EECS 262a
Advanced Topics in Computer Systems
Lecture 8

SOR & LRVM
September 16th, 2022

John Kubiatowicz
Electrical Engineering and Computer Sciences
University of California, Berkeley

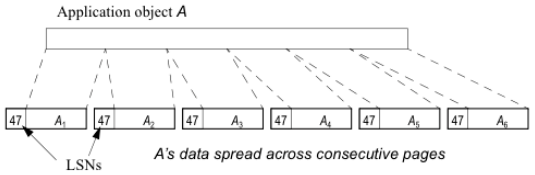
http://www.eecs.berkeley.edu/~kubitron/cs262a

Today's Papers

- Segment-Based Recovery: Write-ahead Logging Revisited
  Sears and Brewer. Appears in *Proceedings of the VLDB Endowment,* Vol 2, No. 1, August 2009, pp 490-501
- Lightweight Recoverable Virtual Memory
  M. Satyanarayanan, Henry H. Mashburn, Puneet Kumar, David C. Steere, and James J. Kistler. Appears in *Proceedings of the 14th ACM Symposium on Operating Systems Principles* (SOSP 1993)

- Thoughts?

Segment-Oriented Recovery



- ARIES works great but is 20+ years old and has some problems:
  - Viewed as very complex
  - No available implementations with source code
  - Part of a monolithic DBMS: can you use reuse transactions for other systems?
  - LSN in the page breaks up large objects (Fig 1), prevents efficient I/O
- DBMS seems hard to scale for cloud computing (except by partitioning)

Original Goals (Stasis)

- Build an open-source transaction system with full ARIES-style steal/no-force transactions
- Try to make the DBMS more "layered" in the OS style
- Try to support a wider array of transactional systems: version control, file systems, bioinformatics or science databases, graph problems, search engines, persistent objects, ...
- Competitive performance
- These goals were mostly met, although the open-source version needs more users and they have only tried some of the usual applications
- Original version was basically straight ARIES
  - Development led to many insights and the new version based on segments

Segment-Oriented Recovery (SOR)

- A segment is just a range of bytes
  - May span multiple pages
  - May have many per page
  - Analogous to segments and pages in computer architecture (but arch uses segments for protection boundaries, SOR uses them for recovery boundaries; in both, pages are fixed size and the unit of movement to disk)
- Key idea – change two of the core ARIES design points:
  - Recover segments not pages
  - No LSNs on pages
    - » ARIES: LSN per page enable exactly once redo semantics
    - » SOR: estimate the LSN conservatively (older) => at least once semantics [but must now limit the actions we can take in redo]

The Big Four Positives of SOR

1) Enable DMA and/or zero-copy I/O
2) Fix persistent objects
3) Enable high/low priority transactions (log reordering on the fly)
4) Decouple the components; enable cloud computing versions

1) Enable DMA and/or Zero-Copy I/O

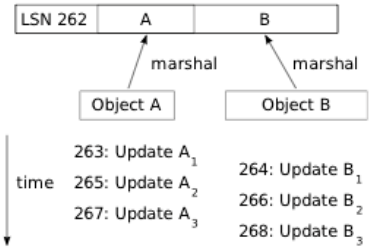
- No LSNs per page mean that large objects are contiguous on disk
- No "segmentation and reassembly" to move objects to/from the disk
- No need to involve the CPU in object I/O (very expensive); use DMA instead
- Any downsides?

2) Fix Persistent Objects

- Problem: consider page with multiple objects, each of which has an in memory representation (e.g., a C++ or Java object)
  - Suppose we update object A and generate a log entry with LSN=87
  - Next we update object B (on the same page), generate its log entry with LSN=94, and write it back to the disk page, updating the page LSN
  - This pattern breaks recovery: the new page LSN (94) implies that the page reflects redo log entry 87, but it does not.
- ARIES "solution":
  - Disk pages (in memory) must be updated on every log write; this is "write through caching" – all updates are written through to the buffer manager page
- SOR solution:
  - There is no page LSN and thus no error
  - Buffer manager is written on cache eviction – "write back caching". This implies much less copying/ marshaling for hot objects.
  - This is like "no force" between the app cache and the buffer manager (!)

3) Enable high/low Priority Transactions (log reordering on the fly)

```
pin page
get latch
newLSN =
    log.write(redo)
update pages
page LSN = newLSN
release latch
unpin page
```



- ARIES: "pin" pages to keep them from being stolen (short term), "latch" pages to avoid race conditions within a page
  - Subtle problem with pages: must latch the page across the call to log manager – in order to assign an LSN atomically with the page write
  - Not possible to reorder log entries at all
  - In theory, two independent transactions could have their log entries reordered based on priority or because they are going to different log disks (or log servers)

3) Enable high/low Priority Transactions (log reordering on the fly)

- SOR does not need to know LSN at the time of the page update
  - Just need to make sure it is assigned before you commit, so that it is ordered before later transactions; this is trivial to ensure – simply use normal locking and follow WAL protocol
- High priority updates: move log entry to the front of the log or to high priority "fast" log disk
- Low priority updates go to end of log as usual
- SOR has no page LSN and in fact no shared state at all for pages => no latch needed

4) Decouple the components; enable cloud computing versions

- SOR decouples three different things:
  - App <-> Buffer manager: this is the write-back caching described above – only need to interact on eviction, not on each update
  - Buffer manager <-> log manager: no holding a latch across the log manager call – log manager call can now by asynchronous and batched together
  - Segments can be moved via zero-copy I/O directly, with no meta data (e.g. page LSN) and no CPU involvement – simplifies archiving and reading large objects (*e.g.*, photos)

Physiological Redo (ARIES)

- Redos are applied exactly once (using the page LSN)
- Combination of physical and logical logging
  - Physical: write pre- or post-images (or both)
  - Logical: write the logical operation ("insert")

SOR Recovery

- Redos are physical
- Normal undos are like redos and set a new LSN (does not revert to the old LSN)
  - Wouldn't work given multiple objects per page!
- To enable more concurrency, do not undo structural changes of a B-Tree (or other index)
  - Instead of a physical undo, issue a new logical undo that is the inverse operation
  - Enables concurrency because we only hold short locks for the structural change rather than long locks (until the end of the transaction)
- Slotted pages:
  - Add an array of offsets to each page (slots), then store records with a slot number and use the array to look up the current offset for that record
  - Allows changing the page layout without any log entries

SOR Redo

- Redos may applied more than once; we go back farther in time than strictly necessary
- Redos must be physical "blind writes" – write content that do not depend on the previous contents
- Undos can still be logical for concurrency
- Slotted page layout changes require redo logging

Core SOR Redo Phase

- Periodically write estimated LSNs to log (after you write back a page)
- Start from disk version of the segment (or from snapshot or whole segment write)
- Replay all redos since estimated LSN (worst case the beginning of the truncated log)
  - Even though some might have been applied already
- For all bytes of the segment:
  - Either it was correct on disk and not changed,
  - Or it was written during recovery in order by time (and thus correctly reflects the last log entry to touch that byte)

Hybrid Recovery

- Can mix SOR and traditional ARIES
- Some pages have LSNs, some don't
- Can't easily look at a page and tell!
  - All the bytes are used for segments
- Log page type *changes* and zero out the page
  - Recovery may actually corrupt a page temporarily until it gets the type correct, at which point it rolls forward correctly from the all zero page
- Example of when to use:
  - B-Trees: internal nodes on pages, leaves are segments
  - Tables of strings: short strings good for pages especially if they change size; long strings are good for segments due to contiguous layout

Summary

- 3 key features:
  - Open-source implementation that decouples components while improving performance and efficiency
    - » https://code.google.com/p/stasis/
  - Preserve ARIES semantics (compatibility is a bonus benefit)
  - Enable QoS for transactions
- Some flaws:
  - "Research" implementation: SOR is only one component of a DBMS system, not a complete practical system implementation
  - Write-back experiment: simple datatype, no error bars, y-axis scale, …

Extra: Why are there LSNs on pages?

- So that we can atomically write the timestamp (LSN) with the page
- Problem: page writes aren't actually atomic anymore
- Solution: make them atomic so that ARIES still works
  - Write some bits into all sectors of a page (8K page = 16 512B sectors); compare those bits with a value store somewhere else. No match => recover the page (but may match and be wrong). Assumes sectors are written atomically, which is reasonable.
  - Write a checksum for each page (including the LSN) and check it when reading back the page
- Both solutions impact I/O performance some

SOR Approach

- Checksum each segment (or group of segments if they are small)
- On checksum failure, can replay last redo, which can fix a torn page (and then confirm using the check-sum); if that doesn't work go back to old version and roll forward
- Blind writes can fix corrupted pages since they do not depend on its contents

Is this a good paper?

- What were the authors' goals?
- What about the evaluation / metrics?
- Did they convince you that this was a good system /approach?
- Were there any red-flags?
- What mistakes did they make?
- Does the system/approach meet the "Test of Time" challenge?
- How would you review this paper today?

BREAK

Lightweight Recoverable Virtual Memory (LRVM)

- Eppinger Thesis (in Related Work [9]):
  - RVM ... poses and answers the question "What is the simplest realization of essential transactional properties for the average application?" By doing so, it makes transactions accessible to applications that have hitherto balked at the baggage that comes with sophisticated transactional facilities.
- Answer:
  - Library implementing No-Steal, No-Force virtual memory persistence, with manual copy-on-write and redo-only logs

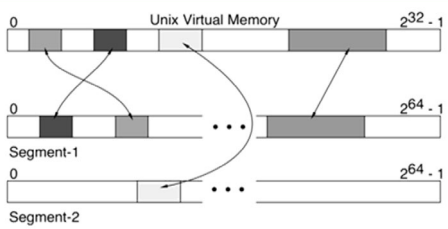
LRVM

- Goal:
  - Allow Unix applications to manipulate persistent data structures (such as the meta data for a file system) in a manner that has clear-cut failure semantics
- Existing solutions
  - Such as Camelot, were too heavyweight, cumbersome to use, and imposed programming constraints
  - Wanted a 'lite' version of these facilities that didn't also provide (unneeded) support for distributed and nested transactions, shared logs, etc.
- Solution
  - A library package that provides only recoverable virtual memory

Lessons from Camelot

- Camelot had an "ugly" object and process model – its componentization led to multiple address spaces and lots of expensive IPC (~600x local call on RISC)
  - Too tied to Mach and complicated log management for multiple applications
- Heavyweight facilities impose additional onerous programming constraints – kernel threads vs. user-level threads
- Size and complexity of Camelot and its dependence on special Mach features resulted in maintenance headaches and lack of portability
  - The former of these two shouldn't be an issue in a "production" system
- However, note that the golden age of CMU Systems learned a lot from the sharing of artifacts: Mach, AFS, Coda...
  - A lot of positive spirit in this paper

LRVM Architecture



- Focus on metadata not data
  - Like Lowest-level consistency for journaling file system, etc..
- External data segments: think mmap of a file into memory
  - But writes do not go back to backing store automatically
- Processes map regions of data segments into their address space – No aliasing/overlap allowed, immediate copy from backing store to main memory

LRVM Architecture (cont'd)

- Changes to RVM are made as part of transactions
- Call *set_range* operation before modifying part of a region
  - Allows a copy of the old values to be made so that they can be efficiently restored (in memory) after an abort
- No-flush commits trade weaker permanence guarantees (bounded persistence) in exchange for better performance
  - Where might one use no-flush commits of transactions?
- No-restore flag == no explicit abort => no undo (ever) except for crash recovery means no need to log old state
- In situ recovery – application code doesn't worry about recovery!

LRVM Architecture (cont'd)

- To ensure that the persistent data structure remains consistent even when loss of multiple transactions ex-post-facto is acceptable – Example: file system
- No isolation, no serialization/concurrency control, no handling of media recovery (but these capabilities can be added)
  - Very systems view; not popular with DB folks...
- Provided capabilities can be used to implement a funny kind of transaction checkpoint
  - Might want/need to flush the write-ahead log before the transaction is done, but be willing to accept "checkpointing'" at any of the (internal) transaction points

LRVM Architecture (cont'd)

- *flush*: force log writes to disk
  - p6: "Note that atomicity is guaranteed independent of permanence." – this confuses DB folks
  - In this context, means that all transactions are atomic (occur *all* or *nothing*), but the most recent ones might be forgotten in a crash – *i.e.*, they change at the time of the crash from "all" to "nothing" even though the system told the caller that the transaction committed
  - Flush prevents this, which only happens with no-flush transactions
- *truncate*: reflect log contents to external data segments and truncate the log

LRVM Implementation

- Log only contains new values because uncommitted changes never get written to external data segments
  - No undo/redo operations
  - Distinguishes between external data segment (master copy) and VM backing store (durable temp copy) => can STEAL by propagating to VM without need for UNDO (since you haven't written over the master copy yet)
- Crash recovery and log truncation: see paper section 5.1.2
- Missing "set range" call is *very* bad – nasty non-deterministic bugs that show up only during some crashes...
  - Might use static analysis to verify set range usage...
- Write-ahead logging: log intent then do idempotent updates to master copy – keep retrying update until it succeeds

LRVM Optimizations

- Intra-transaction: Coalescing *set_range* address ranges is important since programmers have to program defensively
- Inter-transaction: Coalesce no-flush log records at (flush) commit time
  - *E.g.*, for a multi-file copy to same directory only care about last update (subsumes earlier ones)

LRVM Performance

- Beats Camelot across the board
- Lack of integration with VM does not appear to be a significant problem as long as:
  - Ratio of RVM / Physical memory is small (<70%, but <40% best for good locality and <25% for poor locality applications)
  - VM pageouts are rare
  - Slower startup is acceptable (read entire RVM instead of on-demand)
- Log traffic optimizations provide significant savings (20-30%), though not multiple factors savings

3 Key Features about the Paper

- Goal:
  - A facility that allows programs to manipulate persistent data structures in a manner that has clear-cut failure semantics
- Original experience with a heavyweight, fully general transaction support facility led to a project to build a lightweight facility that provides only recoverable virtual memory (since that is all that was needed for the above-mentioned goal)
- Lightweight facility provided the desired functionality in about 10K lines of code vs 60K Camelot, with significantly better performance and portability

A Flaw

- The paper describes how a general facility can be reduced and simplified in order to support a narrower applications domain
- Although it argues that the more general functionality could be regained by building additional layers on top of the reduced facility, this hasn't been demonstrated
- Also allows for “persistent errors” – errors in a set range region aren’t fixable by reboot... they last until fixed by hand
- A lesson:
  - When building a general OS facility, pick one (or a very few) thing(s) and do it well rather than providing a general facility that offers many things done poorly

Distributed Transactions (background)

- Two models for committing a transaction:
  - One-phase: used by servers that maintain only volatile state. Servers only send an end request to such servers (*i.e.,* they don't participate in the voting phase of commit)
  - Two-phase: used by servers that maintain recoverable state. Servers send both vote and end requests to such servers
- 4 different kinds of votes may be returned:
  - Vote-abort
  - Vote-commit-read-only: participant has not modified recoverable data and drops out of phase two of commit protocol
  - Vote-commit-volatile: participant has not modified recoverable data, but wants to know outcome of the transaction
  - Vote-commit-recoverable: participant has modified recoverable data

Is this a good paper?

- What were the authors’ goals?
- What about the evaluation/metrics?
- Did they convince you that this was a good system/approach?
- Were there any red-flags?
- What mistakes did they make?
- Does the system/approach meet the “Test of Time” challenge?
- How would you review this paper today?