Functional Programming Abstractions for Weakly Consistent Systems

KC Sivaramakrishnan

Final Examination
Outline

• Motivation
• Thesis
• Contributions
  – Aneris: A cache-coherent runtime on non-cache-coherent architecture
  – RxCML: A prescription for safely relaxing synchrony
  – Quelea: Declarative programming over eventually consistent memory
• Conclusions
\( \sim 2^3 \) cores

\( \sim 2^4 \) cores

\( \sim 2^6 \) cores

\( \sim 2^0 \text{ to } 2^{10+} \) cores
Memory Consistency Model

Replication, Coherence, Asynchrony, Partial failures, Heterogeneity, ……
Safe and scalable concurrent program

Data Races

Strong Consistency

Prog. Model: Seq. Consist., Linearizability, Serializability, etc.

Impl. Mech: H/W cache coherence, consensus, atomic broadcast, distributed locks

Atomicity violations

Deadlocks
Safe and scalable concurrent program

Data Races

Strong Consistency

Expanding chasm

bottleneck!

Impl. Mech:
H/W cache coherence, consensus, atomic broadcast, distributed locks

Atomicity violations

Deadlocks
Strong Consistency on Multicore

• Caches improve memory access latency
  – Conflict with Multi-processing
• **Hardware Cache Coherence** → Strong consistency
  – DRF programs are sequentially consistent
• Coherence mechanisms have become the bottleneck
  – Power requirements
  – Complexity of coherence hardware
  – Storage requirements of cache meta-data
  – Heterogeneity: GPUs + FPGAs + Co-processors, etc.
Rise of Non-cache-coherent Hardware

• Hardware support
  – No coherence ←……→ coherence islands

• Programmer’s view
  – *No sequential consistency*!
  – MPI, TCP/IP, RDMA, etc.,.
Consistency in the Cloud

- Distributed stores:
  - Shared memory abstraction for the cloud
  - Geo-replication → minimizing latency, tolerate partial failures, availability
- CAP Theorem\[1\] → No highly available, partition tolerant system can provide strong consistency
- *Live with eventual consistency!*

Programming under Eventual Consistency

• 2 distinct concerns
  – Consistency $\rightarrow$ *when* updates become visible

![Diagram showing two replicas and an update event]
Programming under Eventual Consistency

• 2 distinct concerns
  – Consistency $\rightarrow$ *when* updates become visible
Programming under Eventual Consistency

• 2 distinct concerns
  – **Consistency** ➔ *when* updates become visible
Programming under Eventual Consistency

• 2 distinct concerns
  – Consistency $\rightarrow$ *when* updates become visible
  – Convergence $\rightarrow$ *how* conflicting updates are resolved

![Diagram showing two replicas, Kyle, and Stan with their interactions](image-url)
Programming under Eventual Consistency

- 2 distinct concerns
  - Consistency → *when* updates become visible
  - **Convergence** → *how* conflicting updates are resolved

![Diagram showing Kyle and Stan with updates and inconsistency]

Inconsistency leaks to the user
Safe and scalable concurrent program

Relaxed consistency

Partial failures

Latency

Weak Consistency

Asynchrony

Data Races

Non-cache-coherence

Atomicity violations

Deadlocks

Partial failures

Non-cache-coherence
Key Observation

- **Mutation of shared state** causes weak consistency issues
- **Tame shared state mutation** → mitigate weak consistency issues
- **Functional programming**
  - Mutations are *rare and explicit!*
Thesis

Functional programming abstractions simplify scalable concurrent programming under weak consistency

Aneris
- Cache-coherent, seq. consistent prog. model
- Non-cache-coherent Multicore

RxCML
- Strongly Consistent CML Semantics
- Asynchronous Distributed System

Quelea
- Declarative, extensible, high-level prog. model
- Off-the-shelf eventually consistent data store

ISMM ‘12, MARC ‘12, JFP ‘14
PADL ‘14
In submission to PLDI ‘15
Aneris: Coherent Shared Memory on the Intel SCC

Cache Coherent

- No change to programming model

Intel SCC

- Shared memory
- Software Managed Cache-Coherence (SMC)

Cluster of Machines

- Distributed programming
- RCCE, MPI, TCP/IP
- Release consistency + RDMA

Can we program SCC as a cache coherent machine?
Intel SCC: Programmer’s View

No cache coherence (caching disabled)        Software managed cache coherence (release consistency)

Shared memory (off-chip)

L1    L2    L1    L1    L2    L2    L1
Private memory    Private memory    Private memory    Private memory
Core 0    Core 1    Core 2    Core 47
Context: MultiMLton on the Intel SCC

• Parallel extension of MLton
  – A whole-program, optimizing Standard ML compiler
  – Immutability is default, mutations are explicit

• Concurrency model – Asynchronous CML

  \[ \text{send}(c, v) \]
  \[ v \gets \text{recv}(c) \]

• Collectors
  – LC, PRC, SMC
Local Collector (LC)

- Thread-local heap $\Rightarrow$ Circumvent the need for coherence
- *No access to remote core-private memory* $\Rightarrow$ *no need for cache coherence*
- Requires both read and write memory barriers!
  - Write barrier *globalizes*; read barrier handles *forwarding pointers*

Can we eliminate this overhead?

### 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...
Procrastinating Collector (PRC)

• Exploits concurrent functional nature of programming language
  – SML (Mostly functional) → Mutations are rare
    • Write barriers << read barriers
  – ACML → Lots of concurrency

• Eliminate read barriers completely
  – Mutator must never encounter forwarding pointers

• (Rare) Write barriers are more expensive
  1. Immutability: Globalize immutable objects by making a copy.
  2. Dynamic shape analysis: for objects completely in minor heap, globalize and perform minor local GC
  3. Procrastinate: Other objects, suspend threads instead of globalization
Software-managed Coherence (SMC)

- Mutability information + software coherence support

- Integrate cache control instructions into memory barriers
Results

- PRC(SMC) 23%(33%) faster than LC @ 48 cores
- 99% of the memory accesses in SMC are cacheable!
Aneris : Conclusion

• Concurrent FP language and runtime can effectively mask non-cache-coherence

  1. Utilize thread-local heap architecture to circumvent the absence of coherence
  2. Utilize mutability information to optimize for memory/cache hierarchy
  3. Trade concurrency for minimizing GC overheads
Synchronous communication = atomic \{ data transfer + synchronization \}
Can we discharge synchronous communications asynchronously and ensure observable equivalence?

Formalize:

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

Implement:

Distributed Concurrent ML on MultiMLton (Speculative execution)
Concurrent ML

val spawn : (unit -> unit) -> thread_id

val channel : unit -> 'a chan
val send : 'a chan * 'a -> unit
val recv : 'a chan -> 'a

val sendEvt : 'a chan * 'a -> unit event
val recvEvt : 'a chan -> 'a event
val sync : 'a event -> 'a
val never : 'a event
val alwaysEvt : 'a -> 'a event
val wrap : 'a event -> ('a -> 'b) -> 'b event
val guard : (unit -> 'a event) -> 'a event
val choose : 'a event list -> 'a event

Thread creation
Synchronous message passing
First-class events
In this paper, we investigate an alternative semantics for CML: A Prescription for Safely Relaxing Synchrony

1. Introduction

Synchrony complicates program structure and understanding. While the use of asynchrony can help reclaim performance, it also complicates program reasoning. The cost of explicitly accounting for new behaviors introduced by this additional concurrency. Thus, we wish to have the runtime enforce the equivalence:

send synchronously or asynchronously) would be the same.

The ability of communicated data, simplifying program reasoning. The cost of communication can be masked by allowing the sender's continuation to begin execution even if a matching receiver is not yet available. Because asynchrony is introduced only by the runtime, applications do not have to be restructured to explicitly account for new behaviors.

One way to enhance performance without requiring new additions to the core set of event combinators CML supports, is to give programmers the ability to have the runtime enforce the equivalence:

send synchronously or asynchronously) would be the same.

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

Assuming

\[
\text{send(c1,v1)} \quad \text{send(c2,v2)} \quad \text{send(c2,v3)}
\]

\[
\text{recv(c2)} \quad \text{recv(c1)} \quad \text{recv(c2)}
\]

\[
\text{f()} \quad \text{g()} \quad \text{h()}
\]

\[
\text{B} \quad \text{A} \quad \text{C}
\]
Theorem:

Cyclic dependence $\Rightarrow$ divergent behavior
Formalization

Reason axiomatically

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

Well-formed execution

\[ \text{Obs} (\text{WF}_\text{Exec} (P)) \subseteq \{ \text{Obs} (\text{Sync}_\text{Exec} (P)) \} \]

Theorem

- No happens before cycle
- Sensible intra-thread semantics
- No outstanding speculative actions

Recipe for implementation
Implementation

• Dependence graph \equiv Axiomatic execution
  – WF check before observable actions
  – Ill-formed? Rollback and re-execute non-speculatively – Progress!

• Channel consistency
  – Channel state *replicas* at each site
  – Preserve CML semantics – Strong consistency!
  – *Recover strong consistency using speculative execution*

• Mutable references
  – *Cross-site references are prohibited*
  – Checkpoint \rightarrow *local continuation capture + communication log*
Results

- Benchmark: Optimistic OLTP & P2P Collaborative editing

![OLTP Graph](Image)

**OLTP**
- 5.8X faster than sync
- 1.4X slower than async
- @ 48 clients

![Collaborative Editing Graph](Image)

**Collaborative Editing**
- 7.6X faster than sync
- 2.3X slower than async
- @ 6 authors

**Rx-CML → efficient abstraction over high-latency distributed systems!**
RxCML : Conclusion

Concurrent ML

Asynchronous distributed system

Rx-CML: Speculative Execution! (Performance + Consistency)

synchrony

+ Strong consistency of CML

latency
Quelea

• **PL support** for working with eventually consistent data stores

• Problems with existing eventually consistent data stores
  1. Consistency
     • Basic eventual, session guarantees, timeline, causal, sequential, recency, bounded staleness, etc. + Transaction isolation levels!
  2. Convergence
     • LWW register, grow-only counter, and a few more.
     • Lack primitives for operation composition

• Goals
  1. *Automatically map* application-level consistency to store-level consistency
  2. *Let the programmer describe their own* Replicated Data Types (RDTs)
Quelea: Convergence

- **RDT specification language**
  - Object state $\rightarrow$ trace of operation *effects*
    - Trace only-grows
    - No destructive updates $\rightarrow$ conflicts preserved!
  - Operations $\rightarrow$ reduction over trace
    - *Update conflicts are resolved in the operations*

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

Object snapshot (trace of effects)

Read-only returns *Nothing.*
Queuea: Consistency

• **Contract language**
  
  – Express fine-grained app-level consistency

  \[
  \begin{align*}
  x, y, \hat{\eta} \in \text{EffVar} & \quad \text{Op} \in \text{OperName} \\
  \psi \in \text{Contract} & ::= \forall (x : \tau). \psi \mid \forall x. \psi \mid \pi \\
  \tau \in \text{EffType} & ::= \text{Op} \mid \tau \lor \tau \\
  \pi \in \text{Prop} & ::= \text{true} \mid R(x, y) \mid \pi \lor \pi \\
  & \quad \mid \pi \land \pi \mid \pi \Rightarrow \pi \\
  R \in \text{Relation} & ::= \text{vis} \mid \text{so} \mid \text{sameobj} \mid R^+ \\
  & \quad \mid R \cup R \mid R \cap R
  \end{align*}
  \]

  – A **contract enforcement system** assigns correct consistency level

  • Describe store semantics in the *same* contract language

  \[
  \Delta \vdash \psi_{\text{store}} \Rightarrow \psi_{\text{op}}
  \]

  • **Decidable** → Automatically discharged with the help of SMT solver.
Bank Account RDT

• Goal
  – deposit, withdraw and getBalance
  – Balance \( \geq 0 \)

• Effects

\[
data\ Acc = \text{Deposit} \ Int | \text{Withdraw} \ Int | \text{GetBalance}
\]

\[
getBalance :: [\text{Acc}] \rightarrow () \rightarrow (\text{Int, Maybe Acc})
\]

\[
getBalance\ \text{hist} = \begin{cases} 
\text{let res = sum [x | Deposit x} & \leftarrow \text{hist]} \\
- \text{sum [x | Withdraw x} & \leftarrow \text{hist]} \\
in (\text{res, Nothing})
\end{cases}
\]

\[
withdraw :: [\text{Acc}] \rightarrow \text{Int} \rightarrow (\text{Bool, Maybe Acc})
\]

\[
\text{withdraw}\ \text{hist} \ v = \begin{cases} 
\text{if sell } & \text{getBalance}\ \text{hist} () \geq v \\
\text{then (True, Just } & \text{Withdraw} \ v) \\
\text{else (False, Nothing})
\end{cases}
\]
Bank Account Contracts

- Balance $\geq 0$
- Any two withdraw operations must be totally ordered
- A get balance operation witnessing a withdraw must witness all its visible deposits

\[ \psi_w(\hat{\eta}) = \forall (a : \text{Withdraw}). \text{sameobj}(a, \hat{\eta}) \Rightarrow \text{vis}(a, \hat{\eta}) \lor \text{vis}(\eta, a) \lor a = \hat{\eta} \]

\[ \psi_{gb}(\eta) = \forall (a : \text{Deposit}), (b : \text{Withdraw}). \text{vis}(a, b) \land \text{vis}(b, \eta) \Rightarrow \text{vis}(a, \eta) \]

\[ \psi_d(\hat{\eta}) = \text{true} \]
Implementation

Support for coordination-free txns

Off-the-shelf Distributed Store
- Off-the-shelf store
- Failure handling
- Persistence (on-disk)
- Eventual consistency

Shim Layer (RDTs)
- Soft-state (in-mem)
- Datatype operations
- Summarization
- Stronger consistency

Quelea Replicated Store

Obj.oper(args)

res

Business Logic (incl. Txns)

REST API

Clients

We see that majority of the operations and transactions are classified as eventually consistent and RC, reflecting applications, which includes individual RDTs as well as larger applications composed of several RDTs: An eBay-like auction site, an online store with shopping cart functionality and dynamically changing item prices, the accepted price. The checkout process verifies that the customer only pays for that particular object during summarization (usually a recent transaction). Shim layer summarization is straightforward; a summarization thread takes the local lock for that particular object until the global lock is acquired. Shim layer summarization is efficient because it is acting on a causally consistent snapshot of the store and because of the property that relations satisfy in the causal ordering. Dependence tracking is similar to the techniques presented in Lloyd et al. and Bailis et al., and it involves explicitly tracking dependencies introduced between effects due to visibility, session and same transaction, respectively. Dependence tracking is similar to the techniques presented in Lloyd et al., and Bailis et al., and it involves explicitly tracking dependencies introduced between effects due to visibility, session and same transaction, respectively. Dependence tracking is similar to the techniques presented in Lloyd et al., and Bailis et al., and it involves explicitly tracking dependencies introduced between effects due to visibility, session and same transaction, respectively.
Evaluation: Classification

<table>
<thead>
<tr>
<th>Benchmark</th>
<th>LOC</th>
<th>#T</th>
<th>EC</th>
<th>CC</th>
<th>SC</th>
<th>RC</th>
<th>MAV</th>
<th>RR</th>
</tr>
</thead>
<tbody>
<tr>
<td>LWW Reg</td>
<td>108</td>
<td>1</td>
<td>2</td>
<td>2</td>
<td>2</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>DynamoDB</td>
<td>126</td>
<td>1</td>
<td>3</td>
<td>1</td>
<td>2</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>Bank Account</td>
<td>155</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>Shopping List</td>
<td>140</td>
<td>1</td>
<td>2</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>Online store</td>
<td>340</td>
<td>4</td>
<td>9</td>
<td>1</td>
<td>0</td>
<td>2</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>RUBiS</td>
<td>640</td>
<td>6</td>
<td>14</td>
<td>2</td>
<td>1</td>
<td>4</td>
<td>2</td>
<td>0</td>
</tr>
<tr>
<td>Microblog</td>
<td>659</td>
<td>5</td>
<td>13</td>
<td>6</td>
<td>1</td>
<td>6</td>
<td>3</td>
<td>1</td>
</tr>
</tbody>
</table>

- Performance evaluation
  - Amazon EC2 + Cassandra cluster + Quelea shim layer
- Bank account
  - deposit $\rightarrow$ EC, withdraw $\rightarrow$ SC, getBalance $\rightarrow$ CC
  - Compared to all operations tagged SC, Quelea had
    - 1DC $\rightarrow$ 40%(139%) lower(higher) latency(throughput)
    - 2DC $\rightarrow$ 86%(618%) lower(higher) latency(throughput)
Quelea: Conclusions

• PL support for eventual consistency
  – Convergence
    • Grow-only trace of effects
    • Reductions resolve conflicts
  – Consistency
    • Contract language for declarative reasoning
    • SMT solver for contract classification

• Realized on top of off-the-shelf stores!
Summary

Functional programming abstractions simplify scalable concurrent programming under weak consistency

- Immutability
  - Eliminating read barriers
  - Cached shared heap
- Mostly functional nature
  - Small shared heap

- Explicit comm.
  - Simplifies formal reasoning
  - Tractable dep. graph

- Checkpoint
  - Save current continuation & ignore heap

- No destructive updates
  - Sequential reasoning for eventually consistent RDTs

Aneris → RxCML → Quelea
Publications

• Aneris
  – JFP 2014 ➔ MultiMLton language and runtime system
  – ISMM 2012 ➔ Local & Procrastinating collectors
  – MARC 2012 ➔ Software managed coherence
    • Best paper award

• RxCML
  – PADL 2014

• Quelea
  – In submission to PLDI 2015
Future Work

Declarative Consistency!

Contract Inference

Quelea Contracts

Traditional VM

CC multicore

Aneris

Non-CC multicore

Cassandra, Riak, DynamoDB

Geo-distributed compute cluster

SQL Constraints

Not NULL, Unique, Primary Key, Foreign Key, Check, Default

Programming model

Backends

Shared Mem, Fences, Locks, Cond Var

Distr. Mem, SMC, RDMA, MPB

KV interface, Vector clocks, Consensus

Strong Consistency

SQL Constraints

Contract Inference

Quelea Contracts

Traditional VM

CC multicore

Aneris

Non-CC multicore

Cassandra, Riak, DynamoDB

Geo-distributed compute cluster
Thank you!