A Systematic Approach to System State Restoration during Storage Controller Micro-Recovery

Sangeetha Seshadri[*]

- with Lawrence Chiu[†], and Ling Liu[*]

* Georgia Tech †IBM Almaden Research Center

USENIX FAST 2009

Outline

- Storage system availability.
  - Technical challenges.
- Improving firmware availability through micro-recovery.
  - Log(Lock) architecture for system state restoration.
- Evaluation.
- Conclusions.
- Questions.

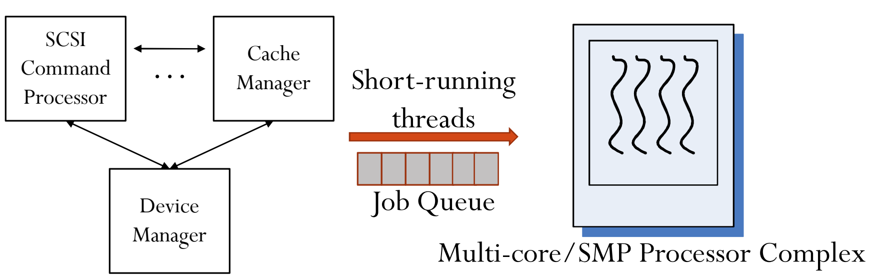
Storage System Availability

- Foundations of modern data centers.
- Extremely high availability expectation.
- Issues:
  - Complex, legacy architectures.
  - Concurrent development, quality assurance processes.
  - Large scale installations – 1000s of components.
  - Multiple applications, different expectations.
    - Failures are the norm, not exception.

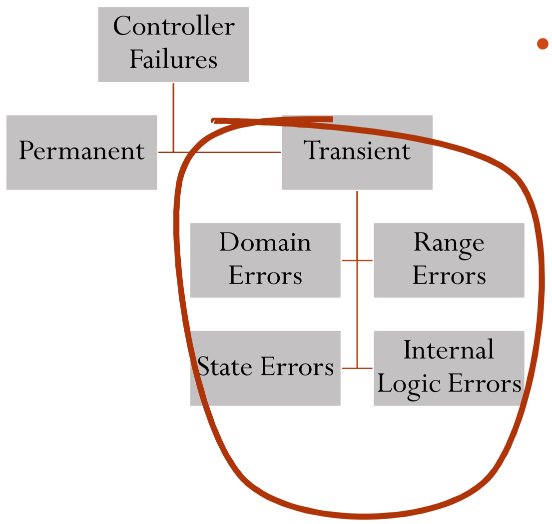
Goal: Improve recovery time in large scale storage systems.
Challenge: Existing failure recovery mechanisms insufficient to deal with scale and complexity.

Storage Controller System Model

- Storage Controllers – RAID, I/O Routing, Error Detection…
- Many interacting components;
- Large number of asynchronous, short-running tasks (~ µsecs).
- Each **task** is executed entirely by one **thread**.

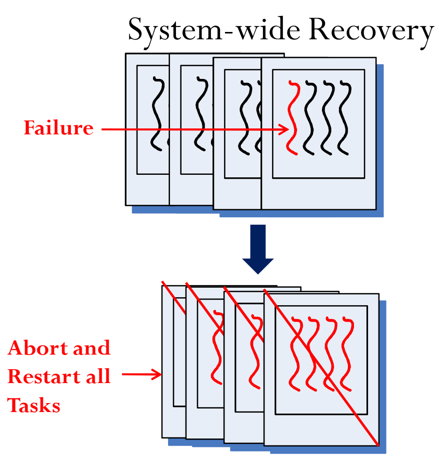


Failure Model



- Focus on service loss.
- Examples:
  - Time-out conditions.
  - Race conditions.
  - Boundary conditions.
  - Insufficient error handling.
  - Queue full condition.
  - Incorrect Linear Redundancy Code (LRC).
  - Unsolicited response from third-party devices.
  - Unknown state caused due to configuration issues.

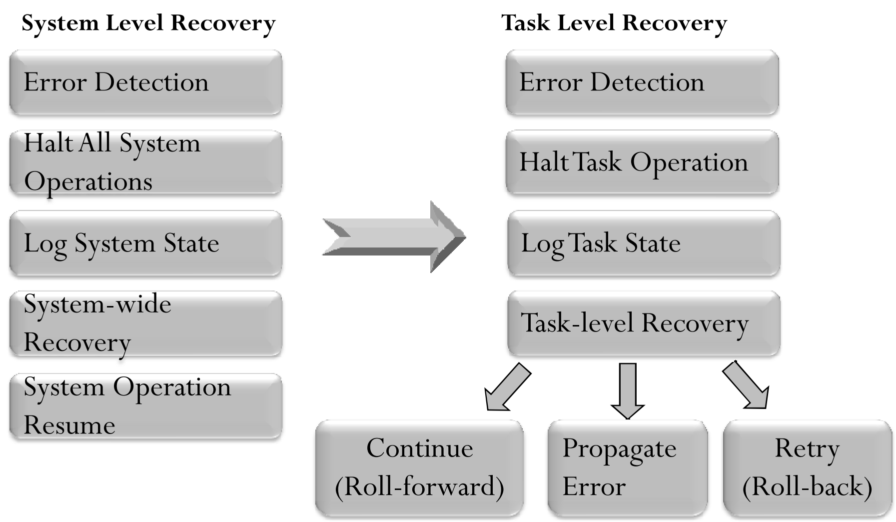
Challenge: Firmware Availability



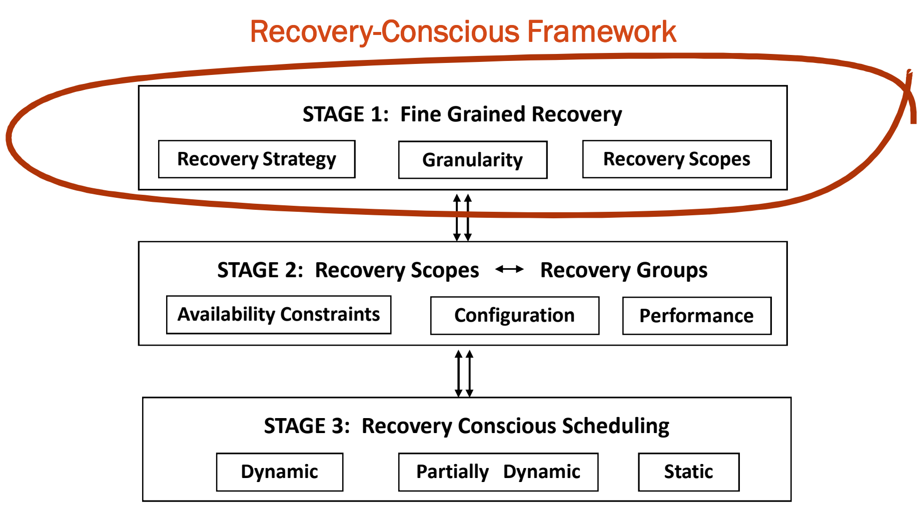
- Failures trigger system recovery.
  - Unavailability ~ 6 seconds (with 8 cores).
  - Does not scale with system size.
- Scalable failure recovery?
  - Legacy architecture. (~ 2M loc)
  - Dynamic dependencies.
  - Complex recovery semantics.
  - Sustain high performance.

Requirements: Retrofittable, dynamic and low overhead.

System-Level vs. Task-Level Recovery



Improving Firmware Availability



State/Resource Dependencies

- Thread interactions:
  - Shared data structures. (Read/Write interactions).
  - Acquiring/releasing resources from a common pool.
  - Interactions with outside world (positioning a disk head, sending response to an I/O) – *Outside world process (OWP)*.
- Capture and account for interactions to ensure
  - State restoration of shared state.
  - Relinquishing shared resources.

Example 1 – Resource Clean Up

```
/* Get cache track to write to cache */
    startSCSICmd();
        └→ processRead();
              └→ getCacheTrack();
                    └→ getTempResource() {
                          ...
                          PANIC
```

- Requires tracking resource ownership.
- Not concerned with reads and writes on the resource.

Example 2 – Dirty Reads

```
R4: /* Update Metadata Location */
    lockWrite( &MetadataLocationLock);
    MetadataLocation = XX;
    unlockWrite( &MetadataLocationLock);
        ...
```

- Metadata location e.g. : checkpoint location.
- If no dirty read, then can undo changes.
- If dirty read has occurred, system-level recovery.

Technical Challenges

- Different contexts have different requirements for recovery.
- For example, threads may care about none or one or more of the following:
  - Resource ownership and clean relinquishing.
  - Dirty reads.
  - Unrepeatable reads.
  - Lost updates.
  - Externally visible actions (such as a response to an user).
- Unlike DB, strict ACID guarantees not required.
- High performance and concurrency is critical.

Need a flexible and lightweight recovery strategy.

Log(Lock) Guided State Restoration

- **Intuition:** Global state protected by locks or similar primitives.
- Lock/Unlock calls can guide understanding of state changes.
- A framework that tracks these calls can alert user to
  - resource ownership,
  - dirty reads, unrepeatable reads and lost updates.
- Incremental approach allows tracking only "interesting entities".

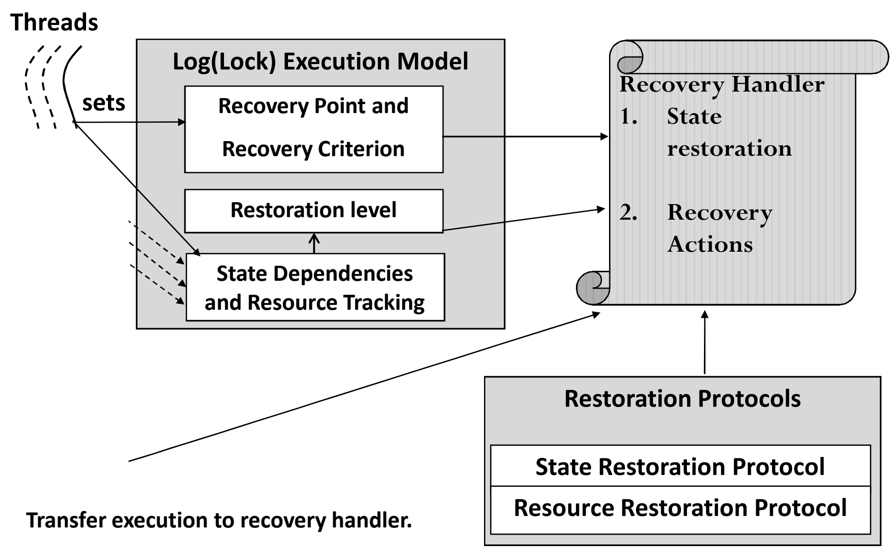
Log(Lock) Overview

- Recoverable thread:
  - Thread which supports micro-recovery.
- Recovery Point p_i:
  - Represents a target starting point for recovery in the event of a failure. Initial system state is a default recovery point.
- Recovery criterion C_i:
  - Associated with a recovery point. Specifies criterion to be satisfied to utilize p_i as a starting point for recovery.
- Restoration Level:
  - Describes failure context.

Recovery Context (Recovery Point p_i, Recovery criterion C_i)

Failure Context (Restoration Level)

Log(Lock) Overview



Deriving Restoration Protocols

- Assume system with only two threads T_1 and T_2
- Let T_1 be the thread that encounters a failure.
- ***W***: *Write*, ***R***: *Read*, ***U***: *Unlock*, ***F***: *Fail*, ***E***: *End*, ***A***: *Acquire*, ***Re***: *Release*
- Events of interest from standpoint of state restoration:
  - Dirty read **(DR)** : $T_1W \rightarrow T_2R \rightarrow T_1F$
  - Lost Update **(LU)**: $T_1W \rightarrow T_2W \rightarrow T_1F$
  - Unrepeatable Read **(UR)**: $T_1R \rightarrow T_2W \rightarrow T_1F$
  - Residual Resources **(RR)**: $T_1R \rightarrow T_1F \wedge T_1U \nrightarrow T_1F$ or
    $T_1W \rightarrow T_1F \wedge T_1U \nrightarrow T_1F$ or
    $T_1A \rightarrow T_1F \wedge T_1\mathrm{Re} \nrightarrow T_1F$
  - Committed Dependency **(CD)**: $T_1W \rightarrow T_2R \rightarrow T_2E \rightarrow T_1F$ or
    $T_1W \rightarrow T_2W \rightarrow T_2E \rightarrow T_1F$ or
    $T_1R \rightarrow T_2W \rightarrow T_2E \rightarrow T_1F$

Recovery Strategies and Context

- Recovery strategies:
  - Single/multi –thread roll-back using a recovery point.
  - Error compensation or roll-forward.
  - System restart (software restart such as warmstart, or hardware restart).
- Restoration Level at instant t, *R(t)*:
  - Failure context.
  - Captures occurrence of events such as DR, LU, UR, RR, CD.
- Recovery point p_i and Recovery Criterion C_i:
  - Recovery context.
  - Specifies the criteria for state to be restored using p_i.
  - Events such as DR, LU, UR, RR, CD that can be handled using p_i.

Resource/State Recovery Protocols

- System state can be restored using recovery point p_i only if *R(t)* meets the recovery criterion C_i on the "residual resources" criterion.
- For single-thread recovery *R(t)* must match C_i .
- If *R(t)* does not meet C_i on read-write conflicts:
  - If event "committed dependency" has occurred, then
    - Only error compensation or system-level recovery possible.
  - Else if "committed dependency" has not occurred
    - Only multi-thread rollback, error compensation or system-level recovery.

Log(Lock) Execution Model

- Log(Lock) maintains the following in main memory:
  - Undo logs: (maintained by developer)
    - Local logs maintained by each recoverable thread.
    - Tracks the sequence of state changes within a single thread.
    - Tracks the creation of recovery points.
    - Tracks resource ownership.
  - ChangeTrack logs: (maintained by the system).
    - Maintained per lock (i.e. per synchronization primitive).
    - Entry made for each lock/unlock call.
    - <Thread#, [Lock|Unlock|Commit], [Read|Write|Commit]>
    - Track concurrent changes.
    - Track commit actions.

Log(Lock) Primitives

- Used by developer to utilize Log(Lock)-based recovery.
  - *startTracking(lock)*
    - Used during normal-path execution.
  - *stopTracking(lock)*
    - Used during normal-path execution.
  - *getRestorationLevel(lock)*
    - Used during failure-recovery in the recovery handler.
  - *getResourceOwnership(lock)*
    - Used during failure-recovery in the recovery handler.

Log(Lock) Undo/Change Track Logs

- **Thread T1:**

  startTracking(*MDataLocationLock);*
  LockWrite (&*MDataLocationLock*);
  mDataLocation = XX;
  UnlockWrite*(&MDataLocationLock)*
  ….

T1 UNDO LOG

timestamp, mDataLocation, oldvalue

- Thread T2:

  …
  LockRead (&*MDataLocationLock*);
  Copy location to local variable.
  UnlockRead*(&MDataLocationLock)*

Global Variables: **CHANGE LOG**

MDataLocationLock →

Evaluation

- Implemented Log(Lock) on enterprise storage controller code with a simulated backend.
- Evaluated Log(Lock) effectiveness and efficiency.
- Highlights:
  - Acceptable overhead & high performance
    - (< 10% impact even while tracking state changes @ 15K times/sec.)
  - Extremely high rate of recovery success (~ 99%) observed.
    - Recovery success: % of time restoration level meets recovery criterion.
  - Significant improvement in recovery time.
    - 35% Throughput drop for a 6 second duration vs 4 seconds downtime.

Experimental Setup

- Enterprise Storage Controller:
  - 4 3.00 GHz Xeon 5160 processors, 12GB memory, IBM MCP Linux.
- Simulating the backend allows control over read/write latencies and setup.
  - 250 LUNS of 100 GB each.
  - Varied Read/Write latencies: 1ms or 20 ms
- Workload – varying read/write %, varying queue depth, varying block sizes.
  - 100% Writes, 50-50% Read-Write, 100% Read.

Metrics

- Efficiency:
  - Impact of Log(Lock) on system performance.
  - Throughput (Iops)
  - Latency (seconds/IO).
- Effectiveness:
  - Ability of Log(Lock) to reduce recovery time.
  - Recovery success.
  - Recovery time.

Methodology

Table 2: State and Resource Access over a 75 minute run with varying workloads

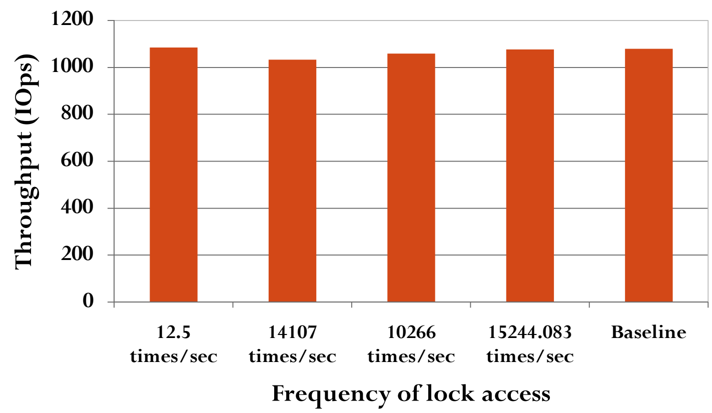
Lock	Contention CPU Cycles	Contention Counter	Number of locks	% contention	Locks/IO
Fiber channel	2654991	578	137196747	4.21293E-06	10.33500111
IO state	219969	76	90122610	8.43296E-07	6.788916609
Resource pool	608103	100	63482290	1.57524E-06	4.782107098
Resource pool state	124965	52	30040757	1.73098E-06	2.262963691
Throttle timer	79848	11	113316	9.7E-05	0.00853607

- Frequent locks $\Longrightarrow$ frequently accessed/modified state.
- Contention $\Longrightarrow$ access by concurrent threads, longer duration of holding locks.

Comparisons

- System-Level Recovery:
  - Reinitializes software, re-drives tasks.
  - No hardware reboot.
- 2-phase locking
  - Commonly used in transactional systems.
  - Locks held for the duration of entire thread.
  - Resulted in lock timeouts and failed to bring system up.

Rate vs Throughput (100% Writes)



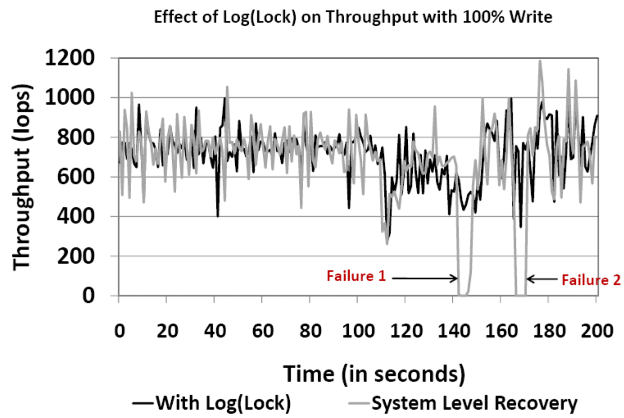
- Acceptable impact on performance.

Recovery Success

Lock	Recovery Criterion	Tracking Calls (times/sec)	#Access (times/sec)	Duration CPU cycles	Recovery Success
Fiber channel	No Residual Resources	3666	15244	20228	100%
IO state	No DR, LU or UR	2500	10266	2894	99.88%
Resource pool	No Residual Resources	10	14107	34642	100%
Resource state	No Residual Resources	5	6675	4806	100%
Throttle timer	No Residual Resources	10	12.59	7258	100%
IO state	No DR, LU or UR	2444	10045	69830	99.38%

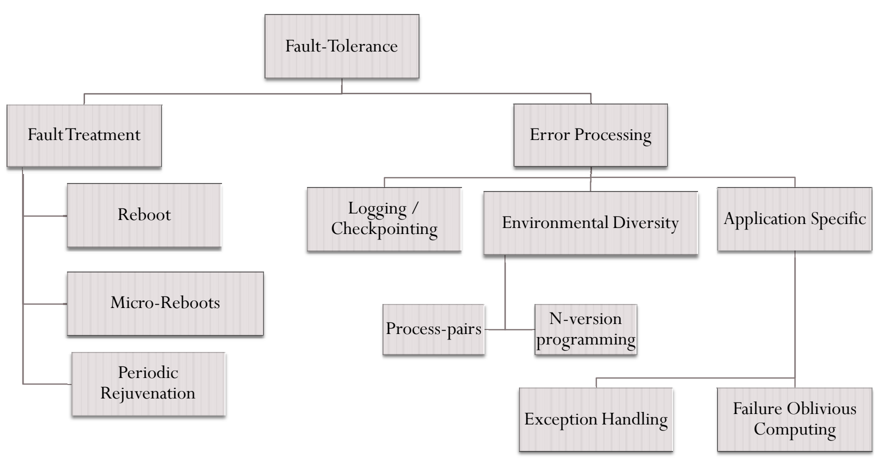
- High recovery success.
- Also due to code architected for high concurrency.

Recovery Time



- 4 seconds downtime reduced to 35% performance impact lasting 6 seconds.

Applicability of Existing Art



Source: Software Fault Tolerance by Kishor S. Trivedi, http://srel.ee.duke.edu/

Conclusion

- Large scale storage systems and services
  - Complex systems, extremely high availability expectations.
  - System-wide recovery processes will not scale.
  - Need scalable and efficient recovery process.
- Contributions:
  - Techniques to perform fine-granularity recovery in legacy systems.
  - Practical and flexible state restoration architecture.
  - Log(Lock)-enabled micro-recovery is effective and efficient.
- Future Work
  - Reduce need for programmer intervention.
  - Evaluate with other highly-concurrent systems.

Questions?

THANK YOU

sangeeta@cc.gatech.edu