# Logging

“We therefore consider this bottleneck as the most dangerous to future scalability”

# ARIES

- Canonical recovery algorithm
- Decouples data writing from log writing

- Write ahead logging
- Restoring database to pre-crash
- Undoing uncommitted txns

# 3 Components

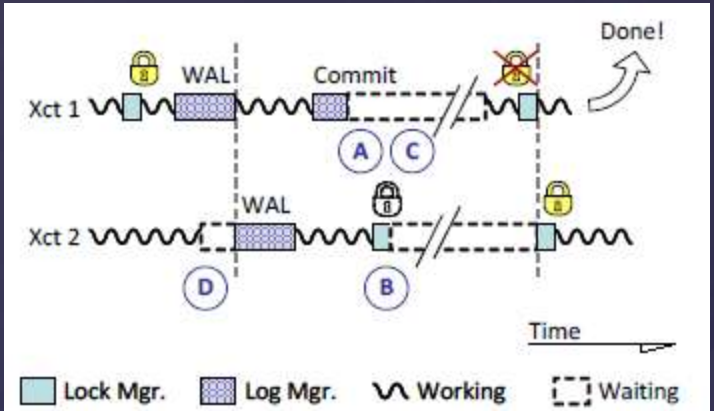
# Multiple Crashes

- Create an additional log while undoing transactions to recover multiple crashes

# Aether

# Motivation

- OLTP txns are small and frequent
- More cores → more contention
- Other bottlenecks disappearing




- I/O delay
- Lock contention
- Schedule overhead
- Log buffer contention

# Convoluted Drawing

# Tradeoffs suck.

Speed + security rocks.

- Critical path
- Decouple to reduce contention
- Commit ≠ return

# Key Ideas

- Critical path
- Decouple to reduce contention
- Commit ≠ return

# Key Ideas

# I/O Delay and Scheduling

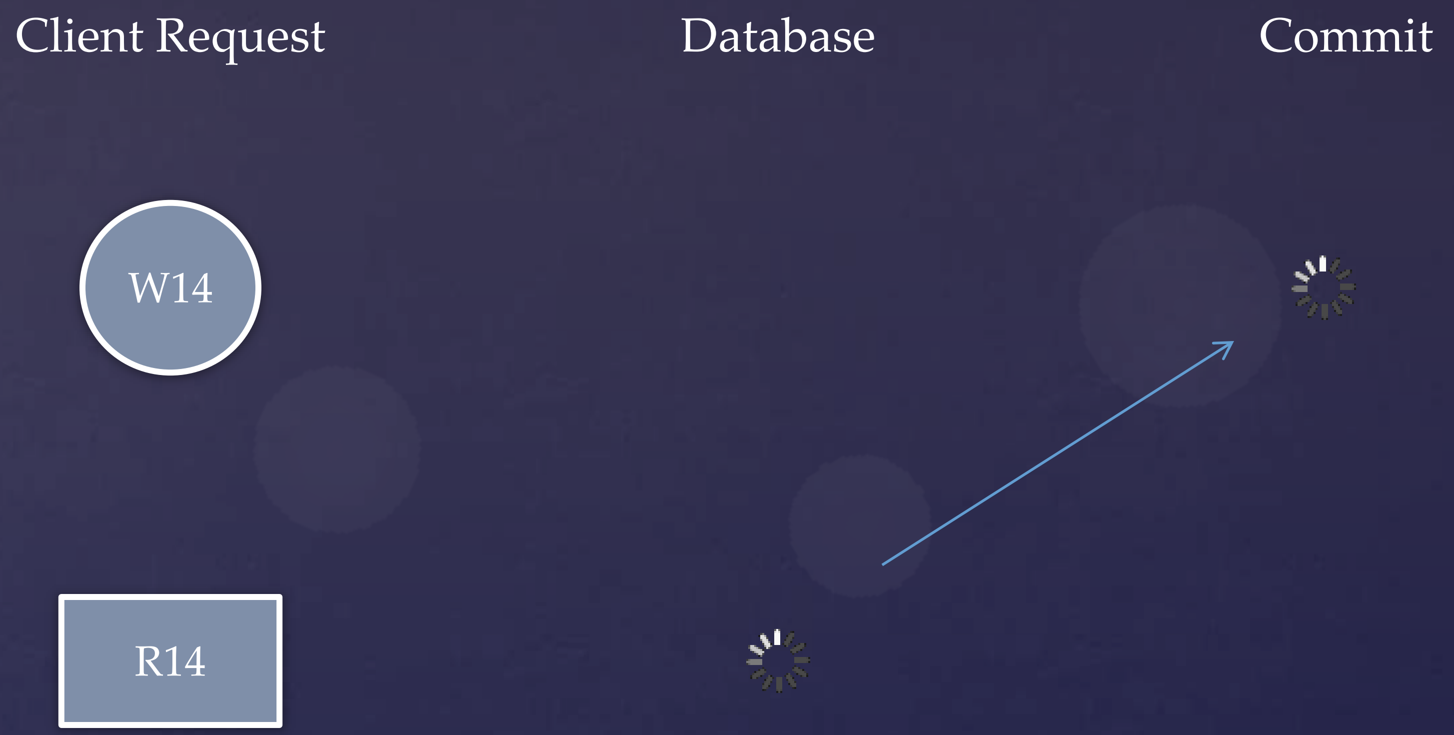
- Release locks on commit
- Track dependencies
- Abort and rollback if necessary

# Early Lock Release

Client Request | Database | Commit

W14

R14



Client Request
Database
Commit
W14
R14

- Decouples transaction commit from scheduling
- Keeps threads busy

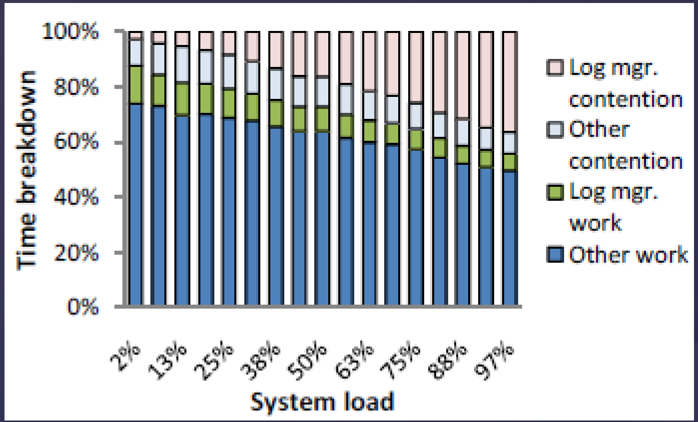
# Flush Pipelining

- Allowing other transactions to proceed speculatively + providing the threads to actually execute them rivals asynchronous commit, but SAFELY

# Combining the Two

# Questions?

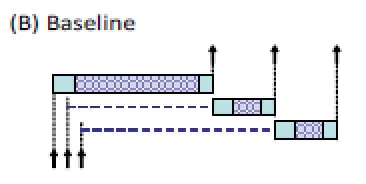
# Scalable Log Buffer




- High core count + high load → bottleneck in the log buffer

# Log Buffer Bottleneck

# The Problem

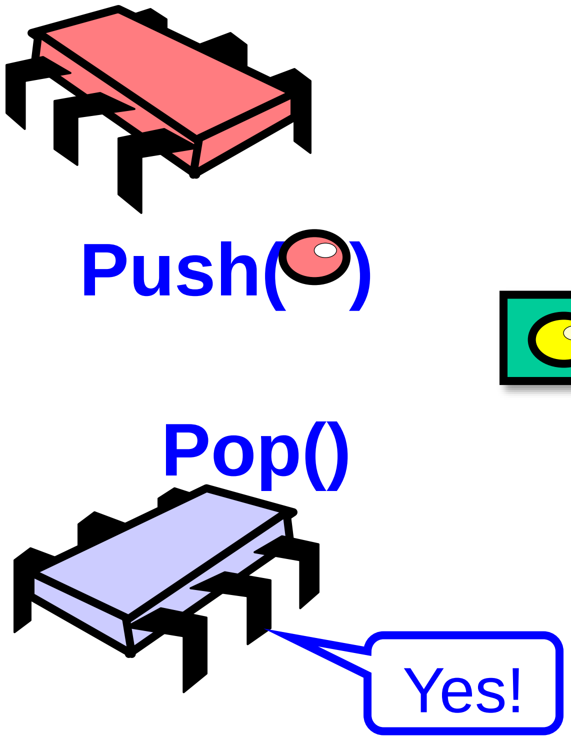


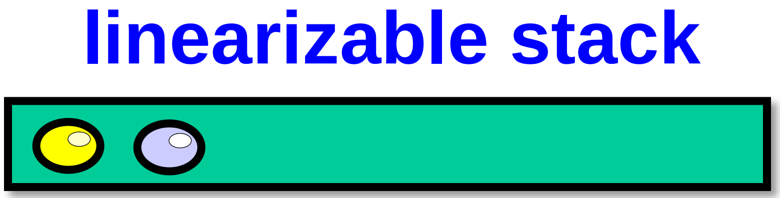

- To write to the log, threads
  - Acquire space in the buffer
  - Fill space
  - Release buffer space for writing

- Group logs together to write to the buffer at once
- Use a consolidation array as a backoff structure

# Solution C

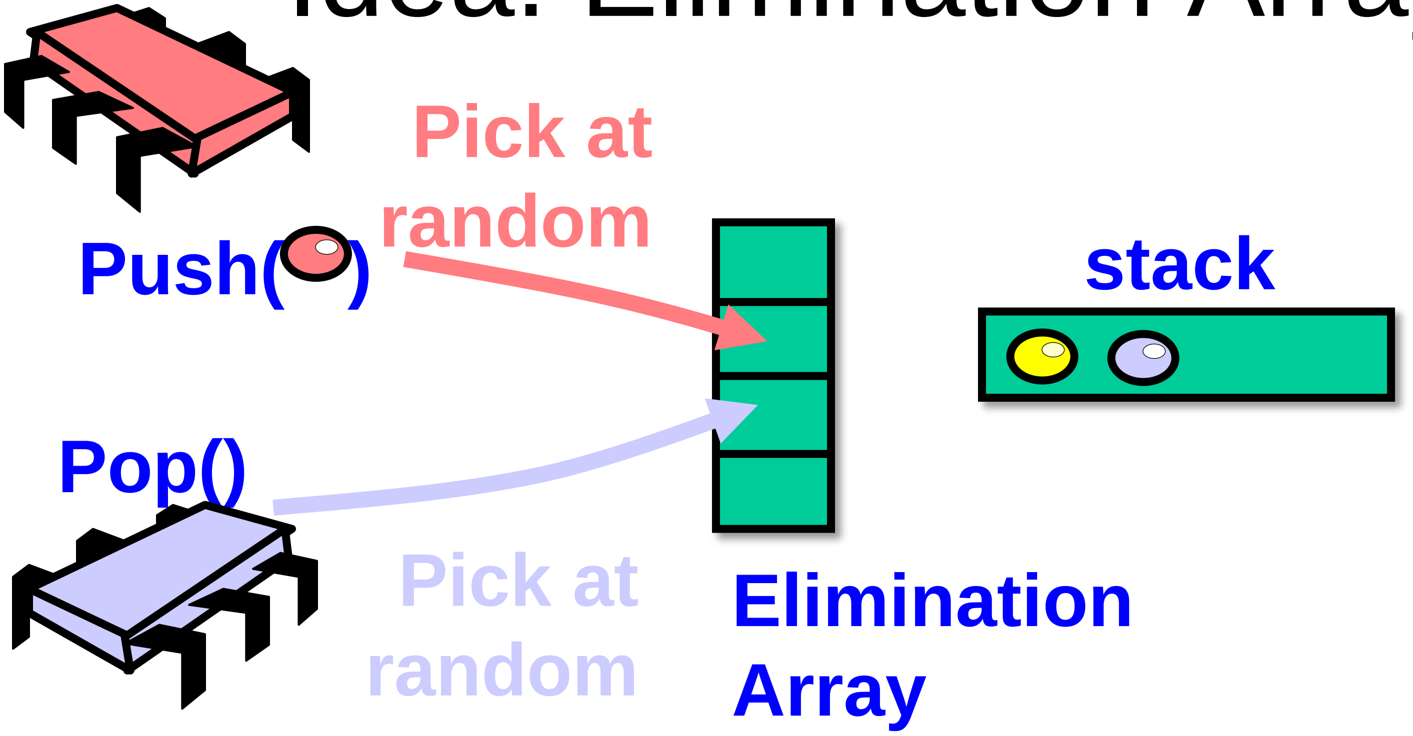
# Observation



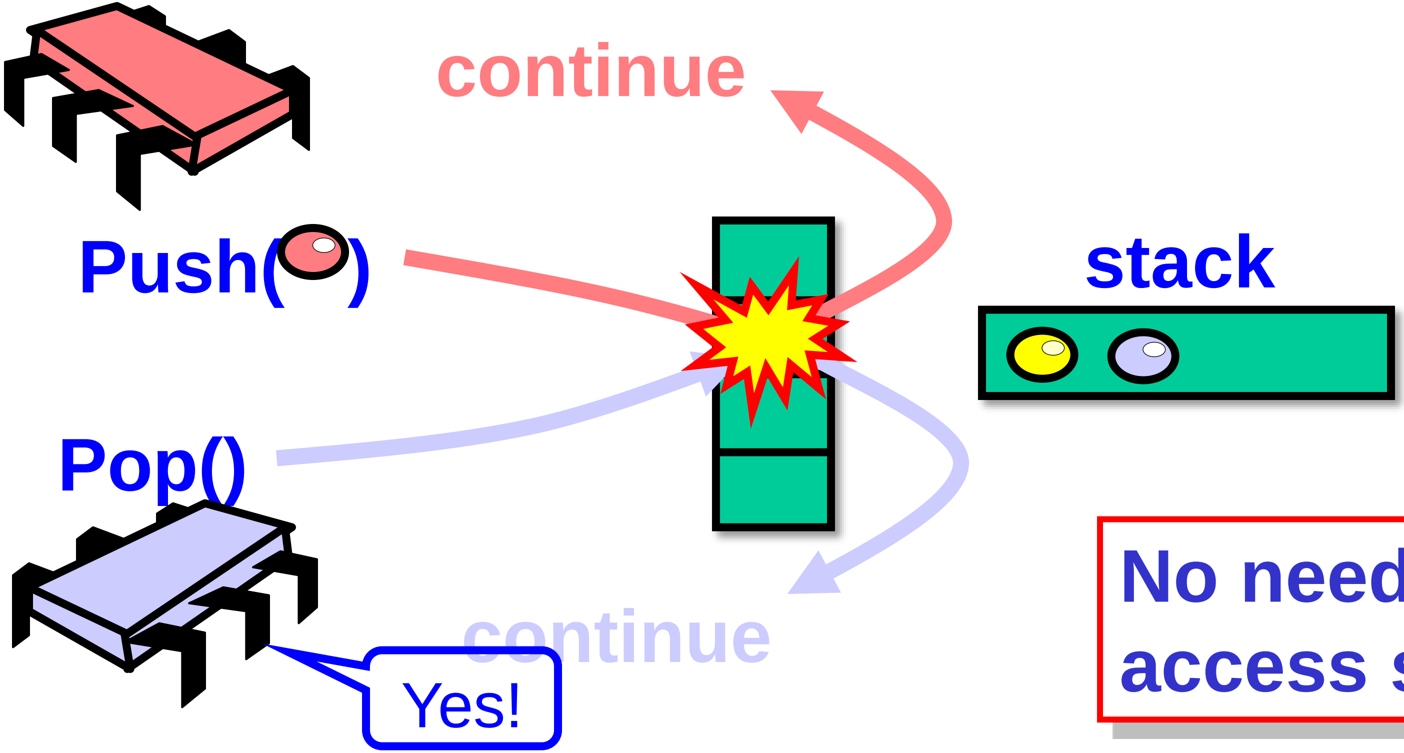




**After an equal number of pushes and pops, stack stays the same**

# Idea: Elimination Array

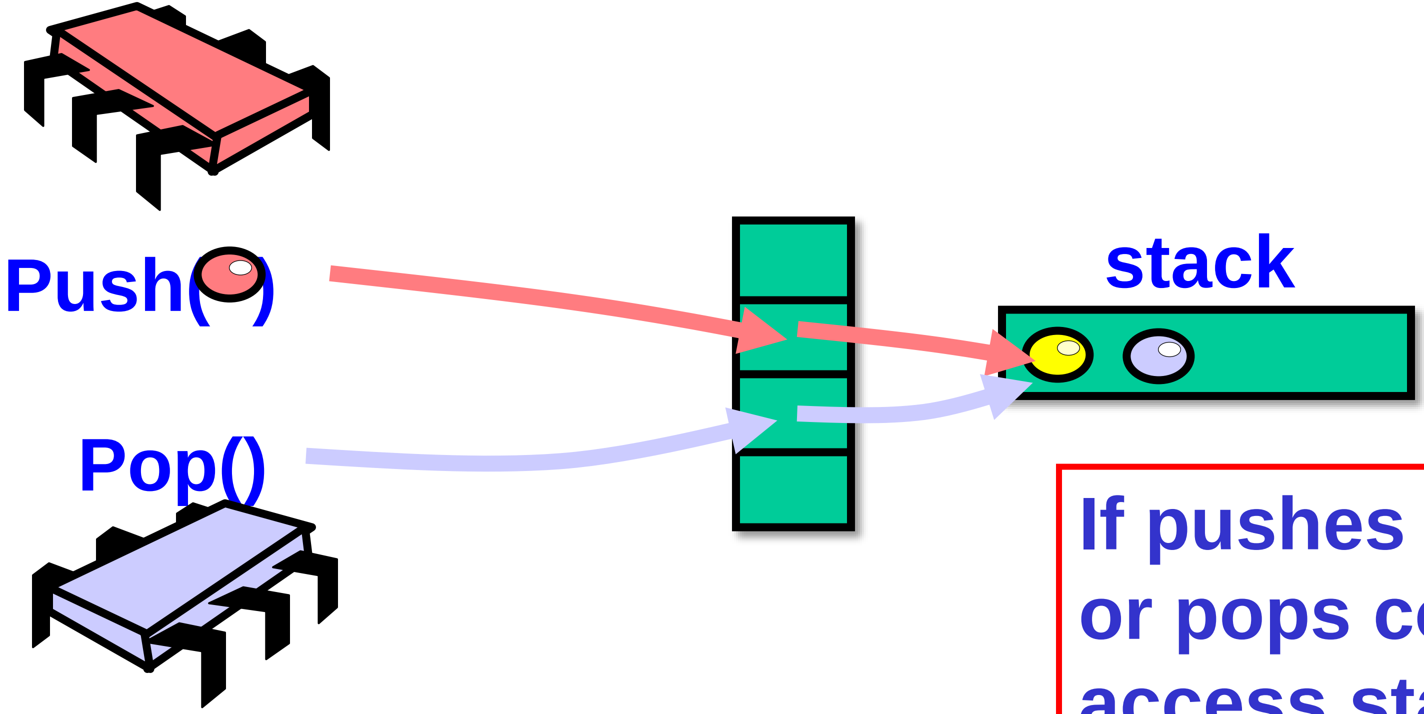


# Push Collides With Pop




**No need to access stack**

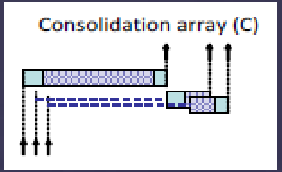
# No Collision




If pushes collide or pops collide access stack

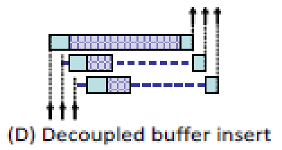
# Elimination-Backoff Stack

- Lock-free stack + elimination array
- Access Lock-free stack,
  - If **uncontended**, apply operation
  - if **contended**, back off to elimination array and attempt elimination



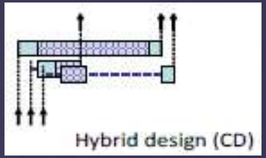

- Consolidation array combines updates rather than eliminating them
- Group leader does all the work
- Great in theory as well as practice

# Consolidation Array




- Buffer fill is not inherently serial
- Must release regions in proper order

# Decoupled Buffer Fill

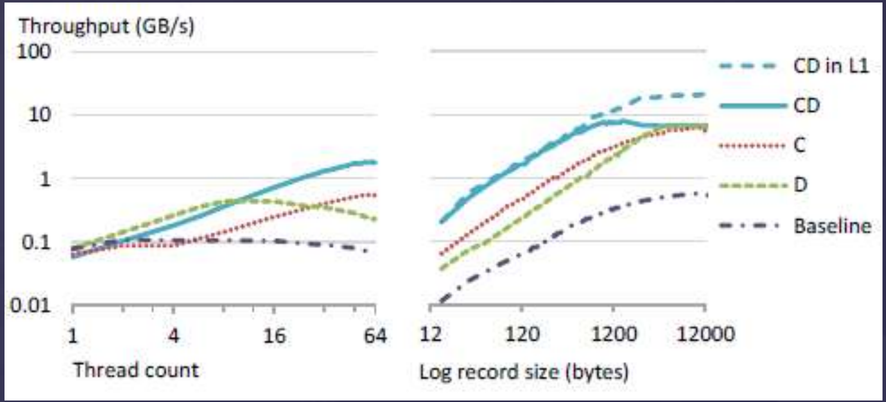



- Hybrid approach
- Consolidate and fill buffer in parallel

# Two > One

# Quick Summary

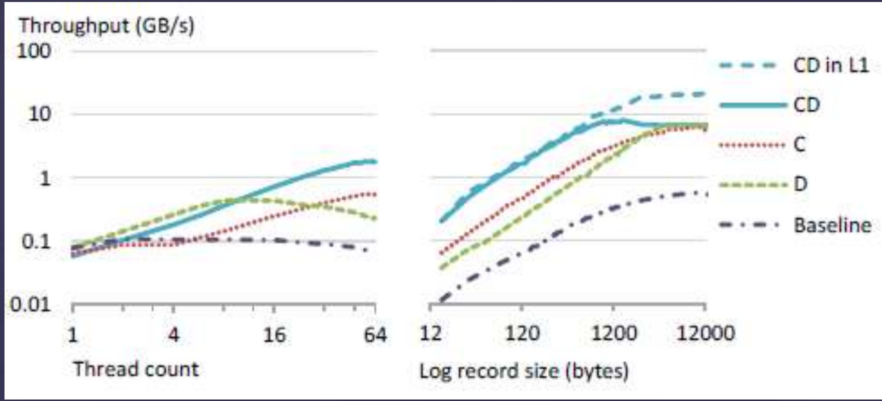
# Does it actually work?




- Left shows record size average held to 120B constant
- Right shows thread count constant (64)

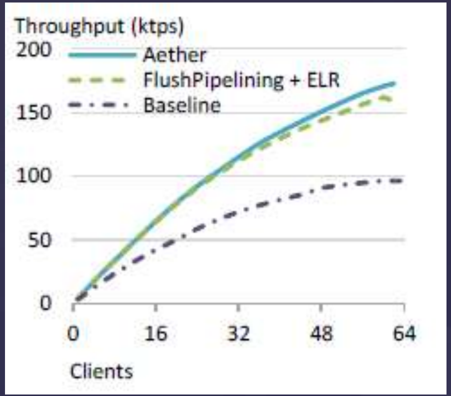
# Experimental Results

# How good?




- Left shows record size average held to 120B constant
- Right shows thread count constant (64)

# Experimental Results




- Flush pipelining + ELR is most important
- Log buffer contention become increasingly important as core-counts grow

# Overall Picture

# Further Optimization

- Delegation can prevent the problem with varying record size
- Threads wake the next in line
- More robust, but performance penalty in the normal case

# In summary…

- Distributed logging
- Higher level (txn + parameters)
- Different (serial excn per thread)
- Txns strongly ordered

# H-Store WAL

- Sharing the log (one log per node)
- Group commit

# H-Store WAL

# Checkpointing

- Natural points of consistently
- At least once per second
- Consistent (not fuzzy)

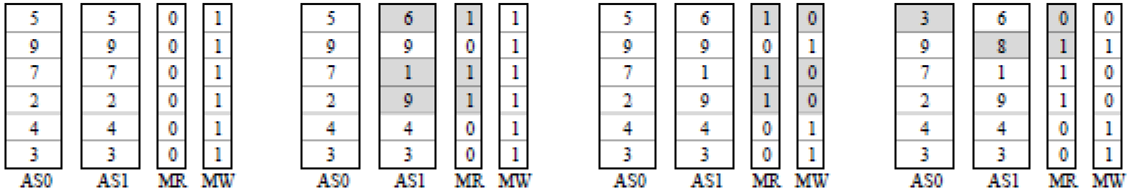
# Frequently Consistent

# Requirements

- Low overhead
- Uniform overhead
- Fast recovery

- Synchronously copies state
- Asynchronously writes to disk
- No checkpoint-specific work

# Naïve-Snapshot




(a) At the beginning of time, $AS_0$ and $AS_1$ contain the same information

(b) During the first checkpoint period, some updates from the Mutator are applied

(c) The state right after switching to the second checkpoint period

(d) In the second checkpoint period, the Mutator applies additional updates

**Figure 1: Wait-Free Zigzag Example**

- Maintain untouched copy of every word for duration of checkpoint
- Bits determine which copy to use

# Wait-Free Zigzag

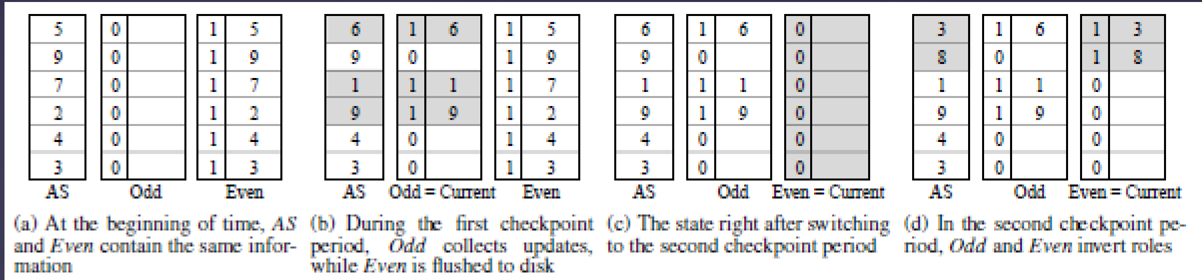



**Figure 2: Wait-Free Ping-Pong Example**

- Maintains extra version of app state
- Swing pointer rather than reset all
- Cache-friendly interleaved version

# Wait-Free Ping-Pong

## Uniform overhead

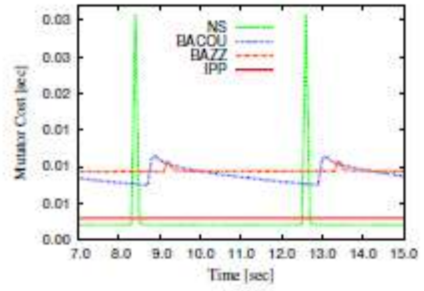



**Figure 9: Latency: 1,280K updates/sec**

## Low overhead

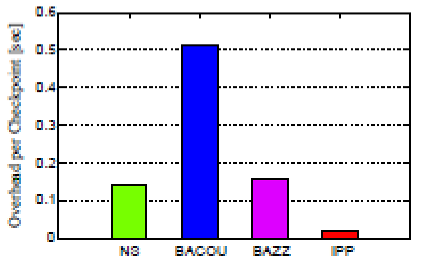



**Figure 10: MMO: Overhead**

## High throughput

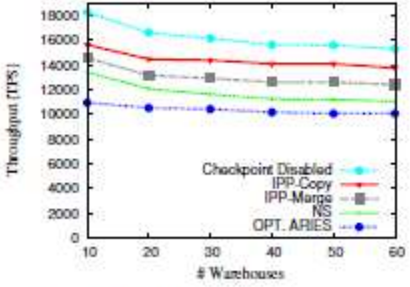



**Figure 12: TPC-C Throughput**

# Results