The classification scheme, presented formally in Figure 5.6, defines rules to judge the consistency class of a contact. For example, the scheme classifies the `getBalance` contract (ψ_{gb}) from Section 5.2 as a CausallyConsistent contract, because the sequent $\Delta \vdash \psi_{\mathsf{cc}} \Rightarrow \psi_{gb}$ is valid in first-order logic (therefore, $\psi_{gb} \leq \psi_{\mathsf{cc}}$), whereas the sequent $\Delta \vdash \psi_{\mathsf{ec}} \Rightarrow \psi_{gb}$ is invalid (therefore, $\psi_{gb} \not\leq \psi_{\mathsf{ec}}$). Since we confine of our contract language to a decidable subset of the logic, validity of such sequents can be decided mechanically allowing us to automate the classification scheme in Quelea.

Along with three straightforward rules that classify contracts into consistency classes, the classification scheme also presents a rule that judges well-formedness of a contract. A contract is well-formed if and only if it is satisfiable under ψ_{sc} - the strongest possible consistency guarantee that the store can provide. Otherwise, it is considered ill-formed, and rejected statically.

5.3.5 Soundness of Contract Classification

We now present a meta-theoretic result that certifies the soundness of classification-based contract enforcement. To help us state the result, we define an operational semantics of the our system described informally in Section 5.1:

$$
\begin{array}{lcll}
op & \in & \texttt{Operation} & \\
\tau & \in & \texttt{ConsistencyClass} & ::= \ \mathsf{ec}, \mathsf{cc}, \mathsf{sc} \\
\sigma & \in & \texttt{Session} & ::= \ \cdot \mid \langle op, \tau \rangle ; \sigma \\
\Sigma & \in & \texttt{Session Soup} & ::= \ \sigma \parallel \Sigma \mid \emptyset \\
 & & \texttt{Config} & ::= \ \mathsf{E}, \Sigma
\end{array}
$$

We model the system as a tuple E, Σ, where the axiomatic execution E captures the data store's current state, and session soup Σ is the set of concurrent client sessions

interacting with the store. A session σ is a sequence of pairs composed of replicated data type operations *op*, tagged with the consistency class τ of their contracts (as determined by the contract classification scheme). We assume a reduction relation of form:

$$\mathsf{E}, \langle op, \tau \rangle; \sigma \parallel \Sigma \overset{\eta}{\hookrightarrow} \mathsf{E}', \sigma \parallel \Sigma$$

on the system state. The relation captures the progress of the execution (from E to E') due to the successful completion of a client operation *op* from one of the sessions in Σ, generating a new effect η. If the resultant execution E' satisfies the store contract ψ_τ (i.e., $\mathsf{E} \models \psi_\tau$), then we say that the store has *enforced* the contract ψ_τ in the execution E'. With help of the operational semantics, we now state the soundness of contract enforcement as follows:

Theorem 5.3.1 (Soundness of Contract Enforcement) *Let ψ be a well-formed contract of a replicated data type operation op, and let τ denote the consistency class of ψ as determined by the contract classification scheme. For all well-formed execution states E, E' such that $\mathsf{E}, \langle op, \tau \rangle; \sigma \parallel \Sigma \overset{\eta}{\hookrightarrow} \mathsf{E}', \sigma \parallel \Sigma$, if $\mathsf{E}' \models \psi_\tau[\eta/\hat{\eta}]$, then $\mathsf{E}' \models \psi[\eta/\hat{\eta}]$*

The theorem states that if a data store correctly enforces ψ_{sc}, ψ_{cc}, and ψ_{ec} contracts in all well-formed executions, then the same store, extended with the classification scheme shown in Figure 5.6, can enforce all well-formed Quelea contracts. The proof of the theorem is given below:

Proof. Hypothesis:

$$\begin{aligned} \mathsf{E}, \langle op, \tau \rangle; \sigma \parallel \Sigma \overset{\eta}{\hookrightarrow} \mathsf{E}', \sigma \parallel \Sigma \quad & H0 \\ \mathsf{E}' \models \psi_\tau[\eta/\hat{\eta}] \quad & H1 \end{aligned}$$

Since τ is the contract class of ψ, by inversion, we have $\psi \leq \psi_\tau$. By the definition of $\leq$ relation:

$$\Delta \vdash \forall \hat{\eta}. \psi_\tau \Rightarrow \psi \quad H2$$

Since η denotes new effect, it is a fresh variable that does not occur free in Δ. From $H2$, after instantiating bound $\hat{\eta}$ with η, we have:

$$\Delta \vdash \psi_\tau[\eta/\hat{\eta}] \Rightarrow \psi[\eta/\hat{\eta}] \quad H3$$

Due to the soundness of natural deduction for first-order logic, $H3$ implies that for all models $\mathcal{M}$ such that $\mathcal{M} \models \Delta$, if $\mathcal{M} \models \psi_\tau[\eta/\hat{\eta}]$ then $\mathcal{M} \models \psi[\eta/\hat{\eta}]$. Since E' is well-formed, we have:

$$\mathsf{E}' \models \Delta \quad H4$$

Proof follows from $H1$, $H3$, and $H4$. ∎

It is important to note that Theorem 5.3.1 does not ascribe any semantics to the reduction relation ($\rightarrow$). As such, it makes no assumptions about how the store actually implements ec, cc and sc guarantees. The specific implementation strategy is determined by the operational semantics of the store, which *defines* the reduction relation for that particular store. The following section describes operational semantics of the store used by the Quelea implementation.

5.4 Operational Semantics

We now describe operational semantics of a data store that implements strong, causal and eventual consistency guarantees. The semantics also serves as a high-level description of our implementation of the store underlying Quelea.

Figures 5.7 and 5.8 present operational semantics as rules defining the reduction relation ($\rightarrow$) over the execution state. Since we now have a concrete store, we extend our system model with Θ, a representation of the store as a map from replicas to their local states. The local state of a replica r (i.e., $\Theta(r)$) is a set of effects that are currently visible at r. An operation *op* performed at replica r can only witness the set of effects ($\Theta(r)$) visible at r. To avoid too many parenthesis in our formal presentation, we represent operation *op*, whose contract is in consistency class τ, as op_τ instead

RDT Specification Language

$$
\begin{array}{lcll}
\delta & \in & \texttt{ReplicatedDatatype} & \\
v & \in & \texttt{Value} & \\
e & \in & \texttt{Expression} & \\
op & \in & \texttt{Operation} & \\
\Lambda & \in & \texttt{OperationDefinition} & \coloneqq \quad op \mapsto e \\
\psi & \in & \mathsf{Contract} & \\
\Psi & \in & \texttt{OperationContract} & \coloneqq \quad op \mapsto \psi \\
 & & \texttt{RDTSpecification} & \coloneqq \quad (\delta, \Lambda, \Psi)
\end{array}
$$

System Model

$$
\begin{array}{lcll}
 & s \in \texttt{SessID} \quad i \in \texttt{SeqNo} & \quad r \in \texttt{ReplID} & \\
\eta & \in & \texttt{Effect} & \coloneqq \quad (s,\ i,\ op,\ v) \\
\mathsf{A} & \in & \texttt{EffSoup} & \coloneqq \quad \overline{\eta} \\
\mathsf{vis}, \mathsf{so}, \mathsf{sameobj} & \in & \texttt{Relations} & \coloneqq \quad \mathsf{A} \times \mathsf{A} \\
\mathsf{E} & \in & \texttt{ExecState} & \coloneqq \quad (\mathsf{A},\mathsf{vis},\mathsf{so},\mathsf{sameobj}) \\
\Theta & \in & \texttt{Store} & \coloneqq \quad r \mapsto \overline{\eta} \\
\tau & \in & \texttt{ConsistencyClass} & \coloneqq \quad \mathsf{ec}, \mathsf{cc}, \mathsf{sc} \\
\sigma & \in & \texttt{Session} & \coloneqq \quad \cdot \mid op_\tau :: \sigma \\
\Sigma & \in & \texttt{Session Soup} & \coloneqq \quad \langle s, i, \sigma \rangle \parallel \Sigma \mid \emptyset \\
 & & \texttt{Config} & \coloneqq \quad (\mathsf{E}, \Theta, \Sigma)
\end{array}
$$

Auxiliary Definitions

$$
\begin{array}{lcl}
\mathsf{oper}(s,\ i,\ op,\ v) & = & op \\
\mathsf{ctxt}(s,\ i,\ op,\ v) & = & (op,\ v)
\end{array}
$$

Figure 5.7. Syntax and states of operational semantics.

of the usual $\langle op, \tau \rangle$. For the sake of clarity, we only consider a single replicated object of well-defined type (for eg: a replicated object of type `BankAccount`) in our

Auxiliary Reduction $\boxed{\Theta \vdash (\mathsf{E}, \langle s, i, op\rangle) \overset{r}{\hookrightarrow} (\mathsf{E}', \eta)}$

[OPER]

$$\frac{\begin{array}{c} r \in \mathsf{dom}(\Theta) \qquad ctxt = \mathsf{ctxt}^*(\Theta(r)) \qquad \Lambda(op)(ctxt) \rightsquigarrow^* v \\ \eta = (s, i, op, v) \qquad \{\eta'\} = \mathsf{A}_{(\mathsf{SessID}=s,\, \mathsf{SeqNo}=i-1)} \\ \mathsf{A}' = \{\eta\} \cup \mathsf{A} \qquad \mathsf{vis}' = \Theta(r) \times \{\eta\} \cup \mathsf{vis} \\ \mathsf{so}' = (\mathsf{so}^{-1}(\eta') \cup \eta') \times \{\eta\} \cup \mathsf{so} \qquad \mathsf{sameobj}' = \mathsf{A}' \times \mathsf{A}' \end{array}}{\Theta \vdash ((\mathsf{A}, \mathsf{vis}, \mathsf{so}, \mathsf{sameobj}), \langle s, i, op\rangle) \overset{r}{\hookrightarrow} ((\mathsf{A}', \mathsf{vis}', \mathsf{so}', \mathsf{sameobj}'), \eta)}$$

Operational Semantics $\boxed{(\mathsf{E}, \Theta, \Sigma) \overset{\eta}{\to} (\mathsf{E}', \Theta', \Sigma')}$

[EFFVIS]

$$\frac{\begin{array}{c} \eta \in \mathsf{A} \quad \Theta' = \Theta \cup [r \mapsto \{\eta\} \cup \Theta(r)] \\ \eta \notin \Theta(r) \qquad \mathsf{E.vis}^{-1}(\eta) \cup \mathsf{E.so}^{-1}(\eta) \subseteq \Theta(r) \end{array}}{(\mathsf{E}, \Theta, \Sigma) \overset{\eta}{\to} (\mathsf{E}, \Theta', \Sigma)}$$

[EC]

$$\frac{\tau = \mathsf{EventuallyConsistent} \qquad \Theta \vdash (\mathsf{E}, \langle s, i, op\rangle) \overset{r}{\hookrightarrow} (\mathsf{E}', \eta)}{(\mathsf{E}, \Theta, \langle s, i, op_\tau :: \sigma\rangle \parallel \Sigma) \overset{\eta}{\to} (\mathsf{E}', \Theta, \langle s, i+1, \sigma\rangle \parallel \Sigma)}$$

[CC]

$$\frac{\tau = \mathsf{CausallyConsistent} \qquad \Theta \vdash (\mathsf{E}, \langle s, i, op\rangle) \overset{r}{\hookrightarrow} (\mathsf{E}', \eta) \qquad \mathsf{E}'.\mathsf{so}^{-1}(\eta) \subseteq \Theta(r)}{(\mathsf{E}, \Theta, \langle s, i, op_\tau :: \sigma\rangle \parallel \Sigma) \overset{\eta}{\to} (\mathsf{E}', \Theta, \langle s, i+1, \sigma\rangle \parallel \Sigma)}$$

[SC]

$$\frac{\begin{array}{c} \tau = \mathsf{StronglyConsistent} \qquad \Theta \vdash (\mathsf{E}, \langle s, i, op\rangle) \overset{r}{\hookrightarrow} (\mathsf{E}', \eta) \qquad \mathsf{E}.A \subseteq \Theta(r) \\ \mathsf{dom}(\Theta') = \mathsf{dom}(\Theta) \qquad \forall r' \in \mathsf{dom}(\Theta').\Theta'(r') = \Theta(r') \cup \{\eta\} \end{array}}{(\mathsf{E}, \Theta, \langle s, i, op_\tau :: \sigma\rangle \parallel \Sigma) \overset{\eta}{\to} (\mathsf{E}, \Theta', \langle s, i+1, \sigma\rangle \parallel \Sigma)}$$

Figure 5.8. Operational semantics of a replicated data store.

formalization. Our semantics are parametric over the specification of this replicated data type. Figure 5.8 formalizes replicated data type (RDT) specification as tuple (δ, Λ, Ψ), where δ is the data type, Λ maps labels (op) of operations on δ to their definitions, while Ψ maps them to their consistency contracts (ψ). The definition of an operation is expected to be a lambda expression, although we do not enforce this in our formalization. For technical reasons, we tag each session with a session identifier (s) and the sequence number (i) of the next operation in the session.

The state of an operational execution (E) is a tuple (A,vis,so,$\mathsf{sameobj}$) where A is a set of effects, and vis, so, $\mathsf{sameobj} \subseteq \mathsf{A} \times \mathsf{A}$ are *visibility*, *session order*, and *same object* relations over effects, respectively. We define an effect (η) as a tuple (s, i, op, v), which records the fact that i^{th} action in session with SessID s, which is an operation op on the replicated object, has been successfully executed on some replica yielding a return value v. Note that the combination of s and i uniquely identifies the effect. Session order relation (so) relates effects generated by the same session. An effect $\eta = (s, i, op, v)$ is said to precede another effect $\eta' = (s', i', op', v')$ in session order if and only if $s' = s$ and $i' \geq i$. Since we only consider one replicated object in our formalization, the $\mathsf{sameobj}$ relation relates every pair of effects in the effect soup (A). An effect generated at a replica becomes visible at rest of the replicas eventually. If we denote the effect generated by the operation op as η_{op}, then $\Theta(r) \times \{\eta_{op}\} \subseteq \mathsf{vis}$. Often, in our formalization, we use vis and so binary relations to obtain a set of effects visible to a given effect η, or set of effects that precede a given effect η in the session order. As a syntactic convenience, whenever R is a binary relation, we write $R(\eta)$ to denote the set of all η' such that $(\eta, \eta') \in R$. Conversely, we write $R^{-1}(\eta)$ to denote the set of all η' such that $(\eta', \eta) \in R$.

Basic guarantee provided by the store is causal visibility, which is captured by the rule [EffVis] as a condition for an effect to be visible at a replica. The rule makes an effect (η) visible at a replica r only after all the effects that causally precede η are made visible at r. It is important to note that that enforcing causal visibility does not require any inter-replica coordination. Any eventually consistent store can

provide causal visibility while being eventually consistent. Therefore, we do not lose any generality by assuming that the store provides causal visibility.

Rule [OPER] is an auxiliary reduction of the

$$\Theta \vdash (\mathsf{E}, \langle s, i, op \rangle) \overset{r}{\hookrightarrow} (\mathsf{E}', \eta)$$

Under the store configuration Θ, the rule captures the progress in execution (from E to E') due to the application of operation op to replica r resulting in a new effect η. The rule first constructs a *context* for the application from the local state ($\Theta(r)$) of the replica, by projecting[4] relevant information from effects in $\Theta(r)$. It then substitutes the definition ($\Lambda(op)$) of the operation for its label (op), and relies on the reduction relation ($\rightsquigarrow$) of the server-side language to reduce the application $\Lambda(op)(ctxt)$ to a value v'. Subsequently, the the attributes of execution state, namely A, vis, so, and $\mathsf{sameobj}$ are extended to incorporate the new effect (η).

If the operation op is $\mathsf{EventuallyConsistent}$, we simply apply the operation to any replica r. Since the store provides causal visibility, eventually consistent operations are satisfiable under any replica. If the operation is $\mathsf{CausallyConsistent}$, the operation can be applied to a replica r only if it already contains the effects of all the previous operations from the same session. This guarantee can be satisfied by applying all operations from the same session to the same logical copy of the database. If such a logical copy is untenable, then the operation might block. Since the store is assumed to converge eventually, the blocked causally consistent operation guaranteed to unblock eventually.

A $\mathsf{StronglyConsistent}$ operation expects sequential consistency. That is, universe of all effects (A) in an execution (E) must be partitionable into a set of effects that *happened before* η and another set that *happened after* η, where η is the effect generated by an strongly consistent operation. The rule [SC] enforces this sequencing in two steps; firstly, it insists that the the strongly consistent operation (op) witness effects of all operations executed so far by requiring the global set of effects A to be a subset

[4] $\mathsf{ctxt*}$ is auxiliary function ctxt extended straightforwardly to set of effects

of local state ($\Theta(r)$) of the replica (r) executing *op*. Secondly, the rule requires the effect (η) generated by *op* to be added to the local state of every other replica in the store, so that further operations on these replicas can witness the effect of *op*. Since both these steps require global coordination among replicas, the store is *unavailable* during the time it is executing *op*.

5.4.1 Soundness of Operational Semantics

We now prove a meta-theoretic property that establishes the soundness of our operational semantics in enforcing ψ_{ec}, ψ_{cc}, and ψ_{sc} consistency guarantees at every reduction step. As a corollary of this result, and Theorem 5.3.1, we have the assurance that QUELEA correctly enforces all well-formed consistency contracts.

First, we prove a useful lemma:

Lemma 5.4.1 (Auxiliary Reduction Preserves Well-Formedness) *For every execution state* E *that is well-formed, if* $\Theta \vdash (\mathsf{E}, \langle s, i, op\rangle) \overset{r}{\hookrightarrow} (\mathsf{E}', \eta)$*, then* E' *is well-formed.*

Proof. Let us denote $\mathsf{E}.\mathsf{A}$, $\mathsf{E}.\mathsf{vis}$, $\mathsf{E}.\mathsf{so}$, and $\mathsf{E}.\mathsf{sameobj}$ as A, vis, so, and $\mathsf{sameobj}$ respectively. Likewise, let us denote $\mathsf{E}'.\mathsf{A}$, $\mathsf{E}'.\mathsf{vis}$, $\mathsf{E}'.\mathsf{so}$, and $\mathsf{E}'.\mathsf{sameobj}$ as A', vis', so', and $\mathsf{sameobj}'$, respectively. By inversion on $\Theta \vdash (\mathsf{E}, \langle s, i, op\rangle) \overset{r}{\hookrightarrow} (\mathsf{E}', \eta)$, we have following hypotheses:

$$r \in \mathsf{dom}(\Theta) \qquad H0$$
$$ctxt = \mathsf{ctxt}^*(\Theta(r)) \qquad H1$$
$$\eta = (s, i, op, v) \qquad H2$$
$$\{\eta'\} = \mathsf{A}_{(\mathsf{SessID}=s,\, \mathsf{SeqNo}=i-1)} \qquad H3$$
$$\mathsf{A}' = \{\eta\} \cup \mathsf{A} \qquad H4$$
$$\mathsf{vis}' = \Theta(r) \times \{\eta\} \cup \mathsf{vis} \qquad H5$$
$$\mathsf{so}' = (\mathsf{so}^{-1}(\eta') \cup \eta') \times \{\eta\} \cup \mathsf{so} \qquad H6$$
$$\mathsf{sameobj}' = \mathsf{A}' \times \mathsf{A}' \qquad H7$$

Since E is well-formed, from the definition of well-formedness and *models* relation, we have the following:

$$\forall(a \in \mathsf{A}).\neg\mathsf{hb}(a,a) \qquad H8$$

$$\forall(a,b \in \mathsf{A}).\mathsf{vis}(a,b) \Rightarrow \mathsf{sameobj}(a,b) \qquad H9$$

$$\forall(a,b,c \in \mathsf{A}).\mathsf{so}(a,b) \ \wedge \ \mathsf{so}(b,c) \Rightarrow \mathsf{so}(a,c) \qquad H10$$

$$\forall(a \in \mathsf{A}).\mathsf{sameobj}(a,a) \qquad H11$$

$$\forall(a,b \in \mathsf{A}).\mathsf{sameobj}(a,b) \Rightarrow \mathsf{sameobj}(b,a) \qquad H12$$

$$\forall(a,b,c \in \mathsf{A}).\mathsf{so}(a,b) \ \wedge \ \mathsf{so}(b,c) \Rightarrow \mathsf{so}(a,c) \qquad H13$$

Since $\mathsf{hb} = (\mathsf{vis} \cup \mathsf{so})^+$, $H8$ is equivalent to conjunction of following assertions:

$$\forall(a \in \mathsf{A}).\neg\mathsf{vis}(a,a) \qquad H14$$

$$\forall(a \in \mathsf{A}).\neg\mathsf{so}(a,a) \qquad H15$$

$$\forall(a,b \in \mathsf{A}).\neg(\mathsf{vis}(a,b) \ \wedge \ \mathsf{so}(b,a)) \qquad H16$$

Since η is fresh, $\eta \notin \Theta(r)$. From $H4$, $H5$, and $H14$, we have the acyclicity property for vis':

$$\forall(a \in \mathsf{A}').\neg\mathsf{vis}'(a,a) \qquad H17$$

Also, $\eta \notin \mathsf{A}$, and from $H4$, $H6$ and $H15$, we have acyclicity for so':

$$\forall(a \in \mathsf{A}').\neg\mathsf{so}'(a,a) \qquad H18$$

Similarly, from the uniqueness of η, and $H5$, $H6$, and $H16$, we have the following:

$$\forall(a,b \in \mathsf{A}').\neg(\mathsf{vis}'(a,b) \ \wedge \ \mathsf{so}'(b,a)) \qquad H19$$

From $H17 - 19$, we prove the acyclicity of hb':

$$\forall(a \in \mathsf{A}').\neg\mathsf{hb}(a,a) \qquad G0$$

The $\mathsf{sameobj}'$ relation is simply the cross product $\mathsf{A}' \times \mathsf{A}'$. Hence following trivially hold:

$$\forall(a,b \in \mathsf{A}').\mathsf{vis}'(a,b) \Rightarrow \mathsf{sameobj}'(a,b) \qquad G1$$

$$\forall(a \in \mathsf{A}').\mathsf{sameobj}'(a,a) \qquad G2$$

$$\forall(a,b \in \mathsf{A}').\mathsf{sameobj}'(a,b) \Rightarrow \mathsf{sameobj}'(b,a) \qquad G3$$

Finally, from $H3$, $H4$, $H6$ and $H10$, we have transitivity for so':

$$\forall(a, b, c \in \mathsf{A}').\mathsf{so}'(a, b) \ \wedge \ \mathsf{so}'(b, c) \Rightarrow \mathsf{so}'(a, c) \quad G4$$

Well-formedness of E' follows from $G0 - 4$. ∎

We now define causal consistency property of the store formally:

Definition 5.4.1 *Given an execution* $\mathsf{E} = (\mathsf{A},\mathsf{vis},\mathsf{so},\mathsf{sameobj})$*, a store* Θ *is said to be causally consistent under an execution if and only if:*

$$\forall(r \in \mathsf{dom}(\Theta)).\forall(\eta \in \Theta(r)).$$
$$\forall(a \in \mathsf{A}).\mathsf{hbo}(a, \eta) \Rightarrow a \in \Theta(r) \quad H20$$

Where, $\mathsf{hbo} = (\mathsf{vis} \cup \mathsf{soo})^+$

The following theorem proves that our operational semantics correctly enforce ψ_{ec}, ψ_{cc}, and ψ_{sc} guarantees:

Theorem 5.4.1 (Soundness Modulo Classification) *For every well-formed execution state* E*, for every store* Θ *that is causally consistent under* E*, and for every contract class* $\tau \in \{\mathsf{ec}, \mathsf{cc}, \mathsf{sc}\}$*, if:*

$$(\mathsf{E}, \Theta, \langle s, i, op_\tau :: \sigma\rangle \parallel \Sigma) \xrightarrow{\eta} (\mathsf{E}', \Theta', \langle s, i + 1, \sigma\rangle \parallel \Sigma)$$

then (i) E' *is well-formed, and (ii)* $\mathsf{E}' \models \psi_\tau[\eta/\hat{\eta}]$

Proof. By case analysis on the derivation:

$$(\mathsf{E}, \Theta, \langle s, i, op_\tau :: \sigma\rangle \parallel \Sigma) \xrightarrow{\eta} (\mathsf{E}', \Theta', \langle s, i + 1, \sigma\rangle \parallel \Sigma)$$

Cases:

- Case [EC]: Hypotheses:

$$\tau = \mathsf{EventuallyConsistent} \quad H0$$
$$\Theta \vdash (\mathsf{E}, \langle s, i, op\rangle) \ \overset{r}{\hookrightarrow} \ (\mathsf{E}', \eta) \quad H1$$

Goal (i) follows from $H1$ and lemma 5.4.1. Goal (ii) is the following:

$$\mathsf{E}' \models \forall a, b.\ \mathsf{hbo}(a, b) \wedge \mathsf{vis}(b, \eta) \Rightarrow \mathsf{vis}(a, \eta) \quad G0$$

Let $\mathsf{A}' = \mathsf{E}'.\mathsf{A}$, $\mathsf{vis}' = \mathsf{E}'.\mathsf{vis}$, $\mathsf{so}' = \mathsf{E}'.\mathsf{so}$, and $\mathsf{hbo}' = (\mathsf{so}' \cup \mathsf{vis}')^+$. By inversion on $H1$, we get the following hypotheses:

$$\begin{aligned} \mathsf{A}' &= \mathsf{A} \cup \{\eta\} && H2 \\ \mathsf{vis}' &= \Theta(r) \times \eta \ \cup \ \mathsf{vis} && H3 \\ r &\in \mathsf{dom}(\Theta) && H4 \\ \mathsf{sameobj}' &= \mathsf{A}' \times \mathsf{A}' && H5 \end{aligned}$$

Since E' defines A' as the universe of values. Therefore, the goal can be rewritten:

$$\forall (a, b \in \mathsf{A}').E' \models [\![\mathsf{hbo}(a, b) \wedge \mathsf{vis}(b, \eta)]\!] \Rightarrow [\![\mathsf{vis}(a, \eta)]\!] \quad G1$$

New hypotheses after *intros*:

$$\begin{aligned} a &\in \mathsf{A}' \quad H6 \\ b &\in \mathsf{A}' \quad H7 \end{aligned}$$

And new goal:

$$E' \models [\![\mathsf{hbo}(a, b) \wedge \mathsf{vis}(b, \eta)]\!] \Rightarrow [\![\mathsf{vis}(a, \eta)]\!] \quad G2$$

Since $(\mathcal{M} \models A \Rightarrow B) \Leftrightarrow (\mathcal{M} \models A \Rightarrow \mathcal{M} \models B)$, we prove $G0$ by proving:

$$(\mathsf{E}' \models \ \mathsf{hbo}(a, b) \wedge \mathsf{vis}(b, \eta)) \Rightarrow (\mathsf{E}' \models \mathsf{vis}(a, \eta)) \quad G3$$

After intros:

$$\mathsf{E}' \models \ \mathsf{hbo}(a, b) \wedge \mathsf{vis}(b, \eta) \quad H8$$

Since E' defines hbo' and vis' as interpretations for hbo and vis respectively, we have:

$$\mathsf{hbo}'(a, b) \wedge \mathsf{vis}'(b, \eta) \quad H9$$

And the goal is:

$$\mathsf{vis}'(a, \eta) \quad G4$$

Inversion on $H9$:

$$\mathsf{hbo}'(a, b) \quad H10$$
$$\mathsf{vis}'(b, \eta) \quad H11$$

Since η is unique, from $H3$ and $H11$ we have the following:

$$b \in \Theta(r) \quad H12$$

Since $a, b \neq \eta$, we have that $\mathsf{hbo}'(a, b) \Rightarrow \mathsf{hbo}(a, b)$. Since Θ is causally consistent under E, using $H10$ and $H12$ we derive the following:

$$a \in \Theta(r) \quad H13$$

Now, from $H3$ and $H13$, we deduce:

$$(a, \eta) \in \mathsf{vis}'$$

which is what needs to be proven ($G4$).

- Case [CC]: Hypotheses:

$$\tau = \mathsf{CausallyConsistent} \qquad H14$$
$$\Theta \vdash (\mathsf{E}, \langle s, i, op \rangle) \overset{r}{\hookrightarrow} (\mathsf{E}', \eta) \quad H15$$
$$\mathsf{E}'.\mathsf{so}(\eta) \subseteq \Theta(r) \qquad H16$$

Goal (i) follows from $H15$ and lemma 5.4.1. We now prove Goal (ii). Expanding the definition of ψ_{cc}, goal is the following:

$$\mathsf{E}' \models \forall a.\ \mathsf{hbo}(a, \eta) \Rightarrow \mathsf{vis}(a, \eta) \quad G5$$

Let $\mathsf{A}' = \mathsf{E}'.\mathsf{A}$, $\mathsf{vis}' = \mathsf{E}'.\mathsf{vis}$, $\mathsf{so}' = \mathsf{E}'.\mathsf{so}$, and $\mathsf{hbo}' = (\mathsf{so}' \cup \mathsf{vis}')^+$. By inversion on $H15$, we get the following:

$$\mathsf{A}' = \mathsf{A} \cup \{\eta\} \qquad H17$$

$$\mathsf{vis}' = \Theta(r) \times \eta \ \cup \ \mathsf{vis} \qquad H18$$

$$\{\eta'\} = \mathsf{A}_{(\mathsf{SessID}=s,\, \mathsf{SeqNo}=i-1)}$$

$$\mathsf{so}' = (\mathsf{so}^{-1}(\eta') \cup \eta') \times \{\eta\} \cup \mathsf{so} \qquad H19$$

$$\mathsf{sameobj}' = \mathsf{A}' \times \mathsf{A}' \qquad H20$$

Expanding $\models$ followed by *intros* on $G5$ yields a context with following hypotheses:

$$a \in \mathsf{A}' \qquad H21$$

$$\mathsf{hbo}'(a, \eta) \quad H22$$

And the goal is the following:

$$\mathsf{vis}'(a, \eta) \quad G6$$

Since *happens-before* is transitive, by inversion on $H22$, we get two cases[5]:

– SCase 1: Hypotheses:

$$(\mathsf{so}' \cup \mathsf{vis}')(a, \eta) \quad H23$$

Note that we Inversion on $H23$ leads to two subcases. In one case, we assume $\mathsf{vis}'(a, \eta)$ and try to prove the goal $G6$. The proof for this case mimics the proof for Case [EC]. Alternatively, in second case, we assume:

$$\mathsf{so}'(a, \eta) \quad H24$$

and prove $G6$. From $H24$ and $H16$, we infer:

$$a \in \Theta(r) \quad H25$$

[5]Recall that we only consider single replicated object in our formalization. Accordingly, for any execution $\mathsf{E} = (\mathsf{A},\mathsf{vis},\mathsf{so},\mathsf{sameobj})$, we have $\mathsf{sameobj} = \mathsf{A} \times \mathsf{A}$. Since, $\mathsf{soo} = \mathsf{so} \cap \mathsf{sameobj}$ and $\mathsf{so} \subseteq \mathsf{A} \times \mathsf{A}$, we use so and soo interchangeably in proofs.

Now, from $H25$ and $H18$ we know:

$$(a, \eta) \in \mathsf{vis}'$$

which is the goal ($G6$). 26

– SCase 2: Hypotheses (after abbreviating the occurrence of$(\mathsf{so}' \cup \mathsf{vis}')^+$ as hbo'):

$$\exists (c \in \mathsf{A}').\mathsf{hbo}'(a, c) \wedge (\mathsf{so}' \cup \mathsf{vis}')(c, \eta) \quad H27$$

Inverting $H27$, followed by expanding $\mathsf{so}' \cup \mathsf{vis}'$:

$$c \in \mathsf{A}' \quad H28$$
$$\mathsf{hbo}'(a, c) \wedge (\mathsf{so}'(c, \eta) \vee \mathsf{vis}'(c, \eta)) \quad H29$$

Inverting the disjunction in $H29$, we get two cases:

∗ SSCase R: Hypothesis is

$$\mathsf{hbo}'(a, c) \wedge \mathsf{vis}'(c, \eta) \quad H30$$

Observe that hypothesis $H30$ and current goal ($G6$) are same as hypothesis $H9$ and goal ($G4$) in Case [EC]. The proof for this SSCase is also the same.

∗ SSCase L: Hypothesis is

$$\mathsf{hbo}'(a, c) \wedge \mathsf{so}'(c, \eta) \quad H31$$

Inverting $H31$:

$$\mathsf{hbo}'(a, c) \quad H32$$
$$\mathsf{so}'(c, \eta) \quad H33$$

From $H33$ and $H16$, we infer:

$$c \in \Theta(r) \quad H34$$

Since $a, c \neq \eta$, we know that $\mathsf{hbo}'(a,c) \Rightarrow \mathsf{hbo}(a,c)$. Since Θ is causally consistent under E, using $H32$ and $H34$ we derive the following:

$$a \in \Theta(r) \quad H35$$

Proof follows from $H35$ and $H18$.

- Case [SC]: Hypotheses:

$$\begin{aligned}
\tau &= \mathsf{StronglyConsistent} && H36\\
\Theta \vdash (\mathsf{E}, \langle s, i, op\rangle) &\overset{r}{\hookrightarrow} (\mathsf{E}', \eta) && H37\\
\mathsf{E}.A &\subseteq \Theta(r) && H38\\
\mathsf{dom}(\Theta') &= \mathsf{dom}(\Theta) && H39\\
\forall r' \in \mathsf{dom}(\Theta').\Theta'(r') &= \Theta(r') \cup \{\eta\} && H40
\end{aligned}$$

Let $\mathsf{A}' = \mathsf{E}'.\mathsf{A}$, $\mathsf{vis}' = \mathsf{E}'.\mathsf{vis}$, $\mathsf{so}' = \mathsf{E}'.\mathsf{so}$, and $\mathsf{hbo}' = (\mathsf{so}' \cup \mathsf{vis}')^+$. Inversion on $H37$ gives:

$$\begin{aligned}
\mathsf{A}' &= \mathsf{A} \cup \{\eta\} && H41\\
\mathsf{vis}' &= \Theta(r) \times \eta \ \cup \ \mathsf{vis} && H42\\
\{\eta'\} &= \mathsf{A}_{(\mathsf{SessID}=s,\, \mathsf{SeqNo}=i-1)} &&\\
\mathsf{so}' &= (\mathsf{so}^{-1}(\eta') \cup \eta') \times \{\eta\} \cup \mathsf{so} && H43\\
\mathsf{sameobj}' &= \mathsf{A}' \times \mathsf{A}' && H44
\end{aligned}$$

Goal (i) follows from $H37$ and Lemma 5.4.1. Expanding the definition of ψ_{sc}, followed by expanding $\models$ relation, and then doing *intros*, we get the following context:

$$a \in \mathsf{A}' \quad H45$$

And the goals are following:

$$\begin{aligned}
\mathsf{sameobj}'(a,\eta) \Rightarrow \mathsf{hbo}'(a,\eta) \vee \mathsf{hbo}'(\eta,a) \vee a = \eta) && G7\\
\mathsf{hbo}'(a,\eta) \Rightarrow \mathsf{vis}'(a,\eta) && G8\\
\mathsf{hbo}'(\eta,a) \Rightarrow \mathsf{vis}'(\eta,a) && G9
\end{aligned}$$

From $H41$ and $H45$, we know that either $a = \eta$ or $a \in \mathsf{A}$. When $a = \eta$:

- $G7$ follows trivially
- From lemma 5.4.1, we know that hbo' is acyclic. Hence $\mathsf{hbo}'(\eta, \eta) = false$. Therefore, $G8 - 9$ are valid vacuously.

When $a \in \mathsf{A}$:

- Intros on $G7$ gives following hypothesis:

  $$\mathsf{sameobj}'(a, \eta)$$

  From $H38$ we know that $a \in \Theta(r)$. Using $H42$, we derive:

  $$\mathsf{vis}'(a, \eta) \quad H46$$

  Introducing disjunction:

  $$(\mathsf{vis}' \cup \mathsf{so}')(a, \eta) \quad H47$$

  Now, since $\mathsf{hbo}' = (\mathsf{vis}' \cup \mathsf{so}')^+$, proof follows from last hypothesis.
- Intros on $G8$ gives following hypothesis:

  $$\mathsf{hbo}'(a, \eta)$$

  From $H38$ we know that $a \in \Theta(r)$. Using $H42$, we derive:

  $$\mathsf{vis}'(a, \eta) \quad H48$$

  Which proves $G8$.
- Intros on $G9$ gives:

  $$\mathsf{hbo}'(\eta, a) \quad H49$$

  From lemma 5.4.1, we know that E' is well-formed. Hence:

  $$\neg\mathsf{hbo}'(a, a) \quad H50$$

Since $\mathsf{sameobj}' = \mathsf{A}' \times \mathsf{A}'$, we have:

$$\mathsf{sameobj}'(a, \eta) \quad H51$$

Using previous hypothesis, we can reuse the proof for $G7$ to derive:

$$\mathsf{hbo}'(a, \eta) \quad H52$$

Since hbo' is a transitive relation, from $H49$ and $H52$, we derive:

$$\mathsf{hbo}'(a, a) \quad H53$$

$H53$ and $H50$ are contradicting hypothesis. Proof follows. ∎

We now show that every configuration of the store that is reachable via the reduction relation ($\rightarrow$) is causally consistent.

Theorem 5.4.2 (Causal Consistency Preservation) *For every well-formed execution state* E*, and a store* Θ *that is causally consistent under* E*, if:*

$$(\mathsf{E}, \Theta, \langle s, i, op_\tau :: \sigma \rangle \parallel \Sigma) \xrightarrow{\eta} (\mathsf{E}', \Theta', \langle s, i+1, \sigma \rangle \parallel \Sigma)$$

then Θ' *is causally consistent under* E'*.*

Proof. By case analysis on on:

$$(\mathsf{E}, \Theta, \langle s, i, op_\tau :: \sigma \rangle \parallel \Sigma) \xrightarrow{\eta} (\mathsf{E}', \Theta', \langle s, i+1, \sigma \rangle \parallel \Sigma)$$

Cases:

- Case [EffVis]: Final execution state is same as the initial (i.e., $\mathsf{E}' = \mathsf{E}$). Therefore, we need to prove that new store configuration Θ is causally consistent under $\mathsf{E} = (\mathsf{A},\mathsf{vis},\mathsf{so},\mathsf{sameobj})$. Hypotheses:

$$r \in \mathsf{ReplID} \quad H0$$
$$\eta \in A \quad H1$$
$$\eta \notin \Theta(r) \quad H2$$
$$\mathsf{E}.\mathsf{vis}^{-1}(\eta) \cup \mathsf{E}.\mathsf{so}^{-1}(\eta) \subseteq \Theta(r) \quad H3$$
$$\Theta' = \Theta \cup [r \mapsto \{\eta\} \cup \Theta(r)] \quad H4$$
$$\forall(r \in \mathsf{dom}(\Theta)).\forall(\eta \in \Theta(r)).\ \forall(a \in \mathsf{A}).\mathsf{hbo}(a,\eta) \Rightarrow a \in \Theta(r) \quad H5$$

From $H4$ and $H5$, it suffices to prove:

$$\forall(a \in \mathsf{A}).\mathsf{hbo}(a,\eta) \Rightarrow a \in \Theta'(r) \quad G0$$

After *intros*, hypotheses:

$$a \in \mathsf{A} \quad H6$$
$$\mathsf{hbo}(a,\eta) \quad H7$$

Goal:

$$a \in \Theta'(r) \quad G1$$

Inversion on $H7$ leads to two cases:

  - SCase *a directly precedes* η : Hypothesis:

$$(\mathsf{vis} \cup \mathsf{so})(a,\eta) \quad H8$$

    From $H3$ and $H8$, we conclude that $a \in \Theta'(r)$.

  - SCase *a transitively precedes* η: Hypothesis:

$$\exists(c \in \mathsf{A}).\mathsf{hbo}(a,c) \wedge (\mathsf{vis} \cup \mathsf{so})(c,\eta) \quad H9$$

Inverting $H9$:

$$c \in A \quad H10$$
$$\mathsf{hbo}(a, c) \quad H11$$
$$(\mathsf{vis} \cup \mathsf{so})(c, \eta) \quad H12$$

From $H3$ and $H12$, we have:

$$c \in \Theta'(r) \quad H13$$

From $H5$, $H13$ and $H11$, we conclude that $a \in \Theta'(r)$.

- Cases [EC] and [CC]: Store configuration (Θ) remains unchanged. Further, no new *happens before* order is added either among existing effects, or *from* the newly generated effect *to* existing effects. Consequently, proof is trivial.

- Case [SC]: Let $\mathsf{E} = (\mathsf{A},\mathsf{vis},\mathsf{so},\mathsf{sameobj})$ and $\mathsf{E}' = (\mathsf{A}', \mathsf{vis}', \mathsf{so}', \mathsf{sameobj}')$. Hypotheses:

$$r \in \mathsf{ReplID} \quad H14$$
$$\tau = \mathsf{StronglyConsistent}(\psi) \quad H15$$
$$\Theta \vdash (\mathsf{E}, \langle s, i, op \rangle) \overset{r}{\hookrightarrow} (\mathsf{E}', \eta) \quad H16$$
$$A \subseteq \Theta(r) \quad H17$$
$$\mathsf{dom}(\Theta') = \mathsf{dom}(\Theta) \quad H18$$
$$\forall r' \in \mathsf{dom}(\Theta').\Theta'(r') = \Theta(r') \cup \{\eta\} \quad H19$$
$$\forall (r' \in \mathsf{dom}(\Theta)).\forall (\eta' \in \Theta(r')).$$
$$\forall (a \in A).\mathsf{hbo}(a, \eta') \Rightarrow a \in \Theta(r') \quad H20$$

The goal:

$$\forall (r' \in \mathsf{dom}(\Theta')).\forall (\eta' \in \Theta'(r')).$$
$$\forall (a \in \mathsf{A}').\mathsf{hbo}'(a, \eta') \Rightarrow a \in \Theta'(r') \quad G2$$

Inverting $H16$:

$$\mathsf{A}' = \mathsf{A} \cup \{\eta\} \qquad H21$$
$$\mathsf{vis}' = \Theta(r) \times \eta \ \cup \ \mathsf{vis} \qquad H22$$
$$\{\eta'\} = \mathsf{A}_{(\mathsf{SessID}=s,\, \mathsf{SeqNo}=i-1)}$$
$$\mathsf{so}' = (\mathsf{so}^{-1}(\eta') \cup \eta') \times \{\eta\} \cup \mathsf{so} \qquad H23$$
$$\mathsf{sameobj}' = \mathsf{A}' \times \mathsf{A}' \qquad H24$$

Using $H19$ and $H20$, we can reduce the goal $(G2)$ to:

$$\forall(r' \in \mathsf{dom}(\Theta')).\forall(a \in \mathsf{A}').\mathsf{hbo}'(a, \eta) \Rightarrow a \in \Theta'(r') \quad G3$$

After *intros*, hypotheses:

$$r' \in \mathsf{dom}(\Theta') \quad H25$$
$$a \in \mathsf{A}' \qquad H26$$
$$\mathsf{hbo}'(a, \eta) \qquad H27$$

Goal:

$$a \in \Theta'(r') \quad G4$$

From $H26$ and $H21$, we know that either $a \in \mathsf{A}$ or $a = \eta$.

- If $a \in \mathsf{A}$, then from $H17$, we know that $a \in \Theta(r')$. However, from $H19$ we know that $\Theta(r') \subset \Theta'(r')$, which lets us conclude that $a \in \Theta'(r')$.
- If $a = \eta$, then $H27$ is $\mathsf{hbo}'(\eta, \eta)$. However, from lemma 5.4.1 we know that E' is well-formed, which means that hbo' is acyclic. Hence, a contradiction. Proof follows from contradiction. ∎

Corollary 5.4.1 (Soundness) *For every well-formed execution state* E*, for every store* Θ *that is causally consistent under* E*, for every contract class* $\tau \in \{\mathsf{ec}, \mathsf{cc}, \mathsf{sc}\}$*, and for every consistency contract* ψ *in the contract class* τ*, If:*

$$(\mathsf{E}, \Theta, \langle s, i, op_\tau :: \sigma\rangle \parallel \Sigma) \xrightarrow{\eta} (\mathsf{E}', \Theta', \langle s, i+1, \sigma\rangle \parallel \Sigma)$$

then (i) E' *is well-formed, (ii)* Θ' *is causally consistent under* E'*, and (iii)* $\mathsf{E}' \models \psi[\eta/\hat{\eta}]$

Proof. Follows from Theorems 5.3.1, 5.4.2, and 5.4.1. ∎

5.5 Transaction Contracts

While contracts on individual operations offer the programmer object-level declarative reasoning, real-world scenarios often involve operations that span multiple objects. In order to address this problem, several recent systems [36, 105, 106] have proposed eventually consistent transactions in order to compose operations on multiple objects. However, given that classical transaction models such as serializability and snapshot isolation require inter-replica coordination, these systems espouse *coordination-free transactions* that remain available under network partitions, but only provide weaker isolation guarantees. Coordination-free transactions have intricate consistency semantics and widely varying runtime overheads. As with operation-level consistency, the onus is on the programmer to pick the correct transaction kind. This choice is further is complicated by consistency semantics of individual operations.

5.5.1 Syntax and Semantics Extension

Quelea automates the choice of assigning the correct and most efficient transaction isolation level. Similar to contracts on individual operations, the programmer associates contracts with transactions, declaratively expressing its consistency specification. We extend the contract language with a new term under quantifier-free propositions - $\mathsf{txn}\ S_1\ S_2$, where S_1 and S_2 are sets of effects, and introduce a new primitive equivalence relation $\mathsf{sametxn}$ that holds for effects from the same transaction. $\mathsf{txn}\{a, b\}\{c, d\}$ is just syntactic sugar for $\mathsf{sametxn}(a, b) \wedge \mathsf{sametxn}(c, d) \wedge \neg\mathsf{sametxn}(a, c)$, where a and b considered to belong to the *current* transaction.

We assume that operations not part of any transaction belong to their own unique transaction. While transactions may have varying isolation guarantees, we make the standard assumption that all transactions provide atomicity. Hence, we include the following axioms in Δ:

- Same transactions is an equivalence relation:

  - $\forall a.\ \mathsf{sametxn}(a, a)$.
  - $\forall a, b.\ \mathsf{sametxn}(a, b) \Rightarrow \mathsf{sametxn}(b, a)$.
  - $\forall a, b, c.\ \mathsf{sametxn}(a, b)\ \wedge\ \mathsf{sametxn}(b, c) \Rightarrow \mathsf{sametxn}(a, c)$.

- Atomicity of transaction:
  - $\forall a, b, c.\ \mathsf{txn}\{a\}\{b, c\}\ \wedge\ \mathsf{sameobj}(b, c)\ \wedge\ \mathsf{vis}(b, a) \Rightarrow \mathsf{vis}(c, a)$.

- Transaction does not span across sessions:
  - $\forall a, b.\ \mathsf{sametxn}(a, b) \Rightarrow \mathsf{so}(a, b)\ \vee\ \mathsf{so}(b, a)\ \vee\ a = b$.

- Transactions are contiguous:
  - $\forall a, b, c.\ \mathsf{sametxn}(a, c)\ \wedge\ \mathsf{so}(a, b)\ \wedge\ \mathsf{so}(b, c) \Rightarrow \mathsf{sametxn}(a, b)$.

The semantics of the atomicity axiom is illustrated in Figure 5.9(a).

5.5.2 Transactional Bank Account

In order to illustrate the utility of declarative reasoning for transactions, consider an extension of our running example to use two accounts (objects) – current (c) and savings (s). Each account provides operations `withdraw`, `deposit` and `getBalance`, with the same contracts as defined previously. We consider two transactions – `save(amt)`, which transfers `amt` from current to savings, and `totalBalance`, which returns the sum of the balances of individual accounts. Our goal is to ensure that `totalBalance` returns the result obtained from a consistent snapshot of the object states. The QUELEA code for the transactions is given below:

```
1 save amt =
2   x ← $(classify ψsv)
3   atomically x $ do
4     b ← withdraw c amt
5     when b $ deposit s amt
```

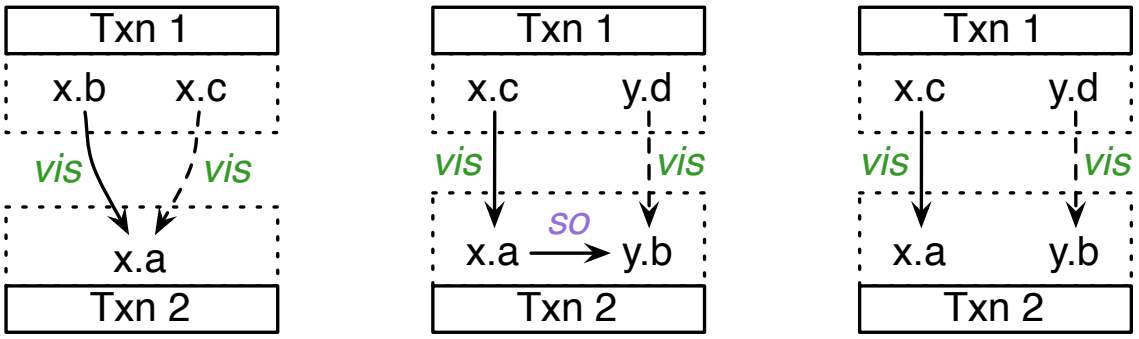
```
1 totalBalance =
2   x ← $(classify ψtb)
3   atomically x $ do
4     b1 ← getBalance c
5     b2 ← getBalance s
6     return b1 + b2
```

ψ_{sv} and ψ_{tb} are the contracts on the corresponding transactions. The function `classify` assigns the contracts *statically* to one of the transaction isolation levels offered by the store; `$()` is meta-programming syntax for splicing the result into the program. The `atomically` construct invokes the enclosing operations at the given isolation level x, ensuring that the effects of the operations are made visible atomically.

While making both transactions serializable would ensure correctness, distributed stores rarely offer serializable transactions since it is unavailable and hinders scalability [105]. As we will see, these transactions can be satisfied with much weaker isolation guarantees. Despite the atomicity offered by the transaction, anomalies are still possible. For example, the two `getBalance` operations in `totalBalance` transactions might be served by different replicas with distinct set of committed `save` transactions. If the first(second) `getBalance` operation witness a `save` transaction that is not witnessed by the second(first) `getBalance` operation, then the balance returned will be less(greater) than the actual balance. It is not immediately apparent which weakest isolation guarantee will be sufficient to prevent the anomaly.

Instead, QUELEA requires the programmer to simply state the consistency requirement as a contract. Since we would like both the `getBalance` operations to witness the same set of `save` transactions, we define the constraint on `totalBalance` transaction ψ_{tb} as:

$$\begin{aligned}\psi_{tb} = \forall a : \texttt{getBalance}, b : \texttt{getBalance},\\ (c : \texttt{withdraw} \vee \texttt{deposit}), (d : \texttt{withdraw} \vee \texttt{deposit}).\\ \mathsf{txn}\{a,b\}\{c,d\} \ \wedge \ \mathsf{vis}(c,a) \ \wedge \ \mathsf{sameobj}(d,b) \Rightarrow \mathsf{vis}(d,b)\end{aligned}$$



(a) Atomicity (b) Monotonic Atomic View (c) Repeatable Read

Figure 5.9. Semantics of transaction contracts. x and y are distinct objects. The dotted line represents the visibility requested by the contracts.

The key idea in the above definition is that the `txn` primitive allows us to relate operations on different objects. The `save` transaction only needs to ensure that the two writes it performs are made visible atomically. Since this is ensured by combining them in a transaction, `save` does not require any additional constraints, and ψ_1 is simply `true`.

5.5.3 Coordination-free Transactions

In order to illustrate the utility of transaction contract classification, we identify three popular coordination-free transaction semantics – Read Committed (RC) [107], Monotonic Atomic View (MAV) [105] and Repeatable Read (RR) [107], and illustrate the classification strategy. Our technique can indeed be applied to a different isolation level lattice.

A transaction with ANSI RC semantics only witnesses committed operations. Let us assume that the store buffers updates until all the updates from the transaction are available at a replica. If the transaction commits, the buffered updates are made visible. Otherwise, the buffered updates are discarded. RC does not entail any further

isolation guarantees. Hence, a store implementing RC does not require inter-replica coordination. We can express RC as follows:

$$\psi_{\mathsf{rc}} = \forall a, b, c.\ \mathsf{txn}\{a\}\{b, c\} \wedge \mathsf{sameobj}(b, c)\ \wedge\ \mathsf{vis}(b, a) \Rightarrow \mathsf{vis}(c, a)$$

Notice that the above definition is the same as the atomicity guarantee of transaction described in Section 5.5.1. The `save` is an example for RC transaction.

MAV semantics ensures that if some operation in a transaction T_1 witnesses the effects of another transaction T_2, then subsequent operations in T_1 will also witness the effects of T_2. MAV semantics is useful for maintaining the integrity of foreign key constraints, materialized views and secondary updates. In order to implement MAV, a store only needs to keep track of the set of transactions S_t witnessed by the running transaction, and before performing an operation at some replica, ensure that the replica includes all the transactions in S_t. Hence, MAV is coordination-free. MAV semantics is captured with the following contract:

$$\psi_{\mathsf{mav}} = \forall a, b, c, d.\ \mathsf{txn}\{a, b\}\{c, d\}\ \wedge\ \mathsf{so}(a, b)\ \wedge\ \mathsf{vis}(c, a)\ \wedge\ \mathsf{sameobj}(d, b) \Rightarrow \mathsf{vis}(d, b)$$

whose semantics is illustrated in the Figure 5.9(b).

ANSI RR semantics requires that the transaction witness a snapshot of the data store state. Importantly, this snapshot can be obtained from any replica, and hence RR is coordination-free. An example for such a transaction is the `totalBalance` transaction. The semantics of RR is captured by the following contract:

$$\psi_{\mathsf{rr}} = \forall a, b, c, d.\ \mathsf{txn}\{a, b\}\{c, d\}\ \wedge\ \mathsf{vis}(c, a)\ \wedge\ \mathsf{sameobj}(d, b) \Rightarrow \mathsf{vis}(d, b)$$

whose semantics is illustrated in the Figure 5.9(c).

5.5.4 Classification

Similar to operation-level contracts, with respect to $\leq$ relation, the coordination-free transaction semantics described here form a total order: $\psi_{\mathsf{rc}} \leq \psi_{\mathsf{mav}} \leq \psi_{\mathsf{rr}}$. The

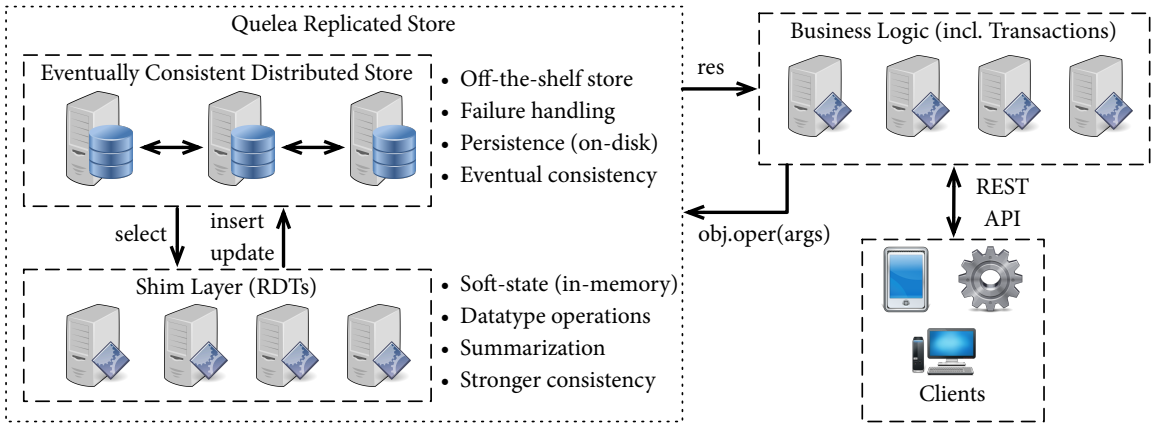


Figure 5.10. Implementation model.

transaction classification is also similar to the operation-level contract classification presented in Figure 5.6; given a contract ψ on a transaction, we start from the weakest transaction contract ψ_{rc}, and progressively compare its strength to the known transaction contracts until we find a isolation level under which ψ can be safely discharged. Otherwise, we report a type error.

5.6 Implementation

Quelea is implemented as a shallow extension of GHC Haskell and runs on top of Cassandra, an off-the-shelf eventually consistent distributed data (or backing) store responsible for all data management issues (i.e., replication, fault tolerance, availability, and convergence). Template Haskell is used implement static contract classification, and proof obligations are discharged with the help of the Z3 [108] SMT solver. Figure 5.10 illustrates the overall system architecture.

Replicated data types and the stronger consistency semantics are implemented and enforced in the *shim layer*. Our implementation supports eventual, causal, and strong consistency for data type operations, and RC, MAV, and RR semantics for transactions. This functionality is implemented entirely on top of the standard interface

exposed by Cassandra. From an engineering perspective, leveraging an off-the-shelf data store enables an implementation comprising roughly only 2500 lines of Haskell code, which is packaged as a library.

5.6.1 Shim Layer

The shim layer maintains a causally consistent in-memory snapshot of a subset of objects in the backing store, by explicitly tracking dependencies introduced between the effects due to visibility, session and same transaction relations. The dependence tracking is similar to the techniques presented in [109] and [35], with the usual optimizations making use of transitivity properties for minimizing the number of dependencies. Shim layer performs the reductions associated with replicated datatype operations corresponding to client requests. As the backing store provides durability, convergence and fault tolerance, each shim layer node simply acts as a soft-state cache, and can safely be terminated at any instant. Similarly, more shim layer nodes can be spawned on demand.

5.6.2 Operation Consistency

Every effect generated as a result of an effectful operation on an object inserts a new row (o, e, vis, txn, val) into the backing store, where o and e are object and (unique) effect identifiers, vis is the set of identifiers of effects visible to this operation, txn is an optional transaction identifier, and val is the value associated with the effect (eg: `Withdraw 50`). The shim layer periodically fetches updates from the backing store for those objects which were accessed since updates were last fetched. Since causally consistent operations require an up-to-date view of the current session, the shim layer node synchronously fetches operations if the causally preceding operations in the current session are not available in the cache. Strongly consistent operations are performed after obtaining exclusive leases on objects. The lease mechanism is im-

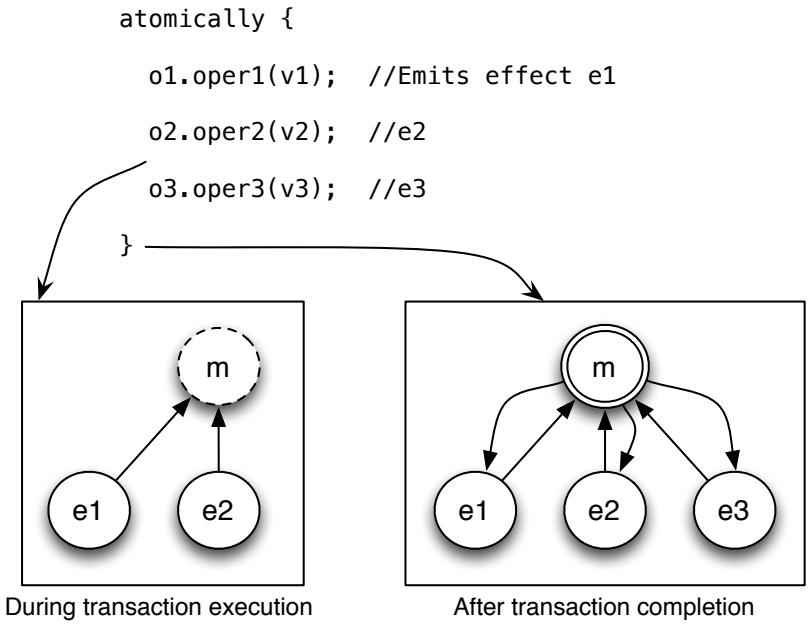


Figure 5.11. Implementing atomicity semantics. Dotted circle represents effects not yet inserted into the backing store.

plemented with the help of Cassandra's support for conditional updates and expiring columns.

5.6.3 Transactions

While Cassandra offers all-or-nothing failure semantics for multiple writes through batching, readers may witness the initial write while the batch is in progress. QUELEA implements atomic visibility by exploiting shim layer causality guarantee – an effect is included only if all the effects if depends on are also included.

Consider the example given in Figure 5.11. For every transaction in QUELEA, we instantiate a special transaction marker effect m. But importantly, do not insert into the backing store. m is included as a dependence to every effect generated in the transaction. In the figure, the graph on the left shows the state of the store in the middle of a transaction. Each circle represents an effect. The dotted circle

indicates that the effect has been instantiated, but has not yet been inserted into the store. Since the causally preceding effect m has not yet been written to the store, no operation will witness $e1$ and $e2$ while the transaction in progress. After the transaction has finished execution, we insert m into the backing store, marking all the effects from the transactions as a dependence for m. Now any replica which includes one of the effects from the transaction must include m, and transitively must include every effect from the transaction. This ensures atomicity and satisfies the RC requirement.

The above scheme prevents a transaction from witnessing its own effects. This might conflict with the causality requirement on the operations. Hence, transactions piggy-back the previous effects from the same transaction for each request. MAV semantics is implemented by keeping track of the set of transaction markers M witnessed by the transaction, and before performing an operation at some replica, ensuring that M is a subset of the transaction markers included at that replica. If not, the missing effects are synchronously fetched. RR semantics is realized by capturing a optimized snapshot of the state of some replica; each operation from an RR transaction is applied to this snapshot state. Any generated effects are added to this snapshot.

5.6.4 Summarization

The main challenge in realizing an efficient implementation of operation-based replicated data types is that the state of the object i.e., the set of effects grows with every effectful operation on the object. If left unchecked, the operations slow down over time, until the shim layer memory or backing store disk runs out of memory. Luckily, the state of the operation-based replicated data type can often be summarized to an *observably equivalent* smaller state. For example,

- A last-writer-wins register with multiple updates where v is the value of the last write is observably equivalent to a register with a single write v.

- A bank account with a series of deposits and withdraws with current balance b is equivalent to a bank account with a single deposit of b.
- A set with collection of add and remove operations is equivalent to a set with a series of add operations of live elements from the original set.

Since the semantics of summarization depends on the semantics of the data type, we expect the programmer to provide a summarization function for each RDT with the following type:

```
1 summarize :: [e] → [e]
```

with the intention that the length of the result is smaller that the length of the argument. We utilize the `summarize` function to summarize the object state both in the shim layer node and the backing store, typically when the number of effects on an object crosses a tunable threshold. Shim layer summarization is straight-forward; a summarization thread takes the local lock on the object, and replaces its state with the summarized state. The shim layer node remains unavailable for that particular object during summarization (usually a few milliseconds).

Compared to the shim layer, summarization in the backing store is more complicated. The main challenge is that unlike the shim layer, summarization cannot run as an atomic operation. Summarization in the backing store involves deleting previously inserted rows and inserting new rows, where each row corresponds to an effect. It is essential that concurrent client operations are permitted, but are not allowed to witness the intermediate state of the summarization process.

To this end, we adopt a novel summarization strategy that builds on the causality property of the store. Figure 5.12 illustrates the summarization strategy. Suppose the original set of effects on an object are $o1$, $o2$ and $o3$. When summarized, the new effects yielded are $n1$ and $n2$. We first instantiate a summarization marker s, and similar to transaction marker, we do not insert it into the store immediately. We insert the new effects $n1$ and $n2$, with strong consistency, including s as a dependence.

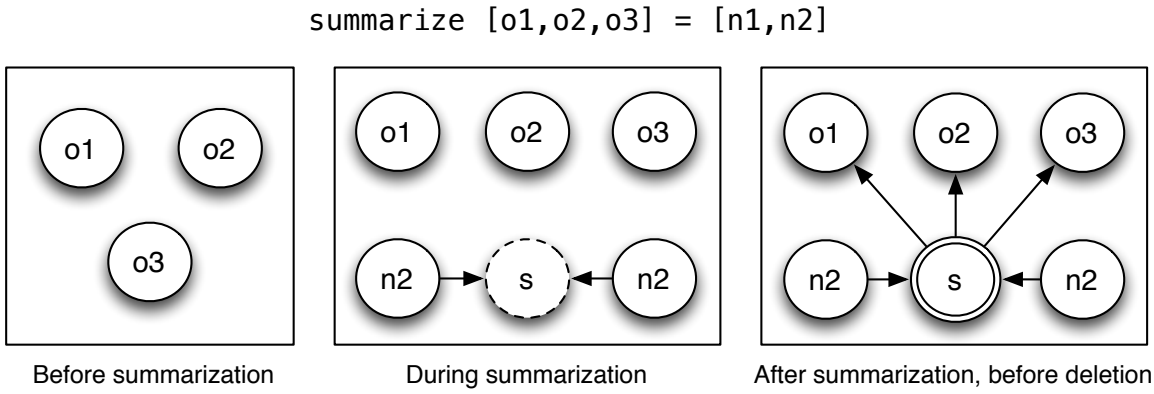


Figure 5.12. Summarization in the backing store. Dotted circle represents effects not yet inserted into the backing store.

Since s is not yet in the store, the new effects are not made visible to the clients. Then we insert s with strong consistency, including the original effects $o1$, $o2$ and $o3$ as dependence. Strongly consistent insertions ensure that a shim layer node witnessing s on some object must also witness $n1$ and $n2$ on the same object. A shim layer node which witnesses all the effects removes the original effects from its cache since they are superseded by the new effects. Finally, the old effects are deleted from the backing store. This process ensures that clients either witness the old or the new effects, but not both; the summarization process appears to be atomic from the clients perspective.

5.7 Evaluation

In this section, we evaluate Quelea programs, report their contract profile and illustrate the performance benefits of fine-grained consistency classification on operations and transactions. We also evaluate on the impact of the summarization. We implemented the following applications, which includes individual RDTs as well as larger applications composed of several RDTs:

- **LWW register**: A last-write-wins register that provides read and write operations. Each write is associated with a timestamp, which is used to resolve conflicting concurrent writes – newer write wins.
- **DynamoDB register**: A integer register that allows eventual and strong puts and gets, conditional puts, increment and decrement operations.
- **Bank account**: Our running example, with savings and current accounts.
- **Shopping list**: Collaborative shopping list which allows adding and deleting items.
- **Online store**: Models an online store with shopping cart and dynamically changing item prices. Checkout process verifies that the customer only pays the accepted price.
- **RUBiS**: An ebay-like auction site [110]. The application allows users to browse items, bid for items on sale, and pay for items from a wallet modelled after a bank account.
- **Microblog**: A twitter-like microblogging site, modelled after Twissandra [111]. The application allows adding a new user, adding and replying to tweets, following, unfollowing and blocking users, and fetching a user's timeline, userline, followers and following.

The distribution of contracts in these applications is given in Table 5.1. We see that majority of the operations and transactions are classified as eventually consistent and RC, respectively. Operation contracts are used to enforce integrity and visibility constraints on individual fields in the tables. Transactions are mainly used to consistently modify and access related fields across tables. In Quelea, the contract classification process is completely performed at compile time and has no overheads at runtime. The proof obligations associated with contract classification is discharged through the

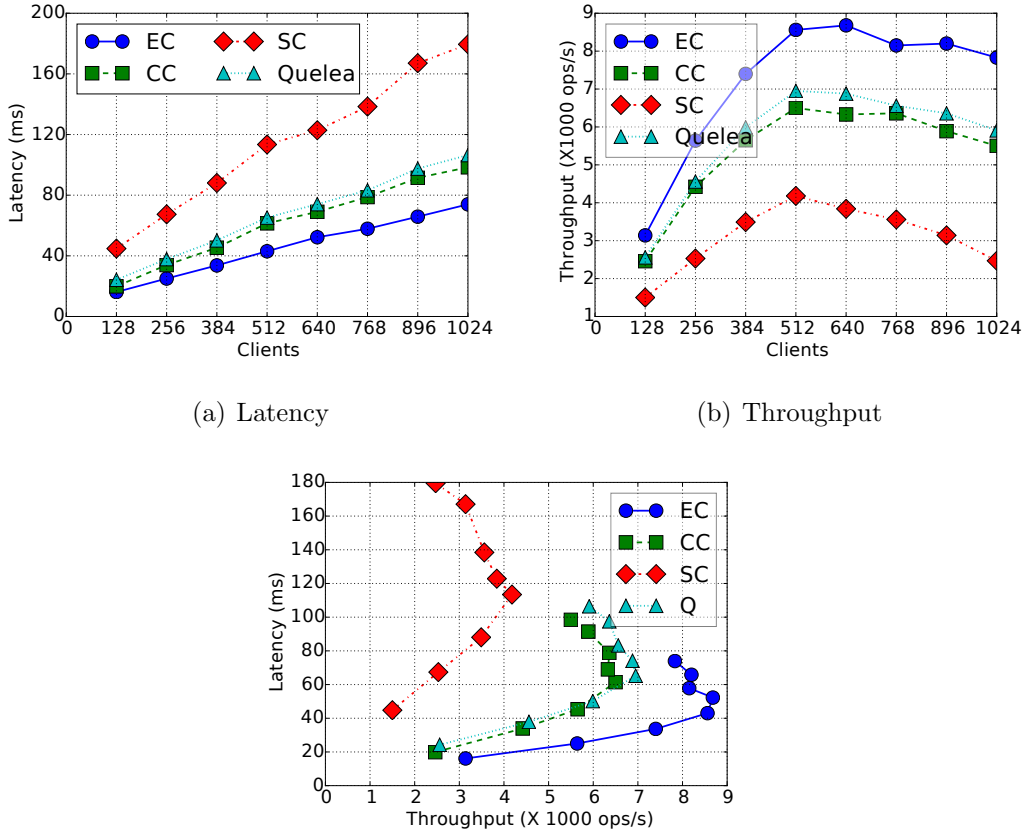
Table 5.1.
The distribution of classified contracts. #T refers to the number of tables in the application. The columns 4-6 (7-9) represent operations (transactions) assigned to this consistency (isolation) level.

Benchmark	**LOC**	**#T**	**EC**	**CC**	**SC**	**RC**	**MAV**	**RR**
LWW Reg	108	1	2	2	2	0	0	0
DynamoDB	126	1	3	1	2	0	0	0
Bank Account	155	1	1	1	1	1	0	1
Shopping List	140	1	2	1	1	0	0	0
Online store	340	4	9	1	0	2	0	1
RUBiS	640	6	14	2	1	4	2	0
Microblog	659	5	13	6	1	6	3	1

Z3 SMT Solver. Across our benchmarks, classifying a contract took 11.5 milliseconds on average.

For our performance evaluation, we deploy QUELEA applications in *clusters*, where each cluster is composed of 5 fully replicated Cassandra replicas within the same datacenter. We instantiate one shim layer node for every Cassandra replica, and place it on the same VM as the Cassandra replica. Clients are instantiated on the same data center as the store, and run transactions. We deploy the each node in the cluster on `c3.4xlarge` Amazon EC2 instances. Our shim layer nodes are multi-threaded, and we allocate 8 CPUs (out of 16 available) for each shim layer node. The clients also run on `c2.4xlarge` instances. We call this `1DC` configuration. For our geo-distributed experiments (`2DC`), we instantiate 2 clusters, each with 5 nodes, and place the clusters on US-east (Virginia) and US-west (Oregon). The average inter-region latency was 85ms.

Figure 5.13 shows the performance of operations in bank account example as we increase the number of clients in `1DC` configuration. Our client workload was generated



(a) Latency

(b) Throughput

(c) Throughput vs. Latency

Figure 5.13. Bank account performance.

using YCSB benchmark [112]. The benchmark uniformly chose from 100,000 keys, where the operation spread was 25% withdraw, 25% deposit and 50% getBalance, which corresponds to the default 50:50 read:write mix in YCSB. We increased the number of clients from 128 to 1024, and each experiment ran for 180 seconds.

The lines marked EC and CC correspond to all operations being assigned EC and CC consistency levels. These levels compromise correctness as withdraw has to be and SC operation. The line SC corresponds to a configuration where all operations are strongly consistent; this ensures application correctness. QUELEA corresponds to our implementation, which classifies operations based on their contracts. Both QUELEA and SC ensure correctness. However, with 512 clients, QUELEA implementation was

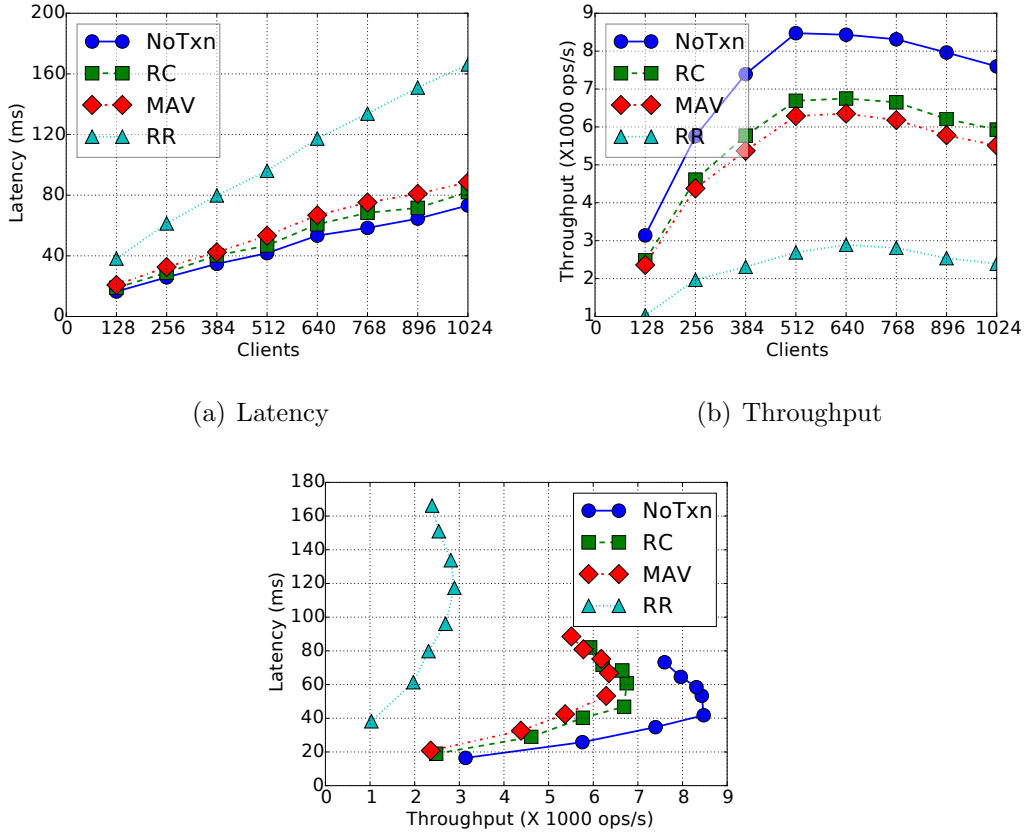
within 41% of latency and 18% of throughput of EC, whereas SC operations had 162% higher latency and 52% lower throughput than EC operations. Observe that in the Figure 5.13(c) which compares throughput vs. latency, there is a point in each case after which the latency increases while the throughput decreases. This indicates a point after which the store becomes saturated with client requests.

In `2DC` configuration (not-shown), the average latency of SC operations with 512 clients increased by 9.4× due to the cost of geo-distributed coordination, whereas Quelea operations were only 2.2× slower, mainly due to the increased cost of `withdraw` operations. Importantly, the latency of `getBalance` and `deposit` remained almost the same. This illustrates the benefit of fine-grained contract classification in Quelea.

We compare the performance of different transaction isolation level choices in Figure 5.14 using the LWW register. The numbers were obtained under 1DC configuration. The YCSB workload was modified to issue 10 operations per transaction, with the default 50:50 read:write mix. Each operation is assumed to have eventual consistency. `NoTxn` corresponds to a configuration that does not use transactions. Compared to this RC is only 12% shower in terms of latency with 512 clients, where as RR is 2.3X slower. The difference between RC and `NoTxn` is due to the meta-data overhead of recording transaction information in the object state. For RR transaction, the cost of capturing and maintaining the snapshot in an RR transaction is the biggest source of overhead.

We also compared (not shown) the performance of EC LWW operations directly against Cassandra (our backing store), which uses last-writer-wins as the only convergence semantics. While Cassandra provides no stronger-than-eventual consistency properties, Quelea was within 30%(20%) of latency(throughput) of Cassandra with 512 clients, illustrating that the programmers only have to pay a minimal overhead for the expressive and stronger Quelea programming model.

Figure 5.15 compares the Quelea implementation of RUBiS in `1DC` configuration against a single replica (NoRep) and strongly replicated (StrongRep) `1DC` deploy-



(a) Latency

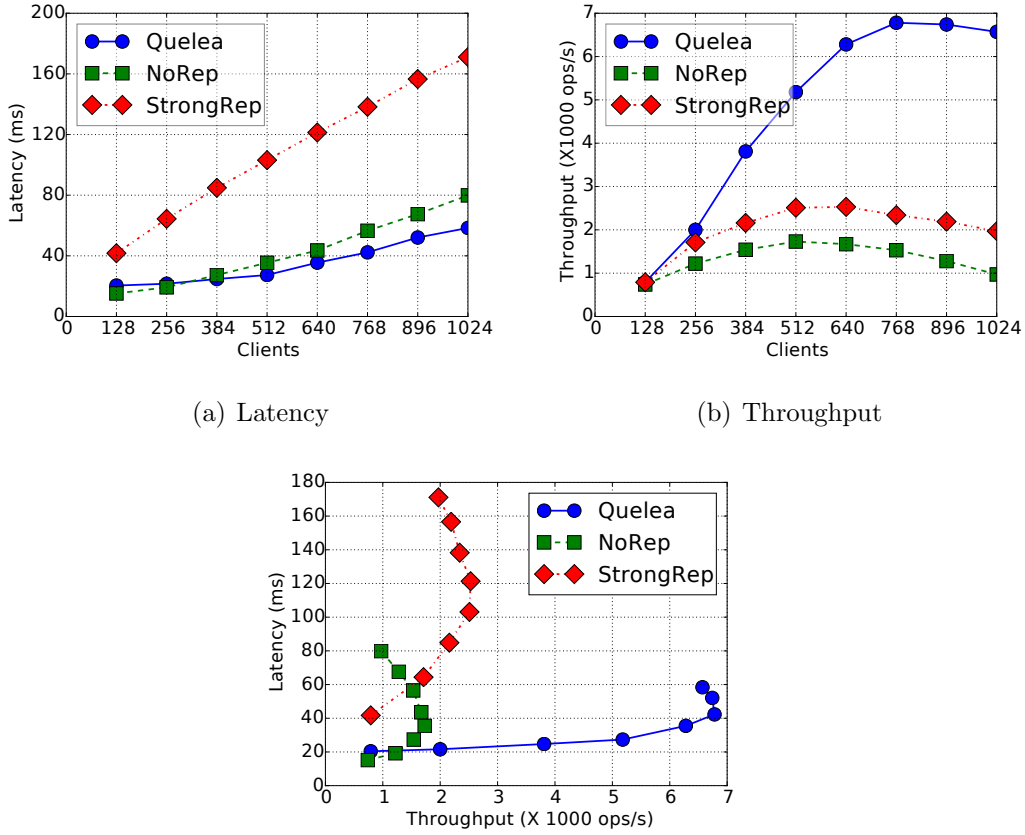
(b) Throughput

(c) Throughput vs. Latency

Figure 5.14. LWW register transaction performance.

ment. The benchmark was RUBiS bidding mix, which has 15% read-write interactions, which is representative of the auction workload. Without replication, NoRep trivially provides strong consistency. However, this deployment does not scale beyond 1750 operations per second. Strong replication offers better throughput at the cost of greater latency due to inter-replica coordination. QUELEA deployment offers the benefit of replication, while only paying the cost of coordination when necessary.

Finally, we study the impact of summarization in Figure 5.16. We utilize 128 clients and a single QUELEA replica, with all the clients operating on the *same* LWW register to stress test the summarization mechanism. The shim layer cache (mem) of operations is summarized every 64 updates, while the updates in the backing store



(a) Latency

(b) Throughput

(c) Throughput vs. Latency

Figure 5.15. Rubis bidding mix performance.

(disk) are summarized every 4096 updates. Each point in the graph represents the average latency of the previous 1000 operations. Each experiment is run for 60s.

The results show that without summarization, the average latency of operations increase exponentially to almost 1 second, and only 13K operations were performed in a minute. Since every operation has to reduce over the set of all previous operations, with a ever growing set, the operations take increasingly more time to complete. With summarization only in memory, the performance still degrades due to the cost of fetching all previous updates from the backing store into the shim layer. Fetching the latest updates is essential for SC operations. With both summarizations enabled, we see that the latency does not increase over time, and we were able to perform 67K

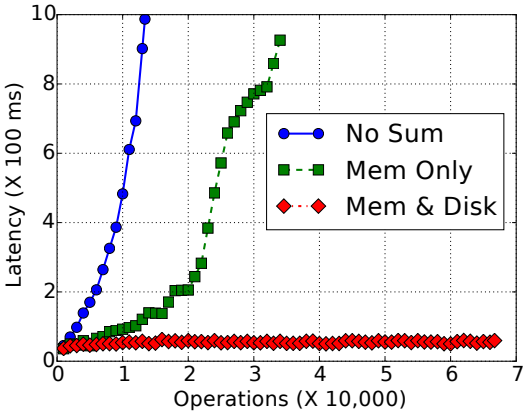


Figure 5.16. Impact of summarization.

operations. This graph illustrates the importance and effectiveness of summarization in QUELEA.

5.8 Related Work

Operation-based RDTs have been widely studied in terms of their algorithmic properties [53, 54], and several systems utilize this model to construct distributed data structures [10, 113, 114]. These systems typically propose to implement the datatypes directly over a cluster of nodes, and only focus on basic eventual consistency. Hence, these systems implement custom solutions for durability and fault-tolerance. QUELEA realizes RDTs stronger consistency models on top of off-the-shelf eventually consistent distributed stores. In this respect, QUELEA is similar to [109] where causal consistency is achieved through a shim layer on Cassandra, which explicitly tracks and enforces dependencies between updates. However, [109] does not support user-defined RDTs, automatic contract classification and transactions.

Since eventual consistency alone is insufficient to build correct applications, several systems [104, 113, 115] propose a lattice of stronger consistency levels. Similarly, traditional database processing systems [107] and their replicated variants [105] propose

weaker isolation levels for performance. In these systems, the onus is on the developer to choose the correct consistency(isolation) level for operations(transactions). QUELEA relieves the developer of this burden, and instead expects contracts expressing declarative visibility requirements.

Our contract language is inspired by the axiomatic description of RDT semantics proposed by [54]. While they use axioms for formal verification of correctness of an RDT implementation, we utilize them as a means for the user to express the desired consistency guarantees in the application. Similar to their work, our contract language does not incorporate real (i.e., wall-clock) time. Hence, it cannot describe store semantics such as recency or bounded-staleness guarantees offered by certain stores [104].

Several conditions have been proposed to judge whether an operation on a replicated data object needs coordination or not. [116] defines *logical monotonicity* as a sufficient condition for coordination freedom, and proposes a consistency analysis that marks code regions performing non-monotonic reasoning (eg: aggregations, such as `COUNT`) as potential coordination points. [117] and [118] define *invariant confluence* and *invariant safety*, respectively, as conditions for safely executing an operation without coordination. [118] also proposes a program analysis that conservatively marks operations as *blue* (coordination not required), while marking the remaining as *red* (coordination required). Unlike QUELEA, these works focus on a coarse-grained classification of consistency as eventual or strong, and do not focus on transaction isolation levels. However, program analyses they propose relieve programmers of the burden to tag operations with consistency levels. Indeed, we do consider automatic inference of consistency contracts from application-specific integrity constraints as the next step for QUELEA.

5.9 Concluding Remarks

In this chapter, we have presented Quelea a shallow Haskell extension for declarative programming over eventually consistent data stores. The key idea of Quelea is the automatic classification of fine-grained consistency contracts on operations and distributed transactions with respect to the consistency and isolation levels offered by the store. Our contract language is carefully crafted from a decidable subset of first-order logic, enabling the use of automatic theorem prover to discharge the proof obligations associated with contract classification. We realize an instantiation of Quelea on top of off-the-shelf distributed store, Cassandra, and illustrate the benefit of fine-grained contract classification by implementing and evaluating several scalable applications.

6 CONCLUDING REMARKS AND FUTURE WORK

A strongly consistent view of data, which enables the programmer to treat parallel and distributed architectures as a centralized system, is at odds with practical concerns such as availability, coherence, latency, and partial failures. Hence, modern multicore and distributed systems only provide weak consistency guarantees, belying the semblance of a centralized system, which complicates concurrent programming. In this dissertation, we presented three novel techniques for programming under weak consistency. Aneris provides a coherent and managed shared memory for programmers on the non-cache coherent Intel SCC processor. $\mathrm{R}^{\mathrm{CML}}_{\mathrm{t}}$ enables synchronous communication to be utilized as an abstraction over asynchronous distributed systems. Quelea permits declarative reasoning about consistency guarantees for programs over eventually consistent data stores.

In this chapter, we present the future work. This discussion is split based on the three contributions.

6.1 Aneris

The main hindrance to scalability of our Aneris collector is the stop-the-world nature of the shared heap collection. While shared heap collections are infrequent when compared to local collections, the pause time for shared heap collections reaches almost one second due to (1) the uncached nature of the collection and (2) the cost of synchronizing all the cores on a barrier. A natural extension to address this issue is to make the shared heap collection concurrent similar to the design by Doligez et al. [67]. In a concurrent shared heap collection, the cores no longer need to synchronize on a

global barrier, and we could envision allocating a few of the available cores specifically for concurrent shared heap collection.

Our globalization strategy lifts the *entire* transitive closure of the globalized object to the shared heap. We have observed that this strategy although simple to implement, globalizes far more memory than is actually shared between the cores. This phenomenon has also been observed by Marlow et al. [46] in their local collector design for Haskell. We can envision a strategy where only portions of the transitive closure are globalized, with further globalization on demand. In this design, we will have pointers from shared heap into local heaps (breaking the heap invariant). We can treat such pointers similar to remembered stacks (Section 3.2.4), adding the pointers into the local heap into a remembered set, so that they can be traced during local GCs. Moreover, the abundance of concurrency in our programming model can mask the latency associated with on-demand globalization, which involves cross-core communication.

Finally, while our GC design is geared for circumventing the absence of cache coherence, we get the added benefit of reduced pause times since each local heap is collected independently. However, our local heap collections are indeed optimized for throughput and optimal memory utilization. If latency is indeed the desired metric, we can envision concurrent and incremental collection for the local heaps. In particular, the independence of collection in the local heaps in Aneris allows the same execution to utilize latency sensitive GC in a collection of cores with throughput optimized GC in others. Thus, Aneris design is well-suited for mixed mode applications such as web-browsers, where both latency and throughput are important for distinct parts of the same program.

6.2 $\mathrm{R}_{\mathfrak{C}}^{\mathrm{CML}}$

While $\mathrm{R}_{\mathfrak{C}}^{\mathrm{CML}}$ provides composable synchronous reasoning for asynchronous distributed systems, the implementation does not address the challenge of partial failures in such

a setting. The key observation we make is that the dependence graph used for monitoring the correctness of speculative execution can be persisted to create a checkpoint [85] to recover from failures in a crash-restart mode. Such an implementation is especially useful in the context of long running data analytics jobs or stateful stream processing applications.

Currently, R^{CML} treats references as side-effecting operations. However, the techniques used for speculative execution can naturally be extended to references [84]. In particular, we will treat the reference write as an effect, and record the old value of the reference written as a node in the dependence graph. If the execution mis-speculates, apart from restoring the thread state with saved continuations, we will restore the state of the references as well.

The R^{CML} model can also provide an alternative strategy for enforcing application-level consistency guarantees for programs on top of eventually consistent distributed stores. Indeed, distributed stores such as Bayou [113] and Google's App Engine datastore utilize speculative execution to recover stronger consistency guarantees. Equipped with Quelea style contracts, R^{CML} can bring speculative execution for user-defined replicated data types.

6.3 Quelea

Contracts in Quelea are written by the programmer by mentally translating the application level consistency specification into visibility constraints on effects. Ideally, we would like to automatically perform the translation from database integrity constraints to contracts capturing visibility obligations. For example, one might wish to express that the balance in a replicated bank account never drops below zero, which entails the visibility constraint that withdraw operations must be totally ordered. The task would then be to discover the *weakest* contract that preserves the invariants. An attractive approach to solving this problem is to utilize counter-example guided invariant synthesis [119] to infer the contracts.

The summarization function for most data types turn out to be straight-forward. Conway et al. [120] describe a program analysis technique to analyze Bloom programs to automatically derive the garbage collection procedure for message-passing programs. It would be interesting to explore the applicability of a similar technique for deriving the summarization function for the RDTs. The combination of these techniques allow programs for eventually consistent distributed stores to be expressed in the same way as traditional database manipulating programs such as SQL or LINQ [121].

In our current work, we have utilized Cassandra as our backing store. However, Quelea itself is an abstract model and can be mapped to a variety of backends. A particularly attractive scenario is utilize Quelea to write programs on top of non-cache coherent multicore processors like the Intel SCC. Since Quelea programming model is built for eventually consistent loosely coupled setting, it can naturally express programs for architectures like SCC. In particular, each core can operate completely locally, and there is no need for the shared heap. The same Quelea program can be compiled to a variety of backends depending upon the deployment platform.

REFERENCES

REFERENCES

[1] Brad Calder, Ju Wang, Aaron Ogus, Niranjan Nilakantan, Arild Skjolsvold, Sam McKelvie, Yikang Xu, Shashwat Srivastav, Jiesheng Wu, Huseyin Simitci, Jaidev Haridas, Chakravarthy Uddaraju, Hemal Khatri, Andrew Edwards, Vaman Bedekar, Shane Mainali, Rafay Abbasi, Arpit Agarwal, Mian Fahim ul Haq, Muhammad Ikram ul Haq, Deepali Bhardwaj, Sowmya Dayanand, Anitha Adusumilli, Marvin McNett, Sriram Sankaran, Kavitha Manivannan, and Leonidas Rigas. Windows Azure Storage: A Highly Available Cloud Storage Service with Strong Consistency. In *Proceedings of the 23rd ACM Symposium on Operating Systems Principles*, SOSP '11, pages 143–157. ACM, New York, NY, USA, 2011.

[2] Amazon Elastic Compute Cloud, 2014. http://aws.amazon.com/ec2/.

[3] Shan Lu, Soyeon Park, Eunsoo Seo, and Yuanyuan Zhou. Learning from Mistakes: A Comprehensive Study on Real World Concurrency Bug Characteristics. In *Proceedings of the 13th International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS XIII, pages 329–339. ACM, New York, NY, USA, 2008.

[4] Aaron Turon. *Understanding and Expressing Scalable Concurrency*. PhD thesis, Northeastern University, Boston, Boston, MA, USA, 2013. AAI3558728.

[5] Hugh C. Lauer and Roger M. Needham. On the Duality of Operating System Structures. *SIGOPS Opererating Systetms Review*, 13(2):3–19, April 1979.

[6] Tim Harris, Simon Marlow, and Simon Peyton Jones. Haskell on a Shared-memory Multiprocessor. In *Proceedings of the 2005 ACM SIGPLAN Workshop on Haskell*, Haskell '05, pages 49–61. ACM, New York, NY, USA, 2005.

[7] John H. Reppy. *Concurrent Programming in ML*. Cambridge University Press, New York, NY, USA, 1999.

[8] Matthew Fluet, Mike Rainey, John Reppy, Adam Shaw, and Yingqi Xiao. Manticore: A Heterogeneous Parallel Language. In *Proceedings of the 2007 Workshop on Declarative Aspects of Multicore Programming*, DAMP '07, pages 37–44. ACM, New York, NY, USA, 2007.

[9] Giuseppe DeCandia, Deniz Hastorun, Madan Jampani, Gunavardhan Kakulapati, Avinash Lakshman, Alex Pilchin, Swaminathan Sivasubramanian, Peter Vosshall, and Werner Vogels. Dynamo: Amazon's Highly Available Key-value Store. In *Proceedings of 21st ACM SIGOPS Symposium on Operating Systems Principles*, SOSP '07, pages 205–220. ACM, New York, NY, USA, 2007.

[10] Avinash Lakshman and Prashant Malik. Cassandra: A Decentralized Structured Storage System. *SIGOPS Operating Systems Review*, 44(2):35–40, April 2010.

[11] Basho Riak, 2014. http://basho.com/riak/.

[12] Milo M. K. Martin, Mark D. Hill, and Daniel J. Sorin. Why On-chip Cache Coherence is Here to Stay. *Communications of the ACM*, 55(7):78–89, July 2012.

[13] Xavier Défago, André Schiper, and Péter Urbán. Total Order Broadcast and Multicast Algorithms: Taxonomy and Survey. *ACM Computing Surveys*, 36(4):372–421, December 2004.

[14] Leslie Lamport. The Part-time Parliament. *ACM Transactions on Computer Systems*, 16(2):133–169, May 1998.

[15] Irving L. Traiger, Jim Gray, Cesare A. Galtieri, and Bruce G. Lindsay. Transactions and Consistency in Distributed Database Systems. *ACM Transactions on Database Systems*, 7(3):323–342, September 1982.

[16] Mike Burrows. The Chubby Lock Service for Loosely-coupled Distributed Systems. In *Proceedings of the 7th Symposium on Operating Systems Design and Implementation*, OSDI '06, pages 335–350. USENIX Association, Berkeley, CA, USA, 2006.

[17] L. Lamport. How to Make a Multiprocessor Computer That Correctly Executes Multiprocess Programs. *IEEE Transactions on Computers*, 28(9):690–691, September 1979.

[18] Maurice P. Herlihy and Jeannette M. Wing. Linearizability: A Correctness Condition for Concurrent Objects. *ACM Transactions on Programming Languages and Systems*, 12(3):463–492, July 1990.

[19] Christos H. Papadimitriou. The Serializability of Concurrent Database Updates. *Journal of the ACM*, 26(4):631–653, October 1979.

[20] Delphine Demange, Vincent Laporte, Lei Zhao, Suresh Jagannathan, David Pichardie, and Jan Vitek. Plan B: A Buffered Memory Model for Java. In *Proceedings of the 40th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '13, pages 329–342. ACM, New York, NY, USA, 2013.

[21] Sewell, Peter and Sarkar, Susmit and Owens, Scott and Nardelli, Francesco Zappa and Myreen, Magnus O. X86-TSO: A Rigorous and Usable Programmer's Model for x86 Multiprocessors. *Communications of the ACM*, 53(7):89–97, July 2010.

[22] Susmit Sarkar, Peter Sewell, Jade Alglave, Luc Maranget, and Derek Williams. Understanding POWER Multiprocessors. In *Proceedings of the 32nd ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '11, pages 175–186. ACM, New York, NY, USA, 2011.

[23] Mark Batty, Scott Owens, Susmit Sarkar, Peter Sewell, and Tjark Weber. Mathematizing C++ Concurrency. In *Proceedings of the 38th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '11, pages 55–66. ACM, New York, NY, USA, 2011.

[24] Stamatis G. Kavadias, Manolis G.H. Katevenis, Michail Zampetakis, and Dimitrios S. Nikolopoulos. On-chip Communication and Synchronization Mechanisms with Cache-integrated Network Interfaces. In *Proceedings of the 7th ACM International Conference on Computing Frontiers*, CF '10, pages 217–226. ACM, New York, NY, USA, 2010.

[25] David Luebke, Mark Harris, Jens Krüger, Tim Purcell, Naga Govindaraju, Ian Buck, Cliff Woolley, and Aaron Lefohn. GPGPU: General Purpose Computation on Graphics Hardware. In *ACM SIGGRAPH 2004 Course Notes*, SIGGRAPH '04. ACM, New York, NY, USA, 2004.

[26] Timothy G. Mattson, Michael Riepen, Thomas Lehnig, Paul Brett, Werner Haas, Patrick Kennedy, Jason Howard, Sriram Vangal, Nitin Borkar, Greg Ruhl, and Saurabh Dighe. The 48-core SCC Processor: The Programmer's View. In *Proceedings of the 2010 ACM/IEEE International Conference for High Performance Computing, Networking, Storage and Analysis*, SC '10, pages 1–11. IEEE Computer Society, Washington, DC, USA, 2010.

[27] Jim Kahle. The Cell Processor Architecture. In *Proceedings of the 38th Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO 38, pages 3–3. IEEE Computer Society, Washington, DC, USA, 2005.

[28] Nicholas P. Carter, Aditya Agrawal, Shekhar Borkar, Romain Cledat, Howard David, Dave Dunning, Joshua Fryman, Ivan Ganev, Roger A. Golliver, Rob Knauerhase, Richard Lethin, Benoit Meister, Asit K. Mishra, Wilfred R. Pinfold, Justin Teller, Josep Torrellas, Nicolas Vasilache, Ganesh Venkatesh, and Jianping Xu. Runnemede: An Architecture for Ubiquitous High-Performance Computing. In *Proceedings of the 2013 IEEE 19th International Symposium on High Performance Computer Architecture (HPCA)*, HPCA '13, pages 198–209. IEEE Computer Society, Washington, DC, USA, 2013.

[29] Eric Brewer. Towards Robust Distributed Systems (Invited Talk), 2000.

[30] Eric Brewer. CAP Twelve Years Later: How the "Rules" Have Changed. *IEEE Computer*, 45(2):23–29, February 2012.

[31] Seth Gilbert and Nancy Lynch. Brewer's Conjecture and the Feasibility of Consistent, Available, Partition-tolerant Web Services. *SIGACT News*, 33(2):51–59, June 2002.

[32] D. B. Terry, M. M. Theimer, Karin Petersen, A. J. Demers, M. J. Spreitzer, and C. H. Hauser. Managing Update Conflicts in Bayou, a Weakly Connected Replicated Storage System. In *Proceedings of the 15th ACM Symposium on Operating Systems Principles*, SOSP '95, pages 172–182. ACM, New York, NY, USA, 1995.

[33] Peter Bailis and Ali Ghodsi. Eventual Consistency Today: Limitations, Extensions, and Beyond. *Queue*, 11(3):20:20–20:32, March 2013.

[34] Wyatt Lloyd, Michael J. Freedman, Michael Kaminsky, and David G. Andersen. Don't Settle for Eventual: Scalable Causal Consistency for Wide-area Storage with COPS. In *Proceedings of the 23rd ACM Symposium on Operating Systems Principles*, SOSP '11, pages 401–416. ACM, New York, NY, USA, 2011.

[35] Wyatt Lloyd, Michael J. Freedman, Michael Kaminsky, and David G. Andersen. Stronger Semantics for Low-latency Geo-replicated Storage. In *Proceedings of the 10th USENIX Conference on Networked Systems Design and Implementation*, NSDI'13, pages 313–328. USENIX Association, Berkeley, CA, USA, 2013.

[36] Yair Sovran, Russell Power, Marcos K. Aguilera, and Jinyang Li. Transactional Storage for Geo-replicated Systems. In *Proceedings of the 23rd ACM Symposium on Operating Systems Principles*, SOSP '11, pages 385–400. ACM, New York, NY, USA, 2011.

[37] Cheng Li, Daniel Porto, Allen Clement, Johannes Gehrke, Nuno Preguiça, and Rodrigo Rodrigues. Making Geo-replicated Systems Fast As Possible, Consistent when Necessary. In *Proceedings of the 10th USENIX Conference on Operating Systems Design and Implementation*, OSDI'12, pages 265–278. USENIX Association, Berkeley, CA, USA, 2012.

[38] Peter Bailis, Aaron Davidson, Alan Fekete, Ali Ghodsi, Joseph M. Hellerstein, and Ion Stoica. Highly available transactions: Virtues and limitations. *PVLDB*, 7(3):181–192, 2013.

[39] KC Sivaramakrishnan, Lukasz Ziarek, and Suresh Jagannathan. MultiMLton: A Multicore-aware Runtime for Standard ML. *Journal of Functional Programming*, 2014.

[40] Kevin O'Brien, Kathryn O'Brien, Zehra Sura, Tong Chen, and Tao Zhang. Supporting OpenMP on Cell. *International Journal of Parallel Programming*, 36(3):289–311, June 2008.

[41] Jaejin Lee, Sangmin Seo, Chihun Kim, Junghyun Kim, Posung Chun, Zehra Sura, Jungwon Kim, and SangYong Han. COMIC: A Coherent Shared Memory Interface for Cell Be. In *Proceedings of the 17th International Conference on Parallel Architectures and Compilation Techniques*, PACT '08, pages 303–314. ACM, New York, NY, USA, 2008.

[42] Mike Houston, Ji-Young Park, Manman Ren, Timothy Knight, Kayvon Fatahalian, Alex Aiken, William Dally, and Pat Hanrahan. A Portable Runtime Interface for Multi-level Memory Hierarchies. In *Proceedings of the 13th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPoPP '08, pages 143–152. ACM, New York, NY, USA, 2008.

[43] Pieter Bellens, Josep M. Perez, Rosa M. Badia, and Jesus Labarta. CellSs: A Programming Model for the Cell BE Architecture. In *Proceedings of the 2006 ACM/IEEE Conference on Supercomputing*, SC '06. ACM, New York, NY, USA, 2006.

[44] Keith Chapman, Ahmed Hussein, and Antony L. Hosking. X10 on the Single-chip Cloud Computer: Porting and Preliminary Performance. In *Proceedings of the 2011 ACM SIGPLAN X10 Workshop*, X10 '11, pages 7:1–7:8. ACM, New York, NY, USA, 2011.

[45] Stefan Lankes, Pablo Reble, Oliver Sinnen, and Carsten Clauss. Revisiting Shared Virtual Memory Systems for Non-coherent Memory-coupled Cores. In *Proceedings of the 2012 International Workshop on Programming Models and Applications for Multicores and Manycores*, PMAM '12, pages 45–54. ACM, New York, NY, USA, 2012.

[46] Simon Marlow and Simon Peyton Jones. Multicore Garbage Collection with Local Heaps. In *Proceedings of the 2011 International Symposium on Memory Management*, ISMM '11, pages 21–32. ACM, New York, NY, USA, 2011.

[47] Sven Auhagen, Lars Bergstrom, Matthew Fluet, and John Reppy. Garbage Collection for Multicore NUMA Machines. In *Proceedings of the 2011 ACM SIGPLAN Workshop on Memory Systems Performance and Correctness*, MSPC '11, pages 51–57. ACM, New York, NY, USA, 2011.

[48] Todd A. Anderson. Optimizations in a Private Nursery-based Garbage Collector. In *Proceedings of the 2010 International Symposium on Memory Management*, ISMM '10, pages 21–30. ACM, New York, NY, USA, 2010.

[49] KC Sivaramakrishnan, Lukasz Ziarek, and Suresh Jagannathan. Eliminating Read Barriers through Procrastination and Cleanliness. In *Proceedings of the 2012 International Symposium on Memory Management*, ISMM '12, pages 49–60. ACM, New York, NY, USA, 2012.

[50] KC Sivaramakrishnan, Lukasz Ziarek, and Suresh Jagannathan. A Coherent and Managed Runtime for ML on the SCC. In *Many-core Applications Research Community (MARC) Symposium at RWTH Aachen University, November 29th-30th 2012, Aachen, Germany*, pages 20–35, 2012.

[51] KC Sivaramakrishnan, Lukasz Ziarek, and Suresh Jagannathan. RxCML: A Prescription for Safely Relaxing Synchrony. In *Proceedings of the 16th International Symposium on Practical Aspects of Declarative Languages*, PADL '14, pages 1–16. Springer-Verlag, Berlin, Heidelberg, 2014.

[52] Swaminathan Sivasubramanian. Amazon DynamoDB: A Seamlessly Scalable Non-relational Database Service. In *Proceedings of the 2012 ACM SIGMOD International Conference on Management of Data*, SIGMOD '12, pages 729–730. ACM, New York, NY, USA, 2012.

[53] Marc Shapiro, Nuno Preguia, Carlos Baquero, and Marek Zawirski. Conflict-Free Replicated Data Types. In *Stabilization, Safety, and Security of Distributed Systems*, volume 6976 of *Lecture Notes in Computer Science*, pages 386–400. Springer-Verlarg, Berlin, Heidelberg, 2011.

[54] Sebastian Burckhardt, Alexey Gotsman, Hongseok Yang, and Marek Zawirski. Replicated Data Types: Specification, Verification, Optimality. In *Proceedings of the 41st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '14, pages 271–284. ACM, New York, NY, USA, 2014.

[55] The MLton Compiler and Runtime System, 2012. http://www.mlton.org.

[56] Lukasz Ziarek, KC Sivaramakrishnan, and Suresh Jagannathan. Composable Asynchronous Events. In *Proceedings of the 32nd ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '11, pages 628–639. ACM, New York, NY, USA, 2011.

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

[58] Martin Elsman. *Program Modules, Separate Compilation, and Intermodule Optimisation*. PhD thesis, University of Copenhagen, 1999.

[59] John C. Reynolds. Definitional Interpreters for Higher-order Programming Languages. In *Proceedings of the ACM Annual Conference - Volume 2*, ACM '72, pages 717–740. ACM, New York, NY, USA, 1972.

[60] Gianfranco Bilardi and Keshav Pingali. Algorithms for Computing the Static Single Assignment Form. *Journal of the ACM*, 50(3):375–425, May 2003.

[61] Carl Bruggeman, Oscar Waddell, and R. Kent Dybvig. Representing Control in the Presence of One-shot Continuations. In *Proceedings of the ACM SIGPLAN 1996 Conference on Programming Language Design and Implementation*, PLDI '96, pages 99–107. ACM, New York, NY, USA, 1996.

[62] Mitchell Wand. Continuation-based Multiprocessing. In *Proceedings of the 1980 ACM Conference on LISP and Functional Programming*, LFP '80, pages 19–28. ACM, New York, NY, USA, 1980.

[63] Glasgow Haskell Compiler, 2014. http://www.haskell.org/ghc.

[64] Patrick M. Sansom. Dual-Mode Garbage Collection. In *Proceedings of the Workshop on the Parallel Implementation of Functional Languages*, pages 283–310. Springer-Verlag, Berlin, Heidelberg, 1991.

[65] A. W. Appel. Simple Generational Garbage Collection and Fast Allocation. *Software: Practice and Experience*, 19:171–183, February 1989.

[66] Guy L. Steele, Jr. Multiprocessing Compactifying Garbage Collection. *Communications of the ACM*, 18(9):495–508, September 1975.

[67] Damien Doligez and Xavier Leroy. A Concurrent, Generational Garbage Collector For A Multithreaded Implementation of ML. In *Proceedings of the 20th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '93, pages 113–123. ACM, New York, NY, USA, 1993.

[68] Bjarne Steensgaard. Thread-Specific Heaps for Multi-Threaded Programs. In *Proceedings of the 2000 International Symposium on Memory Management*, ISMM '00, pages 18–24. ACM, New York, NY, USA, 2000.

[69] Software-Managed Cache Coherence for SCC Revision 1.5, 2012. https://communities.intel.com/message/175008.

[70] Isaías A. Comprés Ureña, Michael Riepen, and Michael Konow. RCKMPI – Lightweight MPI Implementation for Intel's Single-chip Cloud Computer (SCC). In *Proceedings of the 18th European MPI Users' Group Conference on Recent Advances in the Message Passing Interface*, EuroMPI'11, pages 208–217. Springer-Verlag, Berlin, Heidelberg, 2011.

[71] Stephen M. Blackburn and Antony L. Hosking. Barriers: Friend or Foe? In *Proceedings of the 4th International Symposium on Memory Management*, ISMM '04, pages 143–151. ACM, New York, NY, USA, 2004.

[72] Henry G. Baker, Jr. List Processing in Real Time on a Serial Computer. *Communications of the ACM*, 21:280–294, April 1978.

[73] Rodney A. Brooks. Trading Data Space For Reduced Time and Code Space in Real-time Garbage Collection on Stock Hardware. In *Proceedings of the 1984 ACM Symposium on LISP and Functional Programming*, LFP '84, pages 256–262. ACM, New York, NY, USA, 1984.

[74] David F. Bacon, Perry Cheng, and V. T. Rajan. A Real-time Garbage Collector with Low Overhead and Consistent Utilization. In *Proceedings of the 30th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '03, pages 285–298. ACM, New York, NY, USA, 2003.

[75] P.H. Hartel, M. Feeley, M. Alt, and L. Augustsson. Benchmarking Implementations of Functional Languages with "Pseudoknot", a Float-Intensive Benchmark. *Journal of Functional Programming*, 6(4):621–655, 1996. http://doc.utwente.nl/55704/.

[76] Simon Peter, Adrian Schüpbach, Dominik Menzi, and Timothy Roscoe. Early Experience with the Barrelfish OS and the Single-chip Cloud Computer. In *MARC Symposium*, pages 35–39, 2011.

[77] Darko Petrović, Omid Shahmirzadi, Thomas Ropars, and André Schiper. Asynchronous Broadcast on the Intel SCC using Interrupts. In *MARC Symposium*, pages 24–29, 2012.

[78] Richard Jones and Andy C. King. A Fast Analysis for Thread-Local Garbage Collection with Dynamic Class Loading. In *Proceedings of the 5th IEEE International Workshop on Source Code Analysis and Manipulation*, pages 129–138. IEEE Computer Society, Washington, DC, USA, 2005.

[79] Thomas Prescher, Randolf Rotta and Jörg Nolte. Flexible Sharing and Replication Mechanisms for Hybrid Memory Architectures. In *MARC Symposium*, 2011.

[80] Thomas Rauber Andreas Prell. Go's Concurrency Constructs on the SCC. In *MARC Symposium*, pages 2–6, 2012.

[81] J.H. Reppy. *Concurrent Programming in ML*. Cambridge University Press, New York, NY, USA, 2007.

[82] Kenneth P. Birman and Thomas A. Joseph. Reliable Communication in the Presence of Failures. *ACM Transactions on Computer Systems*, 5(1):47–76, 1987.

[83] ZeroMQ: The Intelligent Transport Layer, 2013. http://www.zeromq.org.

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

[85] E. N. (Mootaz) Elnozahy, Lorenzo Alvisi, Yi-Min Wang, and David B. Johnson. A Survey of Rollback-recovery Protocols in Message-passing Systems. *ACM Computing Surveys*, 34(3):375–408, September 2002.

[86] Chi Cao Minh, JaeWoong Chung, Christos Kozyrakis, and Kunle Olukotun. STAMP: Stanford Transactional Applications for Multi-Processing. In *Proceedings of the IEEE International Symposium on Workload Characterization*, IISWC'08, 2008.

[87] Maher Suleiman, Michèle Cart, and Jean Ferrié. Serialization of Concurrent Operations in a Distributed Collaborative Environment. In *Proceedings of the International ACM SIGGROUP Conference on Supporting Group Work: The Integration Challenge*, GROUP '97, pages 435–445. ACM, New York, NY, USA, 1997.

[88] David A. Nichols, Pavel Curtis, Michael Dixon, and John Lamping. High-latency, Low-bandwidth Windowing in the Jupiter Collaboration System. In *Proceedings of the 8th Annual ACM Symposium on User Interface and Software Technology*, UIST '95, pages 111–120. ACM, New York, NY, USA, 1995.

[89] David Wang, Alex Mah, and Soren Lassen. Google Wave: Operational Transformation, 2010. http://www.waveprotocol.org/whitepapers/operational-transform.

[90] Stéphane Martin, Mehdi Ahmed-Nacer, and Pascal Urso. Controlled Conflict Resolution for Replicated Documents. In *CollaborateCom*, pages 471–480, 2012.

[91] Bernadette Charron-Bost, Friedemann Mattern, and Gerard Tel. Synchronous, Asynchronous, and Causally Ordered Communication. *Distributed Computing*, 9(4):173–191, February 1996.

[92] Edmund B. Nightingale, Peter M. Chen, and Jason Flinn. Speculative Execution in a Distributed File System. In *Proceedings of the 20th ACM Symposium on Operating Systems Principles*, SOSP '05, pages 191–205. ACM, New York, NY, USA, 2005.

[93] Brendan Cully, Geoffrey Lefebvre, Dutch Meyer, Mike Feeley, Norm Hutchinson, and Andrew Warfield. Remus: High Availability via Asynchronous Virtual Machine Replication. In *Proceedings of the 5th USENIX Symposium on Networked Systems Design and Implementation*, NSDI'08, pages 161–174. USENIX Association, Berkeley, CA, USA, 2008.

[94] Benjamin Wester, James Cowling, Edmund B. Nightingale, Peter M. Chen, Jason Flinn, and Barbara Liskov. Tolerating Latency in Replicated State Machines Through Client Speculation. In *Proceedings of the 6th USENIX Symposium on Networked Systems Design and Implementation*, NSDI'09, pages 245–260. USENIX Association, Berkeley, CA, USA, 2009.

[95] Tong Li, Carla S. Ellis, Alvin R. Lebeck, and Daniel J. Sorin. Pulse: A Dynamic Deadlock Detection Mechanism Using Speculative Execution. In *Proceedings of the Annual Conference on USENIX Annual Technical Conference*, ATEC '05, pages 3–3. USENIX Association, Berkeley, CA, USA, 2005.

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

[97] Ken Wakita, Takashi Asano, and Masataka Sassa. D'Caml: Native Support for Distributed ML Programming in Heterogeneous Environment. In *Proceedings of the 5th International Euro-Par Conference on Parallel Processing*, Euro-Par '99, pages 914–924. Springer-Verlag, London, UK, UK, 1999.

[98] Peter Sewell, James J. Leifer, Keith Wansbrough, Francesco Zappa Nardelli, Mair Allen-Williams, Pierre Habouzit, and Viktor Vafeiadis. Acute: High-level Programming Language Design for Distributed Computation. *Journal of Functional Programming*, 17(4-5):547–612, July 2007.

[99] Jeff Epstein, Andrew P. Black, and Simon Peyton-Jones. Towards Haskell in the Cloud. In *Proceedings of the 4th ACM Symposium on Haskell*, Haskell '11, pages 118–129. ACM, New York, NY, USA, 2011.

[100] Hans-J. Boehm and Sarita V. Adve. Foundations of the C++ Concurrency Memory Model. In *Proceedings of the 2008 ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '08, pages 68–78. ACM, New York, NY, USA, 2008.

[101] Kevin Donnelly and Matthew Fluet. Transactional Events. In *Proceedings of the 11th ACM SIGPLAN International Conference on Functional Programming*, ICFP '06, pages 124–135. ACM, New York, NY, USA, 2006.

[102] Laura Effinger-Dean, Matthew Kehrt, and Dan Grossman. Transactional Events for ML. In *Proceedings of the 13th ACM SIGPLAN International Conference on Functional Programming*, ICFP '08, pages 103–114. ACM, New York, NY, USA, 2008.

[103] Mohsen Lesani and Jens Palsberg. Communicating Memory Transactions. In *Proceedings of the 16th ACM Symposium on Principles and Practice of Parallel Programming*, PPoPP '11, pages 157–168. ACM, New York, NY, USA, 2011.

[104] Douglas B. Terry, Vijayan Prabhakaran, Ramakrishna Kotla, Mahesh Balakrishnan, Marcos K. Aguilera, and Hussam Abu-Libdeh. Consistency-based Service Level Agreements for Cloud Storage. In *Proceedings of the 24th ACM Symposium on Operating Systems Principles*, SOSP '13, pages 309–324. ACM, New York, NY, USA, 2013.

[105] Peter Bailis, Aaron Davidson, Alan Fekete, Ali Ghodsi, Joseph M. Hellerstein, and Ion Stoica. Highly Available Transactions: Virtues and Limitations. *PVLDB*, 7(3):181–192, 2013.

[106] Sebastian Burckhardt, Daan Leijen, Manuel Fähndrich, and Mooly Sagiv. Eventually Consistent Transactions. In *Proceedings of the 21st European Conference on Programming Languages and Systems*, ESOP'12, pages 67–86. Springer-Verlag, Berlin, Heidelberg, 2012.

[107] Hal Berenson, Phil Bernstein, Jim Gray, Jim Melton, Elizabeth O'Neil, and Patrick O'Neil. A Critique of ANSI SQL Isolation Levels. In *Proceedings of the 1995 ACM SIGMOD International Conference on Management of Data*, SIGMOD '95, pages 1–10. ACM, New York, NY, USA, 1995.

[108] Z3: High-performance Theorem Prover, 2014. http://z3.codeplex.com/.

[109] Peter Bailis, Ali Ghodsi, Joseph M. Hellerstein, and Ion Stoica. Bolt-on Causal Consistency. In *Proceedings of the 2013 ACM SIGMOD International Conference on Management of Data*, SIGMOD '13, pages 761–772. ACM, New York, NY, USA, 2013.

[110] RUBiS: Rice University Bidding System, 2014. http://rubis.ow2.org/.

[111] Twissandra: Twitter clone on Cassandra, 2014. http://twissandra.com/.

[112] Brian F. Cooper, Adam Silberstein, Erwin Tam, Raghu Ramakrishnan, and Russell Sears. Benchmarking Cloud Serving Systems with YCSB. In *Proceedings of the 1st ACM Symposium on Cloud Computing*, SoCC '10, pages 143–154. ACM, New York, NY, USA, 2010.

[113] Karin Petersen, Mike J. Spreitzer, Douglas B. Terry, Marvin M. Theimer, and Alan J. Demers. Flexible Update Propagation for Weakly Consistent Replication. In *Proceedings of the 16th ACM Symposium on Operating Systems Principles*, SOSP '97, pages 288–301. ACM, New York, NY, USA, 1997.

[114] Mahesh Balakrishnan, Dahlia Malkhi, Ted Wobber, Ming Wu, Vijayan Prabhakaran, Michael Wei, John D. Davis, Sriram Rao, Tao Zou, and Aviad Zuck. Tango: Distributed Data Structures over a Shared Log. In *Proceedings of the 24th ACM Symposium on Operating Systems Principles*, SOSP '13, pages 325–340. ACM, New York, NY, USA, 2013.

[115] Cheng Li, Daniel Porto, Allen Clement, Johannes Gehrke, Nuno Preguiça, and Rodrigo Rodrigues. Making Geo-replicated Systems Fast As Possible, Consistent when Necessary. In *Proceedings of the 10th USENIX Conference on Operating Systems Design and Implementation*, OSDI'12, pages 265–278. USENIX Association, Berkeley, CA, USA, 2012.

[116] Peter Alvaro, Neil Conway, Joe Hellerstein, and William R. Marczak. Consistency Analysis in Bloom: a CALM and Collected Approach. In *CIDR 2011, 5th Biennial Conference on Innovative Data Systems Research, Asilomar, CA, USA, January 9-12, 2011, Online Proceedings*, pages 249–260, 2011.

[117] Peter Bailis, Alan Fekete, Michael J. Franklin, Ali Ghodsi, Joseph M. Hellerstein, and Ion Stoica. Coordination-Avoiding Database Systems. *CoRR*, abs/1402.2237, 2014.

[118] Cheng Li, João Leitão, Allen Clement, Nuno Preguiça, Rodrigo Rodrigues, and Viktor Vafeiadis. Automating the Choice of Consistency Levels in Replicated Systems. In *Proceedings of the 2014 USENIX Conference on USENIX Annual Technical Conference*, USENIX ATC'14, pages 281–292. USENIX Association, Berkeley, CA, USA, 2014.

[119] Armando Solar-Lezama. *Program Synthesis by Sketching*. PhD thesis, University of California at Berkeley, Berkeley, CA, USA, 2008.

[120] Neil Conway, Peter Alvaro, Emily Andrews, and Joseph M. Hellerstein. Edelweiss: Automatic Storage Reclamation for Distributed Programming. *PVLDB*, 7(6):481–492, 2014.

[121] Erik Meijer. The World According to LINQ. *Communication of the ACM*, 54(10):45–51, October 2011.

VITA

VITA

KC Sivaramakrishnan was born and brought up in the holy city of Srirangam, India. He obtained a B.Eng. in computer science and engineering from PSG College of Technology, Anna University, Coimbatore, India in May 2008. He obtained an M.S. and a Ph.D. from the Department of Computer Science at Purdue University in May 2011 and December 2014, respectively. His research interests include functional programming languages, concurrent programming, multicore runtime systems and distributed systems. During his Ph.D. studies, he interned with Samsung Research, San Jose, California (May 2010 – Aug 2010) and Microsoft Research, Cambridge, UK (Feb 2012 – May 2012).