

LRVM: Satyanarayanan, et al.

Thesis (in Related Work)

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.

Technical detail: figure out how to truncate log efficiently.

Big Questions

- What is the right interface for persistent data?
- Balance: range of applicability (guarantees), simplicity of architecture, control over performance and behavior.

Their Background

Need not be *your* background!

- Camelot had a yucky object and process model. Its componentization led to lots of IPC. It had poorly tuned log truncation. Was perhaps too much of an embrace of Mach.
 - 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.

Details

- 4 locations of data: RAM+pagefile (x2!), external data segment, log.
 - segment loader is an add-on to ensure that segments end up in the same physical memory each time)
- **map**: maps a segment and a VM range
- **begin_transaction**: sets up an XID. Can choose "no_restore" restore_mode for "wont-abort"! Then no UNDOs are supported. When is this a good idea?
- **set_range**: warn RVM that you're about to write a range. RVM sets up UNDO info by copying the range about to be modded (into VM part of log! not log on disk! Essentially shadow paging.)
- **end_transaction**: Can optionally set commit_mode = no-flush. Flush the WAL later, on demand. "Bounded Persistence". When is this a good idea? Brewer notes that this can support a kind of "savepoint" mechanism.

p6: "Note that atomicity is guaranteed independent of permanence." Huh?

- **flush**: flushes log tail for committed but no-flush transactions
- **truncate**: makes sure committed transactions are reflected in external data segments.

LOG

No-undo, physical redo. I.e. "NO-STEAL, NO-FORCE".

- **What happens on set_range?** Old data copied in VM (log tail), not to external data segment. (No need to bother for no_restore transactions.)
- On commit, new_value records replace old-value records (in the VM log tail). Then these new_value log records are flushed to persistent log.
- On recovery, construct a tree (why?) of changes (REDOs) per segment. Then apply changes to external data segment. At end of recovery, write a status block pointing to an empty log. (Recovery trees must fit completely in memory, right?)
- **Log truncation:** why needed?
 - Epoch Truncation: use the recovery scheme on old committed stuff while doing FW processing. Problem: too much log traffic, bursty.
 - Incremental truncation: push VM pages for old log records to external data segment. Lock pages that are from uncommitted transactions, so we don't have to do undo logging.
 - Page vector: (dirty bit, uncommitted refcount). "Dirty" = committed changes. Refcount incr on set_range, decr on commit or abort.
 - FIFO queue of page mod descriptors, max one per page, which include log offset of 1st record refing that page.
 - **Algorithm:** take 1st descriptor off queue, write out the specified pages, delete descriptor, move the log head to the offset of the next descr. (effectively reclaims the log space corresponding to the previous record).

Optimizations

- Intra-xact: coalesce multiple overlapping set_ranges from the same xact. Note this is common due to defensive programming and libraries.
- Inter-xact: for no-flush only. Postpone overlapping flushes and let the last one only be forced to the log (deleting the earlier log records).

Reflection

- They used it for a bunch of stuff, including persistent storage of disconnected updates to Coda files, and persistent cache history. (hoard database)
- Most common user error is to forget set_range.

Evaluation

- Code size vs. Camelot: 5x-10x smaller.
- Do you pay a penalty for not integrating tightly with VM? TPC-A style benchmark with a wraparound audit trail. (But we thought this wasn't a DBMS thing?)
 - Beats Camelot all over the range.
 - Upshot: with good locality, you can have 40% of memory be active recoverable; with poor locality make it 25%.
 - Camelot suffers from overaggressive log truncation, which forces it to write and rewrite random pages in the external data segment.
- Scalability in terms of # of concurrent something (never specified in paper), which manifests itself in size of recoverable memory relative to physical. Pretty much a measure of the overhead of IPC in Camelot vs inefficient page fault handling in RVM (which is never explicitly measured or discussed in the paper)
- Optimizations buy 20-30% each under their workloads.

Mention of Avalon, but no discussion of the extensive OODB lit.

Related Work

Why is VM simple? Isn't read/write simple? Why don't people use transactions more? Why don't they use persistence more?

Conclusion

"The term "lightweight" in the title of this paper connotes two distinct qualities. First, it implies ease of learning and use. Second, it signifies minimal impact upon system resource usage. RVM is indeed lightweight along both these dimensions. A Unix programmer thinks of RVM in essentially the same way he thinks of a typical subroutine library, such as the stdio package."