Using lightweight formal methods to validate a key-value storage node in Amazon S3

By James Bornholt, Rajeev Joshi, Vytautas Astrauskas, Brendan Cully, Bernhard Kragl, Seth Markle, Kyle Sauri, Drew Schleit, Grant Slatton, Serdar Tasiran, Jacob Van Geffen, Andrew Warfield

Presented by Andrey Satarin, @asatarin
February, 2022

Outline

- Introduction and ShardStore
- Validating a Storage System
- Conformance Checking
- Checking Crash Consistency
- Checking Concurrent Executions
- Other Properties
- Experience and Lessons

Introduction and ShardStore

ShardStore and S3

- The core of S3 are storage node servers
- ShardStore — new key-value storage node
    - 40k lines of code in Rust
    - Crash consistency and concurrency in the implementation
    - Slowly rolling out to replace previous version

Validation goals

- Functional API correcteness

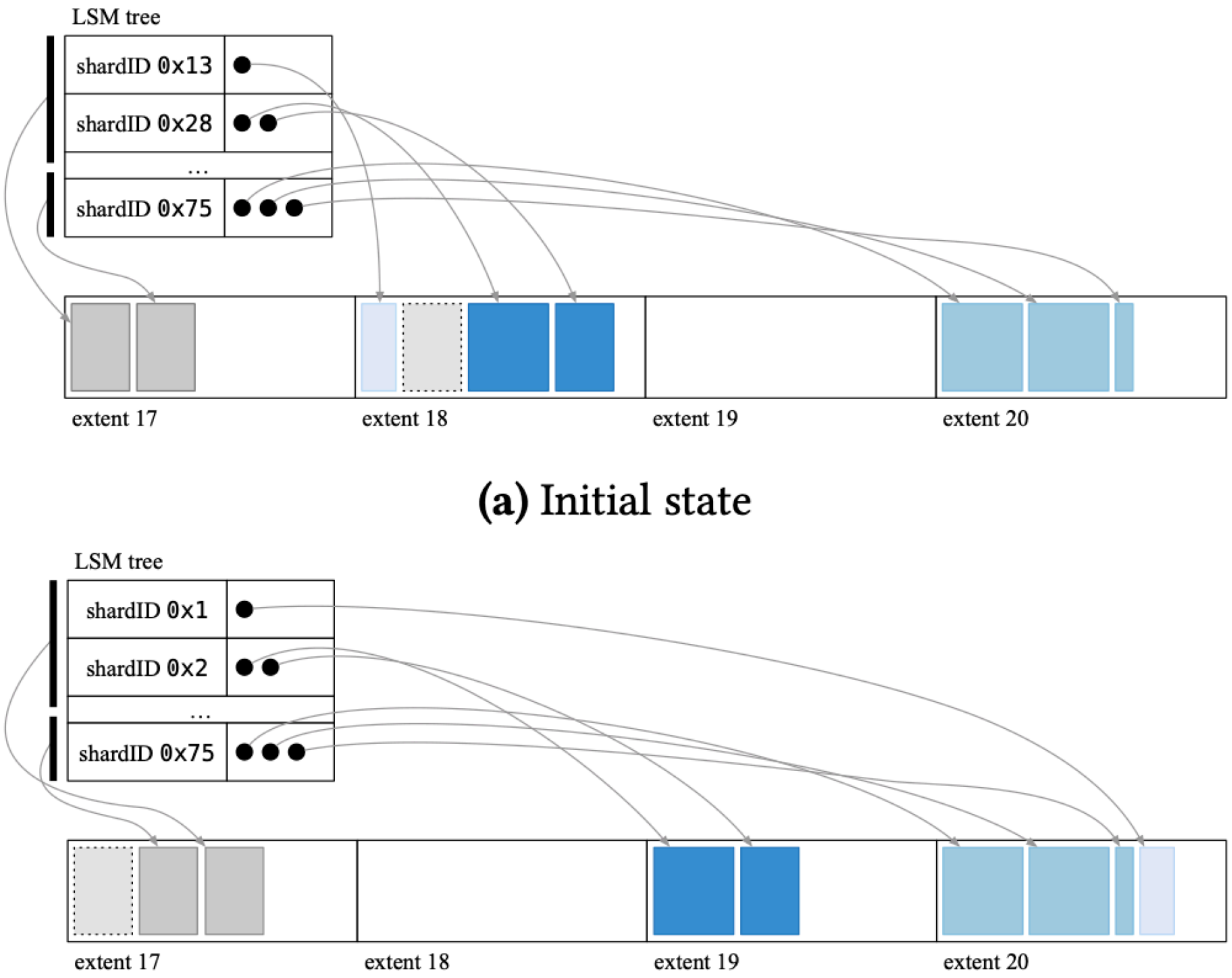
- Crash consistency of on disk data

- Concurrent correctness of API calls and background tasks

Soundness-correctness trade-off — willing to accept weaker guarantees than formal methods

ShardStore

- Log-Structured Merge Tree (LSM)
- Data in chunks, chunks in extents
- More than one log complicates crash consistency
- Garbage collection (GC) in the background



(a) Initial state

(b) After reclamation of extent 18 and LSM-tree flush

Figure 1. ShardStore's on-disk layout

Validating a Storage System

Properties

- Focus on durability and consistency
- Performance and availability is out of scope
- Additional safety properties — undefined behavior, bounds checking, etc

Results must outlive involvement of formal methods experts and be supported by development team in the future

=> lightweight approach to formal methods

Three views on durability

		Section
Sequential	Crash-free	“4 Conformance Checking”
Sequential	With crashes	“5 Checking Crash Consistency”
Concurrent	Crash-free	“6 Checking Concurrent Executions”
Concurrent	With crashes	Out of scope

Reference model

- Executable specification with the same interface in Rust
- 1% of the size of the implementation
- For simplicity omits implementation failures (IO, resource exhaustion, etc)
- Also used as a mock for unit tests, to help keep it up-to-date

Conformance Checking

Property-based testing

- Implementation code refines the model
- Argument bias to steer into interesting states
- Default to random selection, only bias if have quantitative evidence of the benefit
- Code coverage to identify blind spots in tests

Failure injection

- Fail-stop crash
    - Covered in “5 Checking Crash Consistency”
- Disk IO error
    - Relax check against the model
- Resource exhaustion
    - Out of scope for property-based testing

Checking Crash Consistency

Write path

Crash consistency is the primary motivation for this effort

Every put operation has three steps:

1. Write chunked data to an extent
2. Write index entry in the LSM tree
3. Update LSM tree metadata to point to new on-disk index data

Dependency graph

- Inspired by soft updates
- IO scheduler respects dependencies
- Next append only issued if dependency is persisted

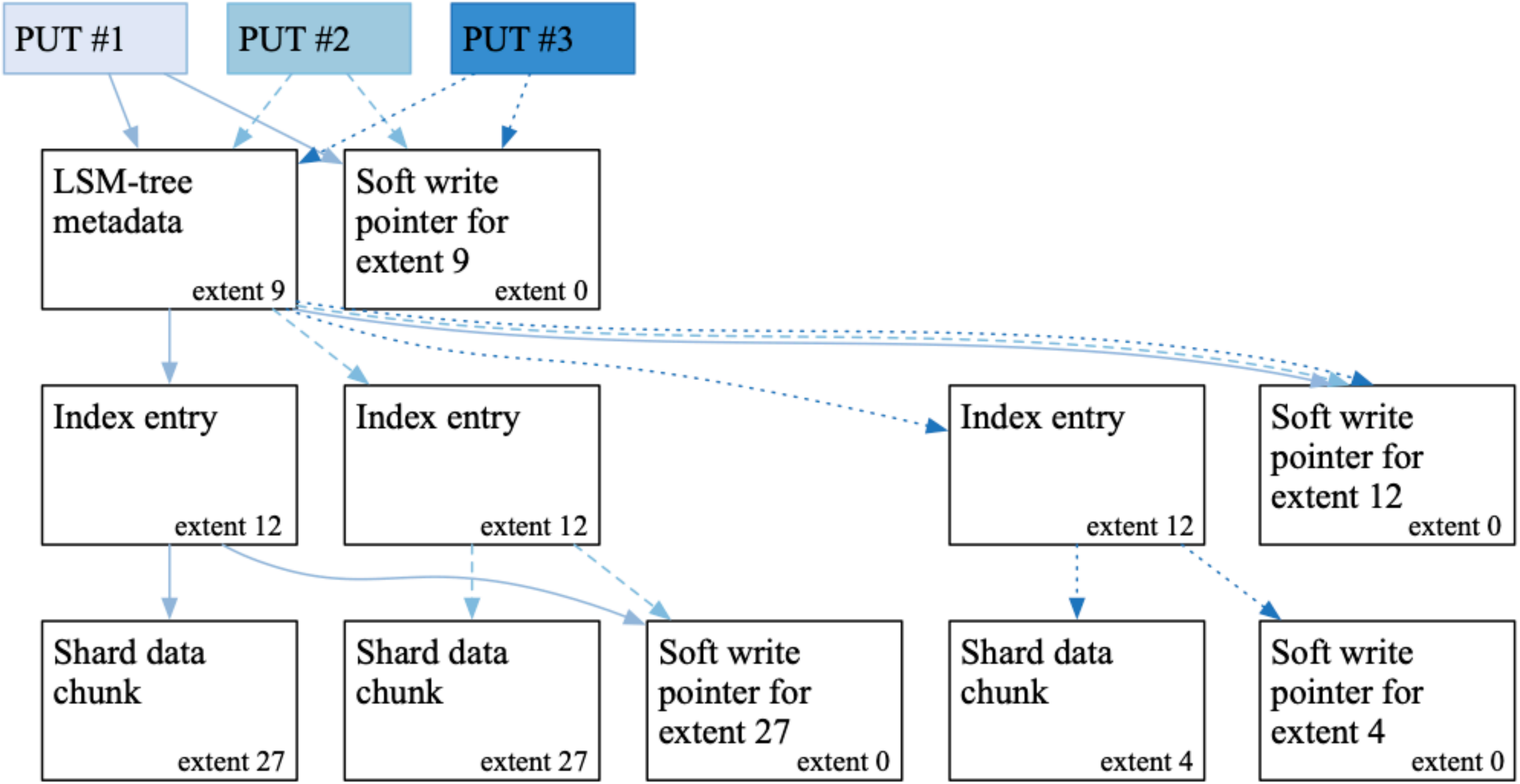


Figure 2 (a) Dependency Graph

Two properties

Persistence — if dependency is persisted, it should be visible after the crash

Forward progress — after non-crash shutdown every operation’s dependency is persistent

Extending property-based testing

- Add new operations to model (e.g. DirtyReboot, IndexFlush)
- Adding block-level crash states proved to be slow and did not uncover new bugs
    - Block level crashes are not used by default

Checking Concurrent Executions

Checking Concurrent Executions

Checking for linearizability

Hand-written harness to validate key properties

- Loom model checker for Rust with sound model checking (slow)
- Shuttle model checker with probabilistic algorithms (faster)

Loom and Shuttle offer a soundness-scalability trade-off

Other Properties

Other properties

- Undefined behavior
  - Miri interpreter for Rust
  - Rust compiler dynamic analysis tools
- Serialization
  - Crux symbolic execution engine to prove panic-freedom
  - Fuzzing

Experience and Lessons

Experience

- Developing the reference model took ~ 2 x 9 months of FM experts
- Non-experts contributed 18% of the model code so far

Benefits:

- Early detection is a great
- Continuous integration/validation keeps the model up-to-date

Limitations

- Hard to evaluate coverage by property-based tests
- Accidental complexity gluing with S3 not covered
- Huge API surface — not everything is covered

Testing distributed systems

Curated list of resources on testing distributed systems
https://asatarin.github.io/testing-distributed-systems/

The end

Contacts

- Follow me on Twitter @asatarin
- https://www.linkedin.com/in/asatarin/
- https://asatarin.github.io/

References

- Self reference for this talk (slides, video, etc)
- "Using lightweight formal methods to validate a key-value storage node in Amazon S3" paper
- Talk at SOSP 2021
- Blog post from Murat Demirbas