EXCITE-VM: Extending the Virtual Memory System to Support Snapshot Isolation Transactions

Heiner Litz
Stanford University
353 Serra Mall
Stanford, CA, 94305
heiner.litz@stanford.edu

Benjamin Braun
Stanford University
353 Serra Mall
Stanford, CA, 94305
bbraun@stanford.edu

David Cheriton
Stanford University
353 Serra Mall
Stanford, CA, 94305
cheriton@cs.stanford.edu

ABSTRACT

Multi-core programming remains a major software development and maintenance challenge because of data races, deadlock, non-deterministic failures and complex performance issues. In this paper, we describe EXCITE-VM, a system that provides snapshot isolation transactions on shared memory to facilitate programming and to improve the performance of parallel applications. With snapshots, an application thread is not exposed to the committed changes of other threads until it receives the updates by explicitly creating a new snapshot. Snapshot isolation enables low overhead lockless read operations and improves fault tolerance by isolating each thread from the transient, uncommitted writes of other threads. This paper describes how EXCITE-VM implements snapshot isolation transactions efficiently by manipulating virtual memory mappings and using a novel copy-on-read mechanism with a customized page cache. Compared to conventional software transactional memory systems, EXCITE-VM provides up to 2.2x performance improvement for the STAMP benchmark suite and up to 1000x speedup for a modified benchmark having long running read-only transactions. Furthermore, EXCITE-VM achieves a 2x performance improvement on a Memcached benchmark and the Yahoo Cloud Server Benchmarks. Finally, EXCITE-VM improves fault tolerance and offers features such as low-overhead concurrent audit and analysis.

1 Introduction

Developing and maintaining a multi-threaded application is a challenge. Developers need to ensure that all shared mutable data items are protected by suitable synchronization, including library routines invoked by the application. This synchronization is hard to get right and when not, leads to difficult-to-diagnose problems such as data races, memory corruption and deadlock. Even when executing correctly, a multi-threaded application can have inscrutable performance, given the complex dynamic interaction between locks that can be occurring. For example, consider a graph application in which threads concurrently modify nodes while some threads analyze shared data to generate a summary. The analysis may have uncertain execution time and an uncertain effect on the rest of the application because of the conflicts between the read locks it needs to acquire and the write locks being requested and held by other threads.

In this paper, we present Elastic Snapshot-Isolation Transaction Extensions in Virtual Memory (EXCITE-VM), a system that enables snapshot isolation semantics to address the issues of multi-core programming. With our system, threads can operate on consistent snapshots of shared memory, enabling safe and protected access, avoiding data races and memory corruption. With snapshot isolation (SI) [6], shared memory does not appear to change for a given thread until it explicitly creates a new snapshot of shared memory. This enables consistent reads to shared memory in the presence of concurrent writes by other threads, reducing synchronization overheads. To continue with the graph example above, with SI, threads can checkout a consistent snapshot of the graph for analysis, while update threads can modify the graph concurrently without competing for locks or executing synchronization code. Snapshots provide further utility such as lock-less global stats generation, and concurrent garbage collection.

Besides improving parallel application development, EXCITE-VM improves fault tolerance by enabling application threads to terminate, for example due to a division by zero exception and subsequently restart without restarting the whole application because none of its writes are visible to other threads until committed. Finally, snapshots address the challenge of creating checkpoints of parallel application state while enabling concurrent writes to shared memory enabling fault tolerance and durability. A thread can simply checkout a snapshot and persist it to non-volatile storage without interfering with other application threads. EXCITE-VM, hence, allows applications to achieve persistence and durability similarly to what is provided by a traditional database without incurring the overheads and code modifications required by the latter.

Having motivated the utility of snapshots, the challenge is to achieve acceptable performance, given the overhead of creating a memory snapshot. To address this concern, EXCITE-VM introduces a novel technique for manipulating virtual memory mappings depending on the snapshot. We propose a new copy-on-read approach supported by an efficient page caching scheme to minimize the cost of generating snapshots. Combining these techniques, EXCITE-VM outperforms conventional STMs on the STAMP [32] benchmark

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than the author(s) must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.

PACT '16, September 11 - 15, 2016, Haifa, Israel

© 2016 Copyright held by the owner/author(s). Publication rights licensed to ACM.

ISBN 978-1-4503-4121-9/16/09…$15.00

DOI: http://dx.doi.org/10.1145/2967938.2967955

suite by up to 2.2x, and up to 1000x on modified benchmarks having long read-only transactions. Furthermore, EXCITE-VM provides up to 2x the performance of Libitm in Memcached and the Yahoo Cloud Server Benchmark.

2 Snapshot Generation

This section discusses existing software approaches to generate snapshots, their shortcomings and how EXCITE-VM improves over them.

2.1 Snapshots in Transactional Memory

SI-STM [43] is a Java based transactional memory system that maintains multiple versions of all shared data objects to reduce transaction abort rates. Whenever referencing an object, a separate data structure is queried which maintains the references to the individual versions. The implementation of SI-STM offers three disadvantages. First, each object dereference necessitates an additional level of indirection, second, conflict validation is performed on the granularity of objects which performs poorly for large objects and third, it requires the application developer to modify every object or class description that needs to be snapshottable.

We observe that to implement the required indirection layer, we can leverage the *virtual to physical address translation* mechanism. Utilizing hardware resources such as the translation lookaside buffer (TLB) and the page walk engine, threads obtain direct access to snapshots without incurring the overhead of an additional indirection layer. Accessing static objects incurs zero overhead while incorporating changes for multiple objects can be batched on a per 4KB page basis. EXCITE-VM also does not require programmers to modify their objects as snapshots are maintained transparently on a page based granularity.

2.2 VM Based Thread Synchronization

DThreads [29] and Conversion [30] are two systems that leverage virtual memory and per-thread page tables for synchronization purposes. The two approaches follow a computation model in which all threads start off with the same consistent state, then diverge into per-thread states by intercepting writes, i.e. Copy-on-Write (CoW), and finally form the next globally consistent state by merging all writes. DThreads and Conversion require additional synchronization logic such as locking, as both only support a last committer wins policy in which multiple writes to the same data item do not conflict but merely overwrite each other. Like Conversion, EXCITE-VM uses a threading model where each thread has its own virtual-to-physical address mapping representing a snapshot. However, EXCITE-VM differs from both DThreads and Conversion in multiple ways. First, EXCITE-VM logs committed changes, allowing threads to efficiently update their snapshots without synchronization overhead. Second, EXCITE-VM supports transactional semantics including an efficient conflict detection scheme and retry capability. Third, EXCITE-VM introduces a novel Copy-on-Read (CoR) mechanism that uses page caching to address the shortcomings of CoW and pure CoR.

Copy-on-Write (CoW) techniques as proposed in prior work [29, 30] have leveraged CoW for isolating threads by applying writes only to thread local memory, effectively hiding them from other threads. To implement this technique, pages are mapped read-only such that every write leads to a trap into the page fault handler which can then map a thread local page, copying the original state of the read only page. Under CoW, reads are served directly from the snapshot with no added overhead.

To enable snapshots via CoW, each thread must update its entire per-thread page table to the committed state at the start of each transaction. Unfortunately, this incurs the cost of invalidating TLB entries to keep the page table consistent. Furthermore, the thread must modify its page table for each modified page, even if it does not access that page in the subsequent transaction. These factors, on top of the page fault cost for writes, make CoW perform poorly, especially for short transactions.

Copy-on-Read (CoR) avoids the high upfront costs of CoW for creating snapshots. In CoR, instead of synchronizing page tables during the start of a transaction, snapshots are generated on the fly whenever a page is read or written. In either case, a page fault occurs and the handler maps in a thread local page which can be read or written to. Though CoR avoids unnecessarily updating pages that are not accessed, it has its costs. First, the first read or write to a page in each transaction will incur an expensive page fault. Second, during commit, all pages that have been accessed within the prior transaction need to be removed from the page table and the corresponding TLB entries invalidated.

While it seems promising to leverage address translation as the indirection layer for implementing snapshots, both CoW and CoR techniques introduce high overheads. EXCITE-VM implements CoR, but avoids its overheads using a page cache that holds onto mapped-in pages over the duration of multiple transactions.

3 EXCITE-VM Architecture

EXCITE-VM leverages the virtual-to-physical address translation mechanism of contemporary processors to enable efficient snapshotting of shared memory state in parallel applications. In order to allow multiple threads to access snapshots at different points in time, each thread in EXCITE-VM has its own page table, and hence, its own virtual-to-physical address mapping. Whenever a thread starts a transaction it acquires a snapshot of the current state of shared memory, generated on the fly via CoR. As a result, in EXCITE-VM, the existence of a set of pages representing the most recent consistent state of shared memory is not required. Instead, every thread maintains only a subset of the shared memory state (the pages it has faulted in) corresponding to a particular version or snapshot. To re-generate a snapshot for a particular page when processing a CoR fault, EXCITE-VM maintains a *global write log*. The global write log represents the central data structure of EXCITE-VM as it contains the history of all writes ever performed on shared memory, similarly as in a log-structured file system [44]. By scanning the log and applying the appropriate writes, any version of any page that existed in shared memory in the past can be re-generated. Once a page is re-generated, it is stored in thread local memory and mapped into the thread's address space so that the subsequent reads and writes to that page incur no overhead. Unfortunately, the cost of re-generating a page for a particular snapshot scales with the length of the log and an unbounded log will eventually consume too much main memory capacity. EXCITE-VM addresses this issue with checkpoints.

3.1 Checkpoints

A checkpoint represents the state of shared memory at a particular point in time EXCITE-VM uses designated checkpointing threads to create checkpoints lazily by periodically applying the write log to the prior checkpoint. After a log entry has been incorporated into the new checkpoint, and after all active threads' snapshots are more recent, the log entry can be either deleted or moved to high capacity, non-volatile storage enabling queries of historical main memory state. Checkpoints bound the cost of re-generating a snapshot to the number of log entries that exist between the last checkpoint and the snapshot to be re-generated. Even with checkpointing, frequent page faulting represents a prohibitive overhead. EXCITE-VM addresses this issue with the *page cache*.

3.2 Page Cache

The page cache is a software cache that buffers memory pages over the duration of multiple transactions. Instead of clearing the faulted-in pages after each transaction, pages remain in the cache so they are available for the next transaction without incurring another page fault. To incorporate the writes of remote threads, the page cache needs to be updated whenever a thread takes a new snapshot by traversing the global log entries that have been committed between the last snapshot of the thread and the current state of memory. While this represents a cost, it is significantly lower as the overhead incurred by the CoW approach which needs to update the entire shared memory state. Furthermore, keeping pages in the cache avoids resetting the access and protection bits of the page table entry reducing TLB flushes. The use of the page cache reduces the number of page faults significantly compared to pure CoR, particularly for applications with high spatial locality in terms of the shared memory pages they frequently access. Sizing the page cache correctly and applying a good replacement policy is important for achieving high performance. A too large page cache that mostly contains unused pages increases update costs and consumes memory, while a too small page cache triggers too many page fault. We study the effect of different cache sizes in Section 6.2.2.

3.3 Transactional Memory Extensions

While providing an efficient snapshotting capability is useful on its own as outlined in Section 1, EXCITE-VM supports snapshot isolation transactional memory (SI-TM) semantics to make snapshots readily accessible by programmers. Snapshots enable to buffer writes without making them visible to other threads providing efficient isolation between threads. As a consequence, the only mechanisms required to support SI-TM are a conflict detection scheme and the ability to rollback transactional state. In SI-TM, reads are always consistent even in the presence of writes and, as a result, SI transactions are only required to abort on write-write conflicts. To guarantee forward progress and to enable batching of the validation overhead into a single operation, EXCITE-VM implements lazy conflict detection. Transactional writes are buffered in a tentative log entry that is only committed to the global log after successful validation. While most other STMs introduce significant overheads to track transactional reads and writes including atomic operations and memory barriers, EXCITE-VM imposes no cost for reads and introduces minimal instrumentation overheads for writes. Validation itself is performed by comparing a transaction's writeset against the log entries that have been committed concurrently by other threads. In case of a conflict, the transaction is aborted by undoing all local writes, dropping the write log entry and by re-executing the transaction.

3.4 Example Transaction Sequence

Figure 1 shows the steps in an EXCITE-VM transaction. The shared global log is shown on the left, whereas per-thread structures, including the timestamp of the current snapshot, the write log of the current transaction, the page cache and its referenced physical pages, and the per-thread page table, are shown on the right. The log contains an entry for each write which indicates the address of the write as well as the delta between the previous and the written value. A page table entry contains the physical address a virtual address is mapped to and whether said page is present or write enabled.

3.4.1 Transaction Begin:

Figure 1a shows how the active thread creates a snapshot at ts=7 by reading the global timestamp counter. As the page cache is outdated (at ts=5) it is updated by applying the writes of log 6 and log 7. The write to address v8 committed at ts=7 hits the page cache and is hence applied to the corresponding cached page.

3.4.2 Transactional Writes:

Figure 1b shows the active thread performing a transactional write to shared memory at address v2. As the page containing v2 is unmapped, a page fault is triggered which allocates a new physical page, then walks the page table to install the new mapping and subsequently inserts the entry into the page cache. The newly created page is now initialized with the content of the checkpoint page (CoR) which is currently at version 3. This page copy is performed using a lockless procedure described further in Section 4.3. The thread then traverses the global log to determine writes (v2 is contained in log 4, 5 and 6) and applies them to the new page. Furthermore, the write is inserted into the write log of the current transaction. Following this, the thread performs a write operation to address v8 that hits the page cache. Handling this write does not require any action besides inserting it into the write log. Note that write logging is performed via the transactional instrumentation logic and not via the page fault mechanism.

3.4.3 Commit:

Figure 1c presents the commit processing of a transaction including validation of the write set against other committed writes. In the example, a remote thread has committed the write log entry (ts=8). As a result, the local thread needs to validate its writes against log 8. As the writesets do not overlap, validation is successful. The transaction obtains a commit timestamp (ts=9) and appends its write log entry to the shared global log. Lastly, infrequently used page cache entries are evicted by clearing the entry from the page cache, unmapping the page in the page table, flushing the hardware TLB and freeing the physical memory page.

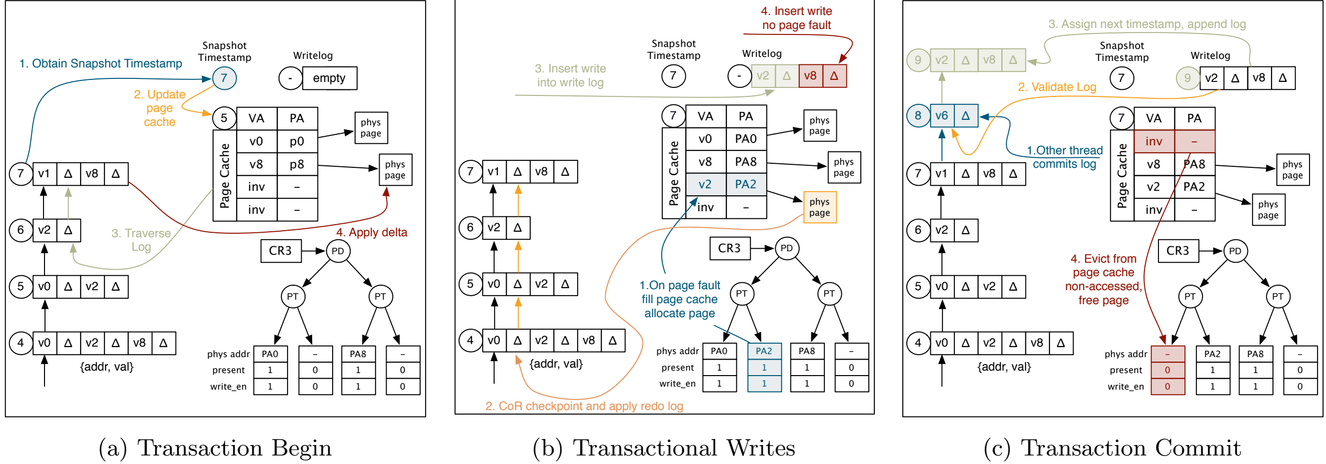

Figure 1: Example Transaction Sequence

4 Implementation

This section describes the implementation of EXCITE-VM, particularly Copy-on-Read, global log management and page caching.

4.1 Copy-on-Read

In order to implement Copy-on-Read, EXCITE-VM must map in the desired version of a page for a thread running transactions whenever a page fault occurs. Instead of modifying the page fault handler in the Linux kernel, we leverage Dune [5], a system that offers safe user-level access to privileged CPU features like page table management. Dune utilizes the virtualization hardware of modern microprocessors to provide virtual machine-like capabilities to basic processes. Dune processes gain access to virtualized hardware, such as a second page table layer for manipulating virtual address mappings, but are still isolated from the host operating system and hence are not more privileged than a usual process. We opted for Dune over a kernel based implementation due to performance advantages as well as development and stability reasons. Managing page tables in a user level library avoids a large part of the kernel specific complexity and reduces context switching overheads, so in particular, Dune reduces the cost of a trap from 2821 cycles to 587 [5]. Dune also facilitated the development process as a crash occurring in a Dune page fault handler does not lead to a kernel panic but instead only exits the Dune process.

An EXCITE-VM thread prepares itself to perform transactions by entering Dune mode and by copying the page table of its caller, similarly to what occurs during a call to `fork()`. During transaction processing, the per thread page table is modified to provide the desired snapshot view of the shared segment to the thread. Process switching is potentially expensive compared to thread switching. To address this issue, we utilize process context IDs (PCIDs) avoiding flushing the TLB when switching between page tables.

The entire CoR logic is implemented within a custom page fault handler which is registered via Dune. When a transaction is started on a thread for the very first time, the presence and write enable bit for all pages is cleared. As a result, the first access to shared memory traps into the EXCITE-VM page fault handler which in return creates a local copy of the page matching the contents of that page in the transaction's snapshot and maps this page with writable permissions into the thread's page hierarchy. Outside of a transaction, we map an empty page table into the thread's page hierarchy for the shared segment. Hence, accesses to the shared segment outside of a transaction will fail, revealing a bug in the application.

4.2 Page Cache

The page cache is implemented as a software cache of configurable size and holds frequently accessed pages which optimistically get updated to a snapshot version during transaction begin. The page cache supports the following simple replacement policy. Every *Scan* transactions, where *Scan* is a configurable parameter, all pages within the page cache whose access bit is cleared are evicted from the cache. Then, the access bit of all remaining pages is cleared and the corresponding TLB entries are flushed. Whenever the page is accessed again by application software, the hardware automatically sets the accessed bit. By configuring *Scan*, an EXCITE-VM user determines the frequency of cache evictions and the resulting memory consumption.

4.3 Global Log

The global log maintains all writes that have been applied to shared memory in chronological order. There exists one global log per application instance running on EXCITE-VM. It is a shared data structure accessed by application threads to re-generate snapshots as well as to perform conflict detection. To avoid being a concurrency bottleneck, the global log is implemented as a doubly linked list supporting lockless reads. New log entries are only appended to the end using an atomic operation. Log entries are removed only by the checkpointing thread and only if they are guaranteed to be older than the oldest snapshot that is currently in use by any thread. The only race condition that exists in this mechanism is between the checkpointing thread and a page faulting thread. If the checkpointing thread updates a particular page while an application thread tries to fault in said page, a race condition might occur. To avoid this issue,

readers implement the sequence lock mechanism [8] enabling concurrent checkpointing and page faulting without obtaining locks. In particular, the checkpointing thread modifies a shared page by setting its version number to dirty, then applies the updates and then increments the version number of the page. The readers on the other hand, check the page version number before and after copying the page and if the two versions are not identical, retries the sequence. The entries of the shared log are identified by a version number which is the unique timestamp assigned to this entry during commit. The writeset tracks writes on a quad-word granularity, whereas each entry is defined by its address and a 64 bit delta value which can be added to roll a version forward (redo) or subtracted to roll a version backwards (undo).

The size of the writeset influences both the snapshot generation and the transaction validation cost. Hence, to reduce the size of the writeset, EXCITE-VM supports a stream optimization which stores writes to consecutive addresses using a base address plus length notation. As a result, frequent operations such as memcpy and memset generate only a single entry within the writeset at the only expense of an additional branch instruction during write logging.

4.4 Libitm Integration

EXCITE-VM exposes transactions to programmers through GCC's libitm [17, 2]. Libitm defines a new keyword, `transaction_atomic{}`, which is used by the programmer to declare transactions. Libitm automatically instruments reads and writes and exposes those to library developers via a modular, internal API. We added SI-TM as a new algorithm to libitm by providing support for three main APIs. To implement `transaction_begin()` a thread reads out the global 64 bit timestamp counter setting the snapshot for this transaction. Before entering the transaction, the thread needs to update its page cache, which is performed by traversing the global log, starting from the snapshot the page cache is currently at (the snapshot of the last transaction). For each write, it queries the page cache and if existent updates the corresponding page. The `transaction_write()` API is called whenever a shared memory location is written and it is handled by buffering the address and value of the write, forming a new log entry. The most costly operation is `transaction_commit()` as it requires to validate the write set against all other committed transactions that overlap in time.

4.5 Commit processing

We support a set of techniques to reduce the validation cost. First, to provide scalability, commit processing is performed by all threads in parallel. By allowing lockless reads, the global log can be accessed by all committers concurrently. Nevertheless, to avoid races between concurrent committers, validation and appending a new log entry needs to be performed atomically. Therefore, committers determine the newest log entry before validation and then recheck for the newest log after validation. If another thread committed a new log in between, then the committer needs to validate against this log as well. This guarantees that a committer always sees all other commit logs even those that are appended during validation. Second, the bloomfilter signature which is contained in each commit log entry is utilized to perform hierarchical validation to cut short the validation sequence in case of no conflict. For example, if the intersection of the bloomfilter of two transactions is empty, there exists no write-write conflict between the two transactions and fine grain word based validation can be skipped. By maintaining the bloomfilters on a page based granularity all writes to a page set the identical filter bits which keeps the filter sparse and avoids false positives. In case no conflict exists between two transactions, this allows to validate all writes within a page using a single operation. Third, supporting streams (a sequence of writes to consecutive addresses) reduces the cost of validating a sequence of writes to a single base-bound check. Fourth, by applying techniques such as loop unrolling, prefetching and vectorization using AVX-256 instructions, EXCITE-VM is capable of checking up to four addresses in parallel.

4.6 Lines of code

EXCITE-VM has been programmed in C/C++ and contains 6330 lines of code, not including blank lines and comments. The GCC patch to enable EXCITE-VM as a new TM algorithm for libitm is 233 lines of code. The dune library adds another 5,898 lines although we utilize only a subset of its components. As a comparison, Linux's virtual memory system (mm) contains 62,877 lines of code as of version 3.13.0.

5 EXCITE-VM Guarantees

We now discuss the guarantees EXCITE-VM provides in terms of fault tolerance, security and consistency.

5.1 Fault Tolerance

EXCITE-VM improves fault tolerance and robustness over conventional multi-threaded systems by tolerating certain programming errors. As threads maintain local copies of the pages that are written during a transaction, uncommitted writes are never visible to other threads. Hence, in case a thread terminates due to a runtime error, the shared state remains unaffected. We assume that the EXCITE-VM system itself does not suffer from runtime errors as it can only protect against runtime errors of the application. The EXCITE-VM commit logic, however, does perform address checking of the write log. So in case a thread tries to commit a writelog that modifies locations outside of shared memory, the log is rejected to avoid segmentation violations induced by EXCITE-VM code. EXCITE-VM provides the means of removing a failed thread from the system, the option of spawning a new thread as well as enabling the other threads to continue operation. Nevertheless, to support fault tolerance in a meaningful way application awareness is required. In particular, the application must be able to tolerate the information loss of the failing thread. For applications that partition the work among threads during startup this might be difficult, whereas for real time server applications that continuously process client requests using multiple threads, it is feasible. For example, Memcached (See Section 5.5) can be extended such that on a thread failure only the in-flight request is lost but that the application can continue to run and accept request subsequently. In summary, EXCITE-VM provides the necessary facilities to considerably improve the robustness of a shared memory system.

5.2 Security

As mentioned in Section 4.1, each thread in EXCITE-VM uses Dune to access virtualized hardware, but these threads still have only the usual privileges and protections. Since each thread in EXCITE-VM has its own virtual-to-physical mapping, threads cannot access each others page caches and hence observe uncommitted writes. To prevent malicious threads gaining access to the transaction processing code and to virtual memory management instructions (which would jeopardize the guarantees above,) application code should be executed in as user, i.e. Dune ring 3, while EXCITE-VM needs to run in non-root, Dune ring 0. As a result, transaction begin and transaction commit operation need to be performed via system calls requiring a ring change which adds an overhead of 587 cycles in case of Dune [5]. Note that write instrumentation can be solely performed in user space as long as EXCITE-VM checks the validity of the write log (e.g. for buffer overflows) during commit.

5.3 Consistency

EXCITE-VM implements the SI isolation level [6], which guarantees consistent reads but permits the write-skew anomaly. A write-skew anomaly may occur when two concurrent transactions update separate variables (so no write-write conflict) but together causing a constraint to be violated. For example, consider the constraint: $a < b + c$. One transaction may modify b while a concurrent transactions modifies c. Though the constraint is satisfied in both thread's snapshots, the constraint may be violated after merging both writes into a new shared state. A write skew can only arise if there is a dependency cycle between reads and writes across two or more concurrent transactions. Such a cycle can be broken by *promoting* a read to be treated as part of the write set, thereby causing a write-write conflict and aborting the second transaction. Prior research [16, 14, 28] presented techniques to detect and remove write skew automatically which we adopt for our system. In particular, EXCITE-VM supports read promotion to avoid write skew anomalies. In our experience, this approach leads to significantly fewer aborts than forcing complete serializability of transactions while reliably preventing anomalies. The performance overhead introduced by read promotion is negligible [28].

In applications where strict serializability is required, programmers can apply locks for synchronization and utilize snapshots for fault tolerance and consistent reads. For example, LevelDB, a fast key-value storage library written by Google follows such a hybrid model by supporting atomic PUT and GET operations while also offering read-only snapshots. Finally, in future work we will support SSI-TM [39] which avoids write-skew by detecting dependency cycles at runtime.

6 Evaluation

We evaluate EXCITE-VM by comparing it to three baselines including SI-STM, Libitm and single global lock. *SI-STM* is a C/C++ re-implementation of Riegel's Java based system [43]. *Libitm* represents the default TM algorithm of GCC/Libitm which utilizes ownership records to guarantee consistency in the case of concurrent shared memory accesses and which is similar in function to TL2 [15]. We also compare to the single *Global Lock* implementation included in Libitm, which is not expected to scale well but which should provide lowest instrumentation overheads. We do not compare against hardware TM implementations such as Intel TSX due to the following reasons. Firstly, at the time of writing this paper no Intel Haswell with more than four cores was available to us and secondly a STAMP comparison between HTM and a TL2 STM has been performed before in [53] which the interested reader can consult for comparison.

We perform our measurements using different software applications, including Microbenchmarks, the STAMP application suite [46], Memcached and the Yahoo Cloud Server Benchmark. To support EXCITE-VM we had to modify the applications moderately by replacing malloc calls for shared data to a version that calls into the EXCITE-VM system. We also had to move all globally defined shared variables onto the heap[1]. From the STAMP suite, we have successfully ported Genome, Kmeans, Labyrinth, Vacation, SSCA2 and Intruder as well as all datastructures. Yada does not work correctly with TMs that deploy lazy conflict detection [46], Bayes we omitted due to the non-deterministic behavior of the application leading to inconsistent results [46]. Porting the applications required a certain understanding of the code at least to the degree of determining shared memory data structures. Porting the applications took us one day per application in average with only few lines of code modified.

We disable two features of EXCITE-VM for these performance comparisons, the first being the mapping of a dummy page table outside of a transaction to catch out-of-transaction accesses (see Section 4.1) and the second being that commit processing is performed in user mode Dune ring 3 (see Section 5.2). These features provide bug-checking and security, but none of the systems we compare against provide either feature. All applications were compiled with GCC 5.0. We run our experiments on an 8-core Intel Xeon CPU E5-2670 running at 2.6 GHz with 20 MBytes of cache running Ubuntu Linux 14.4.

6.1 Microbenchmarks

To compare EXCITE-VM against SI-STM we use two micro benchmarks both taken from the original SI-STM paper. By leveraging the virtual memory system, EXCITE-VM reduces the instrumentation overheads of reads and as a result, outperforms SI-STM by up to 5.9x as shown in Figure 2. The list benchmark evaluates concurrent lookup, insert and delete operations to a linked list initialized with 250 elements. We run 1, 2, 4 and 8 thread configurations and varying the ratio of read-only (lookup) and read-write (insert and delete) operations. This is a read-heavy benchmark as even when using a workload of 100% write transactions each insert and delete operation must partially traverse the list. EXCITE-VM scales well on this benchmark and outperforms the other systems at any write percentage, and by 5.9x under a read-only workload. The bank benchmark is a write-heavy benchmark involving a sequential array of account objects where write transactions transfer money from one account to another and read-all transactions sum up the value of all accounts. We show results from the benchmark

[1]EXCITE-VM currently does not support shared data to reside in the global data/bss segment, however, we plan to add this in a future revision

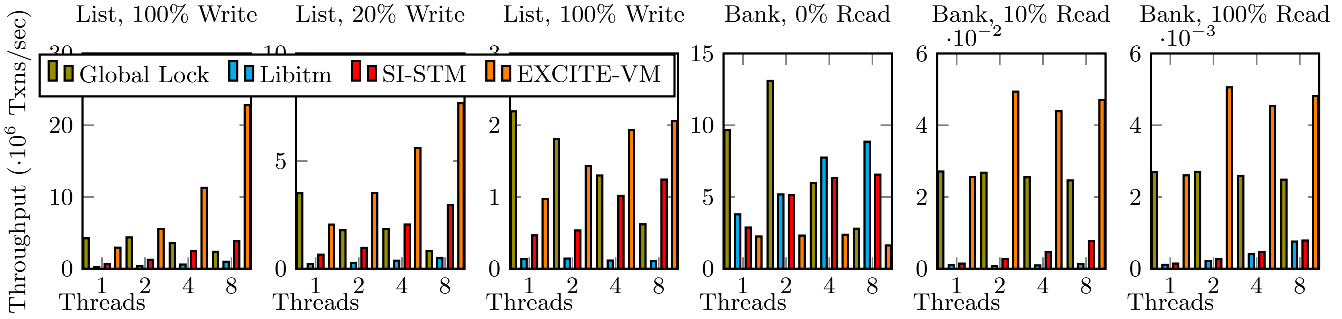

Figure 2: Microbenchmarks

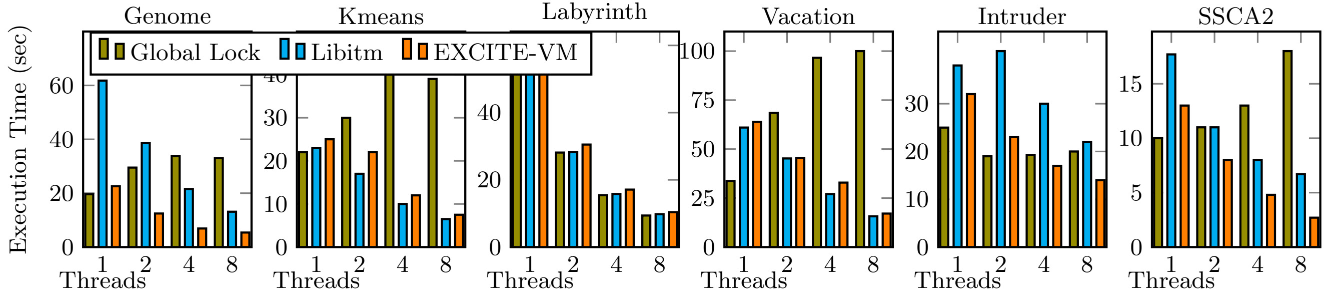

Figure 3: STAMP Execution Time

using 1 million accounts, each a 64-bit integer. When executing 8 threads on a workload of entirely write transactions, EXCITE-VM is 5.5x slower than the highest performing system, Libitm. This is due to the comparably higher cost of writes in EXCITE-VM. However, when read-all transactions are included, EXCITE-VM reaches the bandwidth of the shared last level cache with as few as 2 threads and achieves 6.8x higher throughput than SI-STM or Libitm.

6.2 Applications

6.2.1 STAMP

We evaluate EXCITE-VM using the STAMP benchmark suite and observe up to 2.2x performance improvements over all of the other tested approaches. To evaluate STAMP, we utilized the default configuration parameters with the high contention option. Figure 3 shows the execution time of the STAMP applications on EXCITE-VM. The Genome benchmark is very read intensive, which causes Libitm to spend more than 50% of its execution time on read instrumentation costs as reported by `perf`. In contrast, EXCITE-VM has almost no read overheads and matches the performance of the global lock implementation on a single thread while also scaling well to eight threads. Libitm also scales well EXCITE-VM, however, it is 3x slower for any thread configuration. Kmeans is a write heavy application and, as a result, EXCITE-VM shows a slightly higher single thread overhead than either the global lock or Libitm on this benchmark. Both EXCITE-VM and Libitm scale comparably well. Labyrinth only spends a small amount of time in transactions, furthermore, most of the work can be executed completely parallel without any kind of synchronization. As a result all three systems show good scalability and low overheads. Vacation offers a mixed load of both update-heavy and read-heavy transactions. Hence, EXCITE-VM offers comparable performance to Libitm due to outperforming Libitm on read-heavy transactions but having higher overhead on update-heavy transactions. Both Libitm and EXCITE-VM exhibit good scalability and outperform the global lock. Intruder performs significant number of byte reads which are instrumented on the per access level and which lead to high instrumentation overheads for Libitm. Hence, on this benchmark EXCITE-VM shaves 30% off the runtime of the other tested systems at 8 threads. Read instrumentation for this benchmark under EXCITE-VM shows up as a 10% overhead in `perf`, although this instrumentation is unnecessary for EXCITE-VM and is implemented as an empty virtual function call solely to match the Libitm API. GCC currently does not support link time optimization in combination with transactional memory which would be required to completely remove this overhead. Finally, in SSCA2, EXCITE-VM shows good scalability and, at 8 threads, shows a 2.2x performance improvement over all other tested systems. We do not compare against SI-STM as it cannot run STAMP due to being an object based TM implementation. Neither the original authors nor us have been able to evaluate SI-STM in complex benchmarks due to this limitation.

6.2.2 Memory Overhead

EXCITE-VM introduces potential memory overheads due to the per thread page cache, the per thread page table and the global log. The page tables approximately have a size of 1/512 the of the physical memory in use and in experiments their size has been below 1% of the total memory consumption. The global log size is determined by the oldest live transaction. For the applications we evaluated, the global log has introduced less than 1% memory overheads. The

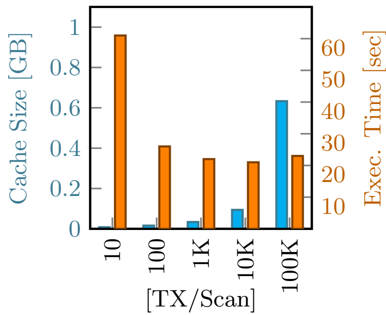

Figure 4: Pagecache 1 Thread

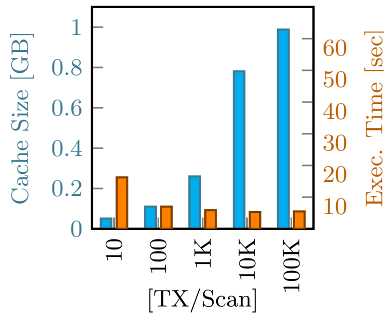

Figure 5: Pagecache 8 Threads

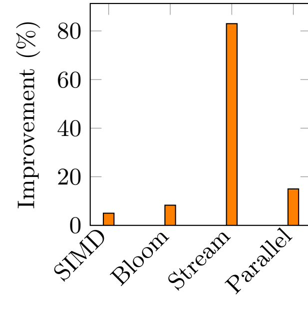

Figure 6: Optimization Techniques

log size can be further bound by generating intermediate checkpoints. To evaluate the memory overhead of the page cache, we studied the effect of different eviction scan periods when running the STAMP application suite to understand the page cache trade-off between memory and performance. A shorter scan period tends to reduce the size of the page cache. Figure 4 and Figure 5 shows Genome at 1 and respectively 8 threads with the ratio of transactions per scan varying between 100 and 100,000, showing the effect on both cache size and execution time. To put the cache size into perspective, note that Genome itself has a memory footprint of 300 MB. For 1 thread and a TX/Scan ratio of 100 the page cache is 8 MB and hence represents an overhead of less than 3%. However, for 8 threads and a 100,000 TX/Scan ratio the cache grows to over 900 MB representing an overhead of 300%. Our current system employs a default TX/Scan ratio of 10,000 which provides close to optimal performance with a memory overhead of 1.9x for 8 threads. This ratio has been proven adequate for the other evaluated applications as well.

Interestingly, the cache size does not necessarily grow linearly in the number of threads, e.g. in Figures 4 and 5 the cache size for 8 threads in the 100,000 case is much less than 8x that for 1 thread. This is because the application partitions some of its state into per-thread working sets, causing some pages to associate with a single thread's page cache. We observed that increasing the scan ratio past 100,000 TX/Scan reduced performance for this experiment, due to the inflicted cost of updating cached pages that are not subsequently accessed. An analysis of the page cache contents revealed that many of the pages in the caches are duplicates, suggesting that further optimizations, such as page sharing and deduplication could improve performance [23, 31]. Genome was representative of results obtained for the other STAMP benchmarks, hence we omit the other applications for brevity.

6.2.3 Impact of Optimization Techniques

The implementation Section discusses thread optimization techniques that improve the performance of EXCITE-VM. We provide the average performance improvement for the STAMP applications running at 8 threads. In particular, we discuss four optimizations: vectorization, bloomfilters, streams and parallel commit.

First, vectorization (SIMD) capabilities of Intel processors both improve validation, by comparing multiple elements of a write set in parallel, as well as log traversal and application, again by applying multiple writes with a single instruction. SIMD is implemented by extracting inner loops into function bodies adhering to the form required by GCC and then enabling auto vectorization. In cases where GCC was unable to auto-vectorize we apply inline assembly. Figure 6 shows a 4% improvement on end-to-end performance for SIMD for Genome (the other STAMP applications behave similar) for eight threads.

Second, we evaluate the 64 bit bloomfilter that is maintained for each write log entry, which allows checking for non-intersection of two write sets with a single operation. By avoiding costly entry-by-entry comparison of writesets in the usual case, bloomfilters reduce validation overhead leading to a 7% improvement as shown in Figure 6.

Third, we evaluate the effect of streams. Streams enable to store a set of writes to consecutive addresses, as emitted by memcpy and memset, as a single pair of start address and length. This optimization reduces the average writeset size and as a result reduces validation cost leading to a mean speedup of 90% or almost 2x.

Finally, we evaluate the effect of performing commits in parallel when executing 8 threads. In the baseline, EXCITE-VM threads obtain a commit lock to perform validation and to append their log entry. In the optimized approach, threads iterate over the log in parallel for validation. Validation and appending a log is still performed atomically, however, if another thread commits a new log concurrently it is sufficient to validate against this new log entry instead of restarting the entire validation process. Parallel commit delivers an improvement of 15%.

6.2.4 Online Analytical Processing

STAMP's Vacation simulates an online transaction processing (OLTP) workload on a large database. Due to the database size, conflicts are rare and both libitm and EXCITE-VM scale equally well. Besides performing OLTP, database operators are generally interested in analyzing their data. For this purpose, we added an online analytical processing (OLAP) workload to Vacation. Concurrent with 7 OLTP threads we execute a single OLAP thread, which chooses one table at random and performs on it range scans of different size to compute an aggregation. The OLAP workload is read-only and requires a consistent view of the database, respectively no concurrent conflicting writes to the same data items can be tolerated. We measure the effect of the OLAP thread on the execution time of the OLTP threads as well as the analysis performance that can be achieved con-

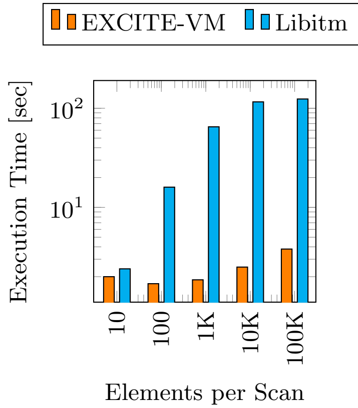

Figure 7: Vacation OLTP

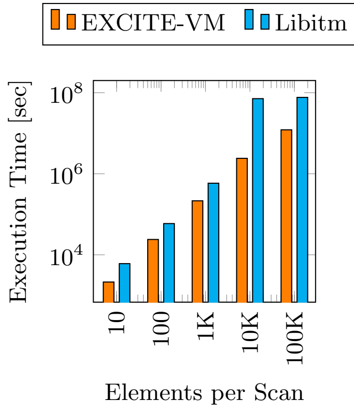

Figure 8: Vacation OLAP

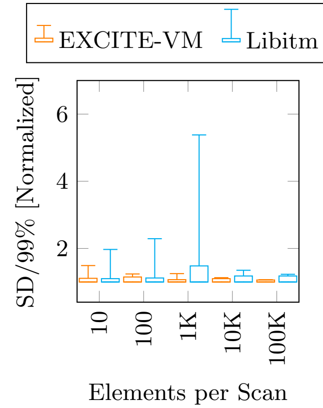

Figure 9: Predictability

currently. Figure 7 shows the effect of the OLAP thread on OLTP performance. As it can be seen, the performance of EXCITE-VM remains relatively stable while Libitm execution time increases by 1000x for long range scans of 100k elements. Even when running a single OLTP thread, the slow down for Libitm is at the same order of magnitude, effectively rendering the OLAP workload impossible to support on Libitm due to conflicting memory accesses. EXCITE-VM only introduces a 1.9x slowdown for the 100k range scan mainly due to cache effects. Profiling revealed that for the 100k scans abort rates of OLTP transactions are unaffected by the OLAP thread, however, the last level cache miss rate increases, presumably long range scans trash the L3 cache. Figure 8 shows the average execution time of an OLAP transaction. Libitm shows a 3x increase in execution time for OLAP over EXCITE-VM for all scan sizes (note the log scale). As it can be seen, Libitm's slowdown of OLTP performance (1000x) is significantly higher than the slowdown for OLAP (3x). The reason, therefore, is that Libitm switches to serial mode after a phase of high contention and as long running OLAP transactions have the same priority to obtain the serial lock, the effect on the OLTP transactions is more significant.

EXCITE-VM gives predictable performance, which is important for applications such as user-facing databases that need to meet service level objectives (SLOs). Figure 9 shows the standard deviation (SD) and maximum 99th percentile (99%) duration of OLAP transactions. For both Libitm and EXCITE-VM SD and 99% increases to a maximum for the 1000 element scans. For even larger scans variability decreases due to the sheer duration of the transactions. Nevertheless, EXCITE-VM significantly outperforms Libitm by up to a factor of 1.4x for SD and 4.3x for the 99 percentile.

6.2.5 Elastic Memcached

We now evaluate Memcached, a popular key-value cache, that was transactionalized to run on Libitm by Ruan et al. [47]. We ported this code to EXCITE-VM, by allocating shared data structures and several global variables on the EXCITE-VM heap. To evaluate the performance we execute the Memcached server with 1 to 8 threads and apply memslap as a load generator using the following parameters: `-concurrency=8 -execute-number=416667 -s localhost:11211 -binary`. Figure 10 shows execution time for Memcached for both Libitm and EXCITE-VM. The two systems show similar performance, while EXCITE-VM shows better scalability for 8 threads.

Furthermore, to demonstrate the fault tolerance capabilities of EXCITE-VM, we built a modified version of Memcached by introducing programmer bugs that trigger a segmentation fault at different locations at an arbitrary randomized point in time. We then ran Memcached 1000 times in this configuration with only the affected thread died in each execution without affecting other server threads and without corrupting the shared state. Memslap detects a timeout in this case for the outstanding request and then continues its operation. This shows the improved fault tolerance provided by EXCITE-VM. We plan to extend Memcached with a functionality that detects failing threads and, in this case, elastically spawns a new thread. As a result failing threads should have a minimal impact on performance.

6.2.6 Yahoo Cloud Server Benchmark

Yahoo Cloud Server Benchmark (YCSB) [12] represents a benchmark suite to test webscale key-value store implementations. It can be used to test scale-up as well as scale-out systems, whereas we focus on scale-up. YCSB defines six workloads which model realistic applications by setting put, get and scan parameters accordingly. For brevity we select only two workloads: Workload A which is an update heavy workload stressing the write performance as well as workload E which executes short range scan operations. We ported YCSB to C++ and perform measurements against the Libitm baseline. Figure 11 and Figure 12 shows the results for the two workloads. In workload A, EXCITE-VM achieves 700,000 operations per second while providing full transactional semantics, e.g. we support multiple atomic put/get operations as a single transaction. Both systems scale well, however, EXCITE-VM shows a slightly higher overhead leading to a slowdown of 5% over Libitm. This workload represent a worst case scenario for EXCITE-VM due to many updates and yet EXCITE-VM shows only modest degradation in performance. Workload E on the other hand performs scans for which EXCITE-VM shows reduced instrumentation overhead and linearly scaling up to eight threads. In particular it shows 2.2x performance increase over Libitm for eight threads.

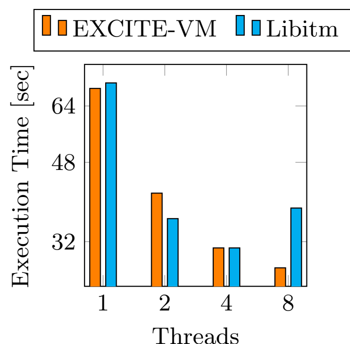

Figure 10: Memcached

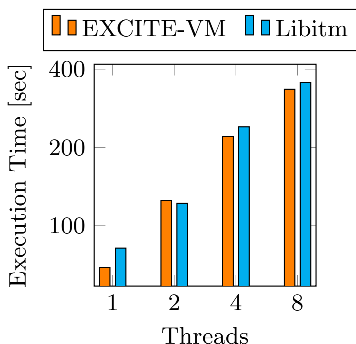

Figure 11: YCSB-A

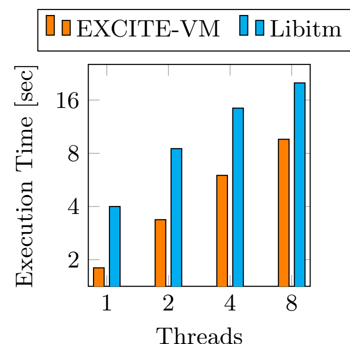

Figure 12: YCSB-E

7 Related Work

Snapshots are supported by HyPer [24] utilizing Linux's fork mechanism to instantiate in-memory database snapshots for read-only transactions. Forking the page table on on each transaction begin is significantly more expensive than creating an updated snapshot with EXCITE-VM. Moreover, there is no mechanism to merge modified snapshots atomically into committed state, as provided in EXCITE-VM. All major databases, PostgreSQL [33], MySQL [36], HANA [50] and OracleDB [22] support snapshot isolation transactions. However, incorporating a database into an application introduces overheads and requires significant changes to the data model. Particularly, instead of relying on standard data structures, as provided for example by the STL, all data needs to be stored in tables. EXCITE-VM, on the other hand, adds these capabilities with low overhead, while avoiding changes to the data model.

SpiceC [18] is a parallel programming technique that enforces isolation between threads by generating thread local copies of shared data on each access. Instrumenting every read operation introduces significant overheads (as in SI-STM), which EXCITE-VM avoids by means of the virtual memory system. Doppler [37] utilizes thread local copies to reduce contention on frequently accessed data structures. In contrast to EXCITE-VM, it is restricted to commutative operations. Burckhardt proposes to program with revisions and isolation types [9] leveraging a fork-join task model to exploit concurrency. Changing a conventional lock-based programming model into a task model requires significant changes whereas EXCITE-VM requires minor modifications. Furthermore, the technique does not support speculation and, as a result, requires the programmer to define a merge function for each data type to handle write-write conflicts, whereas, EXCITE-VM leverages transactions.

There exists a large body of research on *Transactional Memory* [21, 49, 40, 19, 15, 35, 25, 13, 48, 7, 52, 20]. More relevant to our work is unbounded page-based transactional memory (PTM) [10] which leverages the virtual memory system to support transactions overflowing the cache and to avoid aborts due to context switches and exceptions. In contrast to EXCITE-VM, PTM requires hardware modifications and does not provide snapshots. Similarly, SI-TM, [27] which provides the same guarantees as our system, requires hardware modifications of the memory system. Riegel proposes lazy snapshots [42] to provide consistent reads addressing the linearly growing validation overhead, however, the system does not support snapshot isolation semantics.

Memory protection hardware has been proposed to address the privatization problem and to achieve strict atomicity between transactional and non-transactional data accesses [1]. In contrast to our system, which also provides strict atomicity, the work neither addresses fault tolerance nor provides SI. Memory protection techniques are also utilized to enable the concurrent execution of an STM and a bounded hardware to form an efficient Hybrid TM system [4]. Furthermore, virtualization techniques have been previously utilized by TM systems to achieve unbounded transactions [41] and fully virtualized transactions [11]. Similar techniques have been implemented on the operating system level to achieve virtualization [51] and to manage transactional memory as in TxLinux [45]. In distributed systems, Microsoft's CORFU project [3] shares similarities with EXCITE-VM.

8 Conclusions

EXCITE-VM provides a new concurrent programming model supporting snapshot isolation transactions on shared memory. It delivers high performance by enabling threads to directly access shared memory while providing the synchronization-free application programming model provided by inter-process messaging. EXCITE-VM also provides the ability to have individual threads fail and restart without restarting the entire application. Our evaluation shows that this approach achieves performance which is equal, and in most cases superior to, the alternative approaches, especially when applications have a significant number of read-only transactions. In particular, we have shown that for applications leveraging long read-only transactions, EXCITE-VM can be 1000x faster than a conventional TM solution. EXCITE-VM also delivers performance predictability, as read-only transactions never abort or block in the presence of update transactions.

Several implementation techniques are critical to achieving this performance, including the page cache, periodic checkpointing, parallel lazy commit, vectorization, bloom-filters and CoR snapshotting. We also showed that periodically creating a checkpoint for use by read-only transactions reduces the cost of these transactions while allowing the log to be pruned periodically. A natural extension for EXCITE-VM is to provide durability by persisting the log and modified pages to stable storage as part of checkpointing. Another is to allow distributed transactions by distributing the log and the commit operation over the network.

9 References

[1] M. Abadi, T. Harris, and M. Mehrara. Transactional memory with strong atomicity using off-the-shelf memory protection hardware. In *ACM Sigplan Notices*, pages 185–196. ACM, 2009.

[2] A.-R. Adl-Tabatabai, T. Shpeisman, and J. Gottschlich. Draft specification of transactional language constructs for c++, 2009.

[3] M. Balakrishnan, D. Malkhi, T. Wobber, M. Wu, V. Prabhakaran, M. Wei, J. D. Davis, S. Rao, T. Zou, and A. Zuck. Tango: Distributed data structures over a shared log. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*, pages 325–340. ACM, 2013.

[4] L. Baugh, N. Neelakantam, and C. Zilles. Using hardware memory protection to build a high-performance, strongly-atomic hybrid transactional memory. *ACM SIGARCH Computer Architecture News*, 36(3):115–126, 2008.

[5] A. Belay, A. Bittau, A. Mashtizadeh, D. Terei, D. Mazières, and C. Kozyrakis. Dune: Safe user-level access to privileged cpu features. In *Proceedings of the 10th USENIX Conference on Operating Systems Design and Implementation*, OSDI'12, pages 335–348, Berkeley, CA, USA, 2012. USENIX Association.

[6] H. Berenson, P. Bernstein, J. Gray, J. Melton, E. O'Neil, and P. O'Neil. A critique of ansi sql isolation levels. *ACM SIGMOD Record*, 1995.

[7] J. Bobba, K. Moore, H. Volos, L. Yen, M. Hill, M. Swift, and D. Wood. Performance pathologies in hardware transactional memory. In *ACM SIGARCH Computer Architecture News*, pages 81–91. ACM, 2007.

[8] H.-J. Boehm. Can seqlocks get along with programming language memory models? In *Proceedings of the 2012 ACM SIGPLAN Workshop on Memory Systems Performance and Correctness*, pages 12–20. ACM, 2012.

[9] S. Burckhardt, A. Baldassin, and D. Leijen. Concurrent programming with revisions and isolation types. In *ACM Sigplan Notices*, volume 45, pages 691–707. ACM, 2010.

[10] W. Chuang, S. Narayanasamy, G. Venkatesh, J. Sampson, M. Van Biesbrouck, G. Pokam, B. Calder, and O. Colavin. Unbounded page-based transactional memory. In *ACM Sigplan Notices*. ACM, 2006.

[11] J. Chung, C. C. Minh, A. McDonald, T. Skare, H. Chafi, B. D. Carlstrom, C. Kozyrakis, and K. Olukotun. Tradeoffs in transactional memory virtualization. In *ACM SIGARCH Computer Architecture News*, pages 371–381. ACM, 2006.

[12] B. F. Cooper, A. Silberstein, E. Tam, R. Ramakrishnan, and R. Sears. Benchmarking cloud serving systems with ycsb. In *Proceedings of the 1st ACM symposium on Cloud computing*, pages 143–154. ACM, 2010.

[13] P. Damron, A. Fedorova, Y. Lev, V. Luchangco, M. Moir, and D. Nussbaum. Hybrid transactional memory. In *ACM Sigplan Notices*. ACM, 2006.

[14] R. Dias, J. Lourenço, and N. Preguiça. Efficient and correct transactional memory programs combining snapshot isolation and static analysis. In *Proceedings of the 3rd USENIX conference on Hot topics in parallelism (HotPar'11), HotPar*, volume 11, 2011.

[15] D. Dice, O. Shalev, and N. Shavit. Transactional locking ii. In *Proceedings of the 20th international conference on Distributed Computing*, DISC'06, pages 194–208, Berlin, Heidelberg, 2006. Springer-Verlag.

[16] A. Fekete, D. Liarokapis, E. O'Neil, P. O'Neil, and D. Shasha. Making snapshot isolation serializable. *ACM Trans. Database Syst.*, 30(2):492–528, 2005.

[17] P. Felber, C. Fetzer, U. Müller, T. Riegel, M. Süßkraut, and H. Sturzrehm. Transactifying applications using an open compiler framework. *TRANSACT, August*, pages 4–6, 2007.

[18] M. Feng, R. Gupta, and Y. Hu. Spicec: scalable parallelism via implicit copying and explicit commit. In *ACM SIGPLAN Notices*, pages 69–80. ACM, 2011.

[19] L. Hammond, V. Wong, M. Chen, B. D. Carlstrom, J. D. Davis, B. Hertzberg, M. K. Prabhu, H. Wijaya, C. Kozyrakis, and K. Olukotun. Transactional memory coherence and consistency. In *Proceedings of the 31st annual international symposium on Computer architecture*, ISCA '04, pages 102–, Washington, DC, USA, 2004. IEEE Computer Society.

[20] T. Harris, J. Larus, and R. Rajwar. Transactional memory. *Synthesis Lectures on Computer Architecture*, 5(1):1–263, 2010.

[21] M. Herlihy and J. E. B. Moss. Transactional memory: architectural support for lock-free data structures. In *Proceedings of the 20th annual international symposium on computer architecture*, ISCA '93, pages 289–300, New York, NY, USA, 1993. ACM.

[22] K. Jacobs, R. Bamford, G. Doherty, K. Haas, M. Holt, F. Putzolu, and B. Quigley. Concurrency control, transaction isolation and serializability in sql92 and oracle7. *Oracle White Paper, Part*, (A33745), 1995.

[23] K. Jin and E. L. Miller. The effectiveness of deduplication on virtual machine disk images. In *Proceedings of SYSTOR 2009: The Israeli Experimental Systems Conference*, page 7. ACM, 2009.

[24] A. Kemper and T. Neumann. Hyper: A hybrid oltp&olap main memory database system based on virtual memory snapshots. In *Data Engineering (ICDE), 2011 IEEE 27th International Conference on*, pages 195–206. IEEE, 2011.

[25] S. Kumar, M. Chu, C. J. Hughes, P. Kundu, and A. Nguyen. Hybrid transactional memory. In *Proceedings of the eleventh ACM SIGPLAN symposium on Principles and practice of parallel programming*, pages 209–220. ACM, 2006.

[26] T. J. LeBlanc and J. M. Mellor-Crummey. Debugging parallel programs with instant replay. *Computers, IEEE Transactions on*, 100(4):471–482, 1987.

[27] H. Litz, D. Cheriton, A. Firoozshahian, O. Azizi, and J. P. Stevenson. Si-tm: reducing transactional memory abort rates through snapshot isolation. In *Proceedings of the 19th international conference on Architectural support for programming languages and operating systems (ASPLOS-19)*, pages 383–398. ACM, 2014.

[28] H. Litz, R. J. Dias, and D. R. Cheriton. Efficient correction of anomalies in snapshot isolation transactions. *ACM Transactions on Architecture and*

Code Optimization (TACO), 11(4):65, 2015.

[29] T. Liu, C. Curtsinger, and E. D. Berger. Dthreads: efficient deterministic multithreading. In *Proceedings of the Twenty-Third ACM Symposium on Operating Systems Principles*, pages 327–336. ACM, 2011.

[30] T. Merrifield and J. Eriksson. Conversion: multi-version concurrency control for main memory segments. In *Proceedings of the 8th ACM European Conference on Computer Systems*, pages 127–139. ACM, 2013.

[31] D. T. Meyer and W. J. Bolosky. A study of practical deduplication. *ACM Transactions on Storage (TOS)*, 7(4):14, 2012.

[32] C. Minh, J. Chung, C. Kozyrakis, and K. Olukotun. Stamp: Stanford transactional applications for multi-processing. In *Workload Characterization, 2008. IISWC 2008. IEEE International Symposium on*. IEEE, 2008.

[33] B. Momjian. *PostgreSQL: introduction and concepts*, volume 192. Addison-Wesley New York, 2001.

[34] P. Montesinos, L. Ceze, and J. Torrellas. Delorean: Recording and deterministically replaying shared-memory multiprocessor execution efficiently. In *Computer Architecture, 2008. ISCA'08. 35th International Symposium on*, pages 289–300. IEEE, 2008.

[35] K. Moore, J. Bobba, M. Moravan, M. Hill, and D. Wood. Logtm: Log-based transactional memory. In *Proceedings of the 12th International Symposium on High-Performance Computer Architecture (HPCA-12)*. Austin: IEEE Computer Society, 2006.

[36] A. MySQL. *MySQL Administrator_s Guide and Language Reference*. Sams Publishing, 2006.

[37] N. Narula, C. Cutler, E. Kohler, and R. Morris. Phase reconciliation for contended in-memory transactions. In *Proceedings of the 11th USENIX conference on Operating Systems Design and Implementation*, pages 511–524. USENIX Association, 2014.

[38] H. Patil, C. Pereira, M. Stallcup, G. Lueck, and J. Cownie. Pinplay: a framework for deterministic replay and reproducible analysis of parallel programs. In *Proceedings of the 8th annual IEEE/ACM international symposium on Code generation and optimization*, pages 2–11. ACM, 2010.

[39] D. Ports and K. Grittner. Serializable snapshot isolation in postgresql. *Proceedings of the VLDB Endowment*, 2012.

[40] R. Rajwar and J. Goodman. Speculative lock elision: Enabling highly concurrent multithreaded execution. In *Proceedings of the 34th annual ACM/IEEE international symposium on Microarchitecture*, pages 294–305. IEEE Computer Society, 2001.

[41] R. Rajwar, M. Herlihy, and K. Lai. Virtualizing transactional memory. In *Computer Architecture, 2005. ISCA'05. Proceedings. 32nd International Symposium on*. IEEE, 2005.

[42] T. Riegel, P. Felber, and C. Fetzer. A lazy snapshot algorithm with eager validation. In *Distributed Computing*, pages 284–298. Springer, 2006.

[43] T. Riegel, C. Fetzer, and P. Felber. Snapshot isolation for software transactional memory. In *TRANSACT06*, volume 298, 2006.

[44] M. Rosenblum and J. K. Ousterhout. The design and implementation of a log-structured file system. *ACM Transactions on Computer Systems (TOCS)*, 10(1):26–52, 1992.

[45] C. J. Rossbach, O. S. Hofmann, D. E. Porter, H. E. Ramadan, B. Aditya, and E. Witchel. Txlinux: Using and managing hardware transactional memory in an operating system. In *ACM SIGOPS Operating Systems Review*, pages 87–102. ACM, 2007.

[46] W. Ruan, Y. Liu, and M. Spear. Stamp need not be considered harmful. In *Ninth ACM SIGPLAN Workshop on Transactional Computing*, 2014.

[47] W. Ruan, T. Vyas, Y. Liu, and M. Spear. Transactionalizing legacy code: An experience report using gcc and memcached. *SIGPLAN Not.*, 49(4):399–412, Feb. 2014.

[48] B. Saha, A. Adl-Tabatabai, and Q. Jacobson. Architectural support for software transactional memory. In *Microarchitecture, 2006. MICRO-39. 39th Annual IEEE/ACM International Symposium on*. IEEE, 2006.

[49] N. Shavit and D. Touitou. Software transactional memory. *Distributed Computing*, pages 99–116, 1997.

[50] V. Sikka, F. Färber, W. Lehner, S. K. Cha, T. Peh, and C. Bornhövd. Efficient transaction processing in sap hana database: the end of a column store myth. In *Proceedings of the 2012 ACM SIGMOD International Conference on Management of Data*, pages 731–742. ACM, 2012.

[51] M. M. Swift, H. Volos, N. Goyal, L. Yen, M. D. Hill, and D. A. Wood. Os support for virtualizing hardware transactional memory. In *Procs. of the 3rd ACM SIGPLAN Workshop on Transactional Computing*, 2008.

[52] L. Yen, J. Bobba, M. R. Marty, K. E. Moore, H. Volos, M. D. Hill, M. M. Swift, and D. A. Wood. Logtm-se: Decoupling hardware transactional memory from caches. In *High Performance Computer Architecture, 2007. HPCA 2007. IEEE 13th International Symposium on*, pages 261–272. IEEE, 2007.

[53] R. M. Yoo, C. J. Hughes, K. Lai, and R. Rajwar. Performance evaluation of intel® transactional synchronization extensions for high-performance computing. In *Proceedings of SC13: International Conference for High Performance Computing, Networking, Storage and Analysis*, SC '13, pages 19:1–19:11, New York, NY, USA, 2013. ACM.