Fast, fault-tolerant transactions as an OS extension

Benjamin A. Braun, David Cheriton
Stanford University
HPTS 2019
Motivation: Complex, durable state

- Large, ad-hoc shared state, with durability, parallelism

<table>
<thead>
<tr>
<th>Dataset size</th>
<th>Durable</th>
</tr>
</thead>
<tbody>
<tr>
<td>Telecom host registry</td>
<td>TBs, Txns</td>
</tr>
<tr>
<td>Online game</td>
<td>10s of TBs, Txns</td>
</tr>
<tr>
<td>DNA sequence alignment/assembly</td>
<td>100s of GBs, Checkpoints</td>
</tr>
<tr>
<td>Fire/weather simulation</td>
<td>TBs, Checkpoints</td>
</tr>
</tbody>
</table>
Queries can feel limiting & inconvenient to developer

Socket communication round-trip time on data access path
(>2 µs local, >7 µs in-datacenter)

• DBMS aren’t always the right solution
Transactions at the speed of memory

Database libraries arose to serve in-memory data faster.

Transactions in terms of simple, efficient function calls to library

- Libraries offer durability, memory-speed access
Application bugs with database libraries

Buggy application "scribbles" over database datastructures
⇒ data loss, possibly corruption

- The root problem is a lack or isolation
• Database library applies changes. No isolation.
OS isolation: Transactional Virtual Memory

- OS kernel applies changes. Shared memory with isolation. **Transactional virtual memory.**
TVM vs. database libraries, DBMS

• Crash failures: **Protected.**
• Application bugs: **Some protection.**
  – Write scribbles don’t hit durable data unless committed
  – Shared state isolated from application bugs
• Data structure: **Unstructured.**
  – Same API as software transactional memory
  – No support for queries
• Toolkit: **debugger, performance traces, load balancing, resource limiting, access control, work as-is from OS.**
  – Mostly same as libraries
  – DBMS IPC breaks control flow

TVM combines useful properties of both DBMS, libraries
This is a terrible idea

OS locks slower than memory locks (>1.4 µs vs >200 ns)

Puts system call (>400 ns) on the commit path

Puts page fault (>1.2 µs) on the data access path

- Seems dead in the water.
Transaction processing tricks

- Defer locking until commit, snapshot isolation
- Avoid repeated faults, keep pages across txns
- Buffer writes in userspace, read-your-writes cache

Getting the most out of each OS entrance
Giving up on serializability

Given SI-safe dictionary, arrays, queues, can do most tasks

*Conflict-on-free*: Allocator writes zeros on free
  - Fixes anomalies deriving from use-after-free

Automatic correction of snapshot isolation anomalies
[Litz, et.al., 2015]

  - Have tools to avoid snapshot isolation anomalies

• Shared pages, page tables for write-cold data
• Proxy pages for write-hot data

Can use heuristics to balance frequent faults with upkeep
1. We show TVM achieves durable transaction performance >95% that of software transactional memory
2. We demonstrate porting parallel applications to TVM, adding fault-tolerance without massive application changes
3. We prove that TVM can be implemented as a simple extension to a conventional OS
Implementation of TVM for Linux: *Thistle*
Thread
Isolating Transactions as Kernel Extension. (Thistle)
Evaluation

Implementation of TVM for Linux: *Thistle*

Comparison vs DBMS: MySQL, KeyDB

Comparison vs software transactional memory (STM) + write-ahead logging for durability. Systems tested:

- A snapshot isolation variant of TinySTM.
- SI-TM inspired Multi-version concurrency control (MVCC)

Experiment controls:

- Same implementation of write-ahead logging
  - 5µs simulated log persistence latency
- Same implementation of transactional `malloc/free`
- Lock / timestamp table 100K entries, 64-byte granularity
- Snapshot isolation transactions
TVM performance vs DBMS

- Stark contrast between inter-process communication (IPC – DBMS) and in-memory access (TVM).
- Note: KeyDB does not scale past 4 threads

IPC-based isolation a significant overhead for DBMS.
Virtual memory manipulations can take up to 50% runtime
For 8 out of 12 experiments, relative performance is >80%.
Validating reads

Relax requirement for view to match snapshot timestamp

- Timestamp validated cache line reads in userspace

\[
\text{TM\_READ(word* pointer):}
\]
\[
\begin{align*}
\text{word toRet} &= \ast \text{pointer} \\
\text{if } (\text{TIMESTAMP\_TABLE[hash(address)]} > \text{snapshot_ts}) \text{ then abort}
\end{align*}
\]

What do we lose?

- Unmodified code can’t read safely
- Single version \(\implies\) read-after-write aborts
- Sequential reads slower due to validation

Read validation avoids costs of page table manipulation, loses some of the benefits of snapshot isolation
TVM + validating reads vs durable STM

>95% the speed of state-of-the-art STMs on 8 benchmarks

Eliminates most overheads, OS entry/exit up to 10% runtime
Round-based simulations

- Particle simulation, 4GB of particles, sharded by position
- Move particles (non-txnal) then reshard each round (txnal)
- Recovery point every 1s adds <3% overall overhead.
  - Reference: Dumping 4GB striped across 2 SSDs takes ∼1s

TVM gives both checkpoint durability and txnal durability
• Memory-speed KV-store that tolerates bugs, kill signals
• R analysis on a changing data table file (700MB, text)
  – Time to load data table <3% longer than from memory file.

Applications with ad-hoc data format, concurrency control, but don’t need structured queries.
Thistle

- Loadable kernel module for Linux v4.2.1+ and a userspace library `libThistle`
- New filesystem type: `thistle`

<table>
<thead>
<tr>
<th># Lines of code</th>
<th>Thistle kernel module</th>
<th>Thistle userspace library</th>
</tr>
</thead>
<tbody>
<tr>
<td>~13K</td>
<td></td>
<td></td>
</tr>
<tr>
<td>966</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

0 Lines of code changed in Linux.

For comparison: Linux mm subsystem is ~80k lines.

- TVM can be implemented without changes to Linux.
The OS is not the enemy.

I’m looking for an job in industry.
bjmnbraun at gmail dot com
(Backup slides.)
Transaction processing in Thistle

Snapshot isolation transactions
Userspace creates writeset, read-your-writes cache
Characterizing TVM performance

TVM handles two cases especially well, compared to STMs:

- **Heavily read datastructures with concurrent modifiers**
  - Reads to concurrently written state lead to private pages
  - Reads to private pages never abort
  - Private pages never have false sharing
  - \( \sim 1.8x, 2.8x \) speedups on labyrinth, linked-list

- **Sequential access**
  - Once a page is mapped into a threads’ view, reads are free.
  - 1.3x speedup on sequential btree 95% reads

TVM tends to lose for high degrees of write contention.

- No system handles these well
- High contention is an anti-pattern
Application changes to use TVM

- Each thread must be its own process
  - i.e. `pthread_spawn()` -> `fork()`
- Application must place shared data in TVM-managed **files**
  - Modified allocator provided for this purpose
  - Global variables might need to be refactored
- Add `transaction{}` blocks to denote transactions
  - Code transformations achievable with modern compilers

Code changes to use TVM are minor.
Porting STAMP suite to OS-level TM

STAMP suite details: 7 applications, \(\sim\)18K lines of code

<table>
<thead>
<tr>
<th># Lines</th>
<th>Reason for change</th>
</tr>
</thead>
<tbody>
<tr>
<td>85</td>
<td>Add multi-process support (fork, thread barriers, etc.)</td>
</tr>
<tr>
<td>240</td>
<td>Place shared data into transactionally managed segment</td>
</tr>
<tr>
<td>14</td>
<td>Snapshot isolation transaction model (intruder)</td>
</tr>
<tr>
<td>339</td>
<td>Total lines of code changed</td>
</tr>
</tbody>
</table>

Overall: <2% of lines changed to port STAMP to Thistle.
Snapshot-isolation anomaly in intruder

Fragmented Packet, N=7

T1 + T2 +

transaction {
    fragment = QUEUE_POLL(fragmentChannel);
    packet = FIND_PACKET(fragment.packetID);
    exists = LIST_INSERT(packet, fragment);
    if (exists) { goto error; }
    long list_size_after = LIST_GETSIZE(packet);
    if (list_size_after == packet.N) {
        QUEUE.Push(completePackets, packet);
    }
}

• Adding a length field to the packet list resolves the anomaly in just 14 lines of code.
**Performance breakdown**

<table>
<thead>
<tr>
<th>Application</th>
<th>Page fault</th>
<th>Syscall overhead</th>
<th>Check conflicts</th>
<th>Commit</th>
<th>Update</th>
<th>Persistence wait</th>
</tr>
</thead>
<tbody>
<tr>
<td>labyrinth</td>
<td>0.26</td>
<td>0.15</td>
<td>0.00</td>
<td>0.00</td>
<td>0.00</td>
<td>0.00</td>
</tr>
<tr>
<td>kmmeans</td>
<td>0.26</td>
<td>0.15</td>
<td>0.00</td>
<td>0.00</td>
<td>0.00</td>
<td>0.00</td>
</tr>
<tr>
<td>btree seq 95%R</td>
<td>0.26</td>
<td>0.15</td>
<td>0.00</td>
<td>0.00</td>
<td>0.00</td>
<td>0.00</td>
</tr>
<tr>
<td>btree 95%R</td>
<td>0.26</td>
<td>0.15</td>
<td>0.00</td>
<td>0.00</td>
<td>0.00</td>
<td>0.00</td>
</tr>
<tr>
<td>genome</td>
<td>0.26</td>
<td>0.15</td>
<td>0.00</td>
<td>0.00</td>
<td>0.00</td>
<td>0.00</td>
</tr>
<tr>
<td>btree 50%R</td>
<td>0.26</td>
<td>0.15</td>
<td>0.00</td>
<td>0.00</td>
<td>0.00</td>
<td>0.00</td>
</tr>
<tr>
<td>intruder</td>
<td>0.26</td>
<td>0.15</td>
<td>0.00</td>
<td>0.00</td>
<td>0.00</td>
<td>0.00</td>
</tr>
<tr>
<td>linked list</td>
<td>0.26</td>
<td>0.15</td>
<td>0.00</td>
<td>0.00</td>
<td>0.00</td>
<td>0.00</td>
</tr>
<tr>
<td>vac. 80%R</td>
<td>0.26</td>
<td>0.15</td>
<td>0.00</td>
<td>0.00</td>
<td>0.00</td>
<td>0.00</td>
</tr>
<tr>
<td>vacation</td>
<td>0.26</td>
<td>0.15</td>
<td>0.00</td>
<td>0.00</td>
<td>0.00</td>
<td>0.00</td>
</tr>
<tr>
<td>yada</td>
<td>0.26</td>
<td>0.15</td>
<td>0.00</td>
<td>0.00</td>
<td>0.00</td>
<td>0.00</td>
</tr>
<tr>
<td>ssc2</td>
<td>0.26</td>
<td>0.15</td>
<td>0.00</td>
<td>0.00</td>
<td>0.00</td>
<td>0.00</td>
</tr>
</tbody>
</table>

- Page fault at most 26%
- Syscall overheads at most 15% (mostly in syscall entry)

Largest overheads: Page faults, syscalls, and updating page map.
- For yada, update dominated by software page table walks
- For sscat, update dominated by applying writes to private pages

Largest overheads: Page faults, syscalls, and updating page map.
Performance breakdown (detailed)

- Application
- Page fault overhead
- Page fault
- New private page
- Syscall overhead
- Check conflicts
- Log append
- Update page metadata
- Update
- Page cache shrink
- Syscall exit
- Persistence wait
- Checkpoints
- GC checkpoints
Breaking down pgfault overheads

- Poor scaling in `down_read` on `mmap_sem`
- Significant software overheads in `page_fault`, `_handle_mm_fault`.

<table>
<thead>
<tr>
<th>Overhead</th>
<th>Command</th>
<th>Shared Object</th>
<th>Symbol</th>
</tr>
</thead>
<tbody>
<tr>
<td>13.94%</td>
<td>btree_benchmark</td>
<td>btree_benchmark</td>
<td><code>[.]_Z3runm</code></td>
</tr>
<tr>
<td>12.68%</td>
<td>btree_benchmark</td>
<td>[sivfs]</td>
<td><code>[k] sivfs_fault</code></td>
</tr>
<tr>
<td>11.51%</td>
<td>btree_benchmark</td>
<td>[kernel.kallsyms]</td>
<td><code>[k] down_read</code></td>
</tr>
<tr>
<td>10.26%</td>
<td>btree_benchmark</td>
<td>[kernel.kallsyms]</td>
<td><code>[k] page_fault</code></td>
</tr>
<tr>
<td>6.08%</td>
<td>btree_benchmark</td>
<td>[kernel.kallsyms]</td>
<td><code>[k] __handle_mm_fault</code></td>
</tr>
<tr>
<td>5.13%</td>
<td>btree_benchmark</td>
<td>[kernel.kallsyms]</td>
<td><code>[k] swapsgs_restore_regs_and_return_to_usermode</code></td>
</tr>
<tr>
<td>4.99%</td>
<td>btree_benchmark</td>
<td>[kernel.kallsyms]</td>
<td><code>[k] error_entry</code></td>
</tr>
<tr>
<td>4.56%</td>
<td>btree_benchmark</td>
<td>[sivfs]</td>
<td><code>[k] sivfs_snapshot_view_prepare_access</code></td>
</tr>
<tr>
<td>3.64%</td>
<td>btree_benchmark</td>
<td>[kernel.kallsyms]</td>
<td><code>[k] handle_mm_fault</code></td>
</tr>
<tr>
<td>3.50%</td>
<td>btree_benchmark</td>
<td>[sivfs]</td>
<td><code>[k] sivfs_checkpoint_traverse_partial</code></td>
</tr>
<tr>
<td>2.94%</td>
<td>btree_benchmark</td>
<td>[sivfs]</td>
<td><code>[k] __sivfs_snapshot_view_map_cp_children</code></td>
</tr>
<tr>
<td>2.73%</td>
<td>btree_benchmark</td>
<td>[kernel.kallsyms]</td>
<td><code>[k] __radix_tree_lookup</code></td>
</tr>
<tr>
<td>2.25%</td>
<td>btree_benchmark</td>
<td>[kernel.kallsyms]</td>
<td><code>[k] __do_page_fault</code></td>
</tr>
<tr>
<td>1.78%</td>
<td>btree_benchmark</td>
<td>[kernel.kallsyms]</td>
<td><code>[k] handle_pte_fault</code></td>
</tr>
<tr>
<td>1.45%</td>
<td>btree_benchmark</td>
<td>[kernel.kallsyms]</td>
<td><code>[k] sync_regs</code></td>
</tr>
<tr>
<td>1.23%</td>
<td>btree_benchmark</td>
<td>[kernel.kallsyms]</td>
<td><code>[k] up_read</code></td>
</tr>
<tr>
<td>0.99%</td>
<td>sivfs_persisten</td>
<td>[sivfs]</td>
<td><code>[k] sivfs_persistence_threadfn</code></td>
</tr>
<tr>
<td>0.81%</td>
<td>btree_benchmark</td>
<td>[sivfs]</td>
<td><code>[k] sivfs_task_to_state</code></td>
</tr>
<tr>
<td>0.76%</td>
<td>btree_benchmark</td>
<td>[kernel.kallsyms]</td>
<td><code>[k] vmcache_find</code></td>
</tr>
<tr>
<td>0.65%</td>
<td>btree_benchmark</td>
<td>[kernel.kallsyms]</td>
<td><code>[k] __raw_spin_lock</code></td>
</tr>
<tr>
<td>0.53%</td>
<td>btree_benchmark</td>
<td>[kernel.kallsyms]</td>
<td><code>[k] down_read_trylock</code></td>
</tr>
<tr>
<td>0.45%</td>
<td>btree_benchmark</td>
<td>[sivfs]</td>
<td><code>[k] _sivfs_snapshot_view_invalidate_range</code></td>
</tr>
<tr>
<td>0.44%</td>
<td>btree_benchmark</td>
<td>[kernel.kallsyms]</td>
<td><code>[k] pmd_devmap_trans_unstable</code></td>
</tr>
<tr>
<td>0.39%</td>
<td>btree_benchmark</td>
<td>[kernel.kallsyms]</td>
<td><code>[k] find_get_entries</code></td>
</tr>
<tr>
<td>0.32%</td>
<td>btree_benchmark</td>
<td>[kernel.kallsyms]</td>
<td><code>[k] mem_cgroup_from_task</code></td>
</tr>
<tr>
<td>0.28%</td>
<td>tmux</td>
<td>tmux</td>
<td><code>[..]_init</code></td>
</tr>
<tr>
<td>0.27%</td>
<td>btree_benchmark</td>
<td>[kernel.kallsyms]</td>
<td><code>[k] native_queued_spin_lock_slowpath</code></td>
</tr>
<tr>
<td>0.23%</td>
<td>btree_benchmark</td>
<td>[kernel.kallsyms]</td>
<td><code>[k] find_vma</code></td>
</tr>
<tr>
<td>0.22%</td>
<td>btree_benchmark</td>
<td>[kernel.kallsyms]</td>
<td><code>[k] prepare_exit_to_usermode</code></td>
</tr>
</tbody>
</table>
TLB Invalidation costs

Our test machine (Intel x86) provides a single-page TLB flush instruction as well as a full TLB flush instruction.

<table>
<thead>
<tr>
<th></th>
<th>% Speedup:</th>
<th>Entire TLB flush</th>
</tr>
</thead>
<tbody>
<tr>
<td>btree seq</td>
<td>1</td>
<td>95%R/W</td>
</tr>
<tr>
<td>btree</td>
<td>1</td>
<td>95%R/W</td>
</tr>
<tr>
<td>btree</td>
<td>5</td>
<td>50%R/W</td>
</tr>
<tr>
<td>genome</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>intruder</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>kmeans</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>labyrinth</td>
<td>12</td>
<td></td>
</tr>
<tr>
<td>linked list</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>sasca2</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>vacation</td>
<td>2</td>
<td></td>
</tr>
<tr>
<td>vaca. 80%R/W</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>yada</td>
<td>8</td>
<td></td>
</tr>
</tbody>
</table>

• The single-page TLB flush instruction is sufficiently expensive that calling it multiple times is prohibitive.
• Flushing the entire TLB instead after update (when at least one page is changed) provided up to a ∼10% speedup.
• Validating reads eliminates page table manipulation costs, at the price of requiring code modification to read