

GEORGIA INSTITUTE OF TECHNOLOGY

---

# Recoverable Virtual Memory

## CS 6210 Project 3 Report

---

Paras Jain, Manas George

## 1 PROJECT GOAL

The aim of this project is to implement a recoverable virtual memory system. This system allows users to manage persistent memory using a transaction-based API. This mechanism allows clients to create applications with persistent and consistent data structures (e.g. a database).

## 2 API

1. `rvm_init`: initializes the log files and directory for the VM system
2. `rvm_map`: maps a segment file on disk to area in memory
3. `rvm_unmap`: unmaps a segment from memory
4. `rvm_destroy`: cleans up the backing store for a segment
5. `rvm_begin_trans`: creates a transaction
6. `rvm_about_to_modify`: notifies library about incipient modification to memory segments
7. `rvm_commit_trans`: commits the transaction to disk
8. `rvm_abort_trans`: undo all modifications during a transaction
9. `rvm_truncate_log`: shrink the log file and write log data
10. `rvm_verbose`: control log verbosity

## 3 RVM DESIGN AND IMPLEMENTATION

Upon calling `rvm_init`, a directory is created if it does not exist along with a log file. Relevant segment list data structures are created as well. When a mapping is created with `rvm_map`, a segment file is created if it does not exist. This file stores the memory segment in raw binary format. When creating a transaction, a new transaction ID is created. This transaction ID is checked to prevent multiple transactions from editing the same segment. Calling `rvm_about_to_modify` creates a range record. Upon commit, the ranges are actually written out to the log file. Finally, in `truncate`, we read each item from the log file and then apply the changes and write back the data to disk.

This architecture keeps the code simple, which helps ensure that our implementation is more likely to be correct. Give

### 3.1 RVM DATA STRUCTURES

## 4 ANALYSIS OF DESIGN TRADEOFFS

Our implementation is simple which keeps the code relatively bug-free. All library code lives in `rvm.c` in several interface methods, outlined in the API section. Total implementation is on the order of about 350 LOC. This has positive benefits for the library, notably security and verifiability. However, this led to some potential underutilized performance optimizations.

For example, segments are stored in a single global segment list. Operations on this segment list are on the order of  $O(n)$  as each access requires traversing the list to find the relevant segment metadata struct. Using a list keeps implementation simpler and more secure. The list uses `sys/queue.h` which is a very well tested and robust list implementation using compiler macros. However, we could use a hashmap datastructure (e.g. Google sparse hash) which would have a constant access overhead. Looking through the examples, we decided that most implementations did not use a large enough number of segments to make such an optimization worthwhile. Given that our RVM library is intended to be linked directly into the client application as compared to a system-wide service, segment lists would be scoped to the context of a single application and therefore should not be an issue for the kinds of clients using our library. We therefore avoid needing to link another third-party library into our code.

There are several optimizations that could further improve performance of the application. The log currently is read and written to using `fread`. This interface is performant but incurs the cost of a system call for every `range_t` item in the log. A faster approach would be to `mmap` the transaction log which would batch reads to a system-page granularity. This would also have the added benefit of potentially allowing for page-sharing between clients in the future. It is also likely recently written entries in the transaction log will be read shortly after - `mmap` would benefit from this as those pages would likely still be cached in memory.