Logging

“We therefore consider this bottleneck as the most dangerous to future scalability”
 Canonical recovery algorithm
 Decouples data writing from log writing

ARIES
3 Components

- Write ahead logging
- Restoring database to pre-crash
- Undoing uncommitted txns
Create an additional log while undoing transactions to recover multiple crashes

Multiple Crashes
Aether
Motivation

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

- I/O delay
- Lock contention
- Schedule overhead
- Log buffer contention
Tradeoffs suck.
Speed + security rocks.
Key Ideas

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

- Critical path
- Decouple to reduce contention
- Commit ≠ return
I/O Delay and Scheduling
Early Lock Release

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

High core count + high load → bottleneck in the log buffer
To write to the log, threads
- Acquire space in the buffer
- Fill space
- Release buffer space for writing

The Problem
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

Yes!
Idea: Elimination Array

Push()  Pop()

Pick at random

Elimination Array

stack
Push Collides With Pop

Push(  )

Pop()

continue

continue

continue

stack

No need to access stack

Yes!

Yes!
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
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?
Experimental Results

Left shows record size average held to 120B constant
Right shows thread count constant (64)
How good?
Experimental Results

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

- Flush pipelining + ELR is most important
- Log buffer contention become increasingly important as core-counts grow
Delegation can prevent the problem with varying record size
Threads wake the next in line
More robust, but performance penalty in the normal case

Further Optimization
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
Mount untouched copy of every word for duration of checkpoint

Bits determine which copy to use

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

Wait-Free Ping-Pong
Results

Uniform overhead

Low overhead

High throughput

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

Figure 10: MMO: Overhead

Figure 12: TPC-C Throughput