Functional Programming Abstractions for Weakly Consistent Systems

KC Sivaramakrishnan

Final Examination

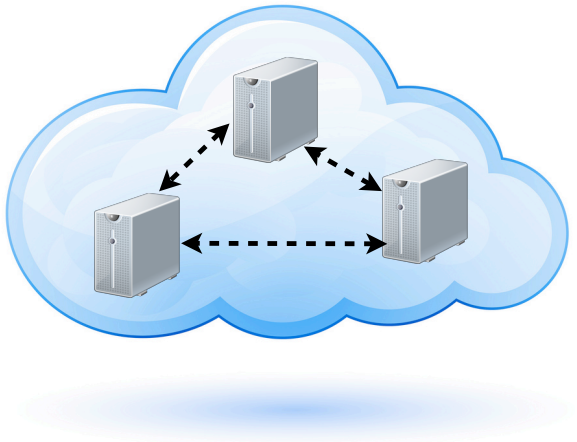
Outline

- 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

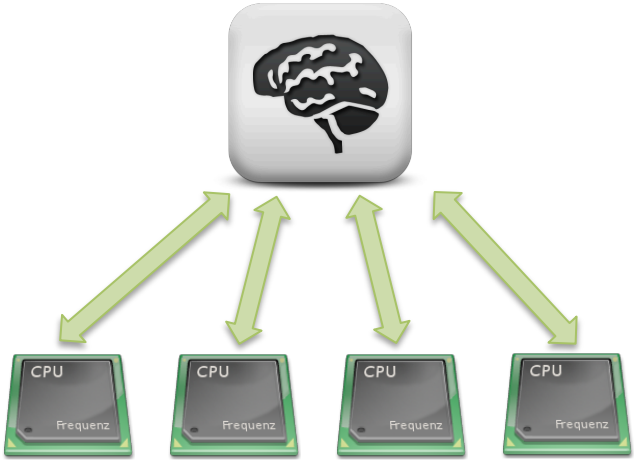
$\sim 2^3$ cores

$\sim 2^4$ cores

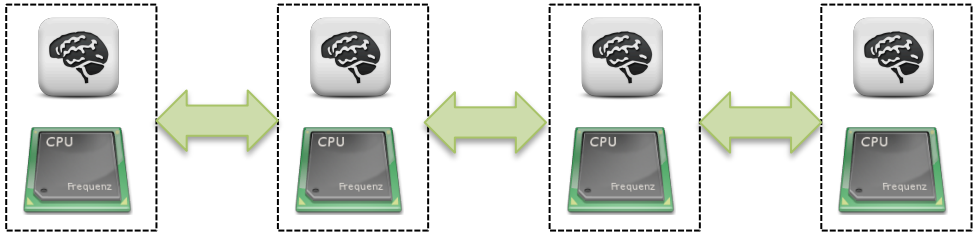
$\sim 2^6$ cores



$\sim 2^0$ to 2^{10+} cores

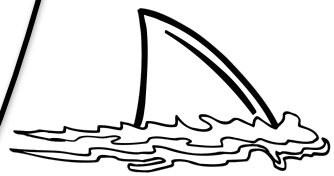


Memory Consistency Model



Replication,
Coherence,
Asynchrony,
Partial failures,
Heterogeneity,
……

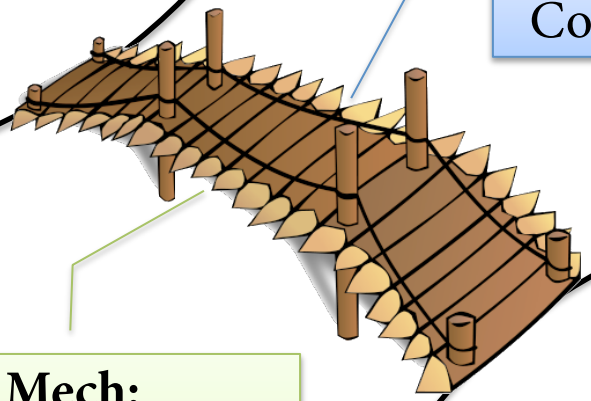
Safe and scalable concurrent program



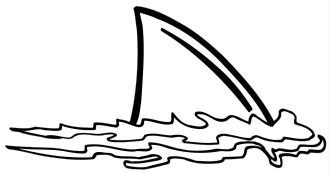
Data Races

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

Strong Consistency



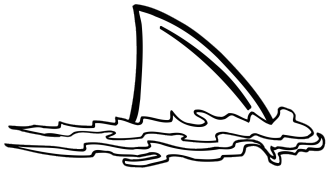
Impl. Mech: H/W cache coherence, consensus, atomic broadcast, distributed locks



Atomicity violations

Deadlocks

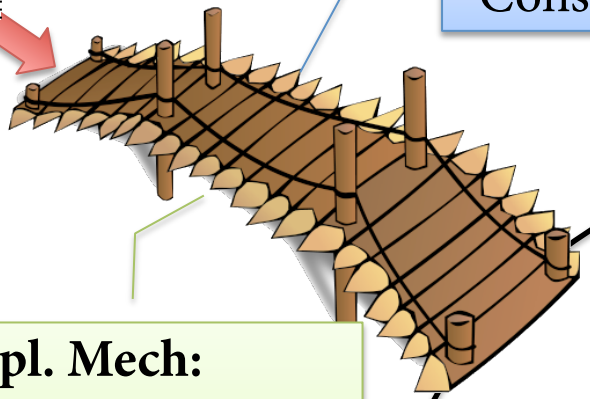
Safe and scalable concurrent program



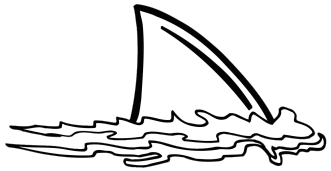
Data Races

Expanding chasm

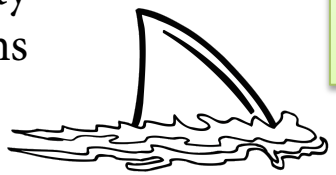
Strong Consistency



bottleneck!

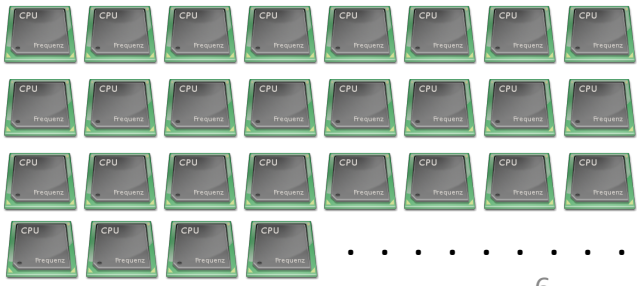


Atomicity violations

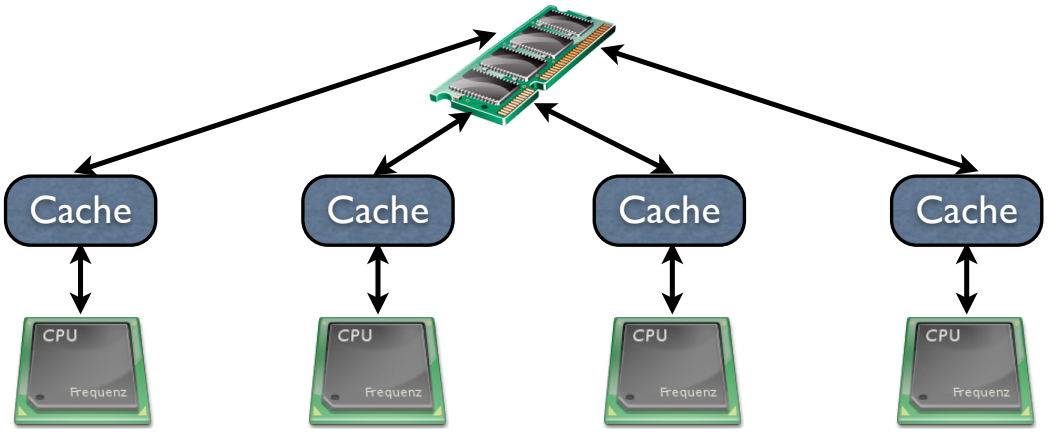


Deadlocks

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

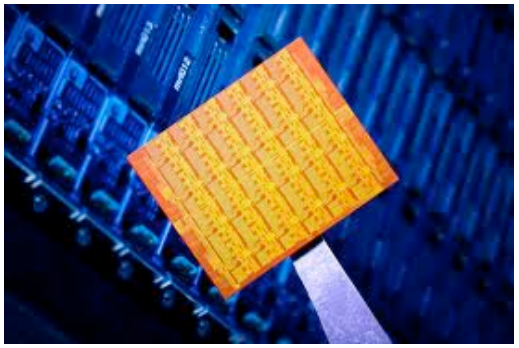


Strong Consistency on Multicore

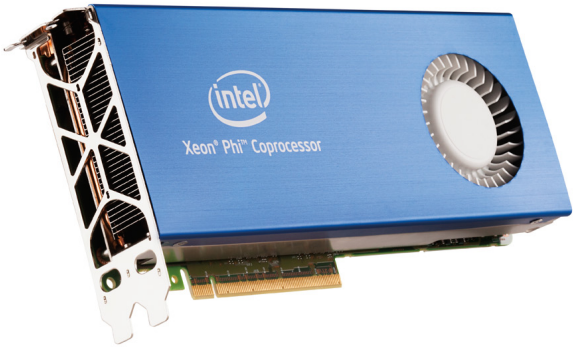


- Caches improve memory access latency
  - Conflict with Multi-processing
- Hardware Cache Coherence → Strong consistency
  - DRF programs are sequentially consistent
- Coherence mechanisms have become the bottleneck
  - Power requirements
  - Complexity of coherence hardware
  - Storage requirements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

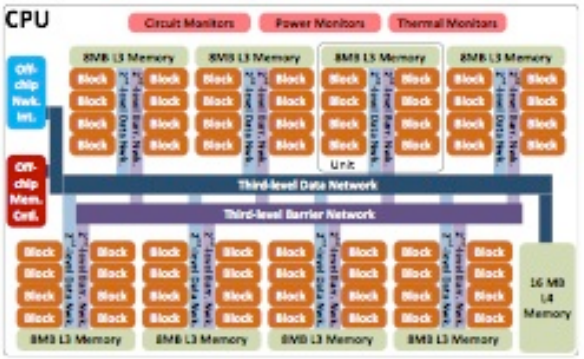
Rise of Non-cache-coherent Hardware



Intel SCC



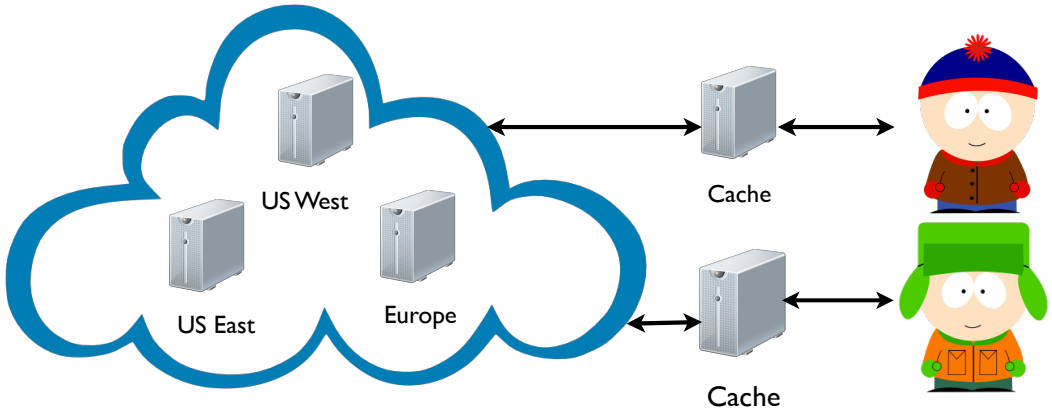
Xeon Phi



Runnymede

- Hardware support
  - No coherence ←……→ coherence islands
- Programmer's view
  - *No sequential consistency!*
  - MPI, TCP/IP, RDMA, etc,.

Consistency in the Cloud

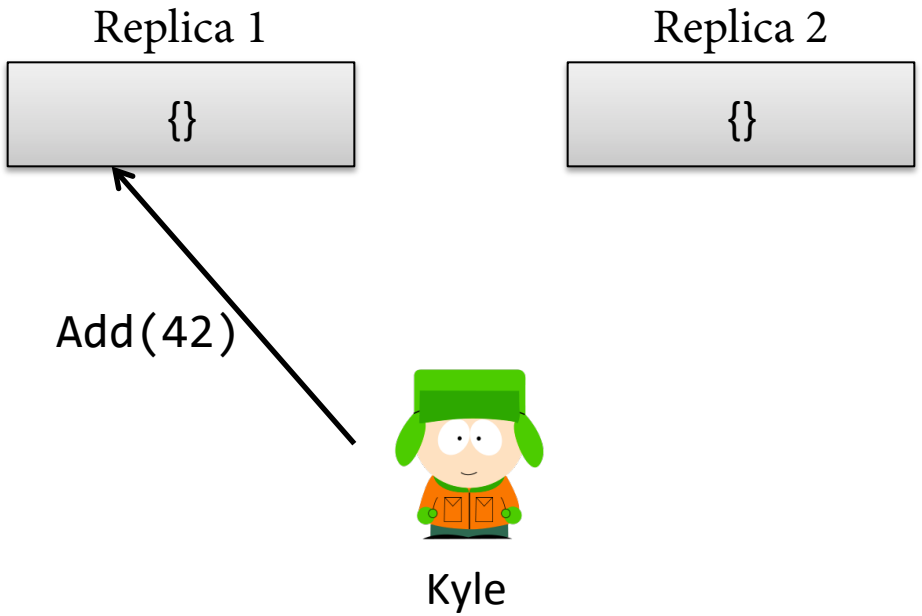


- Distributed stores:
  - Shared memory abstraction for the cloud
  - Geo-replication → minimizing latency, tolerate partial failures, availability
- CAP Theorem[1] → No highly available, partition tolerant system can provide strong consistency
- *Live with eventual consistency!*

[1] N. Lynch el al., "Brewer's conjecture and the feasibility of consistent, available, partition-tolerant web services", ACM SIGACT News, 2002.

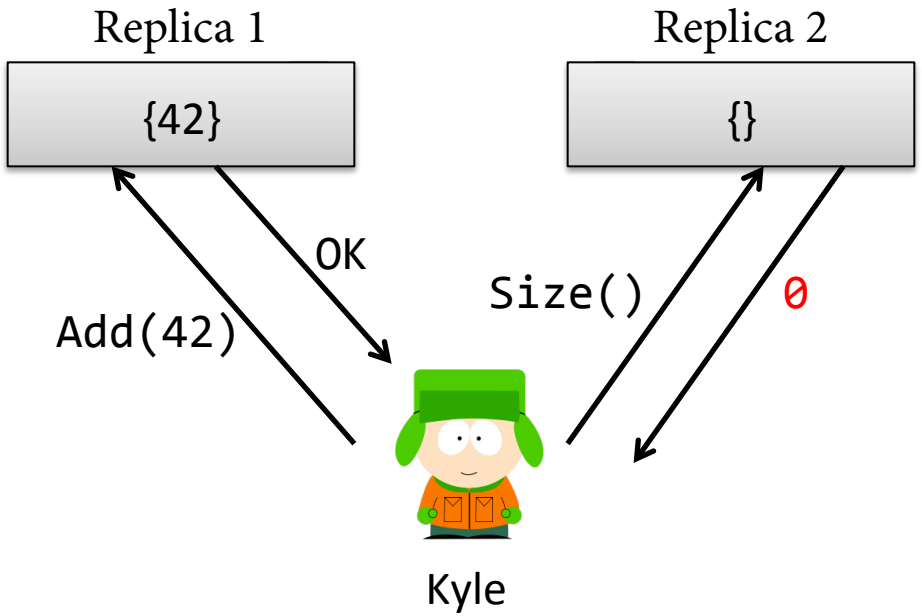
Programming under Eventual Consistency

- 2 distinct concerns
  - **Consistency** → *when* updates become visible



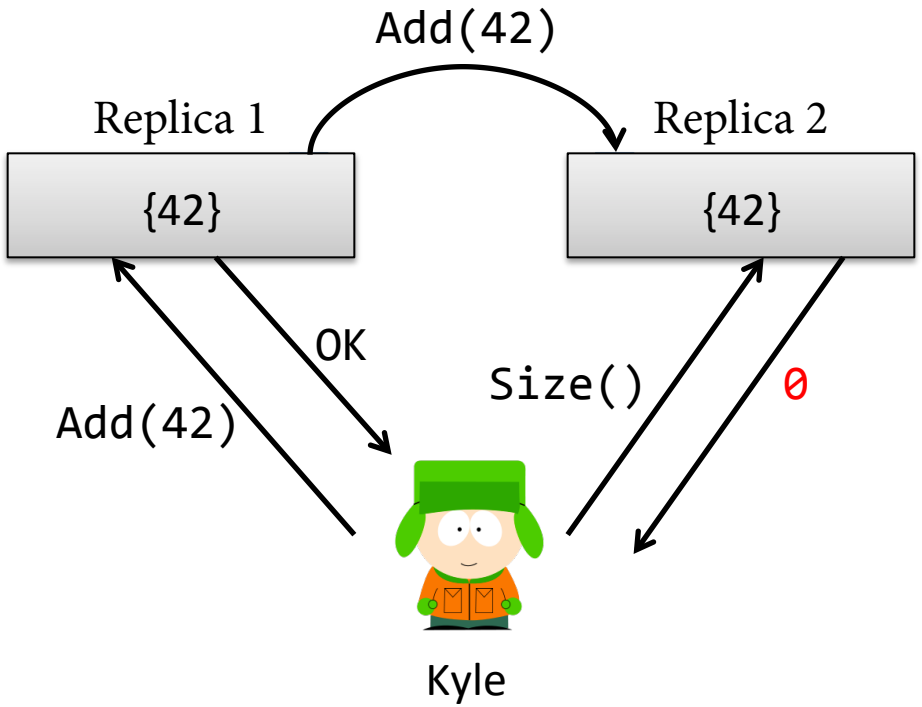
Programming under Eventual Consistency

- 2 distinct concerns
  - **Consistency** → *when* updates become visible



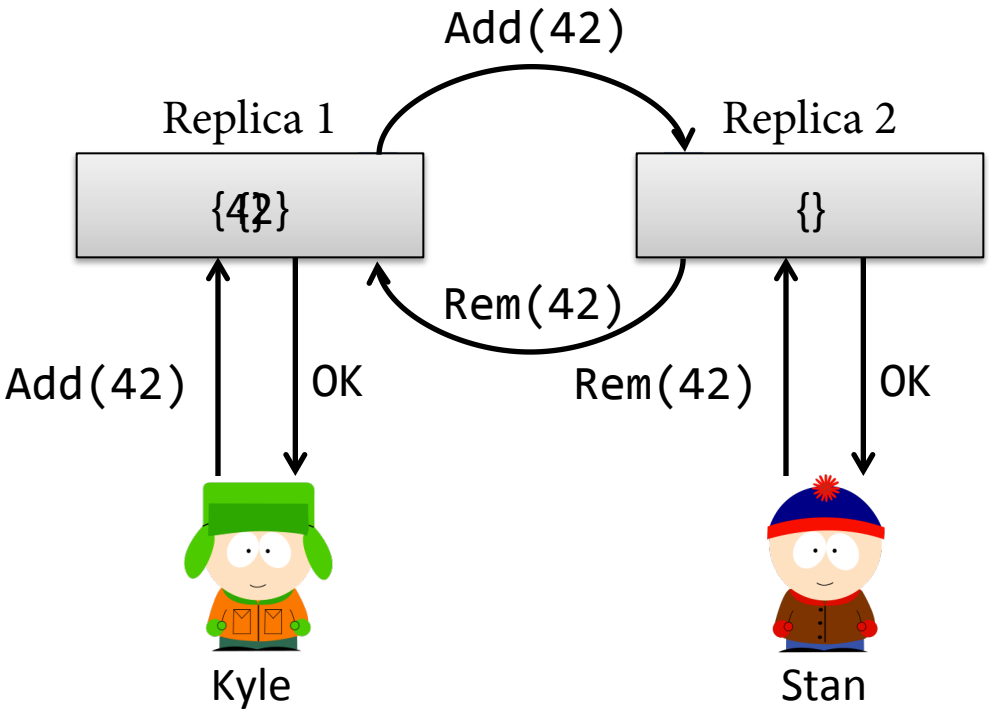
Programming under Eventual Consistency

- 2 distinct concerns
  - **Consistency** → *when* updates become visible



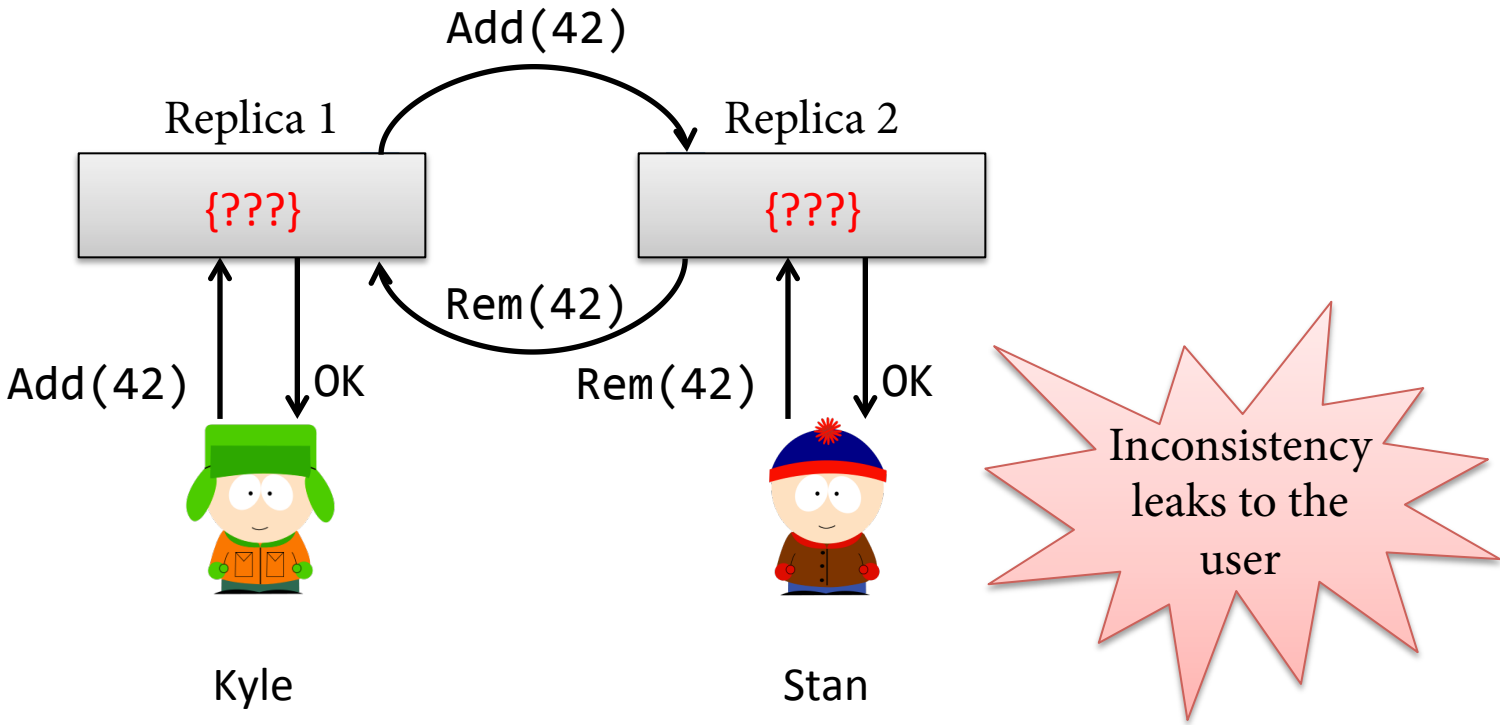
Programming under Eventual Consistency

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



Programming under Eventual Consistency

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



Safe and scalable concurrent program

Relaxed consistency

Partial failures

Latency

Data Races

Weak Consistency

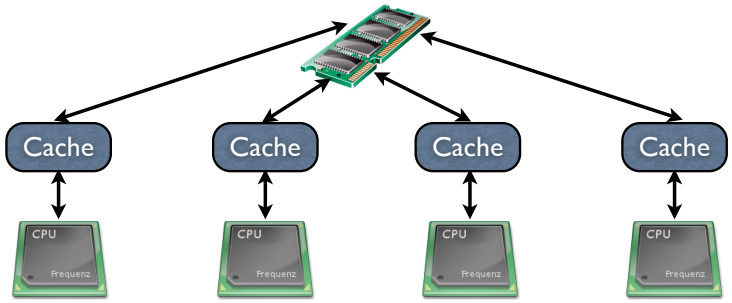
Asynchrony

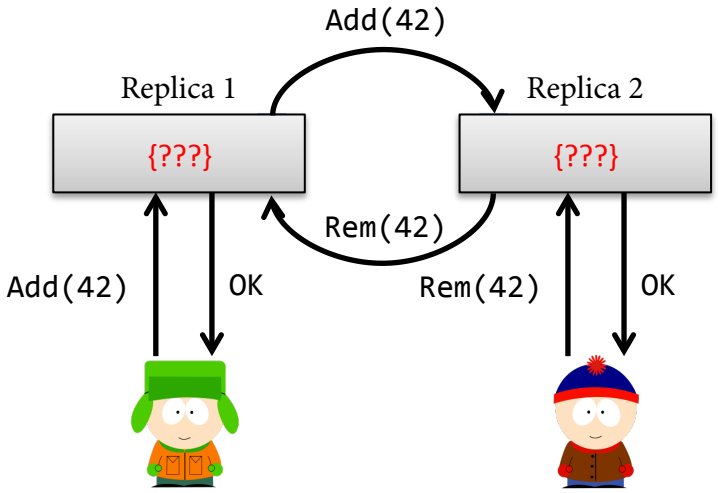
Atomicity violations

Deadlocks

Non-cache-coherence

Key Observation

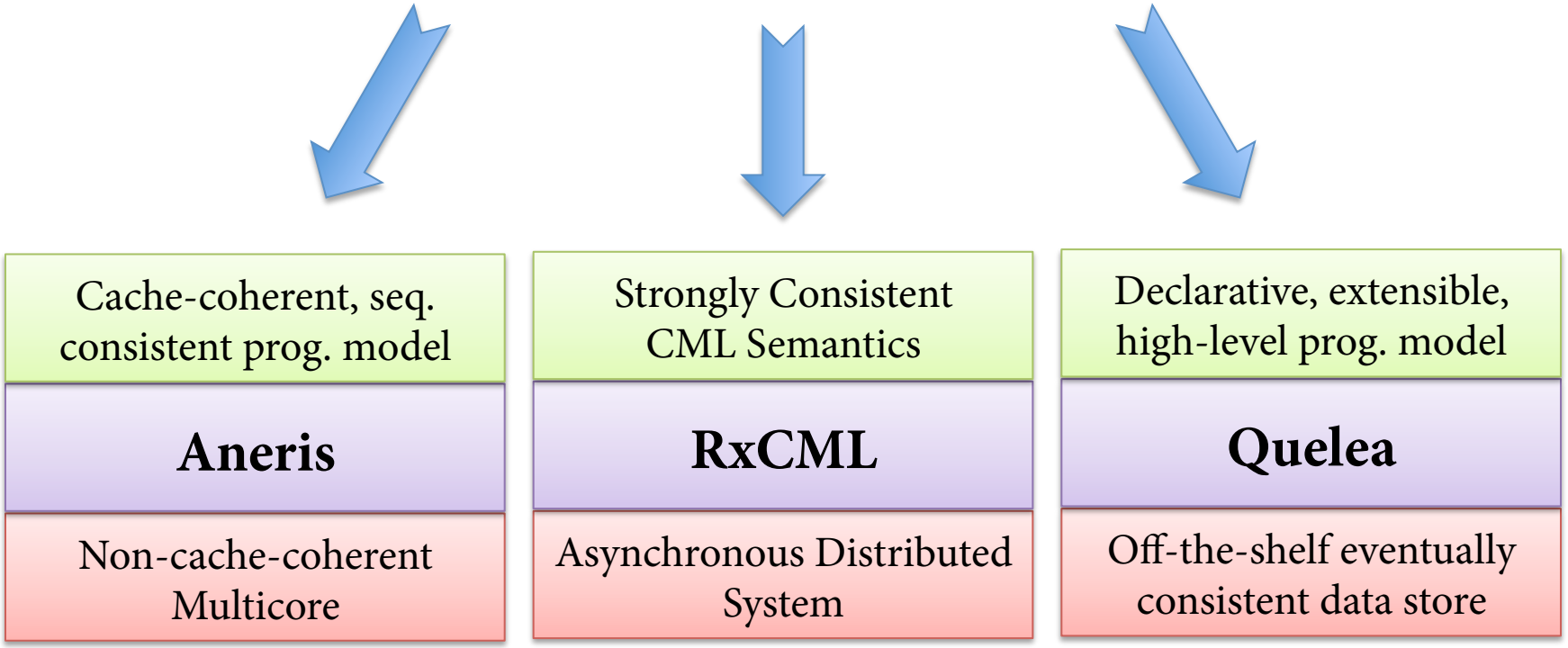




- *Mutation of shared state* causes weak consistency issues
- *Tame shared state mutation* → mitigate weak consistency issues
- Functional programming
  - Mutations are *rare and explicit!*

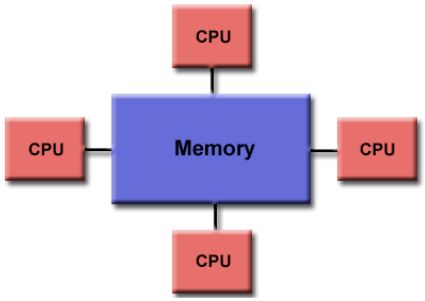
Thesis

Functional programming abstractions simplify scalable concurrent programming under weak consistency



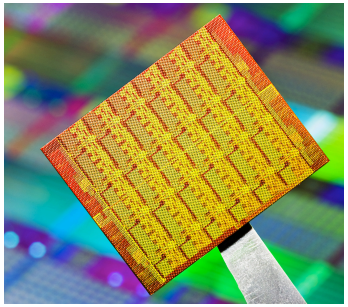
Aneris : Coherent Shared Memory on the Intel SCC

Cache Coherent



??

Intel SCC



Cluster of Machines

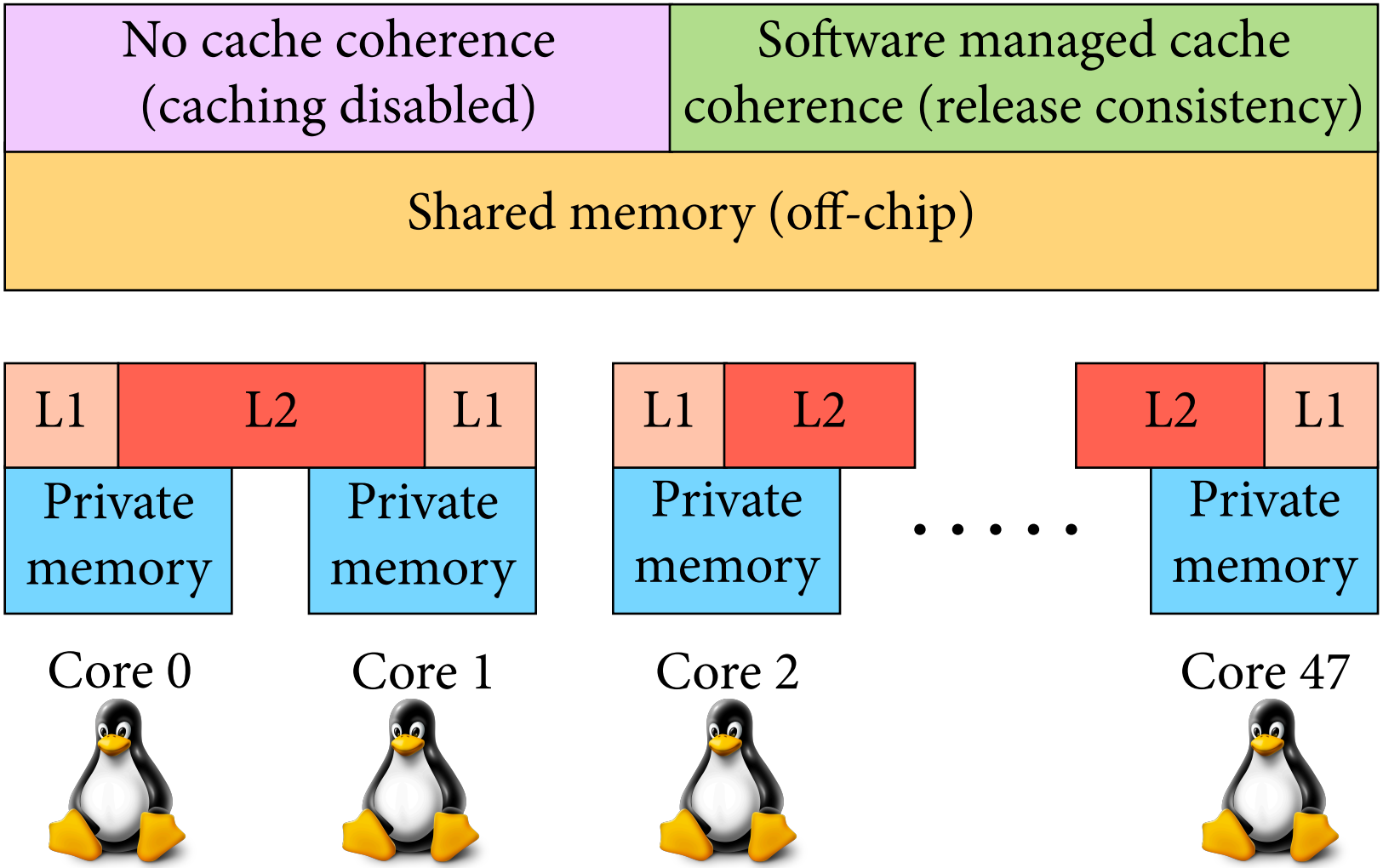
✓ No change to programming model

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

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

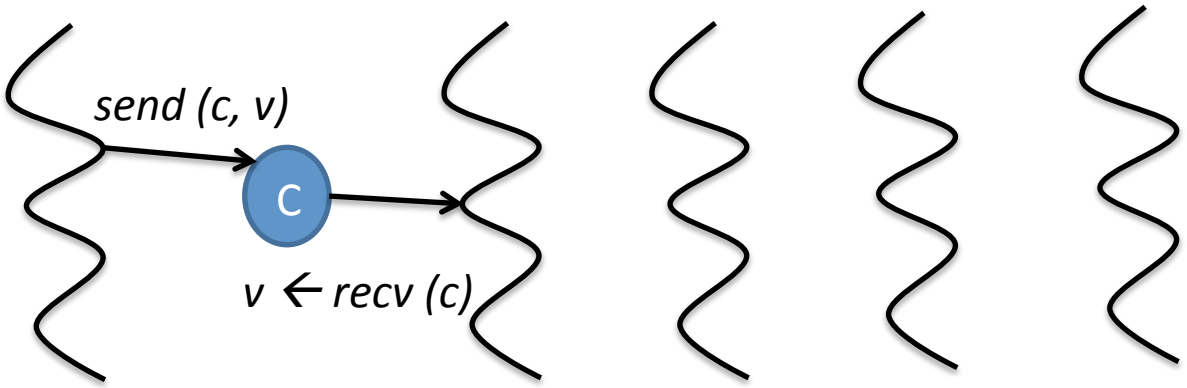
Can we program SCC as a cache coherent machine?

Intel SCC: Programmer's View



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

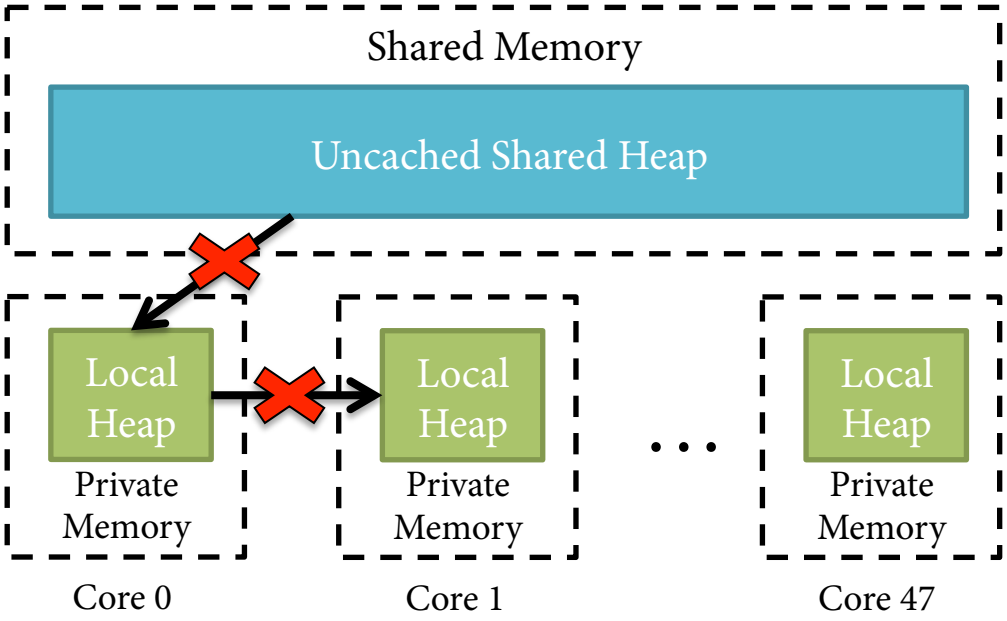


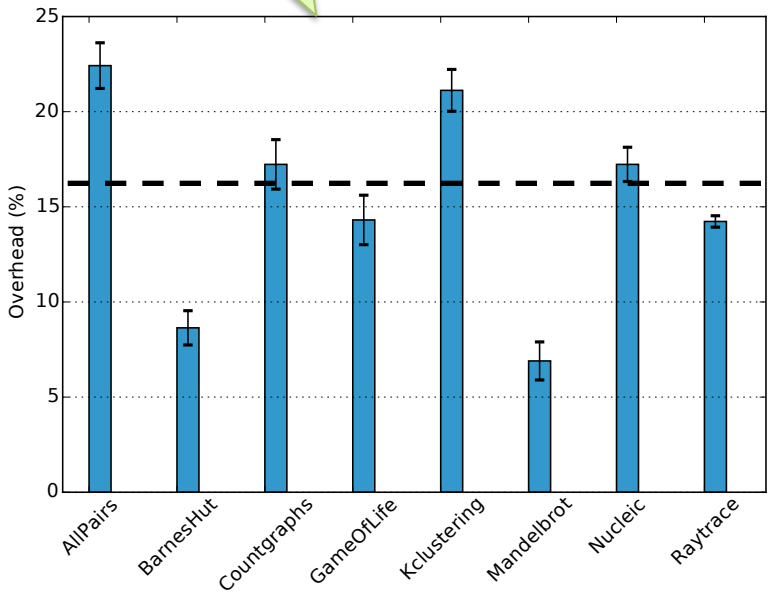
- Collectors
  - LC, PRC, SMC

Local Collector (LC)

- Thread-local heap → Circumvent the need for coherence
- *No access to remote core-private memory → no need for cache coherence*
- Requires both read and write memo
  - Write barrier *globalizes*; read barrier handle *pointers*

Can we eliminate this overhead?



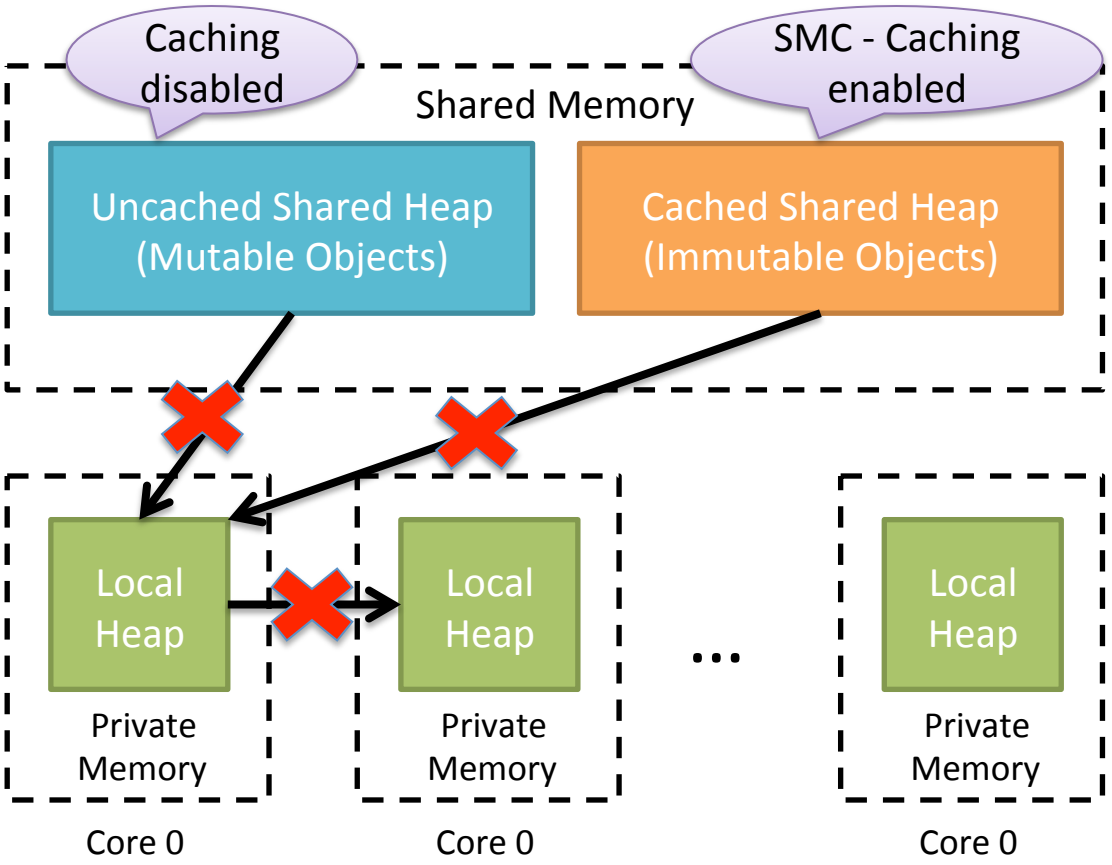


Procrastinating Collector (PRC)

- 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

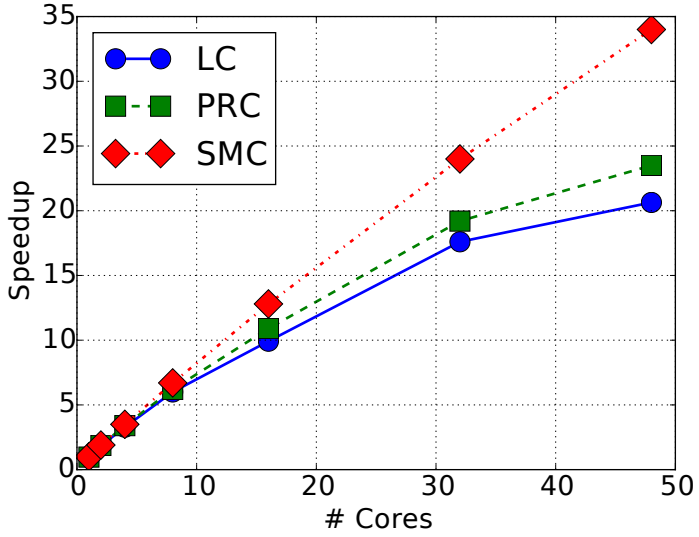
Software-managed Coherence (SMC)

- Mutability information + software coherence support

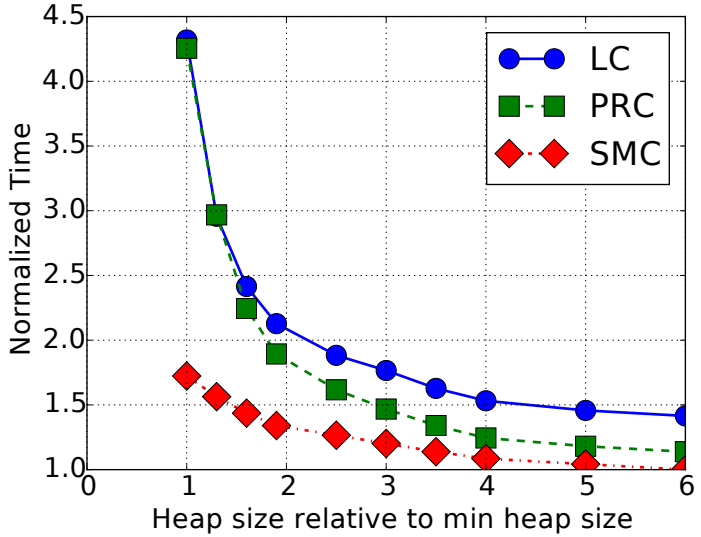


- *Integrate cache control instructions into memory barriers*

Results



Speedup



Heap size vs Total time

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

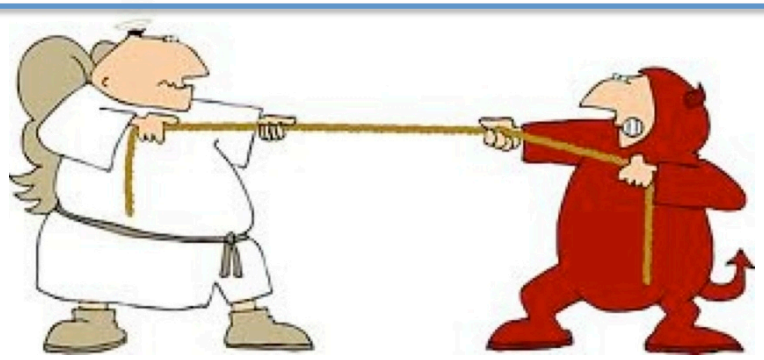
Aneris : Conclusion

- 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

RxCML

Abstraction	Composable Synchronous communication	Transactional memory	Garbage collection
Domain	Asynchronous distributed system	Shared memory concurrency	Memory management

Synchronous communication =
atomic { data transfer +
synchronization }



synchrony latency

Can we discharge synchronous communications asynchronously and ensure observable equivalence?

Formalize:

$$[\![\, \texttt{send}\,(c, v)]\!]k \;\equiv\; [\![\, \texttt{asend}\,(c, v)]\!]k$$

Implement:

Distributed Concurrent ML on MultiMLton
(Speculative execution)

Concurrent ML

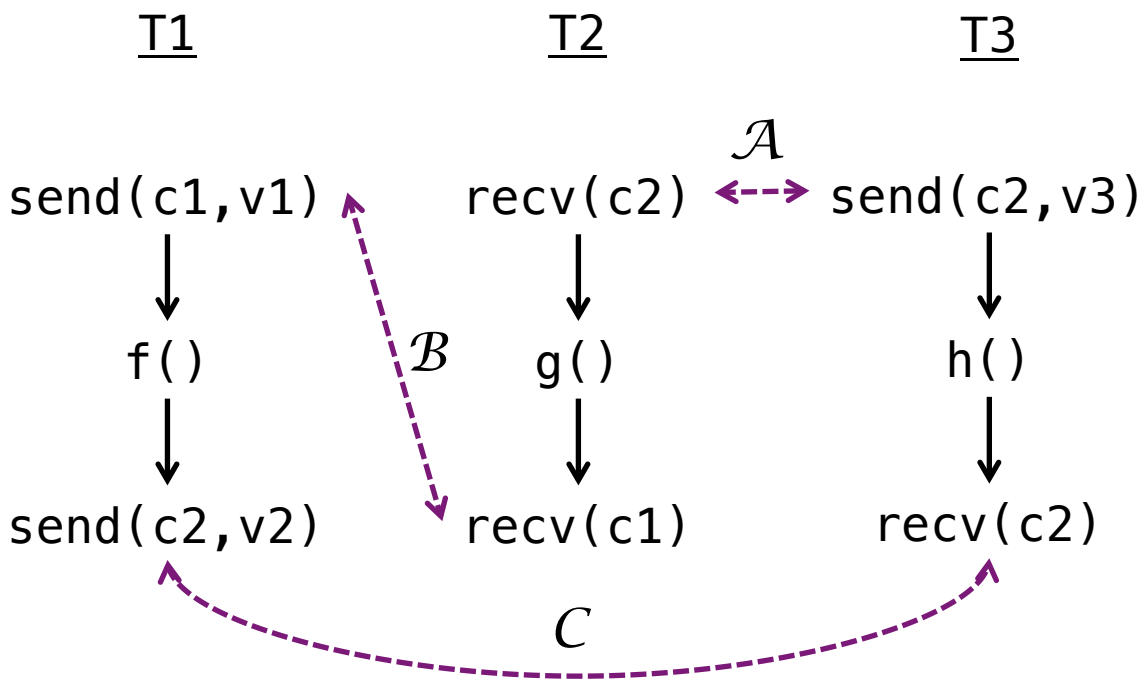
```
val spawn      : (unit -> unit) -> thread_id

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
...
```

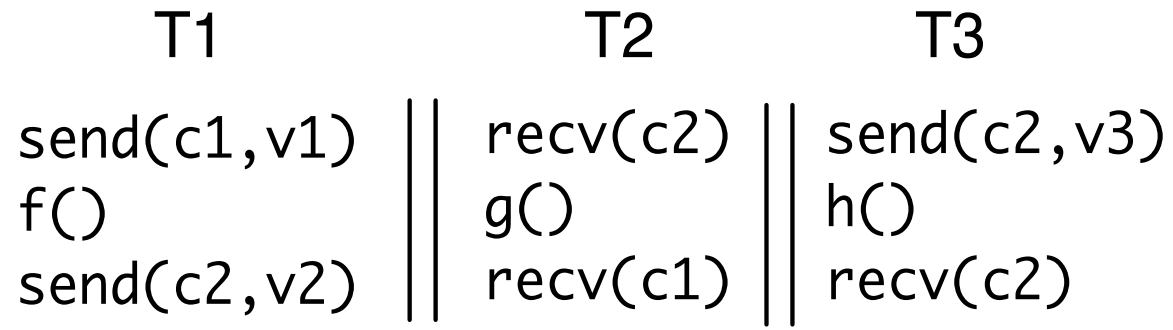
Thread creation

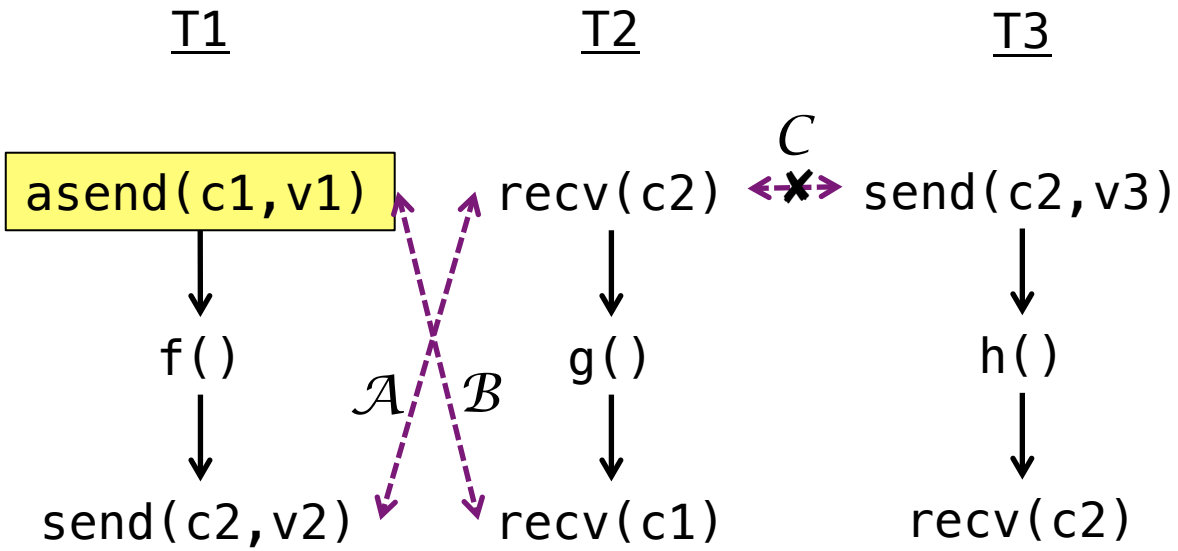
Synchronous message passing

First-class events

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







Theorem:

Cyclic dependence ⇒ divergent behavior

Formalization

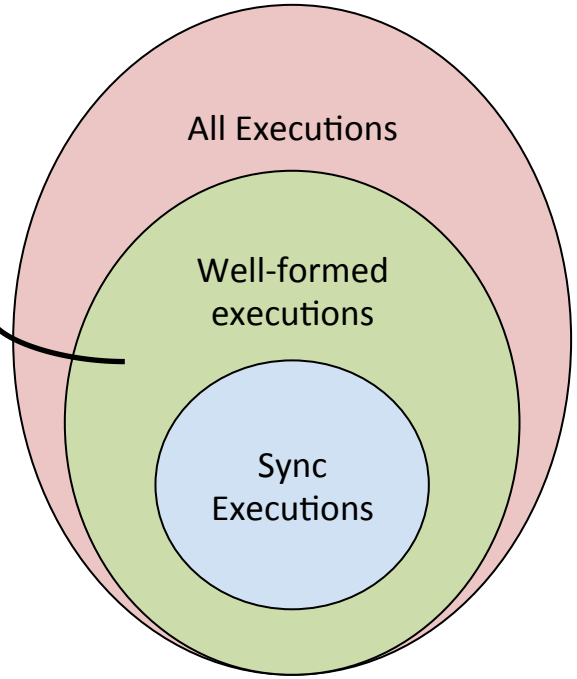
Reason axiomatically $\quad E := \langle P, A, \rightarrow_{po}, \rightarrow_{co} \rangle$

Well-formed execution $\quad Obs\ (WF\_Exec\ (P)) \in \{Obs\ (Sync\_Exec\ (P))\}$

Theorem

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

Recipe for implementation

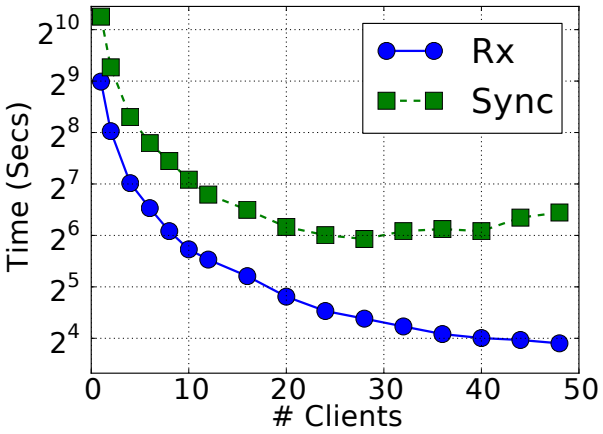


Implementation

- Dependence graph ≡ 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 → *local continuation capture + communication log*

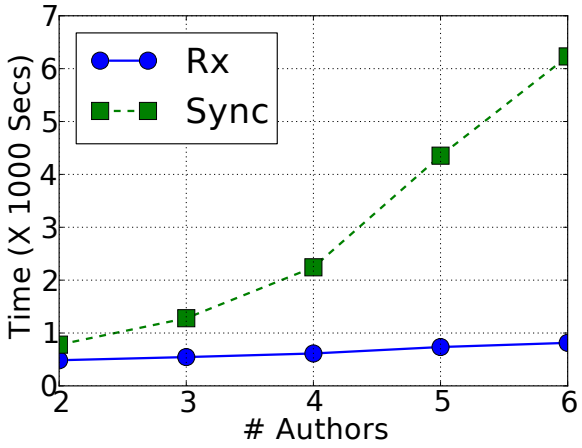
Results

- Benchmark: Optimistic OLTP & P2P Collaborative editing



OLTP

5.8X faster than sync
1.4X slower than async
@ 48 clients



Collaborative Editing

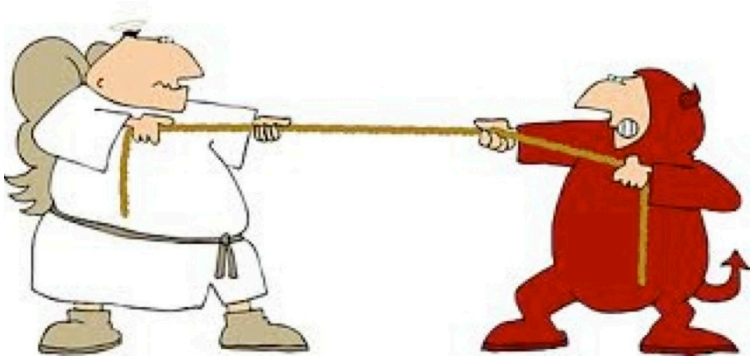
7.6X faster than sync
2.3X slower than async
@ 6 authors

Rx-CML → efficient abstraction over high-latency distributed systems!

RxCML : Conclusion

Concurrent ML

Asynchronous distributed system



synchrony latency

\+

Strong consistency of CML

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

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 → trace of operation *effects*
    - Trace only-grows
    - No destructive updates → conflicts preserved!
  - Operations → reduction over trace
    - *Update conflicts are resolved in the operations*

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

Object snapshot (trace of effects)

Read-only returns **Nothing**.

Quelea: Consistency

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

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

Primitive relations

  - A *contract enforcement system* assigns correct consistency level
    - Describe store semantics in the *same* contract language

$$\Delta \vdash \psi_{store} \Rightarrow \psi_{op}$$

    - *Decidable* → Automatically discharged with the help of SMT solver.

Bank Account RDT

- Goal
  - deposit, withdraw and getBalance
  - Balance >= 0
- Effects

```
data Acc = Deposit Int | Withdraw Int | GetBalance

getBalance :: [Acc] → () → (Int, Maybe Acc)
getBalance hist _ =
  let res = sum [x | Deposit x ← hist]
            - sum [x | Withdraw x ← hist]
  in (res, Nothing)

withdraw :: [Acc] → Int → (Bool, Maybe Acc)
withdraw hist v =
  if sel1 $ getBalance hist () ≥ v
  then (True, Just $ Withdraw v)
  else (False, Nothing)
```

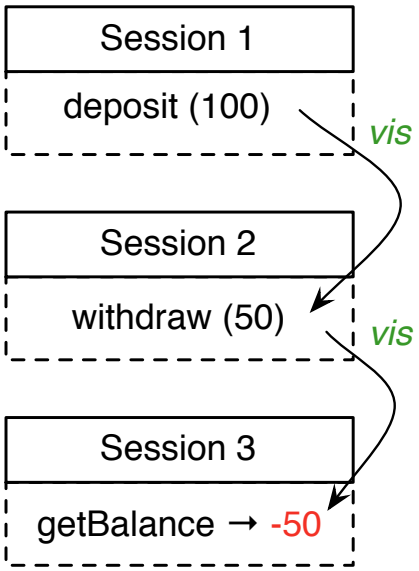
Bank Account Contracts

- Balance >= 0
  - *Any two withdraw operations must be totally ordered*

Effect of current operation

$$\psi_w(\hat{\eta}) = \forall(a : \text{Withdraw}).\ sameobj(a, \hat{\eta}) \Rightarrow vis(a, \hat{\eta}) \vee vis(\eta, a) \vee a = \hat{\eta}$$

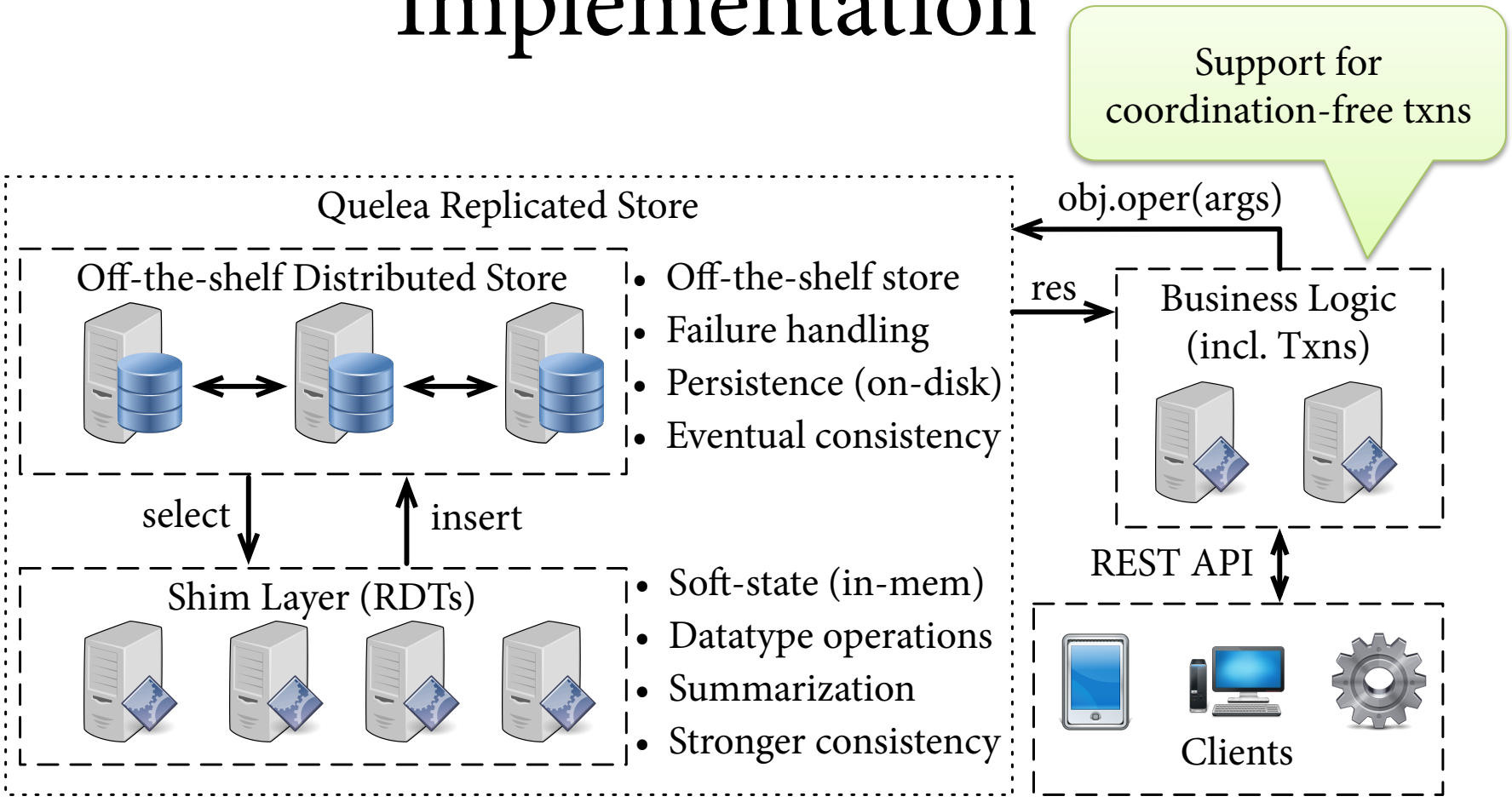
  - *A get balance operation witnessing a withdraw must witness all its visible deposits*



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

$$\psi_d(\hat{\eta}) = \text{true}$$

Implementation



Evaluation: Classification

Operation Classes

Transaction Classes

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 (ebay clone)	640	6	14	2	1	4	2	0
Microblog (twitter clone)	659	5	13	6	1	6	3	1

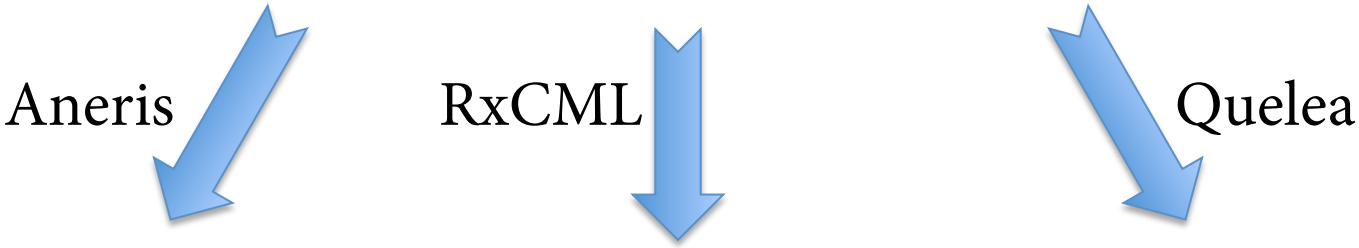
- Performance evaluation
  - Amazon EC2 + Cassandra cluster + Quelea shim layer
- Bank account
  - deposit → EC, withdraw → SC, getBalance → CC
  - Compared to all operations tagged SC, Quelea had
    - 1DC → **40%(139%)** lower(higher) latency(throughput)
    - 2DC → **86%(618%)** lower(higher) latency(throughput)

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

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

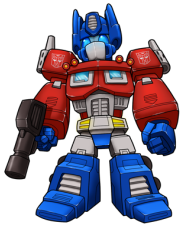
SQL Constraints

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

Contract Inference

Quelea Contracts

Declarative Consistency!



Quelea

Optimistic Concurrency

Strong Consistency

Programming model

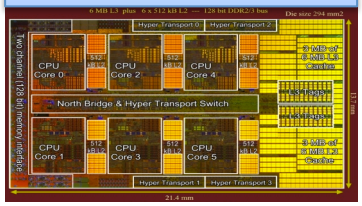
Backends

Shared Mem, Fences, Locks, Cond Var

Distr. Mem, SMC, RDMA, MPB

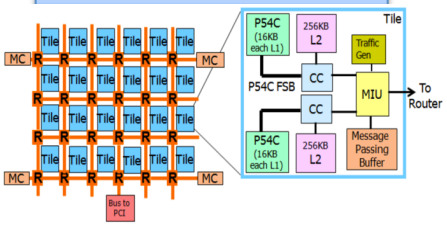
KV interface, Vector clocks, Consensus

Traditional VM



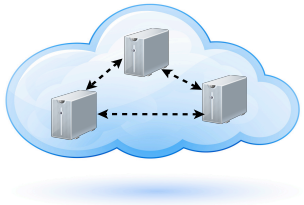
CC multicore

Aneris



Non-CC multicore

Cassandra, Riak, DynamoDB



Geo-distributed compute cluster

Thank you!