Nephele: A Simple Solution for Data Replication

João Carreira, Howard Mao, Nathan Pemberton

May 11, 2015

Abstract

In large-scale distributed systems, failures are the norm rather than the exception. To cope with hardware and software failures, developers mostly make use of two main techniques: persisting data to a non-volatile storage device such as a hard drive or storing data in a distributed storage system such as a DBMS or key-value store. While the first approach is slow and leaves the program's progress disk-bound, the second approach requires the usage of complex APIs that require serialization of user's data structures.

To solve this problem, we built Nephele, a framework that provides efficient replication of in-memory data structures through a simple API. Nephele replicates a program's data to a remote node through RDMA providing snapshots of program's data with latency on the order of a few microseconds. The framework provides a transactional interface to users that guarantees atomicity and durability even in the face of failures.

Nephele consists of two layers: a transactional layer for recoverable virtual memory (RVM) and a remote memory storage layer. The user-facing transaction layer provides an API consisting of 15 methods and is responsible for detecting changes and replicating changes at commit time. The remote memory storage layer is responsible for storing user's data in a remote node over RDMA. For this layer, we have implemented two backends: one using a custom RDMA protocol with a custom server, and one using RAMCloud, an RDMA-optimized key-value store.

To demonstrate the flexibility and performance of our system, we applied our framework to three applications: a genomic assembly program, an in-memory file system, and a vector-matrix multiplication application. We show that our framework provides data replication efficiently through a simple to use API.

1 Introduction

As the number of nodes in distributed systems increases, failures become the rule, not the exception. Because of this, it is important to be able to recover from crashes quickly. In addition, the application should not be made significantly more complicated by the addition of recovery code.

While it would be overly ambitious to try and solve this entire goal, we propose a tool that could simplify such a solution by providing the ability to replicate the application's working state to another failure domain. In this report, we present an implementation of Recoverable Virtual Memory (RVM), an API that allows the programmer to easily add state checkpointing and recovery to their application. RVM operates by detecting modifications to recoverable memory regions and replicating the memory to a remote node.

The proliferation of high-performance RDMA and future disaggregated memory systems offer an opportunity to perform this replication efficiently. For example, the FireBox warehouse-scale computer (an ASPIRE lab project) will have a central pool of universally accessible DRAM and non-volatile memory. Currently there is no FireBox hardware to experiment with, but we do have an infiniband based cluster (Firebox-0). So far, we have implemented two backends, one using the Infiniband RDMA API and one using the RAMCloud key-value store.

1.1 Current Solutions

The need to preserve critical data in the event of a hardware failure is not new. There are a number of popular methods of addressing this problem. One is to checkpoint the entire operating system process using a tool like the Berkeley Lab Checkpoint Restart library [4] or Condor [1]). Process checkpointing is appealing because it requires little to no changes in the application. For this convenience, the technique sacrifices efficiency. All data must be checkpointed, not just the critical state, and operating system state must be quiesced. A common practice to avoid avoid copying the entire program state is to manually serialize critical data structures and write them to a file. While, in theory, this technique copies the minimum amount of data, in practice it can be difficult to identify which state has actually changed. The user is forced to pessimistically replicate most critical state

rvm_cfg_[create/destroy]()	Initialize the system and recover memory if needed
rvm_[alloc/free]()	Allocate a region of recoverable memory
rvm_txn_[start/commit]()	Mark a point of consistency in the program
rvm_[set/get]_usr_data()	Register a pointer to your state.

Table 1: RVM API

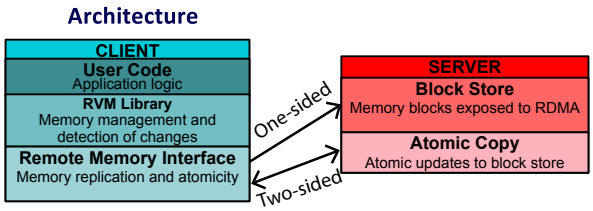

Figure 1: RVM lasagna diagram

on each checkpoint. Manual serialization also leads to significant increases in code complexity. Each data structure must have two definitions, one for on-disk and the other for in-memory. Maintaining this serialization code can be time-consuming and error-prone. Finally, databases are commonly used to replicate critical state. Databases provide clean transactional semantics that can be appealing for high-availability applications. Databases, however, often have a complex interface that requires manual serialization. They also provide more features than are required for state replication, which leads to poor performance.

1.2 RVM Interface

The RVM interface is designed to be as unobtrusive as possible. Users should be able to preserve just their critical state without worrying about re-writing pointers or packing data into a file. To do this our framework requires only that the user identify which memory is considered critical, and identify points of consistency in their code. By marking a point of consistency the user certifies that, if the program we're to restart with the critical memory in the current state, the program would be able to continue execution. Critical state is identified by allocating it from a special recoverable malloc function and consistency points are identified through a transactional interface. Users may also save a special **user_data** pointer that survives failures. This pointer typically stores a state structure that can address the recoverable state in an application-dependent fashion. Table 1 lists the entire required API.

In practice, of course, it is not this simple. Code must be written in such a way that recovery is possible. In addition, while critical local state is preserved, external state (like open files or sockets) is not. RVM is intended as a low-level library that can be exploited by more full-featured recovery libraries. An analogous relationship can be seen in the GASNet [2] library which can be used directly, but is really intended as a low-level interface for global address space languages.

2 Design

The library consists of two main layers on the client side. An upper layer to implement the RVM-specific logic and a lower-layer remote memory (RMEM) backend to implement low-level transport and atomicity. This relationship is described in Figure 1. This modularization allows for a number of different backends to be tried independently of user code and the RVM layer. This enhances portability and simplifies future efforts to improve performance or semantics.

2.1 The RVM API

The RVM layer is responsible for implementing the custom allocator and identifying changes to memory. It is also responsible for tracking commit points.

2.1.1 The Block Table

The RVM layer thinks in terms of *blocks*, fixed-sized regions of memory that are persisted atomically. Most functionality is based around the *block table*, a persistent data structure that keeps track of each allocated block in the system. This table is replicated using the same mechanism as any other recoverable memory. Like a filesystem master boot record, the first page of the block table is always stored with a constant identifier in the RMEM layer. After that, the block table is self-describing and can be recovered using the mechanisms described below.

Each entry in the block table contains a local address where the active block lives on the client and a remote identifier that can be used to identify the block in the RMEM layer.

2.1.2 Initialization and Recovery Procedure

When `rvm_cfg_create()` is called the first time, it initializes the block table to an empty state and persists it in the RMEM layer. When recovering, `rvm_cfg_create()` fetches the first block of the block table from the RMEM layer. The block table is walked from start to finish, fetching each block as it goes.

Even if the block table takes up multiple blocks, each one is fetched in order, ensuring that all data can be found eventually. When RVM fetches a remote block, it must ensure that it is loaded to the same address it was at before failure, otherwise pointers in the data would no longer be valid. The original address is read from the block table and then allocated using the *mmap()* system call. To ensure that these addresses are always available, RVM requires that any OS address space layout randomization be disabled, and that `rvm_cfg_create()` be called before any other local allocations.

2.1.3 Allocation

To ensure that memory is recoverable, the user must allocate it using a special `rvm_alloc()` function. The `rvm_alloc()` function allocates memory both locally and on the remote node. Any modifications to the local pages allocated by `rvm_alloc()` are automatically detected and copied to the remote node at commit time. Detection is achieved through the use of *mprotect()*, a Linux system call that can be used to make the application take an interrupt whenever a page is written. Our custom interrupt handler then marks the page as changed, removes the memory protection and returns. This means that RVM needs to be involved only in the first modification to a page.

2.1.4 Marking a Point of Consistency

The user is required to identify points in their code where the state of recoverable memory is considered ̈consistent:̈ This means that recovery is possible from that particular state. `rvm_txn_commit()` can be called at these points to ensure that memory is atomically persisted. Upon entering `rvm_txn_commit()`, RVM goes through the list of changed pages and copies them to a shadow page in the RMEM layer. This ensures that a consistent version of memory is always available, even if the client crashes during checkpointing. When all the pages have been copied, an `atomic_commit()` function (provided by the RMEM layer) is called to persist the changes.

2.2 The Remote Memory Layer

Underlying the RVM API is the remote memory (RMEM) layer, which provides the basic operations that RVM uses to communicate with the backing data store. The essential operations in the RMEM layer are `malloc()`, `free()`, `put()`, `get()`, and `atomic_commit()`.

The `malloc()` function allocates memory in the backing store. The function arguments include the number of bytes to allocate as well as a unique tag that is associated with that memory region. If the tag has not already been taken, `malloc()` allocates a new memory region and returns the starting address. If a memory region with that tag already exists, the starting address of the previously allocated region is returned.

The `free()` function takes a tag as its argument and frees the memory region associated with that tag.

There are also `multi_malloc()` and `multi_free()` functions which allocate and free multiple memory blocks. Depending on the backend, these functions might coalesce `malloc()` and `free()` requests in order to decrease the number of round trips to the backing store.

The `put()` and `get()` functions copy data to and from the backing store, respectively. These operations are not atomic, so the RVM layer always performs puts and gets onto a shadow page and then copies the data to the real page using `atomic_commit()`.

As mentioned before, `atomic_commit()` takes an array of source tags and an array of destination tags. It instructs the server to copy the data in the source memory regions to the destination regions. This copying is done atomically, so there is no danger of only a portion of the pages being copied due to the client crashing.

2.3 Infiniband Backend

Our primary backend uses Infiniband Remote Direct Memory Access (RDMA) to talk to a server managing a large pool of memory. At startup, the remote memory server maps in a large block of system memory. The server then listens for connections over the Infiniband fabric. When a client connects, the server pins the memory in the page table and registers it with the infiniband drivers. Registering with the infiniband drivers provides a local key and a remote key. The server transmits the remote key and starting address to the client. This allows the client to perform one-sided RDMA operations to the remote memory without the server's mediation.

The `put()` and `get()` commands are implemented using one-sided RDMA writes and reads. The other commands are implemented using two-sided sends and receives. For these commands, the client and server each allocate two message structs: one for sends, and one for receives. In a two-sided transmission, the recipient first posts a receive request to the Infiniband driver. This receive request specifies the local key and address of the receive struct. When the sender posts a corresponding send request using the local key and address of its send struct, the infiniband drivers copy the data from the sender's send struct to the recipient's receive struct and notify sender and recipient of

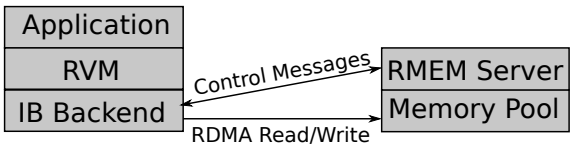

Figure 2: IB Backend Architecture

the operation.

2.3.1 Allocation

The `malloc()` operation is implemented by sending an ALLOC request to the server. When it receives this request, the server will allocate a block of memory form the memory pool and mark it with the given tag. The server then sends a MEMRESP message back to the client containing the starting address of the allocated block.

The `free()` operation is implemented by sending a TXN_FREE request to the server. A key feature of the IB backend is that the server does not immediately perform a free operation when it receives the TXN_FREE message. Instead, it puts the free operation in a queue, which will be processed during an atomic commit. This way, the free operation is transactional. Once the server receives the message and queues the free operation, it sends a TXN_ACK message back to the client, allowing the client to send another command.

There are also MULTI_ALLOC and MULTI_TXN_FREE requests which can encode up to 20 allocation or free requests (this number if configurable at compile time). The server responds to a MULTI_ALLOC request with a MULTI_MEMRESP response, which contains an array of addresses, one for each tag in the MULTI_ALLOC request. The server responds to MULTI_TXN_FREE with a TXN_ACK.

2.3.2 Commit

The `atomic_commit()` operation involves two different message types. The first is the MULTI_TXN_CP message, which instructs the server to copy a set of source blocks to a set of destination blocks. However, as with TXN_FREE, the copy does not occur immediately. When the client sends the server a TXN_GO request, the server performs all requested copies and frees. In our failure model, we assume that the server will not crash. So even if the client crashes after sending TXN_GO, the copies and frees will still be performed to completion. If the client crashes before sending TXN_GO, all of the outstanding copy and free requests will be flushed and no changes will occur.

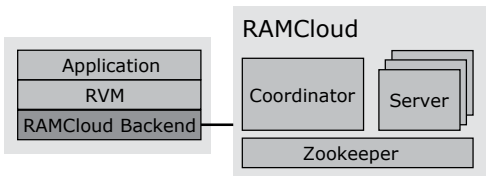

Figure 3: RAMCloud backend layer operating along side RAMCloud.

Figure 4: Diagram of tag/key mapping transformation during commit for a single memory region.

2.3.3 Recovery

If a client reconnects after a crash, the IB server transmits the tag to address mappings left over from the previous run to the client. The mappings are transmitted to the client in groups of twenty through TAG_ADDR_MAP messages. The client acknowledges each TAG_ADDR_MAP message with a STARTUP_ACK message.

2.4 RAMCloud Backend

To investigate the performance and suitability of a key-value store as a block device, we developed an RMEM layer on top of RAMCloud, a low-latency key-value store. In RAMCloud, blobs of memory (values) are identified by keys (strings). When running, RAMCloud is composed of three main executing instances: a coordinator, a server and Zookeeper (see Figure 3). Each server is responsible for storing and serving data (values). The coordinator is responsible for keeping track of all servers alive and for keeping track of where data is stored in the system. A Zookeeper instance is used for leader election and for storing configuration data.

Because RAMCloud's data is referenced by keys, this layer keeps a map structure that associates each tag to a key. This means that each recoverable memory region can be uniquely identified by a key.

To provide atomicity and durability of the tag/key mapping, this backend keeps a special entry in RAMCloud with each tag-key association. This table is read each time the software layer is started and written once for each commit. This entry can be atomically written with a sngle *p*ut operation.

The RMEM's layer operations are implemented in the following way:

Connect During connection the backend creates a RAMCloud client instance that is responsible for establishing a connection to the RAMCloud server. If it is not the first time this connection is performed, i.e., if the client is under recovery, the backend recovers each tag-key mapping. Otherwise, this layer initializes a RAMCloud table and stores an empty master entry in the RAMCloud's server.

Allocate During allocation, first the backend creates a key that identifies the memory being allocated in the RAMCloud server. Secondly, RAMCloud initializes this memory.

Write To write a memory region (identified by a tag) the backend fetches the tag's corresponding key and issues a *p*ut operation with that key and corresponding data.

Read Likewise, to read a memory region the backend issues a *g*et operation with the tag's corresponding key. The data read from RAMCloud is copied to the final destination.

Commit To perform commit, the backend constructs a new master entry where each tag points to the key of the shadow memory being committed (see Figure 4). Likewise, the tag for each of the shadow memory regions is made to point to the key of the old memory region (rephrase). Once this master entry is constructed in the backend, it is written atomically to RAMCloud.

Disconnect To disconnect, the backend clears the main data structures (e.g., local tag-key map).

Our design is simplified by the fact that our framework only replicates data to a single node. This means that we do not have to coordinate replicas.

3 Evaluation

We characterized the performance of our RVM backend implementations by writing several benchmark applications and running them on our Firebox-0 cluster. Firebox-0 is a research cluster built by Berkeley's ASPIRE lab and consists of high-end commercial off-the-shelf server connected by Mellanox Infiniband switches.

3.1 Micro Benchmarks

We used micro-benchmarks to test the performance of RVM commits and RVM recovery. To test commits, we allocated a number of pages, made modifications to each one of them, and measured the time it took for `rvm_txn_commit()` to complete. To test recovery, we first allocated a number of pages and close the connection to the server. We then restart the connection using the recovery flag and time how long it takes `rvm_cfg_create()` to complete. When benchmarking the IB backend, we run three trials for each number of pages, restarting the server after each trial.

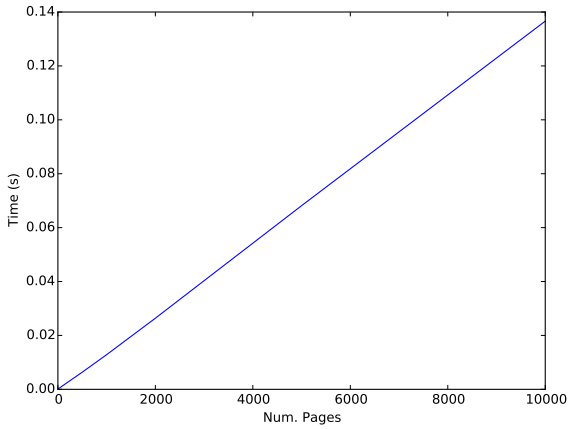

Figure 5: IB Commit Micro-Benchmark Results

3.2 IB Backend

Figure 5 shows the results of the commit micro-benchmark using the IB backend. At the smallest page count, the commit time is around 150 microseconds. At 10K pages, the commit time has increased to 140 milliseconds. As can be seen in the graph, the relation between page count and commit time is fairly linear, with a slope of about 13.7 microseconds per page. From examining the performance of `rvm_txn_commit()` using the profiler, we find that commit time is taken up mostly by sending the MULTI_TXN_CP commands to the server, as well as performing RDMA writes to the remote memory pool. Since these the number of commands and writes that need to be done scales linearly with the number of pages, this accounts for the linear increase.

Figure 6 shows the results of the recovery micro-benchmark using the IB backend. Recovery is quite a bit slower than commit. Recovering a single page takes about 50 milliseconds, while recovering 10,000 pages takes more than 2 seconds. The relationship between number of pages and recovery time is also not entirely linear. The slope seems to increase at about 5000 pages.

From profiling, we find that a large portion of time in recovery is spent fetching the data for the recovered pages using RDMA reads. However, an even larger

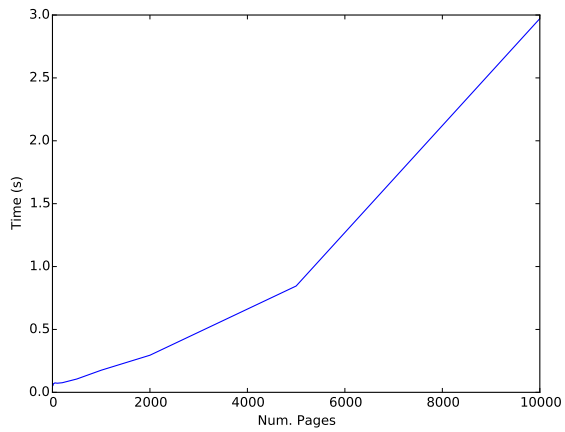

Figure 6: IB Recovery Micro-Benchmark Results

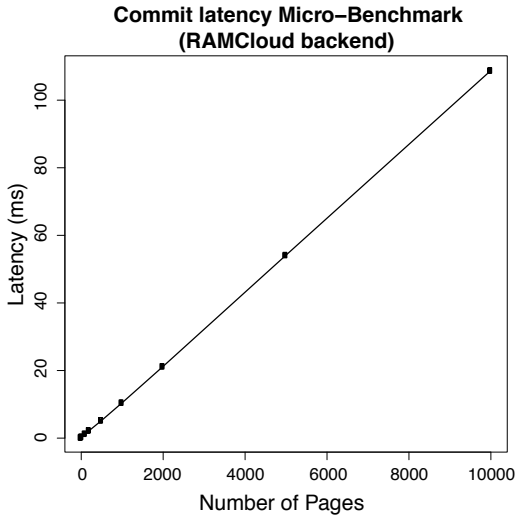

Figure 7: Commit time micro-benchmark when using the RAMCloud backend

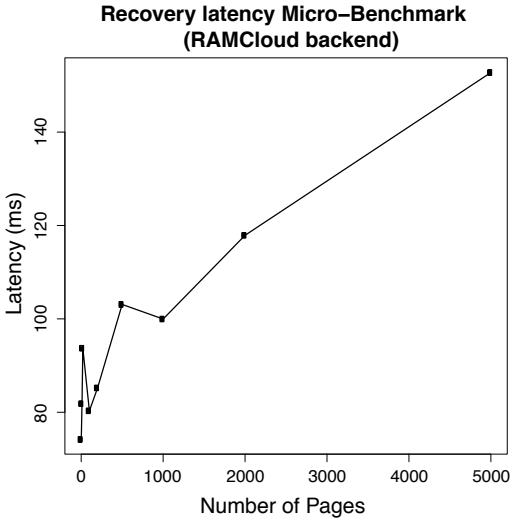

Figure 8: Recovery time micro-benchmark when using the RAMCloud backend

portion of the time is spent registering pages with the Infiniband driver. There is a potential optimization we could make here. A lot of the pages being allocated are contiguous and so can be registered all at once. Instead, we register each page one by one. Similarly, data for contiguous pages could be fetched by a single RDMA read, instead of one read for each page.

We are uncertain what is causing the non-linear increase in recovery time. It is possible that registering pages with the Infiniband driver takes longer as the total number of pages registered grows larger.

3.3 Ramcloud Backend

Figure 7 shows the results of the commit micro-benchmark when using the RAMCloud backend. The end-to-end latency of a commit operation is dominated by the number of pages to be committed, as each page write requires a round-trip to the RAMCloud server. The RAMCloud backend also needs to save the table that contains the mapping between tags and keys in, but this only requires one interaction with the server. The RAMCloud backend requires roughly 10 microseconds to write a page to RAMCloud. As can be seen in the graph, the commit time grows linearly with the number of pages as expected.

Figure 8 shows the results of the recovery micro-benchmark. The end-to-end recovery time is mostly dominated by the time to establish a connection with the RAMCloud server (roughly 80ms). Once the connection is established the RAMCloud backend retrieves the pages previously committed. The time to do this is dominated by the communication with the server, as in the commit benchmark, which grows linear with the number of pages. Because of this we see a linear recovery time increase as the number of pages increases.

3.4 BLCR Microbenchmark

To compare our implementation against an existing tool, we ran similar microbenchmarks for the Berkeley Labs Checkpoint Restart (BLCR) tool. The microbenchmark maps in a given number of pages of memory and makes modifications to it. We measure commit time by having the process call the "cr_checkpoint" program on its own PID and timing how long it takes for the checkpoint program to run. The "cr_checkpoint" program saves the process state to a file. We can measure recovery time by running

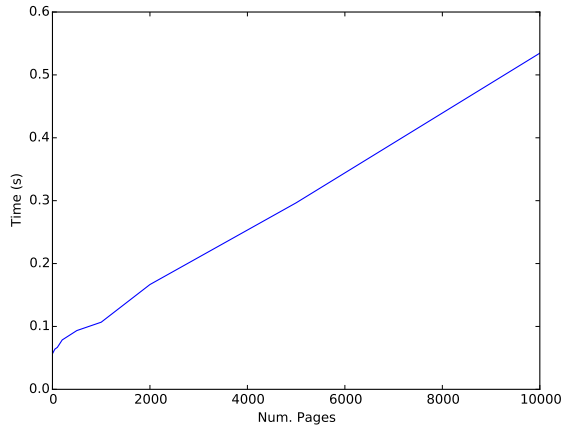

Figure 9: BLCR Commit Micro-Benchmark Results

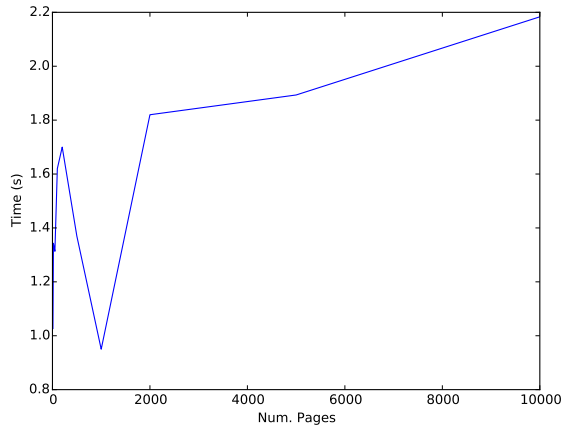

Figure 10: BLCR Recovery Micro-Benchmark Results

"cr_restart" on the checkpoint file and timing how long it takes for the program to run to completion.

As seen in Figure 9, commit latency for BLCR is considerably longer than for either of the two RVM backends. For a single page, commit takes approximately 60 milliseconds, and the commit time increased by 47 microseconds per page.

Recovery time (Figure 10 does not show a clear trend as the number of pages increases, but the time is always rather slow, taking around one to two seconds to restart the application.

3.5 DGEMV

Our first attempt at making a realistic application recoverable was an iterative dense matrix-vector multiply (DGEMV) code. This application repeatedly multiplies a vector by a constant matrix. Since the matrix is constant, the only state that needs to be preserved is the vector and the last successful iteration. This makes DGEMV somewhat of a best-case scenario for traditional recovery schemes. The state is a single large allocation that changes entirely on each iteration. In the following experiments, our RVM implementation is compared with a manual serialization scheme that writes to SSD's on our experimental system. The matrix was chosen to be of dimension 1Mx100 with 100 iterations in order to maximize the critical state and stress the recovery schemes.

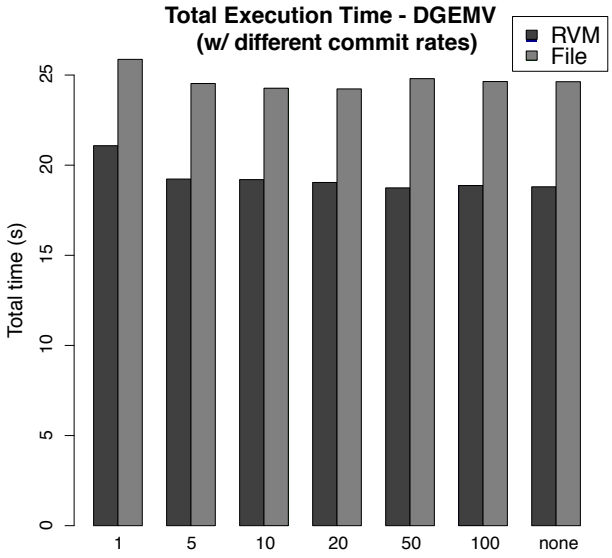

Figure 11: Total time of DGEMV application when committing at different rates

In figure 11 we vary the commit rate (in terms of iterations) from 1 (every time) to 100 (only one commit) without failures. This essentially measures the overhead caused by recovery. In all cases, RVM introduces less overhead. This is likely due to the efficient nature of RVM checkpoints. More interesting, however, is when we introduce failures. In figure 12, we always commit on each iteration, but inject failures after different numbers of iterations (from each iteration up to a single failure). In this case, the file-based backend is faster than RVM for high failure rates, but slower with low failure rates. We believe this is due to a higher startup cost for the file-based approach. It also demonstrates that reading a relatively large, sequential file from an SSD is very performant and competes will with RDMA. We believe that further optimizations in the RMEM layer could make up some of this difference, see Section 5 for some possible performance enhancements.

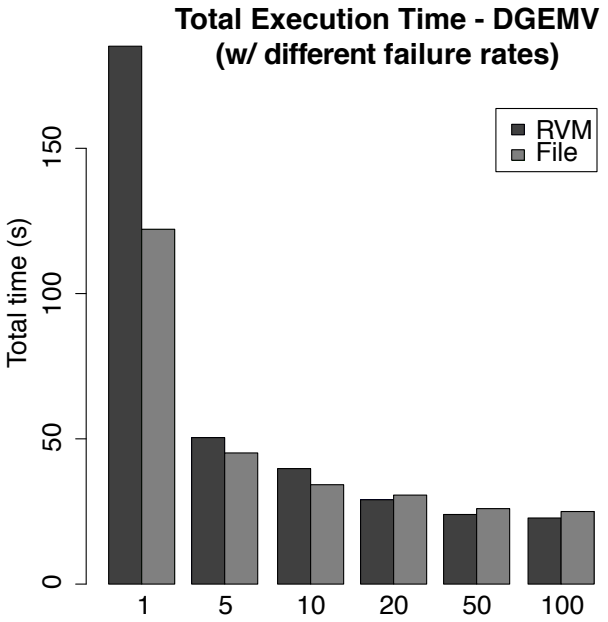

Figure 12: Total time of DGEMV application when failing/recovering at different rates

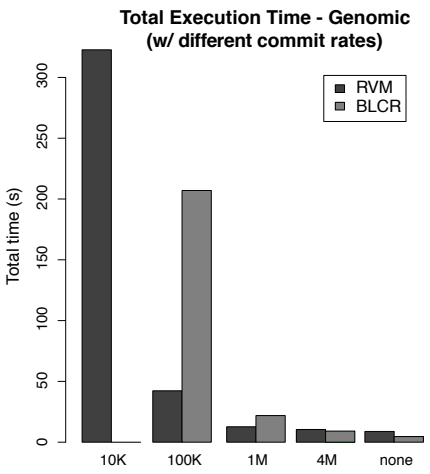

Figure 13: Total time of genomic assembly application when committing at different rates. Total time when running BLCR at a rate of 10K is not shown

3.6 Gene Assembly

For a more complex benchmark, we ported the de novo genome assembly code that was the basis of homework 3. This code involves the use of a large hash-table (rough 100MB with our test input) as well as several linked-lists and other complex data structures. In some ways, this is a best-case scenario for RVM. On each checkpoint, only a fraction of the data is changed, and much of the memory is constant (the input file). While we initially intended to implement a manual serialization as we had done for DGEMV, it quickly became apparent that such an undertaking would be almost as involved as writing the application in the first place. We instead decided to compare it against a popular process-level checkpointer called BLCR [4]. While DGEMV needed only to store the vector and an iteration number, the genome assembly code was considerably more complicated. Execution consists of two main phases. In the build phase, kmers are read from a file and inserted into a hash table. The probe phase probes into this hash table in order to construct the final contigs. To capture this pattern, a global state structure was allocated from recoverable memory. This global state stores shared structures such as the hash table and start-kmer list, which phase is currently being executed (build or probe), and then provides an opaque phase state pointer to be filled in by phase-specific code. The build phase keeps track of where in the input file it was, while probes state is more complex. It needs to preserve which start-kmer it was processing, where in the current contig it was, and where in the output file it was writing. Each phase processed all 4 million kmers, for a total of 8 million processing steps.

In figure 13 we vary the commit rate (in terms of number of processing steps) from every 10 thousand (out of 8 million) up to 4 million (one checkpoint per phase). At higher commit rates, RVM clearly outperforms BLCR. In the exreme, RVM was able to complete in 800 seconds with a commit rate of 10K, while we were forced to cancel the BLCR run after 30 minutes. Even at more reasonable commit rates such as 1 million (8 commits), RVM was nearly twice as fast as BLCR. It's only at the extreme low end of commit rates (4 million and no commit) that BLCR was faster. This is because RVM pays a significant up-front cost to allocate it's large recoverable state. Further analysis showed that of the 10 seconds spent at a 4 million commit rate, a full 5s were spent simply allocating memory, while build took only 2s and probe only 1s. This clearly indicates a performance bottlenect in our system, although we do not believe it is fundemental to the approach. See Section 5 for a discussion of possible performance improvements.

3.7 In-Memory Filesystem (RvmFS)

To demonstrate the flexibility and applicability of our framework we have applied our framework to RvmFS, a VFS compliant in-memory file system (see Figure 14). RvmFS uses main memory as the only storage medium to provide very fast writes and reads. To be able to use our framework we have developed the file system

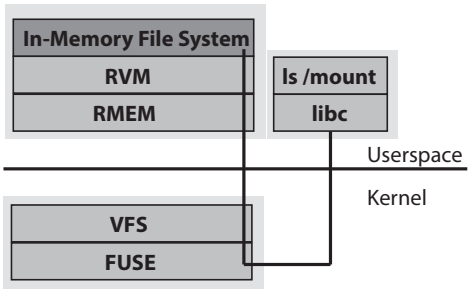

Figure 14: In-memory file system design when using the RVM framework and FUSE library.

using FUSE, a kernel module that allows the creation of user-space file systems. RvmFS performs a commit of the file system in different situations: 1) an inode is created, 2) when a file is closed and 3) when a *sync* operation is performed.

To evaluate the performance of the system we ran a subset of the benchmarks in the Filebench [3] benchmark suite. We compare the performance of the benchmark when running it against an ext4 file system backed by an SSD drive and when running on RvmFS backed by a remote server. Table 2 describes each specific benchmark that we ran and table 3 shows the performance obtained. We show the average latency of each operation in each benchmark.

Benchmark name	Description
file_micro_create	Create an empty file and issue 1024 appends of 1MB each
ramdomread	Random reads (8K) from a file with size 1Mb
openfiles	Creates a fileset with 500 empty files, then proceeds to open each one.

Table 2: Description of the macro-benchmark tests used to evaluate RvmFS.

Discussion As shown in Table 3, RvmFS is considerably slower for some specific benchmarks while obtaining similar performance in others. The poor performance of the file system stems from different reasons. First, some of the file system operations are not optimized for performance. For instance, when extending a file length, RvmFS allocates a new region of memory and copies all the file contents to the new region. Not only copying all the data is slow, this also means that the next commit operation will backup the whole file contents even if only a small part of the file changed. Secondly, RvmFS currently does not support multi-threaded access, common in modern file systems such as ext4. Thirdly, RvmFS is ultimately limited by the performance of RVM. Due to the high overheads involved when backing up large contiguous regions of memory in RVM, RvmFS suffers. Finally, RvmFS is

Benchmark name	Latency per Op. (SSD)	Latency per Op. (RvmFS)
file_micro_create	append-file: 292us	2967.7ms
randomread	Read: 25us	Read: 25us
openfiles	Open/close: 590us	Open: 2842us, Close: 740us

Table 3: Macro benchmark results of RvmFS

not built with locality of storage in mind. This means that simple operations can touch many files.

Next we comment on the performance of each benchmark.

file_micro_create The performance of RvmFS is far from the performance of ext4. This has to do with the inefficiency of RvmFS when extending the length of a file, as previously described.

randomread In this benchmark RvmFS has similar performance to the ext4 file system. Because the file being used to benchmark the read operations is small it can fit in memory. This favours ext4, which can rely on the kernel buffers to directly serve data.

openfiles In this benchmark the close operation in RvmFS is on par with the performance of ext4. On the other hand, the open operation is roughly four times slower.

4 Related Work

Virtual machine / Container checkpoint Systems such as Tardigrade [5] or VMWare provide data fault-tolerance by checkpointing containers and virtual machines, respectively. While these systems can checkpoint a program's data without knowing the program's internals they still have limitations. First, using virtual machines incurs a non-negligible performance overhead on virtualized applications. Secondly, checkpointing an entire virtual machine or container can be much more expensive than necessary. We believe RVM provides a small API that can be used to checkpoint only the data that matters to the user, and thus it can provide better performance with minimal developer effort.

Key-value stores and DBMSs Key-value stores such as RAMCloud and DBMSs such as Postgres can be used to store a program's data to remote nodes and provide similar properties as RVM. While we believe that many of the techniques and lessons used in these systems can be applied in RVM, we think these systems are not a good fit for the replication of in-memory data structures. First, most modern key-value stores do not provide atomic multi-key writes – required to atomically store large memory regions. Secondly, the

API provided by key-value stores despite being simple is not suitable for virtual memory replication, forcing developers to serialize data structures into a suitable format. Likewise, DBMSs sacrifice data access latency in favour of database features that are not required in this context. Additionally, the schema model required by traditional DBMSs does not fit well with arbitrarily-sized regions of memory.

Virtual memory replication Systems like Mojim [7] or LRVM [6] can be used to replicate virtual memory. However, Mojim's interface is considerably more complicated than ours. Mojim provides a virtual filesystem and the user allocates recoverable memory by memory mapping files. Mojim does not ensure that the recoverable pages are mapped into the same space in virtual memory, so pointers cannot be properly recovered. In addition, for each commit, the user must explicitly specify what memory to replicate, as the modified pages are not automatically detected as in our system.

5 Conclusion and Future Work

In this paper, we presented Recoverable Virtual Memory, our solution for easy replication and recovery of application state. We have attempted to provide our implementation with features we believe are highly desirable for the application programmer, such as a simple and understandable API; restoration of virtual address locations, which allows for recovery of complex data structures without the need for serialization; and reasonably low overhead. Of these three goals, we have accomplished the first two, but there is still considerable work to be done in regards to performance.

The most obvious optimization we could make is to coalesce RDMA operations for contiguous pages. In our current infiniband backend, allocation of the backing memory for multiple contiguous pages must be done a page at a time. However, it would be much more efficient to perform a single contiguous allocation on the server side. That way, the number of RDMA write calls and TXN_MULTI_CP messages does not need to increase as the number of pages increases. There are probably other areas for performance improvement that we could discover through more intensive profiling of the benchmark programs.

Another avenue we could explore are alternative backends for the RMEM layer. There are various networking and non-volatile memory technologies that we could investigate, such as SSDs and RDMA over Converged Ethernet. We could also implement different consistency semantics to explore the tradeoffs of performance and consistency.

Finally, we would like to integrate RVM with existing runtimes and recovery frameworks to provide a more complete data replication and recovery solution.

The complete code for our RVM implementation is available on GitHub at https://github.com/zhemao/rmem-server.

References

[1] `https://research.cs.wisc.edu/htcondor/checkpointing.html`.

[2] `http://gasnet.lbl.gov/`.

[3] `http://sourceforge.net/apps/mediawiki/filebench/index.php`, 2013.

[4] J. Cornwell and A. Kongmunvattana. Efficient system-level remote checkpointing technique for blcr. In *Information Technology: New Generations (ITNG), 2011 Eighth International Conference on*, pages 1002–1007, April 2011.

[5] J. R. Lorch, A. Baumann, L. Glendenning, D. Meyer, and A. Warfield. Tardigrade: Leveraging lightweight virtual machines to easily and efficiently construct fault-tolerant services. In *12th USENIX Symposium on Networked Systems Design and Implementation (NSDI 15)*, pages 575–588, Oakland, CA, May 2015. USENIX Association.

[6] M. Satyanarayanan, H. H. Mashburn, P. Kumar, D. C. Steere, and J. J. Kistler. Lightweight recoverable virtual memory. *ACM Trans. Comput. Syst.*, 12(1):33–57, Feb. 1994.

[7] Y. Zhang, J. Yang, A. Memaripour, and S. Swanson. Mojim: A reliable and highly-available non-volatile memory system. In *Proceedings of the Twentieth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '15, pages 3–18, New York, NY, USA, 2015. ACM.