Lightweight Recoverable Virtual Memory

M. SATYANARAYANAN, HENRY H. MASHBURN, PUNEET KUMAR,
DAVID C. STEERE, and JAMES J. KISTLER
Carnegie Mellon University

Recoverable virtual memory refers to regions of a virtual address space on which transactional guarantees are offered. This article describes RVM, an efficient, portable, and easily used implementation of recoverable virtual memory for Unix environments. A unique characteristic of RVM is that it allows independent control over the transactional properties of atomicity, permanence, and serializability. This leads to considerable flexibility in the use of RVM, potentially enlarging the range of applications than can benefit from transactions. It also simplifies the layering of functionality such as nesting and distribution. The article shows that RVM performs well over its intended range of usage even though it does not benefit from specialized operating system support. It also demonstrates the importance of intra- and inter-transaction optimizations.

Categories and Subject Descriptors: D.4.2 [**Operating Systems**]: Storage Management—*virtual memory*; D.4.5 [**Operating Systems**]: Reliability—*fault tolerance*; D.4.8 [**Operating Systems**]: Performance—*measurements*; H.2.2 [**Database Management**]: Physical Design—*recovery and restart*; H.2.4 [**Database Management**]: Systems—*transaction processing*.

General Terms: Design, Experimentation, Measurement, Performance, Reliability

Additional Key Words and Phrases: Camelot, Coda, logging, paging, persistence, RVM, scalability, throughput, truncation, Unix

1. INTRODUCTION

How simple can a transactional facility be, while remaining a potent tool for fault tolerance? Our answer, as elaborated here, is a user-level library with minimal programming constraints, implemented in about 10K lines of mainline code and no more intrusive than a typical runtime library for input-output. This transactional facility, called RVM, is implemented without special-

This work was sponsored by the Avionics Laboratory, Wright Research and Development Center, Aeronautical Systems Division (AFSC), U.S. Air Force, Wright-Patterson AFB, Ohio, under contract F33615-90-C-1465, ARPA order no. 7597. James Kistler is now affiliated with the DEC Systems Research Center, Palo Alto, CA 94301.

Authors' addresses: School of Computer Science, Carnegie Mellon University, 5000 Forbes Avenue, Pittsburgh, PA 15213; J. Kistler, DEC Systems Research Center, Palo Alto, CA 94301.

ized operating system support and has been in use for over two years on a wide range of hardware from laptops to servers.

RVM is intended for Unix applications with persistent data structures that must be updated in a fault-tolerant manner. The total size of those data structures should be a small fraction of disk capacity, and their working set size must easily fit within main memory.

This combination of circumstances is most likely to be found in situations involving the metadata of storage repositories. Thus RVM can benefit a wide range of applications from distributed file systems and databases to object-oriented repositories, CAD tools, and CASE tools. RVM can also provide runtime support for persistent programming languages. Since RVM allows independent control over the basic transactional properties of atomicity, permanence, and serializability, applications have considerable flexibility in how they use transactions.

It may often be tempting, and sometimes unavoidable, to use a mechanism that is richer in functionality or better integrated with the operating system. But our experience has been that such sophistication comes at the cost of portability, ease of use, and more onerous programming constraints. Thus RVM represents a balance between the system-level concerns of *functionality* and *performance* and the software engineering concerns of *usability* and *maintenance*. Alternatively, one can view RVM as an exercise in minimalism. Our design challenge lay not in conjuring up features to add, but in determining what could be omitted without crippling RVM.

We begin this article by describing our experience with Camelot [Eppinger et al. 1991], a predecessor of RVM. This experience, and our understanding of the fault-tolerance requirements of Coda [Kistler and Satyanarayanan 1992; Satyanarayanan et al. 1990] and Venari [Nettles and Wing 1992; Wing et al. 1992], were the dominant influences on our design. The description of RVM follows in three parts: rationale, architecture, and implementation. Wherever appropriate, we point out ways in which usage experience influenced our design. We conclude with an evaluation of RVM, a discussion of its use as a building block, and a summary of related work.

2. LESSONS FROM CAMELOT

2.1 Overview

Camelot is a transactional facility build to validate the thesis that general-purpose transactional support would simplify and encourage the construction of reliable distributed systems [Spector 1991]. It supports local and distributed nested transactions and provides considerable flexibility in the choice of logging, synchronization, and transaction commitment strategies. Camelot relies heavily on the external page management and interprocess communication facilities of the Mach operating system [Baron et al. 1987], which is binary compatible with the 4.3BSD Unix operating system [Leffler et al. 1989]. Figure 1 shows the overall structure of a Camelot node. Each

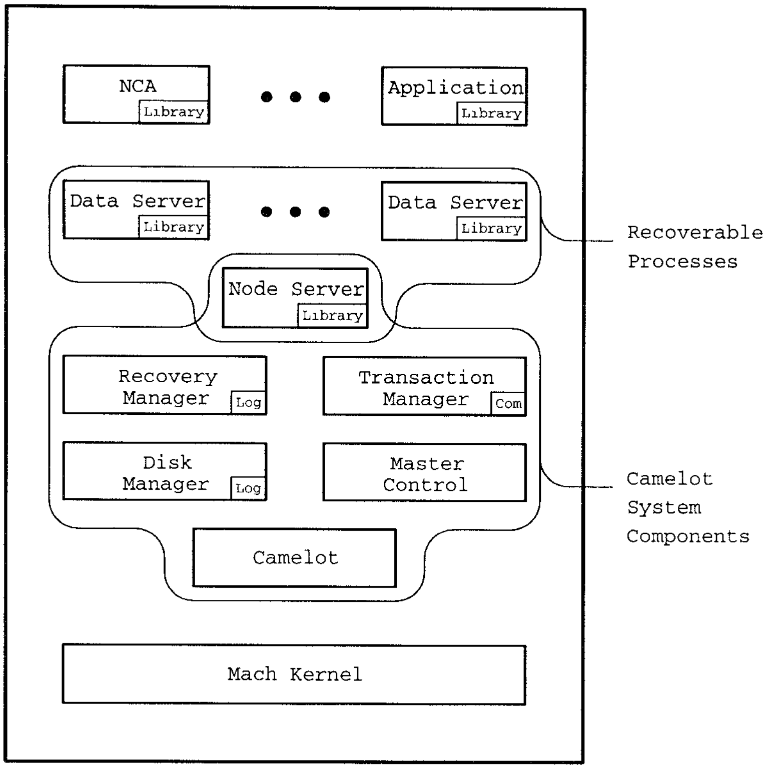


Fig. 1. This figure shows the structure of a Camelot node with application code.

module is implemented as a Mach task, and communication between modules is via Mach's interprocess communication facility (IPC).

2.2 Usage

Our interest in Camelot arose in the context of the two-phase optimistic replication protocol used by the Coda File System. Although the protocol does not require a distributed commit, it does require each server to ensure the atomicity and permanence of local updates to metadata in the first phase. The simplest strategy for us would have been to implement an ad hoc fault tolerance mechanism for metadata using some form of shadowing. But we were curious to see what Camelot could do for us.

The aspect of Camelot that we found most useful is its support for *recoverable virtual memory* [Eppinger 1989]. This unique feature of Camelot enables regions of a process' virtual address space to be endowed with the transactional properties of atomicity, isolation, and permanence. Since we did not find a need for features such as nested or distributed transactions, we realized that our use of Camelot would be something of an overkill. Yet we persisted, because it would give us first-hand experience in the use of transactions and because it would contribute toward the validation of the Camelot thesis.

We placed data structures pertaining to Coda metadata in recoverable memory[1] on servers. The metadata included Coda directories as well as persistent data for replica control and internal housekeeping. The contents of each Coda file was kept in a Unix file on a server's local file system. Server recovery consisted of Camelot restoring recoverable memory to the last committed state, followed by a Coda *salvager* which ensured mutual consistency between metadata and data.

2.3 Experience

The most valuable lesson we learned by using Camelot was that recoverable virtual memory was indeed a convenient and practically useful programming abstraction for systems like Coda. Crash recovery was simplified because data structures were restored in situ by Camelot. Directory operations were merely manipulations of in-memory data structures. The Coda salvager was simple because the range of error states it had to handle was small. Overall, the encapsulation of messy crash recovery details into Camelot considerably simplified Coda server code.

Unfortunately, these benefits came at a high price. The problems we encountered manifested themselves as *poor scalability*, *programming constraints*, and *difficulty of maintenance*. In spite of considerable effort, we were not able to circumvent these problems. Since they were direct consequences of the design of Camelot, we elaborate on these problems in the following paragraphs.

A key design goal of Coda was to preserve the scalability of AFS. But a set of carefully controlled experiments (described in an earlier paper [Satyanarayanan et al. 1990]) showed that Coda was less scalable than AFS. These experiments also showed that the primary contributor to loss of scalability was increased server CPU utilization and that Camelot was responsible for over a third of this increase. Examination of Coda servers in operation showed considerable paging and context-switching overheads due to the fact that each Camelot operation involved interactions between many of the component processes shown in Figure 1. There was no obvious way to reduce this overhead, since it was inherent in the implementation structure of Camelot.

A second obstacle to using Camelot was the set of programming constraints it imposed. These constraints came in a variety of guises. For example, Camelot required all processes using it to be descendants of the Disk Manager task shown in Figure 1. This meant that starting Coda servers required a rather convoluted procedure that made our system administration scripts complicated and fragile. It also made debugging more difficult because starting a Coda server under a debugger was complex. Another example of a programming constraint was that Camelot required us to use Mach kernel threads, even though Coda was capable of using user-level threads. Since kernel thread context switches were much more expensive, we ended up paying a hefty performance cost with little to show for it.

[1] For brevity, we often omit "virtual" from "recoverable virtual memory" in the rest of this article.

A third limitation of Camelot was that its code size, complexity, and tight dependence on rarely used combinations of Mach features made maintenance and porting difficult. Since Coda was the sternest test case for recoverable memory, we were usually the first to expose new bugs in Camelot. But it was often hard to decide whether a particular problem lay in Camelot or Mach.

As the cumulative toll of these problems mounted, we looked for ways to preserve the virtues of Camelot while avoiding its drawbacks. Since recoverable virtual memory was the only aspect of Camelot we relied on, we sought to distill the essence of this functionality into a realization that was cheap, easy to use, and had few strings attached. That quest led to RVM.

3. DESIGN RATIONALE

The central principle we adopted in designing RVM was to value *simplicity* over *generality*. In building a tool that did one thing well, we were heeding Lampson's sound advice on interface design [Lampson 1983]. We were also being faithful to the long Unix tradition of keeping building blocks simple. The change in focus from generality to simplicity allowed us to take radically different positions from Camelot in the areas of *functionality*, *operating system dependence*, and *structure*.

3.1 Functionality

Our first simplification was to eliminate support for *nesting* and *distribution*. A cost-benefit analysis showed us that each could be better provided as an independent layer on top of RVM.[2] While a layered implementation may be less efficient than a monolithic one, it has the attractive property of keeping each layer simple. Upper layers can count on the clean failure semantics of RVM, while the latter is only responsible for local, nonnested transactions.

A second area where we have simplified RVM is *concurrency control*. Rather than having RVM insist on a specific technique, we decided to factor out concurrency control. This allows applications to use a policy of their choice and to perform synchronization at a granularity appropriate to the abstractions they are supporting. If serializability is required, a layer above RVM has to enforce it. That layer is also responsible for coping with deadlocks, starvation, and other unpleasant concurrency control problems.

Internally, RVM is implemented to be multithreaded and to function correctly in the presence of true parallelism. But it does not depend on kernel thread support and can be used with no changes on user-level thread implementations. We have, in fact, used RVM with three different threading mechanisms: Mach kernel threads [Cooper and Draves 1988], coroutine C threads, and coroutine LWP [Satyanarayanan 1991].

Our final simplification was to factor out resiliency to media failure. Standard techniques such as mirroring can be used to achieve such resiliency. Our expectation is that this functionality will most likely be implemented in the device driver of a mirrored disk.

[2] An implementation sketch is provided in Section 8.

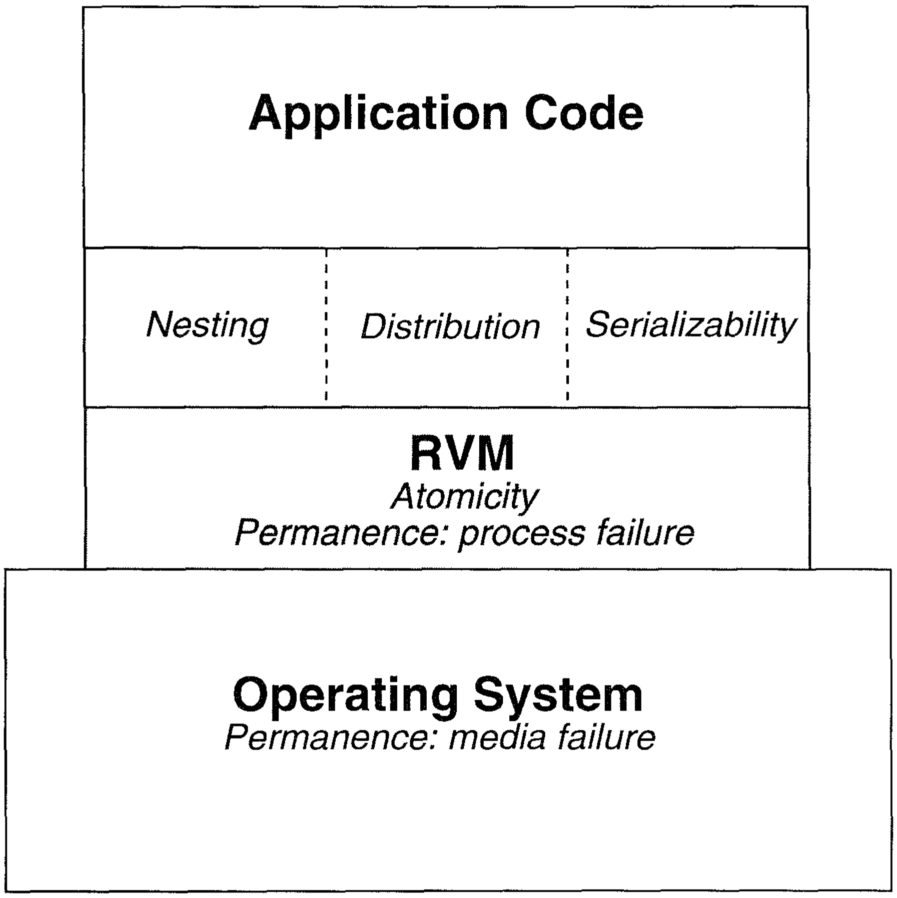


Fig. 2. Layering of functionality in RVM.

RVM thus adopts a layered approach to transactional support, as shown in Figure 2. This approach is simple and enhances flexibility: an application does not have to buy into those aspects of the transactional concept that are irrelevant to it.

3.2 Operating System Dependence

To make RVM portable, we decided to rely only on a small, widely supported, Unix subset of the Mach system call interface. A consequence of this decision was that we could not count on tight coupling between RVM and the VM subsystem. The Camelot Disk Manager module runs as an *external pager* [Young 1989] and takes full responsibility for managing the backing store for recoverable regions of a process. The use of advisory VM calls (pin and unpin) in the Mach interface lets Camelot ensure that dirty recoverable regions of a process' address space are not paged out until the corresponding log records are forced to disk. This close alliance with Mach's VM subsystem allows Camelot to avoid double paging and to support recoverable regions whose size approaches backing store or addressing limits. Efficient handling of large recoverable regions is critical to Camelot's goals.

Our goals in building RVM were more modest. We were not trying to replace tranditional forms of persistent storage, such as file systems and databases. Rather, we saw RVM as a building block for metadata in those systems and in higher-level compositions of them. Consequently, we could assume that the recoverable memory requirements on a machine would only be a small fraction of its total disk storage. This in turn meant that it was acceptable to waste some disk space by duplicating the backing store for recoverable regions. Hence RVM's backing store for a recoverable region, called its *external data segment*, is completely independent of the region's VM swap space. Crash recovery relies only on the state of the external data segment. Since a VM pageout does not modify the external data segment, an uncommitted dirty page can be reclaimed by the VM subsystem without loss of correctness. Of course, good performance also requires that such pageouts be rare.

One way to characterize our strategy is to view it as a *complexity*-versus-*resource usage* tradeoff. By being generous with memory and disk space, we have been able to keep RVM simple and portable. Our design supports the optional use of external pagers, but we have not implemented support for this feature yet. The most apparent impact on Coda has been slower startup because a process' recoverable memory must be read in en masse rather than being paged in on demand.

Insulating RVM from the VM subsystem also hinders the sharing of recoverable virtual memory across address spaces. But this is not a serious limitation. After all, the primary reason to use a separate address space is to increase robustness by avoiding memory corruption. Sharing recoverable memory across address spaces defeats this purpose. In fact, it is worse than sharing (volatile) virtual memory because damage may be persistent! Hence, our view is that processes willing to share recoverable memory already trust each other enough to run as threads in a single address space.

3.3 Structure

The ability to communicate efficiently across address spaces allows robustness to be enhanced without sacrificing good performance. Camelot's modular decomposition, shown earlier in Figure 1, is predicated on fast IPC. Although it has been shown that IPC can be fast [Bershad et al. 1990], its performance in commercial Unix implementations lags far behind that of the best experimental implementations. Even on Mach 2.5, the measurements reported by Stout et al. [1991] indicated that IPC is about 600 times more expensive than local procedure call.[3] To make matters worse, Ousterhout [1990] reports that the context-switching performance of operating systems is not improving linearly with raw hardware performance.

[3] 430 microseconds versus 0.7 microseconds for a null call on a typical contemporary machine, the DECStation 5000/200.

Given our desire to make RVM portable, we were not willing to make its design critically dependent on fast IPC. Instead, we have structured RVM as a library that is linked in with an application. No external communication of any kind is involved in the servicing of RVM calls. An implication of this is, of course, that we have to trust applications not to damage RVM data structures and vice versa.

A less obvious implication is that applications cannot share a single write-ahead log on a dedicated disk. Such sharing is common in transactional systems because disk head movement is a strong determinant of performance and because the use of a separate disk per application is economically infeasible at present. In Camelot, for example, the Disk Manager serves as the multiplexing agent for the log. The inability to share one log is not a significant limitation for Coda, because we run only one file server process on a machine. But it may be legitimate concern for other applications that wish to use RVM. Fortunately, there are two potential alleviating factors on the horizon.

First, independent of transaction-processing considerations, there is considerable interest in log-structured implementations of the Unix file system [Rosenblum and Ousterhout 1992]. If one were to place the RVM log for each application in a separate file on such a system, one would benefit from minimal disk head movement. No log multiplexor would be needed, because that role would be played by the file system.

Second, there is a trend toward using disks of small form factor, partly motivated by interest in disk array technology [Patterson et al. 1988]. It has been predicated that the large disk capacity in the future will be achieved by using many small disks. If this turns out to be true, there will be considerably less economic incentive to avoiding a dedicated disk per process.

In summary, each process using RVM has a separate log. The log can be placed in a Unix file or on a raw disk partition. When the log is on a file, RVM uses the fsync system call to synchronously flush modifications onto disk. RVM's permanence guarantees rely on the correct implementation of this system call. For best performance, the log should either be in a raw partition on a dedicated disk or in a file on a log-structured Unix file system.

4. ARCHITECTURE

The design of RVM follows logically from the rationale presented earlier. In the description below, we first present the major program-visible abstractions and then describe the operations supported on them.

4.1 Segments and Regions

Recoverable memory is managed in *segments*, which are loosely analogous to Multics segments. RVM has been designed to accommodate segments up to 2^{64} bytes long, although current hardware and file system limitations restrict segment length to 2^{32} bytes. The number of segments on a machine is only limited by its storage resources. The backing store for a segment may be a file or a raw disk partition. Since the distinction is invisible to programs, we use the term "external data segment" to refer to either.

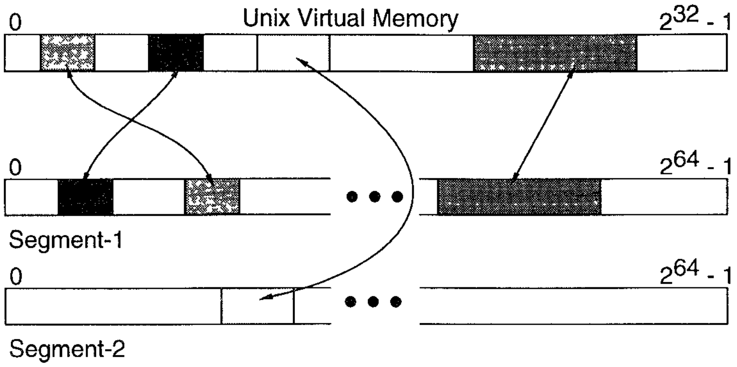


Fig. 3. Each shaded area mapping regions of segments specified during mapping.

As shown in Figure 3, applications explicitly map *regions* of segments into their virtual memory. RVM guarantees that newly mapped data represents the committed image of the region. A region typically corresponds to a related collection of objects and may be as large as the entire segment. In the current implementation, the copying of data from external data segment to virtual memory occurs when a region is mapped. The limitation of this method is startup latency, as mentioned in Section 3.2. In the future, we plan to provide an optional Mach external pager to copy data on demand.

Restrictions on segment mapping are minimal. The most important restriction is that no region of a segment may be mapped more than once by the same process. Also, mappings cannot overlap in virtual memory. These restrictions eliminate the need for RVM to cope with aliasing. Mapping must be done in multiples of page size, and regions must be page aligned.

Regions can be unmapped at any time, as long as they have no uncommitted transactions outstanding. RVM retains no information about a segment's mappings after its regions are unmapped. A *segment loader* package, built on top of RVM, allows the creation and maintenance of a load map for recoverable storage and takes care of mapping a segment into the same base address each time. This simplifies the use of absolute pointers in segments. A *recoverable memory allocator*, also layered on RVM, supports heap management of storage within a segment.

4.2 RVM Primitives

The operations provided by RVM for initialization, termination, and segment mapping are shown in Figure 4a. The log to be used by a process is specified at RVM initialization via the options_desc argument. The map operation is called once for each region to be mapped. The external data segment and the range of virtual memory addresses for the mapping are identified in the first argument. The unmap operation can be invoked at any time that a region is quiescent. Once unmapped, a region can be remapped to some other part of the process' address space.

After a region has been mapped, memory addresses within it may be used in the transactional operations shown in Figure 4b. The begin_transaction

```
initialize(version, options_desc);
map(region_desc, options_desc);
unmap(region_desc);
terminate();
```

(a) Initialization & Mapping Operations

```
begin_transaction(tid, restore_mode);
set_range(tid, base_addr, nbytes);
end_transaction(tid, commit_mode);
abort_transaction(tid);
```

(b) Transactional Operations

```
flush();
truncate();
```

(c) Log Control Operations

```
query(options_desc, region_desc);
set_options(options_desc);
create_log(options, log_len, mode);
```

(d) Miscellaneous Operations

Fig. 4 RVM primitives.

operation returns a transaction identifier, tid, that is used in all further operations associated with that transaction. The set_range operation lets RVM know that a certain area of a region is about to be modified. This allows RVM to record the current value of the area so that it can undo changes in case of an abort. The restore_mode flag to begin_transaction lets an application indicate that it will never explicitly abort a transaction. Such a *no-restore* transaction is more efficient, since RVM does not have to copy data on a set-range. Read operations on mapped regions require no RVM intervention.

A transaction is committed by end_transaction and aborted via abort_transaction. By default, a successful commit guarantees permanence of changes made in a transaction. But an application can indicate its willingness to accept a weaker permanence guarantee via the commit_mode parameter of end_transaction. Such a *no-flush* or "lazy" transaction has reduced commit latency since a log force is avoided. To ensure persistence of its no-flush transactions the application must explicitly flush RVM's write-ahead log from time to time. When used in this manner, RVM provides *bounded persistence*, where the bound is the period between log flushes. Note that atomicity is guaranteed independent of permanence.

Figure 4c shows the two operations provided by RVM for controlling the use of the write-ahead log. The first operation, flush, blocks until all committed no-flush transactions have been forced to disk. The second operation, truncate, blocks until all committed changes in the write-ahead log have been reflected to external data segments. Log truncation is usually performed transparently in the background by RVM. But since this is a potentially long-running and resource-intensive operation, we have provided a mechanism for applications to control its timing.

The final set of primitives, shown in Figure 4d, performs a variety of functions. The query operation allows an application to obtain information such as the number and identity of uncommitted transactions in a region. The set_options operation sets a variety of tuning knobs such as the threshold for triggering log truncation and the sizes of internal buffers. Using create_log, an application can dynamically create a write-ahead log and then use it in an initialize operation.

5. IMPLEMENTATION

Since RVM draws upon well-known techniques for building transactional systems, we restrict our discussion here to two important aspects of its implementation: *log management* and *optimization*. The RVM manual [Mashburn and Satyanarayanan 1992] offers many further details, and a comprehensive treatment of transactional implementation techniques can be found in Gray and Reuter [1993].

5.1 Log Management

5.1.1 *Log Format.* RVM is able to use a *no-undo/redo* value-logging strategy [Bernstein et al. 1987] because it never reflects uncommitted changes to an external data segment. The implementation assumes that adequate buffer space is available in virtual memory for the old-value records of uncommitted transactions. Consequently, only the new-value records of committed transactions have to be written to the log. The format of a typical log record is shown in Figure 5.

The bounds and contents of old-value records are known to RVM from the set-range operations issued during a transaction. Upon commit, old-value records are replaced by new-value records that reflect the current contents of the corresponding ranges of memory. Note that each modified range results in only one new-value record even if that range has been updated many times in a transaction. The final step of transaction commitment consists of forcing the new-value records to the log and writing out a commit record.

No-restore and no-flush transactions are more efficient. The former result in both time and space spacings since the contents of old-value records do not have to be copied or buffered. The latter result in considerably lower commit latency, since new-value and commit records can be spooled rather than forced to the log.

5.1.2 *Crash Recovery and Log Truncation.* Crash recovery consists of RVM first reading the log from tail to head, then constructing an in-memory tree of the latest committed changes for each data segment encountered in the log. The trees are then traversed, applying modifications in them to the corresponding external data segment. Finally, the head and tail location information in the log status block is updated to reflect an empty log. The idempotency of recovery is achieved by delaying this step until all other recovery actions are complete.

Truncation is the process of reclaiming space allocated to log entries by applying the changes contained in them to the recoverable data segment. Periodic truncation is necessary because log space is finite and is triggered whenever current log size exceeds a preset fraction of its total size. In our experience, log truncation has proved to be the hardest part of RVM to implement correctly. To minimize implementation effort, we initially chose to

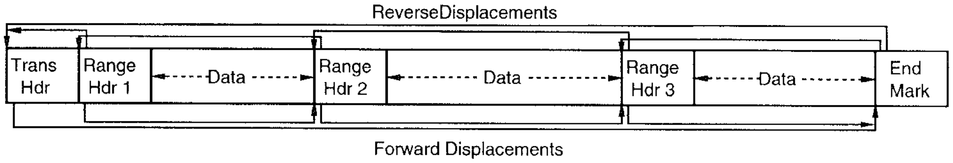


Fig. 5. This log record format of a typical log record read either way.

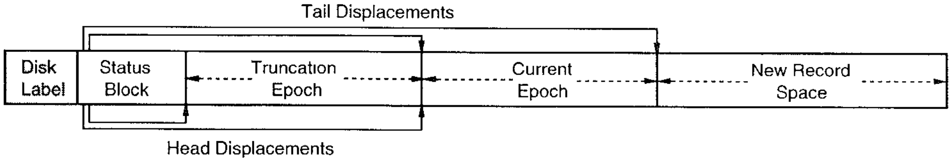


Fig. 6. This figure shows epoch truncation freed for new log records.

reuse crash recovery code for truncation. In this approach, referred to as *epoch truncation*, the crash recovery procedure described above is applied to an initial part of the log while concurrent forward processing occurs in the rest of the log. Figure 6 depicts the layout of a log while an epoch truncation is in progress.

Although exclusive reliance on epoch truncation is a logically correct strategy, it substantially increases log traffic, degrades forward processing more than necessary, and results in bursty system performance. Now that RVM is stable and robust, we are implementing a mechanism for *incremental truncation* during normal operation. This mechanism periodically renders the oldest log entries obsolete by writing out relevant pages directly from VM to the recoverable data segment. To preserve the no-undo/redo property of the log, pages that have been modified by uncommitted transactions cannot be written out to the recoverable data segment. RVM maintains internal locks to ensure that incremental truncation does not violate this property. Certain situations, such as the presence of long-running transactions or sustained high concurrency, may result in incremental truncation being blocked for so long that log space becomes critical. Under those circumstances, RVM reverts to epoch truncation.

Figure 7 shows the two data structures used in incremental truncation. The first data structure is a page vector for each mapped region that maintains the modification status of that region's pages. The page vector is loosely analogous to a VM page table: the entry for a page contains a *dirty bit* and an *uncommitted reference count*. A page is marked dirty when it has committed changes. The uncommitted reference count is incremented as set_ranges are executed and decremented when the changes are committed or aborted. On commit, the affected pages are marked dirty. The second data structure is a FIFO queue of page modification descriptors that specifies the

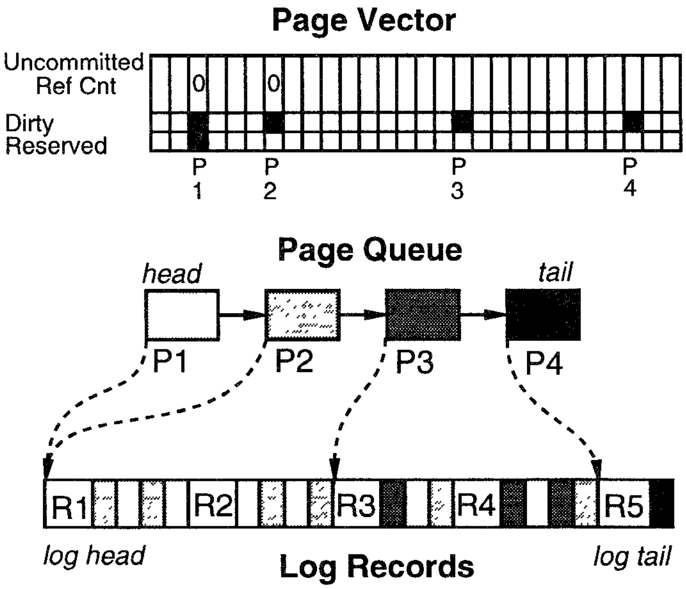


Fig. 7. This figure shows incremental truncation count drop to zero.

order in which dirty pages should be written out in order to move the log head. Each descriptor specifies the log offset of the first record referencing that page. The queue contains no duplicate page references: a page is mentioned only in the earliest descriptor in which it could appear. A step in incremental truncation consists of selecting the first descriptor in the queue, writing out the pages specified by it, deleting the descriptor, and moving the log head to the offset specified by the next descriptor. This step is repeated until the desired amount of log space has been reclaimed.

5.2 Optimizations

Early experience with RVM indicated two distinct opportunities for substantially reducing the volume of data written to the log. We refer to these as *intratransaction* and *intertransaction* optimizations respectively.

Intratransaction optimizations arise when set-range calls specifying identical, overlapping, or adjacent memory addresses are issued within a single transaction. Such situations typically occur because of modularity and defensive programming in applications. Forgetting to issue a set-range call is an insidious bug, while issuing a duplicate call is harmless. Hence applications are often written to err on the side of caution. This is particularly common when one part of an application begins a transaction and then invokes procedures elsewhere to perform actions within that transaction. Each of those procedures may perform set-range calls for the areas of recoverable memory it modifies, even if the caller or some other procedure is supposed to have done so already. Optimization code in RVM causes duplicate set-range calls to be ignored and overlapping and adjacent log records to be coalesced.

Intertransaction optimizations occur only in the context of no-flush transactions. Temporal locality of reference in input requests to an application often translates into locality of modifications to recoverable memory. For example, the command "cp d1 /* d2" on a Coda client will cause as many no-flush transactions updating the data structure in RVM for d2 as there are children of d1. Only the last of these updates needs to be forced to the log on a future flush. The check for intertransaction optimization is performed at commit time. If the modifications being committed subsume those from an earlier unflushed transaction, the older log records are discarded.

6. STATUS AND EXPERIENCE

RVM has been in daily use for over two years on hardware platforms such as IBM RTs, DEC MIPS workstations, Sun Sparc workstations, and a variety of Intel 386/486-based laptops and workstations. Memory capacity on these machines ranges from 12MB to 64MB, while disk capacity ranges from 60MB to 2.5GB. Our personal experience with RVM has only been on Mach 2.5 and 3.0. But RVM has been ported to SunOS and SGI IRIX at MIT, and we are confident that ports to other Unix platforms will be straightforward. Most applications using RVM have been written in C or C + +, but a few have been written in Standard ML. A version of the system that uses incremental truncation is being debugged.

Our original intent was just to replace Camelot by RVM on servers, in the role described in Section 2.2. But positive experience with RVM has encouraged us to expand its use. For example, transparent *resolution* of directory updates made to partitioned server replicas is done using a log-based strategy [Kumar and Satyanarayanan 1993]. The logs for resolution are maintained in RVM. Clients also use RVM now, particularly for supporting *disconnected operation* [Kistler and Satyanarayanan 1992]. The persistence of changes made while disconnected is achieved by storing *replay logs* in RVM, and user advice for long-term cache management is stored in a *hoard database* in RVM.

An unexpected use of RVM has been in debugging Coda servers and clients [Satyanarayanan et al. 1992]. As Coda matured, we ran into hard-to-reproduce bugs involving corrupted persistent data structures. We realized that the information in RVM's log offered excellent clues to the source of these corruptions. All we had to do was to save a copy of the log before truncation and build a postmortem tool to search and display the history of modifications recorded by the log.

The most common source of programming problems in using RVM has been in forgetting to do a set set-range call prior to modifying an area of recoverable memory. The result is disastrous, because RVM does not create a new-value record for this area upon transaction commit. Hence the restored state after a crash or shutdown will not reflect modifications by the transaction to that area of memory. The current solution, as described in Section 5.2, is to program defensively. A better solution would be language based, as discussed in Section 8.

7. EVALUATION

A fair assessment of RVM must consider two distinct issues. From a software engineering perspective, we need to ask whether RVM's code size and complexity are commensurate with its functionality. From a systems perspective, we need to know whether RVM's focus on simplicity has resulted in unacceptable loss of performance.

To address the first issue, we compared the source code of RVM and Camelot. RVM's mainline code is approximately 10K lines of C, while utilities, test programs, and other auxiliary code contribute a further 10K lines. Camelot has a mainline code size of about 60K lines of C and auxiliary code of about 10K lines. These numbers do not include code in Mach for features like IPC and the external pager that are critical to Camelot.

Thus the total size of code that has to be understood, debugged, and tuned is considerably smaller for RVM. This translates into a corresponding reduction of effort in maintenance and porting. What is being given up in return is support for nesting and distribution, as well as flexibility in areas such as choice of logging strategies—a fair trade by our reckoning.

To evaluate the performance of RVM we used controlled experimentation as well as measurements from Coda servers and clients in actual use. The specific questions of interest to us were:

—How serious is the lack of integration between RVM and VM?

—What is RVM's impact on scalability?

—How effective are intra- and intertransaction optimizations?

7.1 Lack of RVM-VM Integration

As discussed in Section 3.2, the separation of RVM from the VM component of an operating system could hurt performance. To quantify this effect, we designed a benchmark inspired by the TPC family of benchmarks [Serlin 1991] and used it in a series of carefully controlled experiments.

7.1.1 *The Benchmark.* The TPC family of benchmarks is stated in terms of a hypothetical bank with one or more branches, multiple tellers per branch, and many customer accounts per branch. A transaction updates a randomly chosen account, updates branch and teller balances, and appends a history record to an audit trail.

In our variant of this benchmark, we represent all the data structures accessed by a transaction in recoverable memory. The number of accounts is a parameter of our benchmark. The accounts and the audit trail are represented as arrays of 128-byte and 64-byte records respectively. Each of these data structures occupies close to half the total recoverable memory. The sizes of the data structures for teller and branch balances are insignificant.

Access to the audit trail is always sequential, with wraparound. The pattern of accesses to the account array is a second parameter of our benchmark. The best case for paging performance occurs when accesses are sequential. The worst case occurs when accesses are uniformly distributed across all accounts. To represent the average case, the benchmark uses an access pattern that exhibits considerable temporal locality. In this access pattern, referred to as *localized*, 70% of the transactions update accounts on 5% of the pages; 25% of the transactions update accounts on a different 15% of the pages; and the remaining 5% of the transactions update accounts on the remaining 80% of the pages. Within each set, accesses are uniformly distributed.

7.1.2 *Results.* Our primary goal in these experiments was to understand the throughput of RVM over its intended domain of use. This corresponds to situations where paging rates are low, as discussed in Section 3.2. A secondary goal was to observe performance degradation relative to Camelot as paging becomes more significant. We expected this to shed light on the importance of RVM-VM integration.

To meet these goals, we conducted experiments for account arrays ranging from 32K entries to about 450K entries. This roughly corresponds to ratios of 10% to 175% of total recoverable-memory size to total physical-memory size. At each account array size, we performed the experiment for sequential, random, and localized account access patterns. Table I and Figure 8 present our results. Hardware and other relevant experimental conditions are described in Table I.

For sequential account access, Figure 8a shows that RVM and Camelot offer virtually identical throughput. This throughput hardly changes as the size of recoverable memory increases. The average time to perform a log force on the disks used in our experiments is about 17.4 milliseconds. This yields a theoretical maximum throughput of 57.4 transactions per second, which is within 15% of the observed best-case throughputs for RVM and Camelot.

When account access is random, Figure 8a shows that RVM's throughput is initially close to its value for sequential access. As recoverable-memory size increases, the effects of paging become more significant, and throughput drops. But the drop does not become serious until recoverable-memory size exceeds about 70% of physical-memory size. The random-access case is precisely where one would expect Camelot's integration with Mach to be most valuable. Indeed, the convexities of the curves in Figure 8a show that Camelot's degradation is more graceful than RVM's. But even at the highest ratio of recoverable to physical-memory size, RVM's throughput is better than Camelot's.

For localized account access, Figure 8b shows that RVM's throughput drops almost linearly with increasing recoverable-memory size. But the drop is relatively slow, and performance remains acceptable even when recoverable-memory size approaches physical-memory size. Camelot's throughput also drops linearly and is consistently worse than RVM's throughput.

These measurements confirm that RVM's simplicity is not an impediment to good performance for its intended application domain. A conservative interpretation of the data in Table I indicates that applications with good locality can use up to 40% of physical memory for active recoverable data,

Table I. This Table Presents Transactional Throughput Significantly

No. of	Rmem	RVM (Trans/Sec)			Camelot (Trans/Sec)		
Accounts	Pmem	Sequential	Random	Localized	Sequential	Random	Localized
32768	12.5%	48.6 (0 0)	47.9 (0 0)	47.5 (0 0)	48.1 (0.0)	41.6 (0 4)	44.5 (0 2)
65536	25.0%	48.5 (0 2)	46.4 (0 1)	46.6 (0.0)	48.2 (0.0)	34.2 (0.3)	43.1 (0 6)
98304	37.5%	48.6 (0 0)	45.5 (0.0)	46.2 (0 0)	48.9 (0 1)	30.1 (0 2)	41.2 (0 2)
131072	50.0%	48.2 (0.0)	44.7 (0.2)	45.1 (0 0)	48.1 (0 0)	29.2 (0.0)	41.3 (0 1)
163840	62.5%	48.1 (0 0)	43.9 (0 0)	44.2 (0.1)	48.1 (0.0)	27.1 (0.2)	40.3 (0 2)
196608	75.0%	47.7 (0 0)	43.2 (0 0)	43.4 (0 0)	48.1 (0 4)	25.8 (1 2)	39.5 (0 8)
229376	87.5%	47.2 (0.1)	42.5 (0.0)	43.8 (0 1)	48.2 (0.2)	23.9 (0 1)	37.9 (0 2)
262144	100.0%	46.9 (0 0)	41.6 (0.0)	41.1 (0.0)	48.0 (0 0)	21.7 (0 0)	35.9 (0 2)
294912	112.5%	46.3 (0 6)	40.8 (0 5)	39.0 (0 6)	48.0 (0 0)	20.8 (0 2)	35.2 (0 1)
327680	125.0%	46.9 (0 7)	39.7 (0 0)	39.0 (0.5)	48.1 (0.1)	19.1 (0.0)	33.7 (0.0)
360448	137.5%	48.6 (0 0)	33 8 (0 9)	40.0 (0 0)	48.3 (0 0)	18.6 (0 0)	33.3 (0 1)
393216	150.0%	46.9 (0 2)	33.3 (1 4)	39.4 (0 4)	48.9 (0 0)	18.7 (0 1)	32.4 (0 2)
425984	162.5%	46.5 (0 4)	30.9 (0 3)	38.7 (0 2)	48.0 (0 0)	18.2 (0 0)	32.3 (0 2)
458752	175.0%	46 4 (0 4)	27.4 (0 2)	35.4 (1 0)	47.7 (0 0)	17.9 (0 1)	31.6 (0 0)

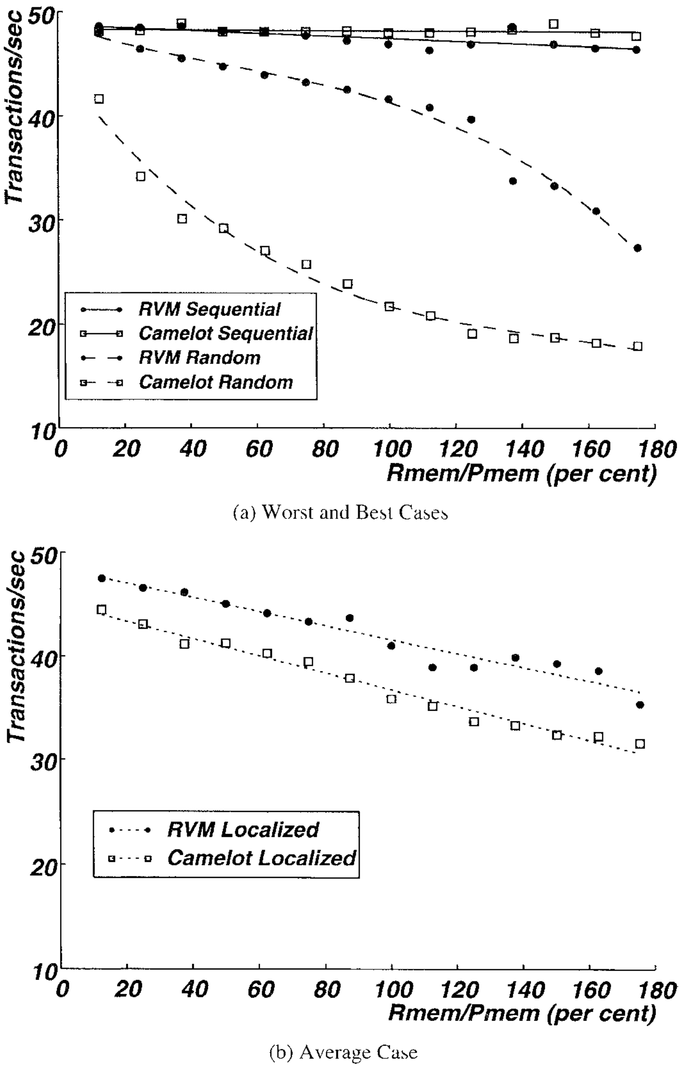
while keeping throughput degradation to less than 10%. Applications with poor locality have to restrict active recoverable data to less than 25% for similar performance. Inactive recoverable data can be much larger, constrained only by startup latency and virtual-memory limits imposed by the operating system. The comparison with Camelot is especially revealing. In spite of the fact that RVM is not integrated with VM, it is able to outperform Camelot over a broad range of workloads.

Although we were gratified by these results, we were puzzled by Camelot's behavior. For low ratios of recoverable to physical memory we had expected both Camelot's and RVM's throughputs to be independent of the degree of locality in the access pattern. The data shows that this is indeed the case for RVM. But in Camelot's case, throughput is highly sensitive to locality even at the lowest recoverable-to-physical-memory ratio of 12.5%. At that ratio Camelot's throughput in transactions per second drops from 48.1 in the sequential case to 44.5 in the localized case, and to 41.6 in the random case.

Closer examination of the raw data indicates that the drop in throughput is attributable to much higher levels of paging activity sustained by the Camelot Disk Manager. We conjecture that this increased paging activity is induced by a less efficient checkpointing implementation for segment sizes of the order of main memory size. During truncation, the Disk Manager writes out all dirty pages referenced by entries in the affected portion of the log. When truncation is frequent and account access is random, many opportunities to amortize the cost of writing out a dirty page across multiple transactions are lost. Less frequent truncation or sequential account access result in fewer such lost opportunities.

7.2 Scalability

As discussed in Section 2.3, Camelot's heavy toll on the scalability of Coda servers was a key influence on the design of RVM. It is therefore appropriate



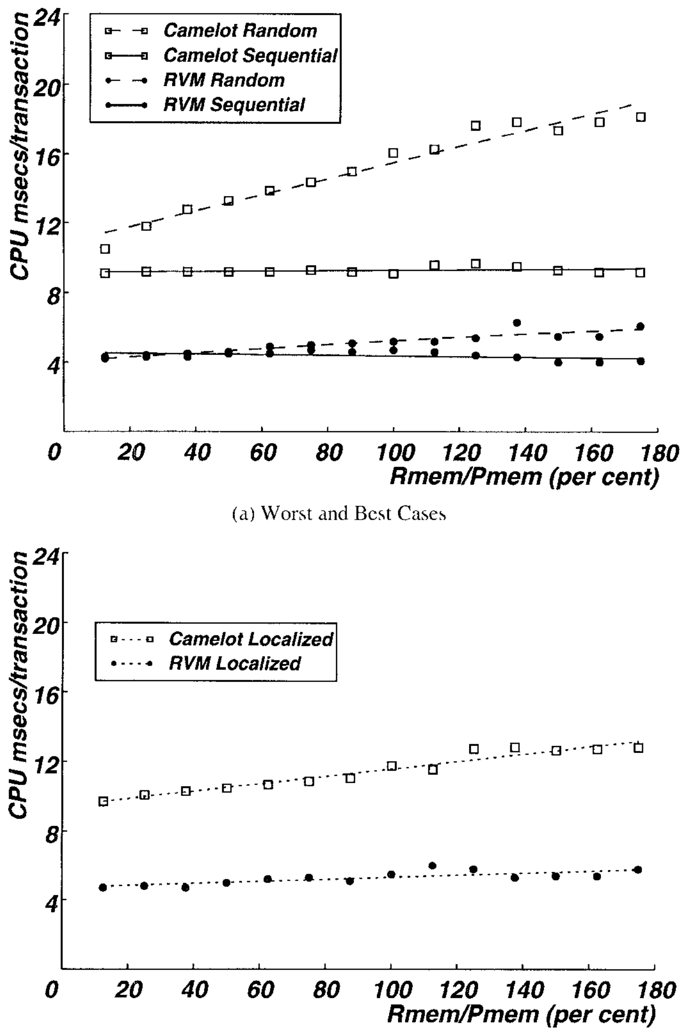
(a) Worst and Best Cases

(b) Average Case

Fig. 8. These plots illustrate transactional throughput worst cases.

to ask whether RVM has yielded the anticipated gains in scalability. The ideal way to answer this question would be to repeat the experiment mentioned in Section 2.3, using RVM instead of Camelot. Unfortunately, such a direct comparison is not feasible because server hardware has changed considerably. Instead of IBM RTs we now use the much faster Decstation 5000/200s. Repeating the original experiment on current hardware is also not possible, because Coda servers now use RVM to the exclusion of Camelot.

Consequently, our evaluation of RVM's scalability is based on the same set of experiments described in Section 7.1. For each trial of that set of experi-



(a) Worst and Best Cases

(b) Average Case

Fig. 9. These plots depict amortized CPU cost per transaction.

ments, the total CPU usage on the machine was recorded. Since no extraneous activity was present on the machine, all CPU usage (whether in system or user mode) is attributable to the running of the benchmark. Dividing the total CPU usage by the number of transactions gives the average CPU cost per transaction, which is our metric of scalability. Note that this metric amortizes the cost of sporadic activities like log truncation and page fault servicing over all transactions.

Figure 9 compares the scalability of RVM and Camelot for each of the three access patterns described in Section 7.1.1. For sequential account access, RVM requires about half the CPU usage of Camelot. The actual values of

Table II. This Table Presents Savings Due to RVM Optimizations, Clients, and Servers

Machine name	Machine type	Transactions committed	Bytes Written to Log	Intra-Transaction Savings	Inter-Transaction Savings	Total Savings
grieg	server	267,224	289,215,032	20.7%	0.0%	20.7%
haydn	server	483,978	661,612,324	21.5%	0.0%	21.5%
wagner	server	248,169	264,557,372	20.9%	0.0%	20.9%
mozart	client	34,744	9,039,008	41.6%	26.7%	68.3%
ives	client	21,013	6,842,648	31.2%	22.0%	53.2%
verdi	client	21,907	5,789,696	28.1%	20.9%	49.0%
bach	client	26,209	10,787,736	25.8%	21.9%	47.7%
purcell	client	76,491	12,247,508	41.3%	36.2%	77.5%
berlioz	client	101,168	14,918,736	17.3%	64.3%	81.6%

CPU usage remain almost constant for both systems over all the recoverable memory sizes we examined.

For random account access, Figure 9a shows that both RVM and Camelot's CPU usage increase with recoverable memory size. But it is astonishing that even at the limit of our experimental range, RVM's CPU usage is less than Camelot's. In other words, the inefficiency of page fault handling in RVM is more than compensated for by its lower inherent overhead.

For localized account access, Figure 9b shows that CPU usage increases linearly with recoverable-memory size for both RVM and Camelot. For all sizes investigated, RVM's CPU usage remains well below that of Camelot's.

Overall, these measurements establish that RVM is considerably less of a CPU burden than Camelot. Over most of the workloads investigated, RVM typically requires about half the CPU usage of Camelot. We anticipate that refinements to RVM such as incremental truncation will further improve its scalability.

RVM's lower CPU usage follows directly from our decision to structure it as a library rather than as a collection of tasks communicating via IPC. As mentioned in Section 3.3, Mach IPC costs about 600 times as much as a procedure call on the hardware we used for our experiments. Further contributing to reduced CPU usage are the substantially smaller path lengths in various RVM components due to their inherently simpler functionality.

7.3 Effectiveness of Optimizations

To estimate the value of intra- and intertransaction optimizations, we instrumented RVM to keep track of the total volume of log data eliminated by each technique. Table II presents the observed savings in log traffic for a representative sample of Coda clients and servers in our environment.

The data in Table II shows that both servers and clients benefit significantly from intratransaction optimization. The saving in log traffic is typically between 20% and 30%, though some machines exhibit substantially higher savings. Intertransaction optimizations typically reduce log traffic on clients by another 20–30%. Servers do not benefit from this type of optimization, because it is only applicable to no-flush transactions. RVM optimizations have proved to be especially valuable for good performance on portable Coda clients, because disks on those machines tend to be selected on the basis of size, weight, and power consumption rather than performance.

7.4 Broader Analysis

A fair criticism of the conclusions drawn in Sections 7.1 and 7.2 is that they are based solely on comparison with a research prototype, Camelot. A favorable comparison with well-tuned commercial products would strengthen the claim that RVM's simplicity does not come at the cost of good performance. Unfortunately, such a comparison is not currently possible because no widely used commercial product supports recoverable virtual memory. Hence a performance analysis of broader scope will have to await the future.

8. RVM AS A BUILDING BLOCK

The simplicity of the abstraction offered by RVM makes it a versatile base on which to implement more complex functionality. In principle, any abstraction that requires persistent data structures with clean local failure semantics can be built on top of RVM. In some cases, minor extensions of the RVM interface may be necessary.

For example, nested transactions could be implemented using RVM as a substrate for bookkeeping state such as the undo logs of nested transactions. Only top-level begin, commit, and abort operations would be visible to RVM. Recovery would be simple, since the restoration of committed state would be handled entirely by RVM. The feasibility of this approach has been confirmed by the Venari project [Wing et al. 1992].

Support for distributed transactions could also be provided by a library build on RVM. Such a library would provide coordinator and subordinate routines for each phase of a two-phase commit, as well as for operations such as beginning a transaction and adding new sites to a transaction. Recovery after a coordinator crash would involve RVM recovery, followed by appropriate termination of distributed transactions in progress at the time of the crash. The communication mechanism could be left unspecified until runtime by using upcalls from the library to perform communications. RVM would have to be extended to enable a subordinate to undo the effects of a first-phase commit if the coordinator decides to abort. One way to do this would be to extend end_transaction to return a list of the old-value records generated by the transaction. These records could be preserved by the library at each subordinate until the outcome of the two-phase commit is clear. On a global commit, the records would be discarded. On a global abort, the library at each subordinate could use the saved records to construct a compensating RVM transaction.

RVM can also be used as the basis of runtime systems for languages that support persistence. Experience with Avalon [Wing 1991], which was built on Camelot, confirms that recoverable virtual memory is indeed an appropriate abstraction for implementing language-based local persistence. Language support would alleviate the problem mentioned in Section 6 of programmers forgetting to issue set-range calls: compiler-generated code could issue these calls transparently. An approximation to a language-based solution would be to use a postcompilation augmentation phase to test for accesses to mapped RVM regions and to generate set-range calls.

Further evidence of the versatility of RVM is provided by the recent work of O'Toole et al. [1993]. In this work, RVM segments are used as the stable to-space and from-space of the heap for a language that supports concurrent garbage collection of persistent data. While the authors suggest some improvements to RVM for this application, their work establishes the suitability of RVM for a very different context from the one that motivated it.

9. RELATED WORK

The field of transaction processing is enormous. In the space available, it is impossible to fully attribute all the past work that has indirectly influenced RVM. We therefore restrict our discussion here to placing RVM's contribution in proper perspective and to clarifying its relationship to its closest relatives.

Since the original identification of transactional properties and techniques for their realization [Gray 1978; Lampson 1981], attention has been focused on three areas. One area has been the enrichment of the transactional concept along dimensions such as distribution, nesting [Moss 1985], and longevity [Garcia-Molina and Salem 1987]. A second area has been the incorporation of support for transactions into languages [Liskov and Scheifler 1983], operating systems [Haskin et al. 1988], and hardware [Chang and Mergen 1988]. A third area has been the development of techniques for achieving high performance in OLTP environments with very large data volumes and poor locality [Good et al. 1985].

In contrast to those efforts, RVM represents a back-to-basics movement. Rather than embellishing the transactional abstraction or its implementation, RVM seeks to simplify both. It 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.

The virtues of simplicity for small databases have been extolled previously by Birrell et al. [1987]. Their design is even simpler than RVM's and is based on new-value logging and full-database checkpointing. Each transaction is constrained to update only a single data item. There is no support for explicit transaction abort. Updates are recorded in a log file on disk, then reflected in the in-memory database image. Periodically, the entire memory image is checkpointed to disk, the log file deleted, and the new checkpoint file renamed to be the current version of the database. Log truncation occurs only during crash recovery, not during normal operation.

The reliance of Birrell et al.'s technique on full-database checkpointing makes the technique practical only for applications which manage small amounts of recoverable data and which have moderate update rates. The absence of support for multiitem updates and for explicit abort further limits its domain of use. RVM is more versatile without being substantially more complex.

Transaction-processing monitors (TPMs), such as Encina [Transarc Corp. 1991; Young et al. 1991] and Tuxedo [Andrade et al. 1989; Unix System Labs 1993], are important commercial products. TPMs add distribution and support services to OLTP backends and integrate heterogeneous systems. Like centralized database managers, TPM backends are usually monolithic in structure. They encapsulate all three of the basic transactional properties and provide data access via a query language interface. This is in contrast to RVM, which supports only atomicity and the process failure aspect of permanence, and which provides access to recoverable data as mapped virtual memory.

A more modular approach is used in the Transarc TP toolkit, which is the backend for the Encina TPM. The functionality provided by RVM corresponds primarily to the recovery, logging, and physical storage modules of the Transarc toolkit. RVM differs from the corresponding Transarc toolkit components in two important ways. First, RVM is structured entirely as a library that is linked with applications, while some of the toolkit's modules are separate processes. Second, recoverable storage is accessed as mapped memory in RVM, whereas the Transarc toolkit offers access via the conventional buffered I/O model.

Chew et al. [1993] have recently reported on their efforts to enhance the Mach kernel to support recoverable virtual memory. Their work carries Camelot's idea of providing system-level support for recoverable memory a step further, since their support is in the kernel rather than in a user-level Disk Manager. In contrast, RVM avoids the need for specialized operating system support, thereby enhancing portability.

RVM's debt to Camelot should be obvious by now. Camelot taught us the value of recoverable virtual memory and showed us the merits and pitfalls of a specific approach to its implementation. Whereas Camelot was willing to require operating system support to achieve generality, RVM has restrained generality within limits that preserve operating system independence.

10. CONCLUSION

In general, RVM has proved to be useful wherever we have encountered a need to maintain persistent data structures with clean failure semantics. The only constraints upon its use have been the need for the size of the data structures to be a small fraction of disk capacity and for the working set size of accesses to them to be significantly less than main memory.

The term "lightweight" in the title of this article connotes two distinct qualities. First, it implies ease of learning and use. Second, it signifies minimal impact on 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.

While the importance of the transactional abstraction has been known for many years, its use in low-end applications has been hampered by the lack of a lightweight implementation. Our hope is that RVM will remedy this situation. While integration with the operating system may be unavoidable for very demanding applications, it can be a double-edged sword, as this article has shown. For a broad class of less demanding applications, we believe that RVM represents close to the limit of what is attainable without hardware or operating system support.

ACKNOWLEDGMENTS

Marvin Theimer and Robert Hagmann participated in the early discussions leading to the design of RVM. We wish to thank the designers and implementors of Camelot, especially Peter Stout and Lily Mummert, for helping us understand and use their system. The comments of our SOSP shepherd, Bill Weihl, helped us improve the presentation significantly.

REFERENCES

ANDRADE, J. M., CARGES, M. T., AND KOVACH, K. R. 1989. Building a transaction processing system on UNIX systems. In *UnıForum Conference Proceedings*

BARON, R. V , BLACK, D. L , BOLOSKY, W., CHEW, J., GOLUB, D. B., RASHID, R. F., TEVANIAN, A., JR. AND YOUNG, M. W. 1987. *Mach Kernel Interface Manual.* School of Computer Science, Carnegie Mellon Univ., Pittsburgh, Pa.

BERNSTEIN, P. A., HADZILACOS, V., AND GOODMAN, N. 1987. *Concurrency Control and Recovery ın Database Systems.* Addison Wesley, Reading, Mass.

BERSHAD, B. N., ANDERSON, T. E , LAZOWSKA, E. D., AND LEVY, H. M. 1990. Lightweight remote procedure call. *ACM Trans. Comput. Syst. 8*, 1 (Feb.).

BIRRELL, A. B., JONES, M. B., AND WOBBER, E. P. 1987. A sımple and efficient implementation for small databases. In *Proceedings of the 11th ACM Symposium on Operatıng System Prıncıples* ACM, New York.

CHANG, A. AND MERGEN, M. F. 1988. 801 Storage: Architecture and programming. *ACM Trans. Comput Syst. 6*, 1 (Feb)

CHEW, K.-M., REDDY, A. J., ROMER, T. H , AND SILBERSCHATZ, A. 1993. Kernel support for recoverable-persistent virtual memory. In *Proceedings of the USENIX Mach III Symposıum.* USENIX Association, Berkeley, Calif.

COOPER, E C. AND DRAVES, R. P. 1988. C threads. Tech. Rep. CMU-CS-88-154, Dept. of Computer Science, Carnegie Mellon Unıv., Pittsburgh, Pa.

EPPINGER, J. L 1989. Virtual memory management for transaction processing systems. Ph.D thesis, Dept. of Computer Scıence, Carnegıe Mellon Univ., Pittsburgh, Pa.

EPPINGER, J. L., MUMMERT, L. B., AND SPECTOR, A. Z. 1991. *Camelot and Avalon.* Morgan Kaufmann, San Mateo, Calif.

GARCIA-MOLINA, H. AND SALEM, K. 1987. Sagas. In *Proceedings of the ACM Sigmod Conference.* ACM, New York.

GOOD, B., HOMAN, P. W., GAWLICK, D. E., AND SAMMER, H. 1985. One thousand transactions per second. In *Proceedıngs of IEEE Compcon.* IEEE, New York.

GRAY, J. 1978. Notes on database operating systems. In *Operatıng Systems· An Advanced Course.* Springer-Verlag, New York.

GRAY, J. AND REUTER, A. 1993. *Transaction Processıng: Concepts and Technıques.* Morgan Kaufmann, San Mateo, Calif.

HASKIN, R., MALACHI, Y., SAWDON, W., AND CHAN, G. 1988. Recovery Management in QuickSilver. *ACM Trans. Comput. Syst. 6*, 1 (Feb.).

KISTLER, J. J. AND SATYANARAYANAN, M. 1992. Disconnected operation in the Coda file system. *ACM Trans. Comput. Syst. 10*, 1 (Feb.).

KUMAR, P. AND SATYANARAYANAN, M. 1993. Log-based directory resolution in the Coda file system. In *Proceedings of the 2nd Internatıonal Conference on Parallel and Distributed Informatıon Systems.* ACM, New York.

LAMPSON, B. W. 1981. Atomic transactions. In *Distributed Systems—Architecture and Implementation*. Springer-Verlag, New York.

LAMPSON, B. W. 1983. Hints for computer system design. In *Proceedings of the 9th ACM Symposium on Operating Systems Principles*. ACM, New York.

LEFFLER, S. L., MCKUSICK, M. K., KARELS, M. J., AND QUARTERMAN, J. S. 1989. *The Design and Implementatıon of the 4.3BSD Unix Operating System*. Addison-Wesley, Reading, Mass.

LISKOV, B. H. AND SCHEIFLER, R. W. 1983. Guardians and actions: Linguistic support for robust, distributed programs. *ACM Tran. Program. Lang. 5*, 3 (July).

MASHBURN, H. AND SATYANARAYANAN, M. 1992. *RVM User Manual*. School of Computer Science, Carnegie Mellon Univ., Pittsburgh, Pa.

MOSS, J. E. B. 1985. *Nested Transactions: An Approach to Reliable Distributed Computing*. MIT Press, Cambridge, Mass.

NETTLES, S. M. AND WING, J. M. 1992. Persistence + Undoability = Transactions. In *Proceedings of HICSS-25*. IEEE Computer Society Press, Los Alamitos, Calif.

O'TOOLE, J., NETTLES, S., AND GIFFORD, D. 1993. Concurrent compacting garbage collection of a persistent heap. In *Proceedıngs of the 14th ACM Symposıum on Operatıng System Prıncıples*. ACM, New York.

OUSTERHOUT, J. K. 1990. Why aren't operating systems getting faster as fast as hardware? In *Proceedings of the USENIX Summer Conference*. USENIX Assocıation, Berkeley, Calif.

PATTERSON, D. A., GIBSON, G., AND KATZ, R. 1988. A case for redundant arrays of inexpensive disks (RAID). In *Proceedings of the ACM SIGMOD Conference*. ACM, New York.

ROSENBLUM, M. AND OUSTERHOUT, J. K. 1992. The design and implementation of a log-structured file system. *ACM Trans. Comput. Syst. 10*, 1 (Feb.).

SATYANARAYANAN, M. 1991. *RPC2 User Guide and Reference Manual*. School of Computer Science, Carnegie Mellon Univ., Pittsburgh, Pa.

SATYANARAYANAN, M., KISTLER, J. J., KUMAR, P., OKASAKI, M. E., SIEGEL, E. H., AND STEERE, D. C. 1990. Coda: A highly available file system for a distributed workstation environment. *IEEE Trans. Comput. 39*, 4 (Apr.).

SATYANARAYANAN, M., STEERE, D. C., KUDO, M., AND MASHBURN, H. 1992. Transparent logging as a technique for debugging complex distributed systems. In *Proceedings of the 5th ACM SIGOPS European Workshop*. ACM, New York.

SERLIN, O. 1991. The history of DebitCredit and the TPC. In *The Benchmark Handbook*. Morgan Kaufman, San Mateo, Calif.

SPECTOR, A. Z. 1991. The design of Camelot. In *Camelot and Avalon*. Morgan Kaufmann, San Mateo, Calif.

STOUT, P. D., JAFFE, E. D., AND SPECTOR, A. Z. 1991. Performance of select Camelot functions. In *Camelot and Avalon*. Morgan Kaufmann, San Mateo, Calif.

TRANSARC CORP. 1991. *Encina Product Overview*. Transarc Corp., Pittsburgh, Pa.

UNIX SYSTEM LABS. 1993. *TUXEDO System Product Overvıew*. Unix System Laboratories, Summit, N.J.

WING, J. M. 1991. The Avalon language. In *Camelot and Avalon*. Morgan Kaufmann, San Mateo, Calif.

WING, J. M., FAEHNDRICH, M., MORRISETT, G., AND NETTLES, S. M. 1992. Extensions to standard ML to support transactions. In *ACM SIGPLAN Workshop on ML and its Applications*. ACM, New York.

YOUNG, M. W. 1989. Exporting a user interface to memory management from a communication-oriented operating system. Ph.D. thesis, Dept. of Computer Science, Carnegie Mellon Univ., Pittsburgh, Pa.

YOUNG, M. W., THOMPSON, D. S., AND JAFFE, E. 1991. A modular architecture for distributed transaction processing. In *Proceedings of the USENIX Wınter Conference*. USENIX Association, Berkeley, Calif.

Received July 1993; revised August 1993; accepted October 1993