CONVERSION: Multi-Version Concurrency Control for Main Memory Segments

Timothy Merrifield
University of Illinois at Chicago
tmerri4@uic.edu

Jakob Eriksson
University of Illinois at Chicago
jakob@uic.edu

Abstract

We present CONVERSION, a multi-version concurrency control system for main memory segments. Like the familiar $Subversion$ version control system for files, CONVERSION provides isolation between processes that each operate on their own *working copy*. A process retrieves and merges any changes committed to the *trunk* by calling `update()`, and a call to `commit()` pushes any local changes to the trunk.

CONVERSION operations are fast, starting at a few microseconds and growing linearly (by less than 1 μs) with the number of modified pages. This is achieved by leveraging virtual memory hardware, and efficient data structures for keeping track of which pages of memory were modified since the last update. Such extremely low-latency operations make CONVERSION well suited to a wide variety of concurrent applications. Below, in addition to a micro-benchmark and comparative evaluation, we retrofit Dthreads [28] with a CONVERSION-based memory model as a case study. This resulted in a speedup (up to 1.75x) for several benchmark programs and reduced the memory management code for Dthreads by 80%.

Categories and Subject Descriptors D.1.3 [*Programming Techniques*]: Concurrent Programming—Parallel Programming

1. Introduction

Designing shared memory programs to target increasingly parallel hardware can be a daunting task. For programs that require consistency guarantees in the face of concurrency, programmers have historically turned to mutual exclusion (locking) and more recently to software transactional memory (STM). However, for programs that can tolerate weaker consistency models, these techniques can be unnecessarily cumbersome and degrade performance.

With this in mind, we present CONVERSION, a kernel-level, multi-version concurrency control system for main-memory segments. CONVERSION provides each process with an isolated local copy of a shared memory segment. A process is free to read from and write to its copy of the segment; any changes remain entirely local to the process itself. By calling `commit()` on the segment, a process can push its changes to a shared version of the segment. The only way that the process sees changes to its segment is by executing an `update()` command to pull changes from the shared version to its local copy.

CONVERSION keeps the latency and overhead of these operations to a minimum by leveraging the virtual memory hardware already present in modern CPUs. Using copy-on-write (COW) memory protection to provide isolation, CONVERSION extends the virtual memory manager in the operating system. A call to `commit()` creates a new version containing any private, COW pages, which are then made accessible to other processes. Retrieving the most recent version is effected through page-table updates rather than content copying, and the number of such updates is kept to the bare minimum by tracking changes at the page level or below (in the case of a write conflict). CONVERSION is built as a Linux kernel module, requiring minor kernel code modifications.

Version controlled memory was first proposed in [16] as a memory model for deterministic concurrency. While CONVERSION shares conceptual similarities with this work, no implementation or evaluation was provided there. CONVERSION offers a practical implementation of version controlled memory and its performance is evaluated against software transactional memory and mutual exclusion. Other recent work, including Dthreads, Grace [6], Determinator [4] and revisions and isolation types [11] use local copies to provide determinism and/or synchronization-free concurrent programming. CONVERSION does not replace these systems, but rather provides a building block upon which such systems can be built. While CONVERSION shares some technical aspects with software transactional memory [23, 24, 37],

Eurosys'13 April 15-17, 2013, Prague, Czech Republic
Copyright © 2013 ACM 978-1-4503-1994-2/13/04. . . $15.00

CONVERSION is a lightweight versioning system, offering no transactional semantics. See §6 for more details on related work. The primary contributions of this paper are as follows:

- The design and implementation of CONVERSION, a kernel-level MVCC system for main memory segments, leveraging virtual memory hardware.
- Micro-benchmark evaluations demonstrating the high performance of the CONVERSION operations `update()`, `commit()` and page faults.
- An evaluation with concurrent data structures, comparing CONVERSION against STM and mutual exclusion.
- An adaptation of Dthreads [28] to using CONVERSION as the underlying memory model, showing improved performance and reduced code complexity.

Below, we first motivate the need for CONVERSION in §2, followed by a detailed description of our approach in §3. Microbenchmarks and a comparative evaluation are offered in §4, and a case-study of porting Dthreads to CONVERSION is offered in §5. A review of the related work is given in §6, and §7 concludes.

2. Who Needs Main-Memory MVCC?

Working with a local copy provides tremendous advantages over standard shared memory, including safe, concurrent, and lock-less access to shared memory contents. The price we pay for these advantages comes in two parts: maintenance overhead and timeliness. CONVERSION keeps the overhead to a minimum. More significant is the fact that changes are only propagated from a writing process to a reading process upon a call to `commit()` (by the writer), followed by a call to `update()` (by the reader). Thus, any given local copy is likely to be somewhat out of date with respect to other processes. However, in many applications working with a slightly out-of-date local copy is inconsequential, or even desirable. Below, we discuss several such applications.

Deterministic Concurrency Systems Deterministic concurrency [3, 4, 6, 28] is the idea of ensuring that parallel programs produce the same output every time. In Dthreads [28] (and other such systems that utilize workspace consistency [2]), each thread essentially operates on a local copy of the heap and globals, and periodically commits its changes to the trunk. Determinism is ensured by enforcing a predetermined commit order among threads: threads perform computations in a parallel phase, but commit their changes in a serial phase. Transitions between the two phases are protected by barriers. This reliance on local copies makes Dthreads a prime candidate for evaluation with CONVERSION. In §5, we demonstrate using CONVERSION to speed up and reduce the code size of Dthreads.

Concurrent, Shared-Memory Data-structures. Consider a main-memory database containing a large, non-partitionable, graph data-structure. The graph is highly dynamic: nodes and edges are added and removed, and edge weights are changed, with high frequency. Now add multiple processes concurrently executing shortest-path queries across this graph.

Long-running queries present a challenging problem to concurrency. Mutual exclusion is clearly undesirable as are read-write locks due to the write-heavy nature of the workload. RCU [32] provides lock-less reading at some performance cost to the writer, but also enforces restrictions on context switches for readers, a problem for systems executing very long-running queries. Moreover, many TM systems have historically struggled to support long-running queries. With TM, long-running, read-only transactions that execute over data structures under constant mutation will often have to abort and retry the transaction. While STM systems with MVCC can greatly reduce this cost, they have traditionally been implemented only in managed languages [26, 36]. A recent proposal for an unmanaged multiversion STM [30] shows promise, however the implementation was not available for comparison at the time of publication.

With CONVERSION, a thread performing a query would simply update its local copy and then execute its query independent of any subsequent updates. Used in this way, CONVERSION can provide the reader with (non-transactional) snapshot isolation [5]. Additionally, unlike STM approaches no instrumentation overhead is required for the thread performing the query which executes at native speed. In §4, we evaluate the use of CONVERSION for a shared-memory, concurrent data-structure and compare this to using software transactional memory, as well as to a single global lock.

Concurrent Garbage Collection Garbage collectors traverse object references starting from one or more known roots, to identify objects that are unreachable. Once an object is no longer reachable from the root, it will remain unreachable until freed. Based on this observation, a garbage collector can operate effectively on a local version of the heap. With CONVERSION, a basic mark-and-sweep garbage collector may be transformed into a concurrent garbage collector. Here, the jarring halt created by stop-the-world garbage collectors is replaced by a brief periodic `update()` of the local copy, followed by a conventional mark-and-sweep operating concurrently on the local copy. A garbage collection specific CONVERSION flag allows the marking performed by a collector to not incur any copy-on-write overhead. Relative to the `fork()`-based technique in [17], CONVERSION promises more natural semantics, and a substantial performance improvement. However, comparing a basic mark-and-sweep collector with CONVERSION, to a state-of-the-art concurrent garbage collector, falls outside the scope of this paper.

X Windows The X11 window system uses shared memory to share images between an application and the window client, when running on the same machine. This is accomplished using a mutual exclusion protocol based on X11 messages. With CONVERSION, no such synchronization is

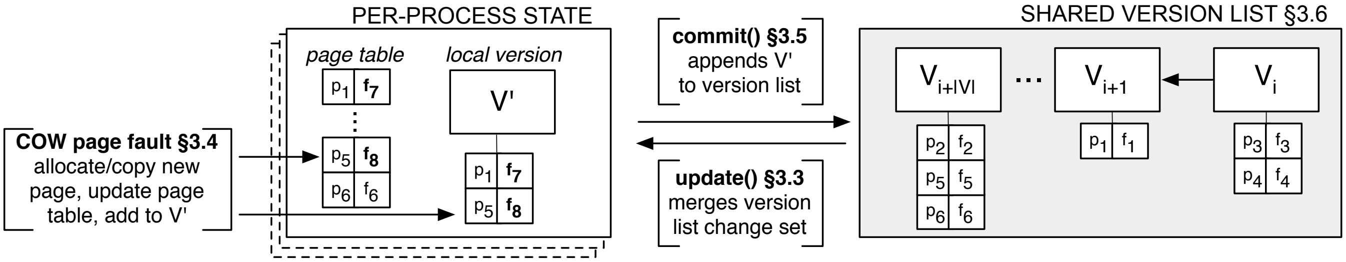


Figure 1. Diagram illustrating CONVERSION operation. Three primary kernel operations: `commit()` and `update()` and COW page faults affect the local version and/or version list of a given segment.

needed. The application may simply call `commit()` whenever it has finished a drawing operation, and the window client may fetch the most recent version of the image at any time by calling `update()`. After using `madvise()` to restrict gratuitous copying of outdated content, we estimate that the amount of work done in this configuration would be similar to the current method, but with improved parallelism and reduced code complexity.

Other Applications While working with a local copy has tremendous benefits, it is not universally applicable. In particular, applications with multiple mutators operating on the same set of data are likely to continue requiring synchronization or transaction support to protect critical sections. CONVERSION interacts well with existing synchronization operations, and a mix of traditional synchronization and CONVERSION may sometimes be the best solution. For example, to ensure mutual exclusion on a variable in version controlled memory, a standard binary semaphore may be used as follows: (a) acquire the semaphore, (b) `update()` the segment, (c) modify the variable, (d) `commit()` the segment and (e) release the semaphore.

3. Design and Implementation

In this section we present the CONVERSION API and describe the operation of these functions behind the scenes. Figure 1 illustrates the three major operations of CONVERSION. CONVERSION segments are memory-mapped as *private* regions, causing the virtual memory hardware to execute a COW page fault when the process attempts to write. Thus, an executing process gradually accumulates a list of private pages for the working copy, until `commit()` is called to push the local changes to a shared version list. Upon calling `update()`, a change set is retrieved from the version list, and applied to the local version.

3.1 The CONVERSION API

Table 1 outlines the main functions of the CONVERSION API. The function `s=checkout(name, size, flags)` is used to create a new segment, or attach to an existing segment. Additionally, the `flags` tells CONVERSION whether the segment should be file-backed or anonymous shared memory, and other performance-tuning specifics. Calling `checkout()` maps a *local copy* of the segment into the process' address space, starting at address s. This local copy reflects the most recently committed version at the time that `checkout()` was called. Any writes to the local copy remain private, using copy-on-write memory protection, until the process calls `commit()`. `commit()` writes any local changes to the central version repository of the segment, making these changes available to other processes. Here, changes are tracked at page granularity. `commit()` fails with an error if the repository contains any updates that have not been merged into the local copy. Finally, `update()` fetches any new changes committed to the repository, and merges these changes into the local copy.

3.2 Merging Change-Sets

Any time the local copy and the repository were both changed between calls to `update()`, conflicts may arise between the two change-sets, which must be resolved. CONVERSION offers two basic but automatic resolution methods.

First, the user may specify the size of an *editing unit*, ϵ, for a given segment. By default, $\epsilon = 8$, or one (64 bit) word. CONVERSION automatically merges any ϵ-disjoint change-sets. For example, with a $\epsilon = 1$ byte, single bytes may be independently changed within a segment, and a call to `update()` will automatically replace the original bytes in the local copy with any updated bytes from the repository, as long as no overlap exists between the modified byte offsets in the local copy and the repository. §3.3 describes our implementation of a fast change-set merge.

Second, if the change-sets are not ϵ-disjoint, CONVERSION will simply favor the local copy. That is, if an editing unit at a certain offset was modified in both the local copy and the repository, `update()` will *not* modify the local editing unit. Upon subsequent `commit()`, the value from the local copy is then written to the repository, overwriting the earlier value.

In some applications, this is clearly not the desired outcome. For these, CONVERSION offers two alternatives. Used correctly, CONVERSION is compatible with conventional synchronization. Thus, concurrent writes to shared variables may be avoided using mutual exclusion. Alternatively, `update()` can be passed the flag CV_VALIDATE, which causes `update()` to fail with an error code instead of

Function	Description
`s=checkout(name, size, flags)`	Called by each process to create a new segment or map in an existing segment.
`update(s)`	Updates a local working copy to the most recent version, merging any concurrent changes.
`commit(s)`	Writes any changes in the working copy to the *trunk*.

Table 1. Main functions of the CONVERSION API. Applications use these functions to interact with the version control subsystem. The functions are library functions, serving as wrappers for system calls.

merging if non-disjoint change-sets are detected. This could be used as a signal to retry the local computation, in some ways similar to STM.

Finally, in some applications it may be possible to design around the ϵ-disjointness requirement by introducing additional, per-process variables.

3.3 Operation of `update()`

Whenever `update()` is called, the caller's view of the segment is updated to reflect the most recent version in the repository. `update()` is able to do this very quickly by virtue of two facts:

- For page-disjoint changes, no page contents are copied or scanned during `update()`. Instead `update()` modifies the caller's page table to point to the most recent version of each page.
- Page table modifications are limited to those entries that actually need updating, making run-time complexity independent of the total size of the segment.

The operation of `update()` is supported by a *version list* data structure (the version list is discussed in detail in §3.6). When updating to a version V_i, the version list provides the set of changes $C_i = \{(p_1, f_1), \ldots, (p_j, f_j)\}$ recorded since the last update, until the present time. Here, each pair (p_k, f_k) denotes a virtual page offset (from the beginning of the segment) p_k where a change was recorded, and a physical frame number f_k where the most recently settled version of the page is stored. C_i contains at most one such pair per unique p_k. `update()` can retrieve the C_i from the version list in $O(|C_i|)$ time (see §3.6).

`update()` is performed in kernel space with direct access to the caller's page table. For each entry in C_i, `update()` must modify the page table entry, decrement the reference count on the page previously located at p_k and flush the translation look-aside buffer (TLB) entry for the address at p_k. `update()` is quite fast for page-disjoint updates, with the time taken per entry in C_i at approximately $400ns$ (see §4.1 for more detail).

For pages that contain changes both in the local copy and in the change set, `update()` needs to determine, for each editing unit, which version to use. To support non-page disjoint, concurrent edits, CONVERSION maintains a reference to the original version of each page in the local copy. The merging process then proceeds as follows. For each editing unit, if the local version differs from the original, the local version is used. Otherwise, the version from the change set is used (see §4.1 for performance discussion). Note that without the original for reference, determining which of two concurrently modified versions changed at a certain word offset is not possible.

3.4 Page Fault Handling, Copy-on-Write

CONVERSION leverages the virtual memory subsystem to provide system-level isolation and conserve memory. Due to this, the "local copy" accessed by a process is no copy at all: it is simply a duplicate mapping of the same physical memory frames referenced by the version list. Any frames of memory that are shared between processes, or between the version list and a process, are marked read-only in their respective page table entries. When a write to such a page is attempted:

- A new (private) page frame is allocated (see Fig. 1), and initialized with the contents of the original frame.
- The relevant entry in the writer's page table is pointed at the new frame, with R/W permissions.
- A new entry (p_i, f_i) is added to V', the local change set.
- To support merging, a reference to the original page is stored in the change set and the reference count is increased to avoid garbage collection.

In our experiments these operations added only 100 ns of overhead to the normal Linux copy-on-write page fault handling.

3.5 Operation of `commit()`

Calling `commit()` triggers the addition of a new version, V_i, to the version list. Then, for each entry $(p_k, f_k) \in V'$:

- The entry (p_k, f_k) is added to V_i, indicating that page p_k was modified since the previous call to `commit()`, and the location of the new page contents, frame f_k.
- If there was a previous entry in the version list for page p_k, then this entry can be removed. The reference count for the old entry is decremented.
- The calling process' page table entry corresponding to page p_k is set to write-protected, forcing a copy-on-write page fault on the next write attempt to this page.

- To support persistent segments (see §3.1), several Linux kernel data structures including the page cache, rmap and the LRU lists may be updated.

The processing of each entry takes constant time, so the runtime of `commit()` is $O(|V'|)$. In our experiments, `commit()` took approximately 800 ns per entry.

3.6 The Version List

The version list is the primary data structure of CONVERSION. Its purpose is two-fold: (a) to maintain, for every version in use, a list of pages that has changed since that version was created, and (b) to hold references to physical pages that are part of the most recent version, but that may not be in use by any current process. In the kernel, the version list is protected by a read-write lock. However, in §3.7 we remove the read-write lock to increase concurrency.

`new_version()`	$O(1)$	Adds a new, empty version to the version list.
`add_entry(p,f)`	$O(1)$	Adds an entry (p_k, f_k) to the most recent version.
`change_set(V)`	$O(\lvert C\rvert)$	Returns a change set $C = \{(p_1, f_1), ..., (p_{\lvert C\rvert}, f_{\lvert C\rvert})\}$ containing every page modified since version V.
`V=top_version()`	$O(1)$	Returns a reference to the most recent version.

Table 2. Interface of the version list data structure. 2nd column indicates asymptotic run-time complexity.

Table 2 describes the main functions used by CONVERSION to manipulate the version list. Additional functions exist for reference-counted garbage collection of old versions.

The version list is a doubly linked list representing the set of versions $\mathbf{V} = \{V_1, V_2, ...\}$, where each version V_i is represented by a set of entries (p_k, f_k), see Fig 1. Thus, given a version V_i, the full set of changes is the union of all the changes since V_i was committed or,

$$C_i = \bigcup_{k=i+1}^{|\mathbf{V}|} V_k, \tag{1}$$

where the union operator retains only the most recent entry (which refers to the most recent version of the page). It is common for different versions among $V_{i+1...k}$ in Eq. 1 include changes pertaining to the same page. For example, if a writer updates the same page between every call to `commit()`, the end result may be a series of versions which all contain an entry for that one page.

The key insight to a fast `change_set()` implementation is the fact that only the most recent version, for each virtual memory page, is of interest: CONVERSION does not support updating to any version other than the most recently committed one. Thus, to ensure that we can compute Eq. 1 in $O(|C_i|)$ time, we need to pro-actively prune any previous entry from the list, leaving only the most recent entry.

For each call to `add_entry()`, the version list is first examined to find any existing entry (in any version) referring to the page in question. If such an entry exists, it is removed before the new entry is added. For example, in Fig. 1, once the local version V' is committed, the p_5 entry in version $V_{i+|V|}$ is removed. Since this is done for every insertion, there can be at most one existing entry in the list. To find any existing entry quickly, we keep a separate lookup table using the existing radix-tree implementation in the Linux kernel [29]. Once a version V_i has had all of its entries pruned, it must eventually be removed. We perform this task during `update()` as we traverse the version list. In §3.7, we discuss deferring deletion to support lock-free updates.

3.7 Increasing Concurrency

All CONVERSION operations for a given segment depend on a shared data structure: the version list. To avoid races between threads, the design above used a single read-write lock to protect the version list. However, this coarse-grained synchronization creates scalability issues for many workloads. In this section we discuss modifications to `update()` and `commit()` that greatly increase concurrency and improve scalability.

3.7.1 Allowing concurrent `update()` and `commit()`

It is unsafe to allow `update()` to traverse the version list while `commit()` is concurrently pruning out-dated entries for each page it is committing. To solve this problem, instead of actually removing the entry from the list, `commit()` can mark the entry as obsolete. This allows us to defer garbage collection of entries until a time that is guaranteed not to conflict with threads performing `update()`.

To support this we modify the version list entry to be a 3-tuple (p_k, f_k, o_k) where o_k stores the *obsoleted-by version*. At the start of `commit()`, we acquire a version list mutex and pre-determine the version V_i that will be ours. Now (with the mutex released), as we commit entries (p_k, f_k) in V' we lookup the previous entry for page p_k in the version list and (if it exists) mark it obsolete as of version i. We need to acquire the version list mutex one more time at the end of `commit()` to mark our new version as available to updaters and export the latest version number to userspace (see §3.8).

During `update()`, before performing any page table operations we now check o_k to ensure that the entry is not obsolete with respect to the version we are updating to. If it is obsolete, then there is a newer version further along in the version list and we can avoid the unnecessary work and continue the update.

We perform the garbage collection of obsolete entries in a background kernel thread that is invoked periodically. By tracking the oldest version in use by any ongoing `update()`, the garbage collector can safely reclaim obsolete entries.

3.7.2 Concurrent committers

The design above does not permit concurrent `commit()` operations: one process must finish its `commit()` phase before the next may begin. Below, this restriction is relaxed by allowing concurrent operation on separate pages. Pages with only a single committer are committed concurrently—for all other pages, commits are performed in version order (recall that a thread acquires a new version number whenever it calls `commit()`). A new *version map* data structure is used to establish the arrival order and allow committers to determine when it is safe to commit a given page.

The global *version map* $M = \{(max_1, cur_1), ...\}$ holds max_j, the number of the most recently created version that affects page j (including ongoing calls to `commit()`), and cur_j, the number of the most recent version to *finish* committing changes to page j. Thus, if $max_j = cur_j$, no thread is currently committing changes to page j.

At the start of `commit()` we acquire the version list mutex, and claim the next available version list entry V_k. With the lock still held, for each page entry $(p_j, f_j) \in V'$ we examine the corresponding version map entry (max_j, cur_j). If $max_j \neq cur_j$, then another thread is in the process of committing this page and we will have to wait for it to finish. We then store a thread-local variable $wait_j = max_j$, to indicate which thread we are waiting for. Otherwise, we set $wait_j = 0$. Finally, we set $max_j = k$ to indicate that our version also affects page j.

We may now release the version list mutex and commit all pages for which $wait_j = 0$. For each page j that we commit, we update the version map and set $cur_j = k$ to signify that our thread is done with this page. Finally, we must commit the pages for which $wait_j > 0$. The committer spins over these pages, for each page checking if $cur_j = wait_j$, at which point it is our turn to commit page j and set $wait_j = 0$. Notice that the actual committing of entries (the bulk of the work in `commit()`) is now performed lock-free. We evaluate the impact of these changes on scalability in §4.2.

3.8 Making fewer system calls

Avoiding system calls when possible can significantly reduce the amount of time spent on Conversion-related overhead. In our experiments, a Linux system call doing little work (`getpid()`) took on average 60 ns. To eliminate superfluous system calls we make additional meta-data available in user space, similar to techniques used for the futex implementation in Linux [20]. For example, by making the latest committed version number available in user space, calls to `update()` can use this meta-data to decide whether a system call will be fruitful.

4. Evaluation

Our evaluation of Conversion consists of three parts: (1) a micro-benchmark analysis of the primary Conversion operations, (2) an analysis of Conversion performance for concurrent access of data structures and (3) a case study on retrofitting Dthreads [28] with Conversion. All experiments were performed on a machine with four 2.00GHz Intel Xeon E7-4820 8-core processors and 128GB of main memory. Where times in microseconds are shown, $1\mu s = 2000$ cycles.

Operation	**latency**
COW fault w/o Conversion	$2.4\ \mu s$
COW fault w/ Conversion	$2.5\ \mu s$
`commit()`	$3{-}6\mu s{+}0.8\ \mu s{*}pages$
persistent `commit()`	$3{-}6\mu s{+}1.2\ \mu s{*}pages$
`update()`	$3{-}6\mu s{+}0.4\ \mu s{*}pages$
`update()` w/ merging	$3{-}6\mu s{+}5.2\ \mu s{*}pages$

Table 3. Total cost of Conversion and related operations in our micro-benchmark experiment.

4.1 Micro-benchmarks

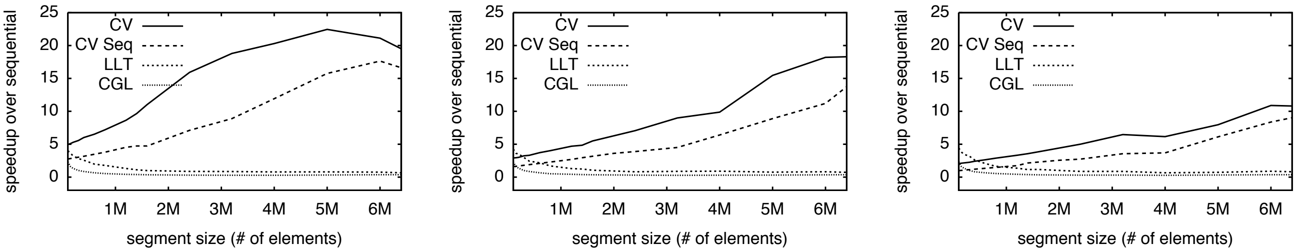
Table 3 describes Conversion latency (and related operations) at a low level. These latencies were determined using two processes: one generates 100 COW page faults, performs a `commit()` operation then sleeps; while the other calls `update()` (retrieving 100 pages each time) between sleep intervals. Our experiments show the latency of a COW page fault to be 2.57 μs with Conversion and 2.48 μs with the Conversion module removed; adding less than 100 ns (3 %) of additional overhead per fault.

Except for a constant cost of 3–6 μs, the run-time of Conversion operations is linear in the number of pages that are to be committed/updated/merged. Thus, we report the latency of these operations on a per-page basis. Both non-persistent `commit()` and `update()` are fast, performing a single page operation in less than 1 μs per page (excluding the constant cost). Note that while the merging performance listed reflects a single modified word in a page, the work required for additional modified words (an additional write) is marginal compared to the cost of detecting such changes, which is incurred in either case. `commit()` operations on persistent segments add additional overhead as we must update several Linux kernel data structures as pages transition from swap-backed to file-backed.

These experiments were conducted with two processes executing on the same physical CPU. When executing with a higher number of processes across several processors on a NUMA system, these latencies may increase.

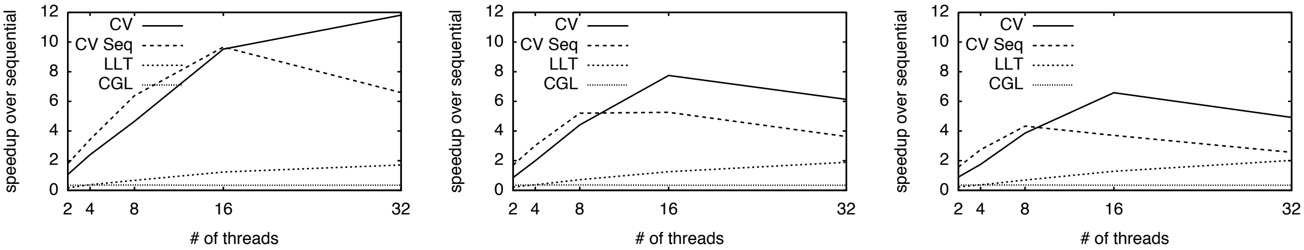
4.2 Comparative Eval: Concurrent Data Structures

Through multi-versioning, Conversion provides support for long-running, read-only queries that execute at native speed. When combined with traditional synchronization (locks) to support multiple writers, Conversion has the potential to greatly improve the throughput of workloads containing a mix of updates and long-running queries. In



(a) 74% lookups, 25% writes and 1% queries (b) 49% lookups, 50% writes and 1% queries (c) 25% lookups, 74% writes and 1% queries

Figure 2. Performance of CONVERSION when compared to LLT (STM) and CGL (STM with a single lock). A total of 32 threads perform a mix of lookups, writes and long running queries on an array with a variable number of elements. As the array size grows, LLT struggles with transactional aborts while CGL suffers from lock contention. CONVERSION outperforms the others by allowing concurrent writes and lookups/queries.



(a) 74% lookups, 25% writes and 1% queries (b) 49% lookups, 50% writes and 1% queries (c) 25% lookups, 74% writes and 1% queries

Figure 3. Scalability of CONVERSION, LLT (STM) and CGL (STM with a single lock). A varying number of threads perform a mix of lookups, writes and long running queries on an array with 2M elements. CONVERSION scales well when the number of writes (`commit()` operations) is low. When threads perform frequent commits there is a great deal of contention for the version list lock.

this section, we present data comparing CONVERSION with existing STM approaches and analyze the performance of CONVERSION operations on a highly-concurrent data structure.

4.2.1 Experimental Description

Below, we evaluate CONVERSION both with (CV) and without (CV Seq) the concurrency optimizations described in §3.7. We compare against Lazy-Lazy-Timestamp (LLT), a word-based TL2-like [18] algorithm available from the RSTM [31] library, and a single global lock (CGL) also from RSTM. We experimented with other STM algorithms provided by RSTM, but the results did not significantly improve beyond LLT.

In this experiment (based on a program provided by RSTM) each thread performs one of three operations on an array of integers: (1) write to (increment) a single, random integer; (2) lookup the value of a single, random integer; (3) perform a query over the entire segment. The query performed is a $find\_max$ operation, with running time linear to the size of the array. We use the notation 25:74:1 to denote a workload of 25% lookups, 74% writes and 1% queries.

4.2.2 Throughput

Fig. 2 shows the throughput for 32 threads performing workloads of 74:25:1, 49:50:1 and 25:74:1. The throughput is presented as the speedup over a single thread performing no synchronization. For LLT, as the array size increases the performance suffers as the write-heavy workload triggers conflicts with long running queries. This causes frequent aborted transactions and thus reduced throughput. At lower sizes LLT still aborts transactions, however the likelihood of a conflict and the penalty of an abort are decreased. We see a similar throughput pattern for the global lock. As query times take longer, more time is spent waiting to acquire the lock from threads executing queries.

The results for CONVERSION show that as the segment size grows, CONVERSION's performance continues to improve over the alternatives. CONVERSION overtakes the competition at 50K, 300K and 500K elements for workloads of 74:25:1, 49:50:1 and 25:74:1 respectively. This can be attributed to CONVERSION's resilience to long running queries, allowing the writes to continue unabated. As the number of elements in the array grows, this phenomenon is increasingly

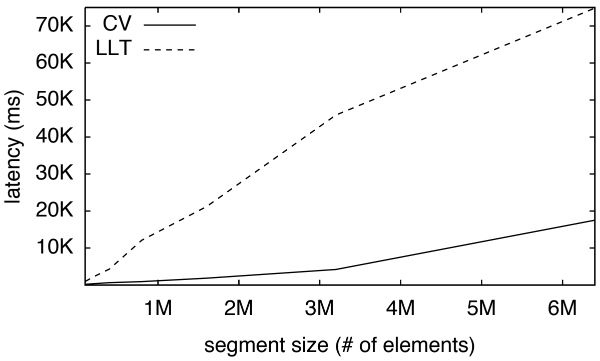


Figure 4. The running time (in ms) of long running queries with CONVERSION and the word-based STM. When using CONVERSION, queries executes without instrumentation and approximately 4-10× faster.

exaggerated. The performance disparity between read-heavy and write-heavy workloads is described in more detail in §4.2.3

While some of CONVERSION's advantage over LLT can be attributed to the avoidance of aborted transactions, another important performance impact is the cost of STM instrumentation. Fig. 4 shows the query execution times for LLT and CONVERSION over variable array size. We observe that STM instrumentation causes queries to execute approximately 4-10× slower than the native execution speeds seen with CONVERSION.

4.2.3 Scalability

Fig. 3 illustrates the scalability of our experiment for workloads of 74:25:1, 49:50:1 and 25:74:1 over an array of 2M elements. With a workload of mostly reads (74:25:1), CONVERSION continues to scale up to 32 threads. This can be attributed to the lock-less nature of `update()`, which is the only CONVERSION operation required for reading from the segment. Because `update()` operations require no synchronization with other threads, read-heavy workloads will scale well with CONVERSION.

However, as the percentage of writes increases, CONVERSION scalability begins to suffer for several reasons. First, a greater number of writes increases the total number of modified pages in the system at any given time. This means that for every call to `update()` there is now more work to do. This cost is compounded by the number of threads in the system. For example, with 32 active threads the number of pages to update will be approximately double the number of pages with 16 active threads. Second, with more writers attempting to commit there is greater contention for acquisition of the version list lock. Unfortunately, the ticket spin locks used in the Linux kernel are non-scalable, leading to acquisition times proportional to the number of spinning threads [10].

By comparing with the sequential version (CV Seq), we can see at what point the concurrency optimizations begin to prove fruitful. For fewer threads, the increased complexity (and latency) of the concurrent algorithms allows the sequential version to perform better. This is true up to 8 threads for 49:50:1 and 25:74:1, and 16 threads for 74:25:1 where CV Seq benefits from its use of a read-write lock

5. Case Study: DTHREADS

It is well known that deterministic execution greatly helps to improve debugging, testing, and more generally the ability to *reason* about a multi-threaded program [9]. Several systems [3, 4, 6, 28] that enforce determinism have directly implemented (or closely resembled) workspace consistency (WC) [2]. In these systems, threads execute in isolation from one another, unable to see mutating operations by other threads. At static synchronization points, modifications by each thread are communicated to others in a deterministic fashion. Because both the timing of synchronization and the communication ordering are deterministic, only a single interleaving is possible.

Below we describe and evaluate CONVERSION as a memory system for deterministic execution by retrofitting an existing deterministic run-time: Dthreads [28].

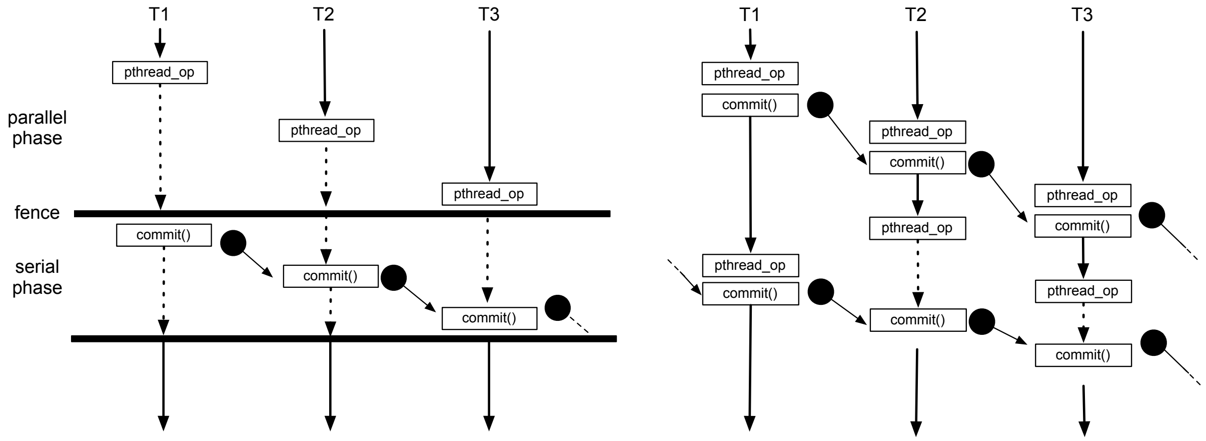
Dthreads is a deterministic version of the pthreads library. Programs written for pthreads can be linked against Dthreads and executed deterministically without modification to the user code. Dthreads utilizes pthreads operations ($create$, $exit$, $mutex\_lock$, etc...) as deterministic synchronization points between threads and reimplements each of these operations in order to enforce deterministic execution.

5.1 DTHREADS: Unmodified Operation

In Dthreads, threads can share variables located either in global memory or on the heap. For each of those areas of memory, Dthreads maintains a local segment and a shared segment which are memory mappings backed by the same physical memory. As a thread executes, it operates on the local segment exclusively, triggering COW page faults on writes. Dthreads tracks the dirty pages in user space via signal handlers. When page table entries in the local segment are invalidated (via $mprotect()$), the resulting page fault will populate the segment with a page from the shared physical memory. This allows Dthreads to control exactly when updates to the shared segment become visible to a user process.

When a thread encounters a pthread operation ($create$, $exit$, $mutex\_lock$, etc...), Dthreads uses that opportunity to commit thread-local changes to shared memory. To ensure determinism, commit order is enforced by a *token*, which is passed between threads, see Fig. 5(a). In order to commit, the thread must first acquire the token.

During commit, the list of dirty pages is traversed and the contents are written to the public memory-mapped segment. If Dthreads detects that another thread has written to the same page since the last commit, the pages are merged



(a) The original Dthreads execution model. A thread executes in the *parallel phase* until a pthreads operation occurs, then waits (represented by a dotted line) at the fence for the arrival of the other threads. After the fence, threads commit their changes in the order dictated by the *token* (shown as a black disk).

(b) The execution model of Dthreads with Conversion after the fence has been removed. Threads that possess the token can commit immediately with no regard to the other threads. If a thread does not possess the token it must block and wait until it receives the token.

Figure 5. Comparison of the Dthreads execution model with and without Conversion. Dashed lines indicate periods spent waiting. With Conversion, waiting is nearly eliminated in this example allowing significantly more progress to be made.

	Dthreads	**Dthreads w/** Conversion
Memory class LOC	578	95
Commit function LOC	57	4

Table 4. Lines of code for the Dthreads class that implements the memory model vs. the revised model using Conversion. Conversion significantly decreases code complexity and reduces most of the class to boiler-plate code.

on a per-word basis where conflicts favor the thread that committed last (as determined by the token). Additionally, in order for the most recent version to be made visible to the user process, page table entries for the private segment are invalidated via $mprotect()$.

Given this design, it is not possible to have the commits of one thread occurring while other threads are executing. If $T1$ is executing while $T2$ is committing, $T2$ may commit a page which is read by $T1$ during its execution. To avoid this race condition, Dthreads uses a fence (barrier) to synchronize threads ahead of their commits. Each round of thread execution, followed by fence synchronization and a commit is referred to as a *transaction*. As shown in Figure 5(a), a thread arriving at the fence will wait until all other threads have arrived (completing the *parallel phase*), and then commit its changes once it acquires the token (the *serial phase*).

5.2 DTHREADS with CONVERSION

Replacing Dthreads memory architecture with Conversion yields the following improvements:

1. Conversion *significantly* simplifies the memory management code of Dthreads (see Table 4). In the original design, Dthreads tracks dirty pages, creates and stores ”twin” reference pages, manages page protection levels and implements a number of optimizations (see [28]). Leveraging the Conversion API can replace all of that with a few function calls.
2. The isolation that Conversion provides allows threads to commit while other threads are executing. This lets us *remove the fence entirely* and rely on only the token for deterministic commit, increasing parallelism for certain classes of applications.
3. Conversion handles page faults and bookkeeping in kernel space, which our experiments show to be approximately 2× faster than the user space signal handling of Dthreads.

Below, we describe how Dthreads is retrofitted with Conversion in more depth. §5.3 analyzes the performance of our implementation.

We start by replacing the private and public segments (for the heap and globals) with a single Conversion segment for each. The parallel phase executes as before, however now Conversion provides isolation between threads. The commit operation for a thread requires just two Conversion operations: an `update()` followed by a call to `commit()`. The `update()` will first merge the changes made in the current thread with the latest committed version and `commit()` will of course create a new version.

As described above, the original Dthreads design requires a fence to ensure that one thread's commits do not affect another thread still executing in the parallel phase. With Con-

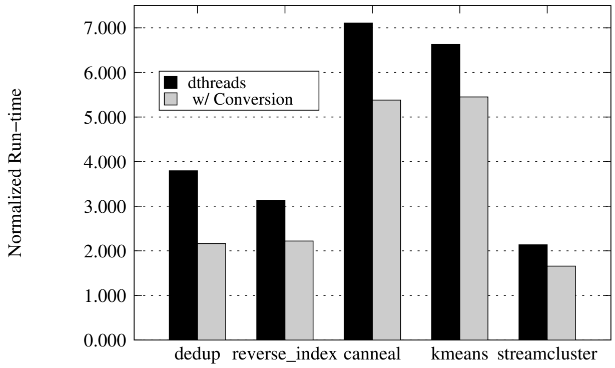


Figure 6. Execution time for Dthreads and with and without CONVERSION, normalized to the pthreads execution time.

VERSION this problem no longer exists; threads are completely isolated from the activities of other threads.

Thus, it is now safe to remove the fence. Figure 5(b) demonstrates the effect of this new execution model on the earlier example shown in Figure 5(a). At the start, Thread T1 possesses the token and is the first to perform a pthread operation, triggering a commit. Because T1 holds the token, and its commit can not affect other threads (because of the isolation inherent in CONVERSION), it performs the commit *wait-free* and continues execution of user code. The other threads perform commits in a similar fashion, forming a *waterfall* pattern. It is important to note that none of the threads spend any time waiting when the waterfall pattern occurs.

The waterfall pattern represents the ideal (parallel) conditions of execution. Another possible outcome is shown in Figure 5(b) after the first set of commits has completed. Here, T2 executes a second pthread operation but cannot perform a commit without the token, which currently belongs to T1. T2 must now wait for T1 to perform a pthread operation and commit its changes before it can acquire the token and commit. Still, at the point where the original Dthreads finishes its first serial phase, the CONVERSION-enhanced Dthreads has almost completed a second round of commits in this example. Such results depend heavily on workload, and are not representative.

5.3 Evaluation

Mirroring the approach in [28], we evaluate our work using the PARSEC [8] and Phoenix [35] benchmark suites. For benchmarks that are mostly data parallel (little synchronization between threads), Dthreads performance rivals pthreads. In some cases, Dthreads can even outperform pthreads due to the elimination of false sharing [28]. We do not consider these benchmarks further, as CONVERSION cannot further improve their performance. However, it is important to note that CONVERSION does not add any additional overhead to these programs and their performance is nearly identical to Dthreads.

benchmark	token (ms)	fence (ms)	commit (ms)	trans. count	page faults
`dedup`	6.3K	17K	251	91K	359K
w/ CONVERSION	10K	0	1.2K	”	434K
`canneal`	49K	25K	14K	2.1K	3.6M
w/ CONVERSION	63K	0	24K	”	3.6M
`reverse_index`	2.8K	5.5K	125	76K	132K
w/ CONVERSION	3.6K	0	415	”	152K
kmeans	17K	112K	127	8.3K	49K
w/ CONVERSION	22K	8.7K	120	”	55K
streamcluster	5.5K	11K	409	260K	135K
w/ CONVERSION	7.8K	0	3.0K	”	137K

Table 5. Statistics for relevant benchmarks for Dthreads and Dthreads w/ CONVERSION. For each benchmark CONVERSION reduces Dthreads related overhead and improves concurrency. †The fence time for `kmeans` w/ CONVERSION includes time for atomic forking of processes (see §5.3.3).

Several of the PARSEC benchmarks are incompatible with Dthreads [28] and additionally we were unable to get the `ferret` benchmark to compile to a 32 bit executable (a requirement for the Dthreads version we are using). Performance results were derived by executing each program using between 8 and 64 threads, choosing whichever provides the fastest execution time.

Figure 6 shows the execution times for original Dthreads and Dthreads w/ CONVERSION normalized to the pthreads execution time. For all five benchmarks, CONVERSION provides a performance improvement for Dthreads. This is particularly true for the `dedup` benchmark, where CONVERSION provides a speedup of 1.75×.

5.3.1 Impact of Removing the Fence

For some benchmarks with a high number of transactions, CONVERSION can provide increased parallelism. Table 5 details benchmark statistics, including the total number of transactions and page faults, and the time (in ms) spent performing Dthreads tasks. For `dedup`, Dthreads w/ CONVERSION exclusively uses the token to establish the commit order, yielding a 2.3× speedup in Dthreads overhead. A similar pattern can be seen for `reverse-index`.

Interestingly, high numbers of transactions *do not* always lead to better performance with CONVERSION. For CONVERSION to improve performance, the programs' use of synchronization operations must lend itself to the waterfall pattern described in §5.2. For example, the threads in `dedup` use a pipeline model to perform work in parallel. Here, if a thread arrives at the synchronization point (the lock acquisition) and acquires the token, it can commit, release the lock, and execute its pipeline stage without blocking. Alternatively, programs like `streamcluster` that make heavy use of barriers benefit less from CONVERSION's removal of the fence.

Here, the user's barrier acts as a stand-in for the fence, forcing threads to wait until all commits have completed.

5.3.2 Commit Performance

Looking at Table 5, we can see that the commit process with CONVERSION is slower than the original Dthreads by 2-7x for benchmarks generating a significant number of page faults. This is the cost that CONVERSION pays for providing absolute isolation. As described in §5.2, the commit process for Dthreads w/ CONVERSION consists of first invoking `update()`, followed by `commit()`. For the original Dthreads design there is no analog to `update()`, as the latest version is already available via the shared mapping. Additionally, the latency of `update()` is typically higher than `commit()`, as all the other thread's committed updates must be retrieved.

However, CONVERSION mitigates this cost by allowing commits to occur while others are still executing. Also, as indicated in Table 5, the total cost of commits is typically low compared to the cost of synchronization (the fence and token). Finally, this cost can be further offset by the reduced page fault overhead, which can be seen in the performance of `canneal`.

5.3.3 Supporting Fork-Join

The Phoenix `kmeans` benchmark, which utilizes a fork-join model of parallelism, exposed a performance issue with forking processes. Dthreads performs thread creation as an atomic operation to enforce determinism. Newly created threads wait for the parent to finish forking all new threads and begin execution once the parent calls `join()` (or another pthreads operation). The total cost of the atomic `fork()` is made particularly acute by the size of the segments and the number of page table entry copies that must be made.

To solve this problem, we mark CONVERSION segments with the *MADV_DONTFORK* flag through the madvise system call. We then use the `update()` operation to provide equivalent functionality but in parallel after the fork has completed. This improved fork performance for the `kmeans` benchmark by over 10x.

6. Related Work

Multi-Version Concurrency Control (MVCC) [7, 34] has been described and used in multiple areas in the past. In database management systems, MVCC is a common feature used to improve concurrency of database transactions. Here, a transaction t operates on a version v_t of the database as it existed at the beginning of t. Before committing t, a validation technique checks t's accesses for serializability [33] with respect to any transactions committed after version v_t. In databases, transactional semantics are implicit in MVCC implementations. The focus of CONVERSION is main-memory concurrent versioning only, and it offers no built-in transactional semantics.

Another very popular use of multi-version concurrency control is in version controlled source code and document repositories such as CVS [13] or Subversion [15]. Here, authors maintain local copies of a document, and periodically commit their edits to a central repository. This is conceptually quite similar to CONVERSION although the merging process differs substantially. Whereas CONVERSION operates on fixed-size memory segments and provides a basic byte-by-byte merging function, source code repositories typically deal with variable size documents, and relies on the author to manually resolve any merging conflicts.

Finally, some Software Transactional Memory (STM) implementations, including Clojure [26] use MVCC. By checking for serializability, rather than concurrent access, such systems are able to improve STM concurrency. Current MVCC-based STM implementations are all object-based and intended for higher-level languages. By contrast, CONVERSION operates on bytes and pages and is well suited to both low- and high-level languages. However, as CONVERSION does not offer any transactional semantics it is only a small part of a full STM system. By reducing its scope, CONVERSION is able to avoid the high cost of current STM approaches [12], and is readily implementable as a Linux kernel module running on contemporary PC hardware. Using CONVERSION to implement a full STM system is feasible, but the page-level granularity of CONVERSION may significantly degrade performance. An STM implementation based on CONVERSION falls outside the scope of this paper.

6.1 Support for Concurrent Memory Access

Several alternatives to using shared memory with mutual exclusion primitives have been proposed in the past, including hardware and software transactional memory [1, 19, 22–25, 37], deterministic concurrency support [4, 6, 28], and memory snapshots [14].

Software Transactional Memory (STM) [23, 24, 37] is implemented by expanding loads and stores inside transactions into transactional versions of the same, either at compile-time (for low-level language implementations) or at runtime (for high-level languages). This incurs a significant sequential overhead which (so far) has prevented the widespread use of STM.

Grace [6], Dthreads [28], and Determinator OS [4] use copy-on-write techniques similar to those presented here, but for the purpose of enforcing deterministic execution of multi-threaded programs. Revisions and isolation types [11] provide isolation and conflict merging, but (like Grace) enforce a strict fork-join model with an emphasis on determinism.

6.2 Memory Consistency Models

Version controlled memory was first proposed in [16] as a memory model for deterministic concurrency. While CONVERSION shares conceptual similarities with this work, no implementation or evaluation was provided and the authors make reference to a "steep performance cost." Additionally, CONVERSION shares some commonalities with consistency models from distributed shared memory, most notably release consistency (RC) [21] and lazy release consistency (LRC) [27]. In RC, access to shared memory variables by each process are book-ended with `acquire` and `release` operations (similar to `update()` and `commit()`). While in RC a `release` operation immediately pushes updates out to other processes, in LRC the updates are propagated on demand after an `acquire` is invoked. However, contrary to CONVERSION (L)RC enforces mutual exclusion between `acquire` and `release` and does not allow for the "local copy" design of CONVERSION.

Workspace consistency (WC) [2] is more recent work, providing a consistency model that enforces determinism. In WC, threads work on their local copies and synchronize between *matching* `acquire` and `release` operations. In other words, updates are propagated to a specific other thread where any conflicts are merged by the thread performing the `acquire`. Unlike WC, CONVERSION does not enforce an up-front ordering of thread communication. However, as shown in §5.2, CONVERSION allows similar functionality to be enforced at the application layer.

7. Future Work and Conclusion

In §4 we saw that CONVERSION struggles to scale for workloads that generate contention in `commit()`. This was due to both the version list lock and the non-scalability of ticket spin locks in the Linux kernel. In future work, we look towards further concurrency improvements for `commit()` with the goal of making the operation lock-less.

We have presented the design, implementation and evaluation of CONVERSION, a kernel-level multi-version concurrency control system for main-memory segments. CONVERSION operations are fast, starting at a few microseconds and growing linearly (by less than 1 μs) with the number of modified pages since the last version was retrieved. We have demonstrated the merits of CONVERSION by retrofitting and evaluating Dthreads [28] with CONVERSION, providing a significant increase in performance and a reduction in code size for memory management.

Our CONVERSION implementation takes the form of a Linux kernel module and requires very little modification to the existing kernel. The source code for CONVERSION will be made available for free download from our web site by the time of publication.

8. Acknowledgments

This material is based upon work supported by the U.S. National Science Foundation under Grants CNS-1017877 and CNS-1149989.

References

[1] M. Abadi, T. Harris, and M. Mehrara. Transactional Memory with Strong Atomicity using Off-The-Shelf Memory Protection Hardware. PPoPP '09, 2009.

[2] A. Amittai, B. Ford, and Y. Zhang. Workspace Consistency: A Programming Model for Shared Memory Parallelism. 2nd WoDet, 2011.

[3] A. Aviram and B. Ford. Deterministic OpenMP for Race-Free Parallelism. HotPar, 2011.

[4] A. Aviram, S.-C. Weng, S. Hu, and B. Ford. Efficient Eystem-Enforced Deterministic Parallelism. OSDI, 2010.

[5] 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*, 24(2):1–10, 1995.

[6] E. D. Berger, T. Yang, T. Liu, and G. Novark. Grace: Safe Multithreaded Programming for C/C++. OOPSLA '09, 2009.

[7] P. Bernstein and N. Goodman. Multiversion Concurrency Control–Theory and Algorithms. *ACM TODS*, 8(4):465–483, 1983.

[8] C. Bienia, S. Kumar, J. P. Singh, and K. Li. The PARSEC Benchmark Suite: Characterization and Architectural Implications. In *PACT*, PACT '08, 2008.

[9] R. L. Bocchino, Jr., V. S. Adve, S. V. Adve, and M. Snir. Parallel Programming Must be Deterministic by Default. HotPar, 2009.

[10] S. Boyd-Wickizer, M. F. Kaashoek, R. Morris, and N. Zeldovich. Non-Scalable Locks are Dangerous. In *Proceedings of the Linux Symposium*, Ottawa, Canada, July 2012.

[11] S. Burckhardt, A. Baldassin, and D. Leijen. Concurrent Programming with Revisions and Isolation Types. OOPSLA '10, 2010.

[12] C. Cascaval, C. Blundell, M. Michael, H. W. Cain, P. Wu, S. Chiras, and S. Chatterjee. Software Transactional Memory: Why Is It Only a Research Toy? *Queue*, 6:46–58, September 2008.

[13] P. Cederqvist, R. Pesch, et al. Version Management with CVS. *Available online with the CVS package. Signum Support AB*, 1992.

[14] J. chung, W. Baek, and C. Kozyrakis. Fast Memory Snapshot for Concurrent Programming without Synchronization. ICS, 2009.

[15] B. Collins-Sussman, B. Fitzpatrick, and C. Pilato. *Version Control with Subversion*. O'Reilly Media, Incorporated, 2004.

[16] D. Hower and M. Hill. Hobbes: CVS for Shared Memory. In *Workshop on Determinism and Correctness in Parallel Programming*, 2011.

[17] A. Demers, M. Weiser, B. Hayes, H. Boehm, D. Bobrow, and S. Shenker. Combining Generational and Conservative Garbage Collection: Framework and Implementations. POPL '90, 1990.

[18] D. Dice, O. Shalev, and N. Shavit. Transactional Locking II. *Distributed Computing*, pages 194–208, 2006.

[19] D. Dice and N. Shavit. What Really Makes Transactions Faster. In *TRANSACT*, 2006.

[20] H. Franke, R. Russell, and M. Kirkwood. Fuss, Futexes and Furwocks: Fast Userlevel Locking in Linux. In *Ottawa Linux Symposium*, 2002.

[21] K. Gharachorloo, D. Lenoski, J. Laudon, P. Gibbons, A. Gupta, and J. Hennessy. *Memory Consistency and Event ordering in Scalable Shared-Memory Multiprocessors*, volume 18. ACM, 1990.

[22] 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. ISCA, 2004.

[23] T. Harris and K. Fraser. Language Support for Lightweight Transactions. OOPSLA, 2003.

[24] M. Herlihy, V. Luchangco, M. Moir, and W. N. Scherer, III. Software Transactional Memory for Dynamic-Sized Data Structures. PODC, 2003.

[25] M. Herlihy and J. Moss. Transactional Memory: Architectural Support for Lock-Free Data Structures. *ACM SIGARCH Computer Architecture News*, 21(2):289–300, 1993.

[26] R. Hickey. The Clojure Programming Language. In *Symposium on Dynamic languages*, page 1. ACM, 2008.

[27] P. Keleher, A. Cox, and W. Zwaenepoel. *Lazy Release Consistency for Software Distributed Shared Memory*, volume 20. ACM, 1992.

[28] T. Liu, C. Curtsinger, and E. D. Berger. Dthreads: Efficient Deterministic Multithreading. SOSP, 2011.

[29] R. Love. *Linux Kernel Development*. Novell Press, 2nd edition, 2005.

[30] L. Lu and M. L. Scott. Unmanaged Multiversion STM. 2012.

[31] V. Marathe, M. Spear, C. Heriot, A. Acharya, D. Eisenstat, W. Scherer III, and M. Scott. Lowering the Overhead of Nonblocking Software Transactional Memory. In *Workshop on Languages, Compilers, and Hardware Support for Transactional Computing (TRANSACT)*, 2006.

[32] P. McKenney and J. Slingwine. Read-Copy Update: Using Execution History to Solve Concurrency Problems. In *Parallel and Distributed Computing and Systems*, pages 509–518, 1998.

[33] C. Papadimitriou. The Serializability of Concurrent Database Updates. *Journal of the ACM (JACM)*, 26(4):631–653, 1979.

[34] C. Papadimitriou and P. Kanellakis. On Concurrency Control by Multiple Versions. *ACM TODS*, 9(1):89–99, 1984.

[35] C. Ranger, R. Raghuraman, A. Penmetsa, G. Bradski, and C. Kozyrakis. Evaluating MapReduce for Multi-core and Multiprocessor Systems. In *HPCA*, pages 13–24, 2007.

[36] T. Riegel, C. Fetzer, and P. Felber. Snapshot Isolation for Software Transactional Memory. *TRANSACT06*, 298, 2006.

[37] N. Shavit and D. Touitou. Software Transactional Memory. PODC, 1995.