**Advanced Topics in Operating Systems, CS262a**

**Prof. Eric A. Brewer (with help from Joe Hellerstein)**

# Lightweight Recoverable Virtual Memory

**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.

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 heavy-weight. 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:

- Overhead of multiple address spaces and constant IPC between them was significant.
- Heavyweight facilities impose additional onerous programming constraints.
- 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.)

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.

Architecture:

- Focus on metadata not data
- External data segments: think mmap of a file into memory
- Processes map regions of data segments into their address space. No aliasing allowed.
- Changes to RVM are made as part of transactions.
- set_range operation must be called 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 in exchange for better performance. Where might one use no-flush commits of transactions?
- No-restore == no explicit abort => no undo (ever) except for crash recovery
- in situ recovery (application code doesn’t worry about recovery!)

- 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, no handling of media recovery (but these can be added). Very systems view; not popular with DB folks...

These 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.

- flush: force log writes to disk.
  - p6: "Note that atomicity is guaranteed independent of permanence." This confuses DB folks. In this context, it means that all of the 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.

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.
- 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 (retry the update until it succeeds)

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.

Performance:

- Beats Camelot across the board.
- Lack of integration with VM does not appear to be a significant problem as long as ration of Real/Physical memory doesn't grow too large.
- Log traffic optimizations provide significant (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, 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.

Background on distributed transactions:

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