Multiversioned Page Overlays: Enabling Faster Serializable Hardware Transactional Memory

Ziqi Wang
Carnegie Mellon University
ziqiw@cs.cmu.edu

Michael A. Kozuch
Intel Labs
michael.a.kozuch@intel.com

Todd C. Mowry
Carnegie Mellon University
tcm@cs.cmu.edu

Vivek Seshadri
Microsoft Research India
visesha@microsoft.com

***Abstract*—Practical and efficient support for *multiversioning* memory systems would offer a number of potential advantages, including improving the performance and functionality of *hardware transactional memory* (HTM). This paper presents a new approach to multiversioning support (Multiversioned Page Overlays) along with a new HTM design that it enables: OverlayTM. Compared with existing HTM designs, OverlayTM takes advantage of multiversioning to reduce unnecessary transaction aborts while providing full serializable semantics (in contrast with multiversioning HTMs that improve performance at the expense of being vulnerable to write skew anomalies). Our performance results demonstrate that OverlayTM is especially advantageous in read-heavy workloads.**

I. Introduction

Hardware Transactional Memory (HTM) [1] has gained considerable traction in recent years due to its usefulness in avoiding synchronization bugs while writing parallel software. While today's commercial HTM implementations [2]–[6] are helpful for programmers, they typically experience more transaction aborts than are strictly necessary due to limitations such as the sizes of the caches (or other hardware buffers), the conservative nature of eager conflict detection, etc. To help address these limitations, there has been research on techniques for supporting *unbounded transactions* [7]–[10] as well as *lazy conflict detection* [11]–[14].

Challenge: Achieving both High Performance and Strong Semantics with HTM. Across the spectrum of HTM designs, there is a fundamental tension between providing the *highest performance* and providing the *strongest correctness guarantees* (aka "semantics") to programmers. This is roughly analogous to the performance-versus-correctness tradeoffs with memory consistency models [15]: programmers would prefer the strong semantics of sequential consistency (to help avoid correctness bugs), but weaker consistency models can offer higher performance. As illustrated in Fig. 1, the strongest semantic model for HTM is full serializability of transactions. At the other end of the spectrum is unsynchronized transactions, which offer high performance but no isolation guarantees.

Snapshot isolation is an interesting semantic model in the middle of this spectrum. By maintaining a consistent snapshot of memory that a transaction can read from throughout its execution, snapshot isolation eliminates the need to check for read-write conflicts across transactions. Hence HTM designs based upon snapshot isolation (e.g., SI-TM [14]) have been shown to outperform previous HTM designs with

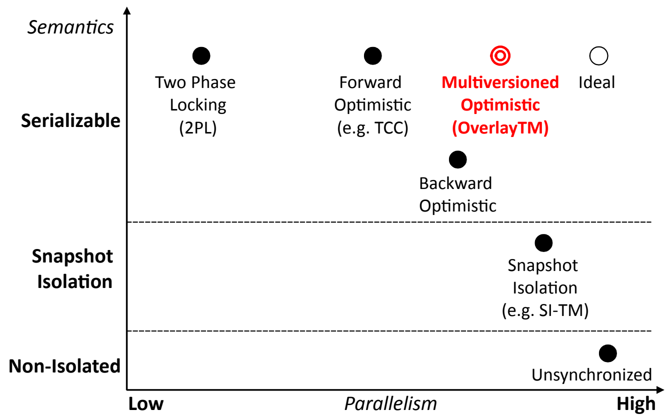

Fig. 1: **Trade-off Between Parallelism and Semantics for HTM**

serializable semantics due to reduced abort rates (especially in workloads with frequent read-only transactions). While the weaker semantic model of snapshot isolation offers performance advantages, it does so at the cost of enabling concurrency bugs due to *write skew anomalies* [14]. For example, we observe that one of the STAMP benchmarks [16] (`genome`) does not run correctly under snapshot isolation (when compiled with standard libraries) due to a write skew anomaly.

Similar to how researchers have explored techniques for closing the performance gap between sequential consistency and relaxed memory consistency models [18], our goal in this paper is to offer the strong transactional semantics of full serializability while closing as much of the performance gap with snapshot isolation (e.g., SI-TM) as possible. Similar to snapshot isolation, our design also maintains snapshots of memory, but we use them to accelerate performance while still providing the strong semantics of full serializability. For both snapshot isolation and our design, however, a key technical challenge is efficiently maintaining these snapshots through some form of *multiversioned memory system*.

Multiversioned Page Overlays (MPO): Efficient Support for Multiversioned Memory. Our multiversioning design builds upon *page overlays* [19]. The page overlays framework maps each virtual page to a default physical page (similar to today's virtual memory systems), but it *additionally* enables the optional mapping of *cache-line-sized* portions of the virtual address space to alternate physical addresses (called "overlays"). During a memory access, if an overlay exists for the particular

TABLE I: Qualitative comparison of OverlayTM with prior work.

Design	Conflict Detection	Serializable Semantics	Read-Only Optimization	Hardware Multiversioning	Unbounded Transactions	No Broadcasting	No Commit-Time Cache Flush
2PL [1]	Two Phase Locking	✓	✕	✕	✕	✓	✓
TCC [17]	Forward OCC	✓	✕	✕	Partly	✕	✓
SI-TM [14]	Snapshot Isolation	✕	✓	✓	✓	✓	✕
OverlayTM	Backward OCC	✓	✓	✓	✓	✓	✓

virtual address, it will access that overlay; otherwise, the default physical page will be accessed. In the original page overlays design, only a single overlay can exist for a given virtual address. As described in detail later in Section III, we extend this design to support *multiple* overlays per virtual address where the desired overlay is specified through an *Overlay ID (OID)*. To handle version requests from multiple processors, we also propose adding a *version directory* that tracks which version is cached by which processor. The version directory is comparable to a coherence directory, and can be implemented with similar hardware costs. The original page overlays paper [19] discussed six interesting use cases for overlays beyond HTM: i.e. overlay-on-write, sparse data structures, fine-grained deduplication, checkpointing, fine-grained metadata management and flexible super-pages. Each of these techniques can potentially benefit from MPO (e.g., when N processes are sharing the same data structures, etc.).

While it would be relatively straightforward to use MPO's multiversioning support to implement an HTM with snapshot isolation, our goal was to deliver comparable performance but with much stronger (fully serializable) semantics. We call our new HTM design *OverlayTM*.

OverlayTM: A Fast Serializable HTM that Combines Multiversioning with Optimistic Concurrency Control. A key performance benefit of an HTM based upon snapshot isolation (e.g, SI-TM) compared with previous serializable HTM designs is that *read-only transactions* can always successfully commit under snapshot isolation. This is because the snapshot guarantees the consistency of any data that is read throughout the read-only transaction. As illustrated in the qualitative comparison in Table I, our OverlayTM design also leverages multiversioning to enable read-only transactions to commit successfully. However, a key difference is that our conflict detection in OverlayTM is not based upon snapshot isolation (SI), but rather upon *backward optimistic concurrency control* (backward OCC). As we describe in detail later in Section IV, this key difference means that OverlayTM detects conflicts that would cause serializability to fail.

Compared with most of today's commercial HTM implementations that have pessimistic two-phase locking (2PL) conflict detection, OverlayTM offers better performance by enabling conflicting transactions to co-exist through OCC. By combining backward OCC with MPO's multiversioning support, OverlayTM not only ensures that transactions are isolated from each other, it also improves performance by allowing read-only transactions to successfully commit. Hence within the spectrum of HTM designs, OverlayTM appears near the upper-right corner in Fig. 1, achieving strong semantics (full serializability) and relatively high parallelism.

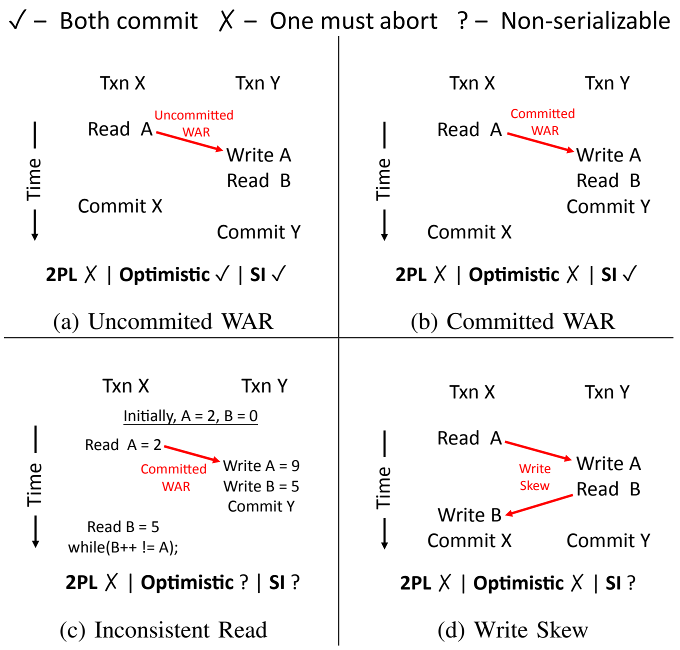

Fig. 2: **False Aborts and Anomalies in Conflict Detection** – As semantics are relaxed, more read-write interleavings become possible. In the meantime anomalies may also occur.

This paper makes the following contributions:

- We propose **Multiversioned Page Overlays (MPO)**, a novel technique based upon Page Overlays [19] for creating memory snapshots and supporting multiversioning on the entire address space with relatively low cost.
- We propose **OverlayTM**, a multiversion-based HTM design that supports unbounded transactions and fast read-only transactions. OverlayTM requires relatively modest hardware changes.
- We evaluate the performance of OverlayTM. Our results show that OverlayTM often offers signficant reductions in abort rates compared with previous HTM proposals, and it achieves performance that is either better than or comparable to the state of the art. Compared with SI-TM, OverlayTM achieves similar performance while offering *serializability* (as opposed to *snapshot isolation*, which risks write skew anomalies).

II. Background: Conflict Detection in HTM

Conflict detection in HTM has a close relationship with concurrency control algorithms in transaction processing.

Herlihy et al. [1], one of the earliest HTM proposals, uses a variant of Two-Phase Locking (2PL) in which the lock holder gives up the lock and aborts when the lock is requested. Read and write locks on individual cache lines map to read-shared and read-exclusive requests in invalidation-based coherence protocols such as MESI. Fig. 2a gives an example of read-write conflict in 2PL. X must abort because it will receive a read-exclusive request when Y writes A. LogTM [7], [20] implements 2PL in the coherence directory as "sticky bits". Instead of forcing transactions to abort on conflict, LogTM allows processors to be stalled until the lock holder completes. VTM [8] extends coherence-based 2PL by adding an auxiliary data structure called XADT. Speculative status of in-cache blocks are represented by their coherence states as in the baseline 2PL protocol, while those evicted from the cache are entered into the XADT. The XADT is searched against incoming coherence requests to ensure that conflicts with evicted lines are still detected.

Detecting conflicts eagerly (as in 2PL) may introduce unnecessary aborts or stalls, since conflicts are only harmful when they form cycles. TCC [11], [17] and Bulk [12], [21] overcome this problem by detecting conflicts lazily before transaction commit. This is similar to Optimistic Concurrency Control (OCC) commonly seen in database engines [22]–[24]. OCC divides transaction execution into three phases: read, validation and write. In the read phase, the transaction body is executed. Speculative data is buffered and invisible to other transactions. Every transaction should maintain a local read- and write-set (RW set) for transactionally accessed data. In the validation phase, each transaction attempts to commit by testing its RW sets against the RW sets of concurrent transactions– those recently committed and possibly also those still executing. In the write phase, speculative data is made globally visible. Based on the validation algorithm, OCC can be further divided into two categories: backward and forward. Backward OCC tests the read set of the committing transaction with write sets of concurrent transactions that committed before the validation begins. Forward OCC, on the other hand, lets the validating transaction broadcast its write set to running transactions whose read sets are then tested with the broadcasted write set. Validation succeeds in both cases if all intersections are empty sets.

OCC provides better parallelism than 2PL, as illustrated by Fig. 2a in which a write does not commit until after an earlier transactional read commits, forming an uncomitted Write After Read (WAR) dependency. In fact, with OCC, uncommitted writes never incur conflicts as they are "invisible" to other transactions. Committed WAW conflicts are also harmless because OCC serializes writes to the same address with global coordination[1] (e.g. commit token in TCC, bus arbitrator in Bulk). However, committed WAR conflicts (where a transaction including a write to A commits before a transaction including an earlier read of A) will cause the reading transaction to abort during validation, as shown in Fig. 2b.

[1]In practice, synchronization occurs at the granularity of cache lines.

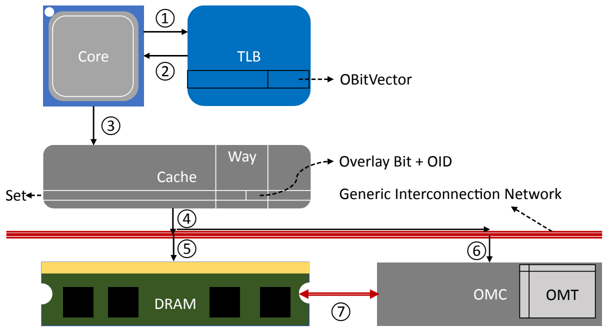

Fig. 3: **Page Overlays** – This diagram shows how memory system handles overlay. ① TLB lookup using VA; ② TLB outputs either translated PA or VA based on OBitVector; ③ Cache lookup using TLB output; ④ On cache miss (or eviction), include `(OID, VA)` in the fetch (eviction) request; ⑤ If address is PA (or not an overlay cache line), send request to the main memory; ⑥ If address is VA (or is an overlay cache line), send request to OMC; ⑦ OMC queries OMT using `(OID, VA)`, obtains PA, and fetches from memory.

Despite increased parallelism, naive forward and backward OCC have implementation issues that hinder their adoption. Forward OCC, as is the case with TCC and Bulk, requires a broadcasting medium to perform validation, which is expensive on modern multicore architecture. Backward OCC, on the other hand, may expose inconsistent states that will never be seen during serialized execution, causing undefined behavior [25]. As shown in Fig. 2c, *txn* X reads an old value of A but a new value of B, and then begins the loop. Although the validation algorithm can correctly identify this as a WAR conflict and prevents *txn* X from committing at the end, there is no guarantee that *txn* X may have a chance to validate, due to the infinite loop (assuming integers are sufficiently long) caused by reading the "partial writes" of *txn* Y. By contrast, in a serialized execution, no matter which of the two transactions execute first, the value of B is always smaller than the value of A, meaning that both can terminate within a finite number of cycles.

To further relax the semantics and reduce aborts, SI-TM [14] proposes Snapshot Isolation (SI) as the conflict detection algorithm. SI provides even higher parallelism than OCC, since it only checks for WAW conflicts and allows committed WAR conflicts, as shown in Fig. 2b. However, SI does *not* provide serializability, and this approach can introduce hard-to-reason-about anomalies, such as *write skew* (see Fig. 2d). No serial schedule can produce the final state in this example, since there is a dependency cycle between *txn* X and *txn* Y.

Several conflict detection algorithms have also been designed to track dependency information as conflicts are detected and to abort transactions only when absolutely necessary, i.e. when a conflict cycle is about to occur. These designs typically involve changing existing coherence protocols as in DATM [26] and HMTX [27], adding non-trivial tracking infrastructure as in OmniOrder [28], WnGTM [29], and EazyHTM [30], or timestamping the entire address space as in SONTM [31]. Such proposals are complicated, hard to verify, and often only provide limited speedup due to hardware resource constraints, lack of global information, and unrealistic assumptions.

III. Multiversioned Page Overlays

In this section, we discuss our Multiversioned Page Overlays (MPO) framework. We first provide a review of the basic Page Overlays [19] framework. We then discuss how we extend it to support multiversioning.

A. Overview of Page Overlays

As illustrated in Fig. 3, the Page Overlays paper [19] describes a fine-grained, general-purpose virtual memory technique intended for optimizing common memory management tasks such as copy-on-write [32], sparse data structures, deduplication, etc.

1) Overlay Address Mapping: In the Page Overlays design, every virtual memory page may be associated with two different backing stores: the usual physical memory frame and an alternative "overlay" frame, which may be sparsely populated and stored compactly in a reserved portion of main memory. Additionally, an **Overlay Bit Vector (OBitVector)**, indicating which cache lines should be fetched from the usual physical frame and which should be fetched from the overlay, is associated with each overlay frame and cached in the Translation Lookaside Buffer (TLB). The MMU checks the OBitVector in parallel with normal address lookup when an address translation is requested. If the corresponding bit is set, indicating an access to the overlay, normal address translation is aborted, and the MMU directly outputs the virtual address as the "overlay address". Otherwise, the MMU produces the usual physical address.

2) Overlay Cache Lookup: Using untranslated virtual addresses to access an unmodified cache has undefined result, as most modern L1 caches are Virtually Indexed and Physically Tagged (VIPT). To avoid false hits when virtual addresses are used, cache tags are extended with **Version Tags** consisting of two fields: A one bit **Overlay flag** to indicate whether the address tag is a virtual address, and a 15-bit **Overlay ID (OID)** which originally meant to be a process ID. On receiving a lookup request, the cache uses the address provided by MMU to perform a tag check. Given a tag match, a hit is signaled only in one of the following two cases: (i) Address is virtual, the Overlay bit is set, and OID from the instruction matches tag OID; (ii) Address is physical, and Overlay bit is clear.

3) Overlay Memory Controller: When a line is evicted from the LLC, the cache controller checks the Overlay bit in the cache tag. If the bit is clear, then a normal write back to the main memory is scheduled. Otherwise, the controller sends it to a special device, the **Overlay Memory Controller (OMC)**. Cache misses are handled similarly; the processor sends the cache line fill request to either main memory or the OMC based on whether the MMU outputs a normal physical line fetch or overlay line fetch.

The OMC is a memory-backed device connected to the inter-processor communication network. It maintains a separate virtual-to-physical mapping table, the **Overlay Memory Table (OMT)**. Compared with a page table, the OMT has two unique features. First, OMT mappings have cache line guanularity, thus enabling more compact memory management than paging. Second, the OMT maps the overlay address (the virtual address *augmented with* a corresponding OID) into a physical address, i.e. `(OID, VA)` $\rightarrow$ `PA`.

On receiving a request from the cache, the OMC queries the OMT using the virtual address and OID in the request. The cache line is then written back to (fetched from) the main memory using the physical address. The OS is responsible for allocating a chunk of memory for the OMC to use. Memory management within the chunk, however, is performed solely on hardware by the OMC for efficiency reasons. The OMC also maintains the OBitVectors. On a page fault, the MMU needs to access the OMC to fetch the relevant OBitVector.

Because the overlay system provides a seperate mapping for each OID-address tuple, we can repurpose this design to support *versions* of memory at a cache line granularity. We describe this multiversion scheme in the section that follows.

B. Extensions to Support Multiversioning

1) Version Instructions: **Versions** are timestamped, immutable overlays, logically ordered by the OIDs. To enable version access, we extend the ISA by adding four version instructions. We first describe the semantics of the four version instructions, and then present a practical implementation in Section III-B2. Version instructions take an implicit operand, the *operation timestamp (ots)*. This operand is supplied by a special register, *current ots*, which is part of the model-specific register file. This register can be loaded either manually, or by transaction begin instructions as we will see later in Section IV. The semantics of version instructions are as follows:

vload *addr* (Versioned Load): Load the most recent version (numerically greatest timestamp) with timestamp $\leq$ *ots*. If such version does not exist, load from the main memory.

vstore *addr, val* (Versioned Store): For first write, make a copy of a previous version by issuing *vload* with the *vstore*'s *ots*. Later writes are performed on the new (copied) cache line. Note that dirty versions remain speculative, and are invisible to other processors until version commit.

vcommit *ts* (Version Commit): Atomically commit all speculative versions in the cache, thus making them accessible to other processors. The OID of these versions are changed to *ts*, the instruction's explicit operand. Note that version commit does not force versions to be written back to the OMC.

vabort (Version Abort): Atomically discard all speculative versions whose OID $=$ *ots*.

For the sake of generality, MPO also provides *overlay load (oload)* and *overlay store (ostore)* to access a specified version. An exception is raised if the version does not exist. Implementation of *oload* and *ostore* in MPO is trivial. In the following sections, we focus on versioned operations.

One major difficulty of implementing these instructions is that committed versions can be scattered in multiple caches. For example, assume that processor #0, #1, #2 committed version 100, 104 and 102 respectively on the same address. Later on, version 100 was written back to the OMC due to an eviction. Now consider what if processor #3 issues versioned load with *ots* = 103. If processor #3 sends a versioned load request to the OMC, it can only read version 100, because the OMC is not aware of larger versions in the caches of processor #1 and #2. On the other hand, the correct result can be obtained, if processor #3 broadcasts the versioned load request to all other processors and the OMC. After receiving all responses, processor #3 performs a local sort, and can finally read the correct version, 102.

Sending broadcasts solves the correctness issue, but is expensive on large systems. To solve this problem, we leverage the observation that the challenge of selecting the right version to read is essentially a coherence problem. Instead of maintaining consistent content between different caches and the main memory as in classical cache coherence problems, which is unnecesary for OMC as versions are immutable, coherence needs to be maintained between caches and the OMC on the number of versions and their OIDs.

2) Version Directory: We propose a *version coherence* mechanism that tracks in-cache versions at a lower cost than broadcasting. We add a directory to the OMC, called a **Version Directory**. The version directory operates similarly to a cache coherence directory: one bit is reserved for each processor on every cache line sized memory block; If a bit is set, then the corresponding processor owns a version on the *virtual* address. Version instructions can be implemented efficiently with a version directory as follows:

vload *addr*: Perform overlay read. If cache misses, the processor sends a *versioned load* request to the OMC. On receiving the request, OMC reads the directory, and sends a *version query* request to processors that have a "1" bit. Processors check their own caches on receiving the version query from OMC. If one or more *committed* versions has OIDs $\leq$ *ots*, the OIDs are sent back to the OMC in a single packet. Otherwise the processor replies NACK. After receiving all responses, the OMC sorts all versions (including versions in OMC itself) by their OID. The version is then fetched by OMC using a *version fetch* message, and forwarded back to the requestor. The OID in requestor's cache tag is set to *ots*, to ensure later accesses will hit.

vstore *addr, val*: Perform overlay write. If cache misses, issue *vload* with *vstore*'s *ots* first, and then write to the local copy. The OID in cache tag is set to *ots*. A *Speculative* bit in cache tag is also set. Cache lines with *Speculative* bit set do not respond to version coherence messages. The directory bit is set for the requestor by the OMC on seeing the *versioned load* request.

vcommit *ts*: For cache lines with *Speculative* bit set and OID $=$ *ots*, change the OID to *ts*, and flash-clear the bit. This process is local – no message is sent.

vabort: Assemble an *abort request* which contains the addresses of all cache lines that have *Speculative* bit set, and send the request to OMC. On receiving the abort request, OMC clears the directory bit for addresses in the request. Note that if multiple versions on the same address exist in the cache, the address signature must only be set for the address if all versions are speculative. Otherwise the directory bits must

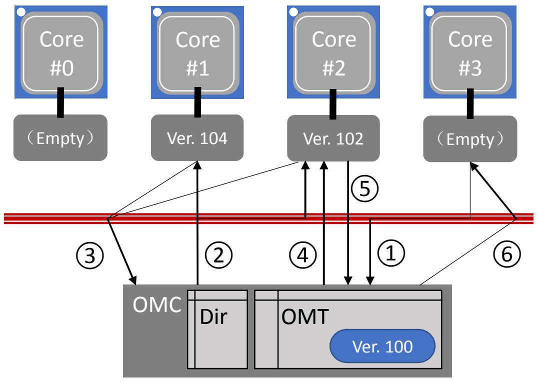

Fig. 4: **Sequence of Messages on Versioned Store by Processor #3** – In the normal access path using the directory, six messages and three steps are needed to resolve a *vstore* miss.

remain unchanged. The processor also flash-invalidates all cache lines with *Speculative* bit set.

3) Version Directory Example: To better understand how the version directory works, we now present an example, illustrated in Fig. 4. We assume that processor #1 and #2 have committed version 104 and 102 respectively. Processor #0 has committed version 100, and then evicts the version to the OMC. All versions are on address A.

Now processor #3 issues a versioned store operation with *ots* = 103 on address A. First, processor #3 needs to search its own cache for an overlay whose OID = 103. In our case this will be a cache miss, as processor #3 has never read or committed version 103. On a cache miss, processor #3 issues a *versioned load* request to the OMC, which contains the operation's *ots* (①). On receiving the *versioned load* request, the OMC reads the directory entry for address A, and finds out that the bits for processor #1 and #2 are set. In the meantime, the OMC also searches the OMT, and finds out that version 100 is in the main memory. The OMC then sends a *version query* request to processor #1 and #2 (②). On receiving the *version query*, both processors search their private cache for versions whose OID $\leq$ the operation's *ots*, 103. In our case, processor #2 replies 102, and processor #1 replies NACK (③) since 104 is greater than 103. After receiving the reply messages, the OMC sorts all version OIDs. There are only two versions: Version 100 in the main memory, and version 102 from processor #2. The OMC then selects the largest version, which is 102. Since version 102 is from processor #2, the OMC sends a version fetching request to processor #2 (④), and the latter replies with cache line data (⑤). Finally, the OMC forwards the cache line to processor #3 (⑥). Processor #3 allocates a cache line entry for version 102, and changes the OID of the line to 103. The speculative bit of the line is also set. The next time a versioned load or store operation with *ots* = 103 is issued, the line that the processor just read will be hit. After processing a versioned store operation, the OMC also sets the directory bit for the processor that issues the write.

4) Decoupled Metadata and Data Writeback: In the naive directory-based design, each cache miss triggered by versioned operation will take six steps on the network to resolve in the

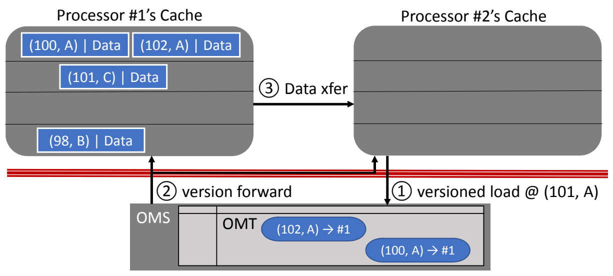

Fig. 5: **Fast Access Path with Decoupled Matedata Writeback** – Accessing a frequently accessed address only takes three steps and four messages

worst case: Two between the requestor and OMC, two for *version query*, and another two for *version fetch* (see Fig. 4). By contrast, a normal cache miss can be resolved in only three steps: one for the initial *GETS/GETX* request, and another two for *invalidation* and *ACK* (on a store miss).

To close the gap, we propose *decoupled metadata writeback* for frequently accessed addresses, which leverages point-to-point query as much as possible to avoid extra rounds of message exchange. This technique works as follows. When the OMC receives a *versioned load* request generated by a *vstore* miss from processor P on address VA, it immediately inserts `(ots, VA)` $\rightarrow$ `P` into the mapping without data (recall that *ots* is in the request), and does not set the directory bit. Multiple entries may be created this way for the same VA but different ots. The next time OMC receives a *versioned load* request for VA, since the direcory bit is clear, no *version query* message is sent to any of the processors. Instead, the OMC will perform an OMT lookup, sorts all entries on the requested address by their version, and locates the correct version to read using the version read rule. If the selected version in the mapping is of the form `(VER, VA)` $\rightarrow$ `P`, OMC will notify processor P to forward version VER to the requesting processor, by sending P a *version forward* message. On receiving this message, processor P will initiate a cache-to-cache transfer of the specified version to the requesting processor, taking advantage of the high-bandwidth inter-processor link.

With decoupled metadata writeback, fulfilling a cache miss on frequently accessed addresses only takes three steps: One for the initial *versioned load* request, one to notify the version owner with *version forward* message, and one cache-to-cache transfer carrying the requested data (in parallel with the reply message to the requestor). In total, only four messages are sent over the network, which is also comparable to a normal cache miss (see Fig. 5).

On eviction of committed versions, both the metadata and data are written back to the OMC. The cache controller checks whether the line is the only locally cached version of the address. If no other version is found, the controller sets a "clear directory" bit in the eviction message, indicating that the version directory should clear the bit for the processor. On receiving the eviction message from processor P, the OMC will insert an entry `(OID, VA)` $\rightarrow$ `DATA` to the mapping, in which version data is also stored. We postpone discussion of uncommitted version eviction to Section IV-E under the context of OverlayTM.

5) Discussion: In this section we discuss implementation issues with MPO and version directory. We show that both can be implemented rather efficiently on modern hardware.

Directory Overhead: Given the locality of computation, the version directory only needs to quickly access bit vectors for a small subset of addresses at any given moment. The directory can therefore be implemented as a sparse hash table with a cache hierarchy for accelerating accesses to recently used entries.

Scalability: In larger systems, the version directory can be partitioned into *slices*, each responsible for an address range. A hash function needs to be applied to the address to generate the target slice address before a version request is sent.

Verification Cost: No MESI state is maintained for versions, since they are immutable. Moreover, the MPO design only adds incremental changes; existing cache coherence and eviction policy are not modified.

OMT Bandwidth: To avoid overloading the OMT by inserting an entry for every *versioned load* request generated by *vstore* misses, we propose adding a small cache to the OMT, such that the mapping `(ots, VA)` $\rightarrow$ `P` is only inserted into the cache. When an entry is evicted from the OMT cache, the OMC sets the directory bit for processor P on address VA, and then just discards the entry. Frequently accessed addresses will remain in the cache and therefore have lower protocol latency, while the rest use the version directory.

6) Comparison with Existing Multiversioning Designs: We compare MPO with two previous transactional multiversioning designs: HICAMP [33] and SI-TM [14]. HICAMP also features hardware supported immutable versions. Objects in the main memory are organized into segments, a B+Tree-like structure that enables fast content-based lookup. HICAMP natively supports Snapshot Isolation transactions. It involves, however, radical hardware and software redesign, and changes the programming paradigm entirely. On the contrary, MPO is just an extension of existing virtual memory system, and can be easily disabled by the Operating System. More importantly, programs do not suffer any performance overhead from not using MPO.

SI-TM embeds a Multiversion Manager (MVM) in the LLC controller, which intercepts line misses and evictions from upper level caches, and rewrites the physical address in the request with a translated version address. Upper level caches, as opposed to our design, are unaware of multiversioning, and only store the address tag. Since versions are not *self-contained* as in MPO, on SMT switch and transaction commit, upper level caches must evict all dirty lines in the private cache. This operation is on the critical path, because future version accesses with different timestamps may hit the wrong version, due to the fact that different versions on the same address have the same tag. MPO, on the other hand, tags every line in the private cache with OID, and uses a customized coherence protocol to enforce the version read rule. Version commit is therefore instantaneous, as suggested in Table I.

IV. OverlayTM: MPO + Conflict Detection

Transactional memory designs must address two challenges: Version management and conflict detection. Multiversioned Page Overlays (MPO) provides a solution to HTM version management using version instructions such as *vload* and *vstore*, as we have seen in Section III. In this section, we fianlize OverlayTM by introducing its conflict detection hardware: the commit queue.

A. Commit Queue

The **Commit Queue (CommitQ)** is a hardware structure in the OMC that buffers write sets of committed transactions. We do not discuss the implementation of RW sets in detail. We assume they are fixed-length bloom filters (our experiments show that bloom filters of 2KB with a good hash function achieves almost perfect conflict detection in most cases, which is consistent with previous work [20], [31]). The cache controller may compress the RW set before sending them in order to optimize bandwidth [21]. Set intersections can be computed efficiently on hardware using bitwise AND, with the possibility of false positives. Processors maintain per-transaction read and write sets (RW sets). These local RW sets are updated accordingly as processors issue *vload* and *vstore* instructions, and cleared when new transaction begins. Note that *partially* written lines should be added to *both* the read and the write set.

The CommitQ accepts two inputs: an RW set bloom filter and a validation timestamp. Each entry of the CommitQ also has an *entry timestamp*, with a comparator that outputs “1” if the validation timestamp is less than the entry timestamp. There is also a RW set comparator on every CommitQ entry which allows quick intersection tests against the input RW set. The RW set comparator outputs “1” if two bloom filters have non-empty intersections. The results of the tests are first AND’ed with the output of the timestamp comparator respectively, and then OR’ed together as the final output signal. A logical “1” indicates validation failure. A block diagram of the CommitQ is depicted in Fig. 6.

B. Transaction Begin

In order to serialize transactions on a globally agreed order, the OMC maintains a *global timestamp (gts)* counter, the width of which is identical to the OID in cache tags (i.e. 15 bits in most cases). At transaction begin, the processor acquires a *begin timestamp (bt)* by sending a *begin request* to the OMC. The OMC fetch-increments *gts*, and replies with the new value, which is also an unique identifier of the transaction. The processor then loads the *current ots* register (see Section III-B1) such that *vload* and *vstore* use *bt* as the implicit operand, essentially accessing a snapshot at logical time *bt*.

The OMC also maintains a list that tracks the *bt* of uncommitted transactions. The newly allocated *bt* is inserted into this list when a new transaction begins, removed when it is committed or aborted. In later sections, we will see that this list plays an important role in handling overflowed versions and performing garbage collection.

C. Transaction Commit

At transaction commit, the processor sends a *commit request* message to the OMC. This message includes the read set, write set, and the *bt* of the transaction. The OMC validates the transaction by dispatching the *read* set and the *bt* to the CommitQ. If the CommitQ indicates a validation success, the OMC fetch-increments *gts*, and then sends a *committed* message back to the requestor. The *committed* message contains the new value of *gts* as the transaction’s *commit timestamp (ct)*.

On receiving the *committed* message, the processor injects a *vcommit* into the pipeline. The explicit operand of *vcommit* is the *ct* in the *committed* message, meaning that all speculative versions created by the transaction will become visible to transactions started after logical time *ct*. In the meantime, the CommitQ allocates an entry for the newly committed transaction, and stores both the *ct* and the *write* set of the transaction into the new entry.

If, on ther other hand, the CommitQ indicates a validation failure, the OMC will send an *aborted* message to the requestor. On receiving the message, the processor performs version abort, which invalidates all speculative versions. Directory bits for speculative versions are cleared by the OMC in the background. This does not affect correctness, but only incurs some extra traffic for a short time.

OverlayTM’s commit protocol only validates one transaction at a time, serializing concurrent commit requests. Transaction begin and version requests that are not on committed addresses, however, are unaffected as long as the network guarantees ordered delivery between *committed* and *version query* messages. In Section VII-B we show that the serialization penalty is minimal for most workloads, and hence does not constitute a bottleneck in most cases.

D. Garbage Collection

CommitQ entries are garbage collected (GC’ed) if no uncommitted transaction can be aborted by the entry. The CommitQ maintains the smallest uncommitted *bt* as a low-water mark (recall that the OMC maintains a list of uncommitted transactions), and removes an entry if its *entry timestamp* is smaller than the low-water mark.

Versions are also deleted when they are no longer accessible by active transactions. A version becomes inaccessible when there exists a larger version on the same address, and no uncommitted *bt* exists in-between. Due to the complexity of this task, we propose using a software handler for version cleanup when the OMS runs out of space.

E. Supporting Overflowed Transactions

In OverlayTM, a transaction overflows if one or more uncommitted versions are evicted from the last-level cache (LLC). The OMC will insert this speculative version into the OMT as described in Section III-B4 as if it were a committed version. To ensure proper isolation, the OMC will check the list of uncommitted *bt* when serving a *versioned load* request, and exclude this speculative version from the response. Unlike some other TM designs, there is no time consuming “version

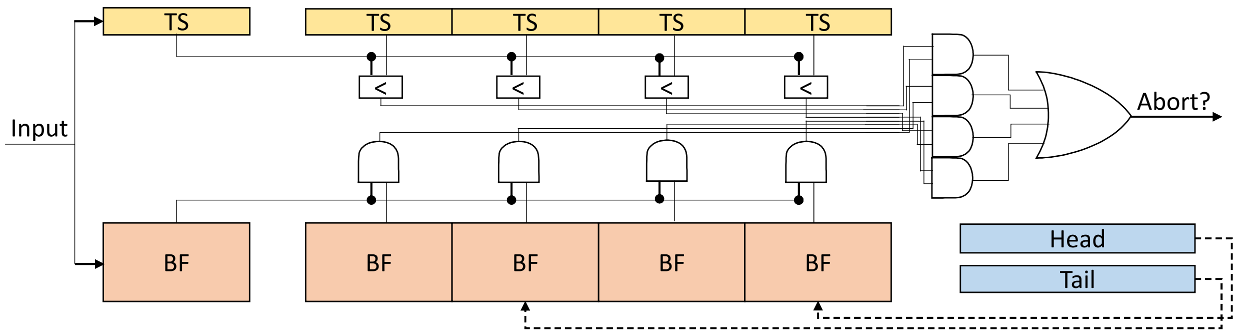

Fig. 6: **Commit Queue** – Implements Backward OCC validation algorithm; The committing write set is compared with read sets of transactions committed after it starts.

walk" on transaction commit, and therefore, transaction commit is always a fast operation.

Compared with SI-TM [14], VTM [8], etc., OverlayTM's overflow handling requires no dedicated logging hardware for spilling speculative data to a private log. This should be attributed to the fact that versions in OverlayTM are self-contained, and that every transaction has its own unique identifier. Overflowed transactions also do not complicate the conflict detection protocol, because OverlayTM decouples conflict detection from version management by using bloom filters. Compared with TCC [17] and EazyHTM [30], overflowed transactions in OverlayTM does not block concurrent transactions by entering "invincible state", which scales better.

F. Timestamp Wrap-Around

On platforms with 48 bit physical address [34], Page Overlays can support at most 15 bits OID [19]. In a saturated system, we expect OIDs to wrap-around quite often, because most transactions will use two OIDs, one for begin and another for commit. In this section we propose a solution for timestamp wrap-around using *phase variable*.

To detect global timestamp (*gts*) wrap-around, OMC maintains one additional **Phase Variable** counter, initialized to zero at startup time. The phase variable is wide enough (e.g. 64 bits) such that in practice it never overflows. Every time *gts* overflows, the OMC increments the phase variable. On receiving a *begin request*, the OMC also includes the phase variable value in the reply message. On receiving the reply, processors save the phase variable in *begin phase* model-specific register, which is part of the context. On every *version request* and on commit, the processor piggybacks the value of *begin phase* register in the *commit request* message. The OMC checks whether the *begin phase* is identical to the current phase variable. If not, the OMC instantly replies *aborted*, because the transaction was started in a stale phase, and timestamps have already wrapped around since then.

G. Optimizations

In this section, we present two optimizations that leverage OverlayTM's ability to read from a consistent snapshot.

Read-Only Commit: A transaction can always commit successfully if it does not write to shared states (the detection of which can be implemented on hardware or compilers). Serializability is still achievable, with the possibility that the read-only transaction may not see the most up-to-date data. Section VII-A1 shows that the performance improvement can be significant for certain read-dominant workloads.

Early Release: OverlayTM implements early release [35] by simply not adding certain reads into the read set. MPO's snapshot read semantics guarantee that the read image is consistent, while not adding them to the read set prevents the transaction from being aborted by committed WAR conflicts. For some applications, early release provides better concurrency without sacrificing correctness. In Section VII-A1, we demonstrate that this feature can reduce OverlayTM aborts on STAMP `labyrinth` by almost $5\times$.

H. Scaling to Large Systems

The centralized conflict detection protocol introduced in Section IV-C may not scale well on future large systems due to CommitQ contention. In this section we present several techniques that enable higher parallelism within the protocol for better scalability.

Parallel Validation: Instead of only serving one commit request at a time, the CommitQ hardware may implement k copies of validation logic (i.e. AND gates and integer comparator in Fig. 6). During validation, k waiting entries in the receiving buffer are selected and then validated against CommitQ entries. In addition, the read sets of k validating transactions are checked with each other's write sets. The CommitQ must guarantee that if transaction X is assigned a smaller *ct* than Y, the write set of X must have empty intersection with the read set of Y. Since multiple commit requests are validated in parallel, the commit latency can be reduced by at most k times, improving overall throughput.

Eager Abort: As described in Section III-B2, on receiving *version query* messages, processors check their own cache for committed versions whose OID $\leq$ request *ots*. We slightly extend the protocol as follows: If the processor finds a committed version on the requested address with OID $>$ *ots*, it indicates in the reply message that a larger committed version exists. The OMC will immediately abort the transaction by sending an *aborted* message back to the requestor, since a committed WAR violation has been detected. This alleviates contention on the OMC if aborts are frequent.

Hierarchical Conflict Detection: In large systems where the address space is partitioned between several nodes (e.g. NUMA), OverlayTM hardware components can also be partitioned, such that every node has an OMC and CommitQ handling requests only within that node. While intra-node conflicts are resolved lazily, inter-node conflicts cannot be detected because they are validated by distinct CommitQ. We propose resolving these conflicts eagerly using version coherence. Recall that during a cross-node version access, the *version read* request is sent to the OMC on the remote node, which will then be forwarded to processors on that node. On receiving such a cross-node version request, processors owning a conflicting uncommitted version on the requested address will abort immediately. Note that, in this scheme, one additional bit per processor is used by the directory to mark speculative reads, and inter-node version requests generated by *vstore*s are forwarded to *both* speculative readers and writers. The *gts* counter is maintained cooperatively by multiple OMC devices using regular coherence (i.e. only one OMC may have permission to increment *gts* at a time).

To see why hierarchical conflict detection is correct, let us assume there is a non-serializable execution where transaction X commits on transaction Y's read set. It must be that Y reads address A before X commits. There are two possibilities: either X executes *vstore* A before Y's *vload* A or the opposite. In both cases, the version request protocol can recognize that there is an uncommitted versioned load or store on another node, and one of them will abort, a contradiction!

V. Hardware Cost Summary

Because our design is built on top of *page overlays* [19], it includes the same three sources of hardware overhead: (i) the OMT Cache, (ii) wider TLB entries (to store the `OBitVector`), and (iii) wider cache tags (due to the wider physical address space). (The hardware overhead of this baseline page overlay design is discussed in detail in [19].)

We have three additional sources of hardware overhead: (i) our extensions to page overlays to support *multiversioning* (discussed in Section III-B), (ii) our hardware *commit queue* (discussed in Section IV-A), and (iii) additional registers in each processor. While the last two are specific to supporting OverlayTM, hardware for multiversioning support is generally useful for applications beyond transactional memory.

Our extension adds additional control logic into OMC to support version coherence. Similar to cache coherence, the control logic can be implemented as a state machine. Since versions are immutable, we expect the state machine to be simpler than the one used for cache coherence. In the commit queue, the most significant portion of hardware is the array of bloom filters. As shown by [20], bloom filters of size 2KB can achieve almost perfect conflict detection. Assume that the number of entries in the commit queue is equal to the number of processors. For a 16-core system, the commit queue will need 32KB (16 * 2KB) storage in total. Each processor maintains a read set, a write set, and registers to store the *bt*, *ct*, *gts*, and *RO* bit. This adds an extra 4KB plus a few bytes for each processor.

TABLE II: Simulation parameters

CPU cores	16 cores 4-way superscalar @ 3GHz
CPU L1-D/I caches	32KB, 64B lines, 8-way, 4 cycles
CPU L2 cache	256KB, 64B lines, 8-way, 8 cycles
CPU L3 cache	32MB, 64B lines, 16-way, 30 cycles
Memory Controllers	4
DRAM Latency	100 cycles
zSim Phase Length	200 cycles

VI. Experimental Framework

A. Simulation Platform

We extended zSim [36] to simulate OverlayTM. We chose zSim not only for its simulation speed, but also because its *execution-driven* approach is important for correctly modeling execution paths when transactions need to retry. Our simulation parameters are shown in Table II.

The execution binaries that we simulate are compiled using the Intel TSX Restricted TM (RTM) interface [34], [37]. We instrument three RTM instructions:

- **XBEGIN**: The processor enters speculation mode, after taking a snapshot of the current context. This instruction also takes the address of the abort handler. On a transaction abort, the control flow will transfer to the abort handler, after the context is restored.
- **XEND**: Commits the current transaction.
- **XABORT**: Aborts the current transaction. This instruction carries a user-defined return code that the abort handler can access. The abort code is put into `EAX`, together with several hardware set status bits.

B. Simulated HTM Designs

We compare OverlayTM against 2PL (requestor-wins) and SI-TM to illustrate the performance differences between each of the underlying concurrency control algorithms. Both 2PL and SI-TM are scalable and use simple hardware extension, which is similar to OverlayTM. We also simulate an idealized version of TCC, for which we assume *instant commit* (no bus arbitration) and *unbounded support* (no serialized commit when transactions overflow). This gives a lower bound of TCC on realistic hardware. We also make the following assumptions to ensure fairness of comparison. First, all HTMs detect conflicts at aligned 8-byte word granularities (to help avoid false sharing conflicts). Second, we assume that the hardware can overlap long operations during speculation, e.g. cache tag walk before commit, because this process is highly implementation-dependent (in practice this assumption works well, as shown in [31]).

C. Benchmarks

Our simulation runs STAMP benchmark [16] with recommended parameters. Moreover, to further evaluate the feasibility of OverlayTM on a broad range of workloads, three data structures are used:

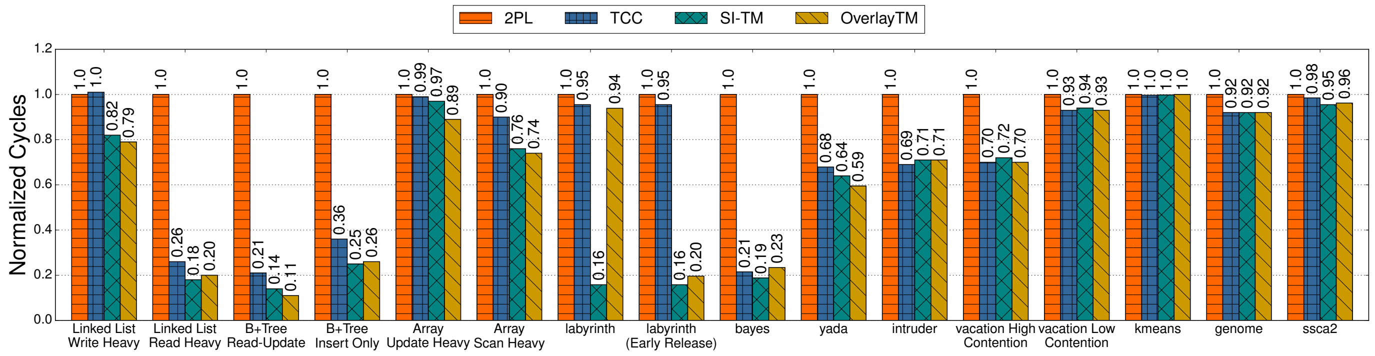
Fig. 7: **Normalized Cycles** – All numbers are normalized to 2PL.

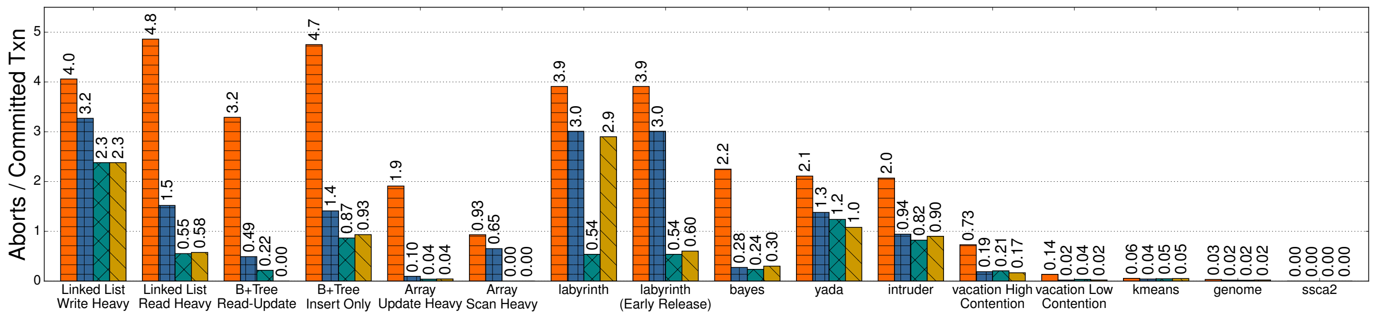
Fig. 8: **Aborts Per Committed Transaction**

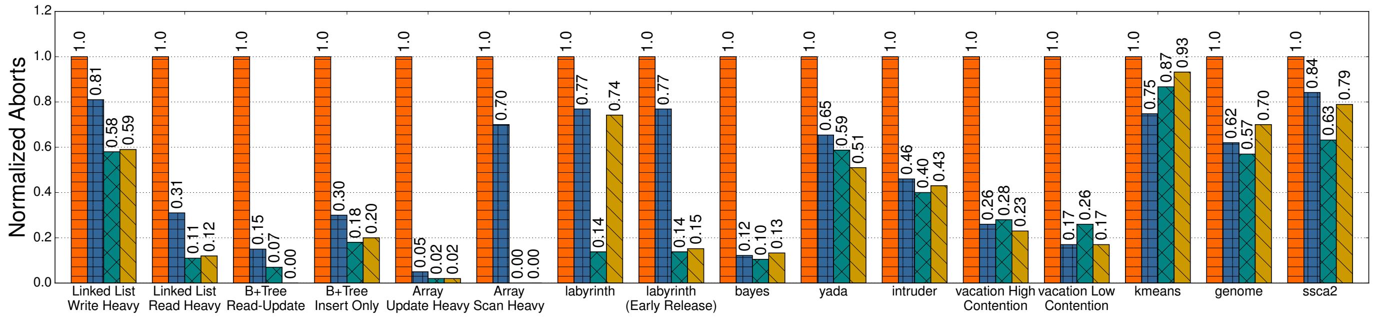
Fig. 9: **Normalized Aborts** – All numbers are normalized to 2PL.

- **Linked List**: A singly linked list. Each node has an 8 byte key and 8 byte payload field. Threads first generate a random number k between 0 and the current length minus one, and then traverse k nodes before they finally insert, delete or read the node after the current node.
- **B+Tree**: A standard B+Tree with 4KB nodes. Both key and payload are 8 bytes. Threads insert, update or read on certain "hot spots".
- **Array**: An array of integers with $(m+n)$ worker threads. m threads perform linear scan on the array, n threads write randomly chosen array entries.

All random numbers are drawn from the `rand` function in standard C library. We repeat each test case five times using the same random seed, and then take the median.

All workloads adopt the lock elision algorithm recommended by [37]. Critical sections are backed by a single state-of-the-art spin lock. Transactions elide the lock by reading the lock variable once started. On transaction abort, the abort status code is checked. The transaction will restart if: (i) the abort is caused by transient conditions such as conflicts; and (ii) the retry counter has not reached zero. Otherwise, the fall-back path will be executed, and the lock is physically acquired. In this case, speculative transactions are blocked by the lock before they can start.

VII. Experimental Evaluation

A. Performance Analysis

We run each of the HTM with STAMP and data structure workloads and present simulation results in Fig. 7 (normalized cycles), Fig. 8 (aborts per committed transaction) and Fig. 9 (normalized aborts). We discuss these numbers in the following.

1) Overall: In 11 out of the 16 workloads (all except array update-heavy, `vacation` low, `genome`, `kmeans`, `ssca2`), OverlayTM outperforms 2PL by more than 20%, which demonstrates a great advantage of lazy conflict detection over eager 2PL. Furthermore, in 8 out of the 16 workloads, OverlayTM outperforms TCC with both less cycles and lower abort rates, which shows the effectiveness of multiversioning.

Most interestingly, OverlayTM, with its ability to commit read-only transactions regardless of concurrent writes, is comparable in most cases with SI-TM (23% more aborts and 11% more cycles, except `labyrinth`). We consider this as the biggest merit of our design: Achieving similar or better performance than Snapshot Isolation with strong semantics.

2) Linked List: The initial list has 256 nodes. Worker threads eiher insert, delete, or read a node as described in Section VI-C. Write-heavy workload has 20% read, 40% insert, and 40% delete; Read-heavy workload has 80% read, 10% insert, and 10% delete.

The linked list workload models a broad range of commonly used data structures including sorted list, chaining hash table, etc. Our configuration features high contention and high read-write ratio, since threads read a "prefix" of all nodes in the list before they finally stop. This behavior makes them vulnerable to WAR conflicts incurred by any updating transaction on the prefix.

In Fig. 9, we can see that both eager conflict detection (2PL) and forward OCC (TCC) suffer from high abort rates as well as more wasted cycles. As explained by Section II, 2PL and TCC are sensitive to committed WAR conflicts on the prefix, while multiversioning HTM, i.e. SI-TM and OverlayTM, commits read-only transactions using the snapshot they take at transaction begin. In addition, 2PL has more aborts than TCC, due to the fact that 2PL also aborts on uncommitted conflicts. Both OverlayTM and SI-TM achieves a 20% speedup with 40% less aborts.

3) B+Tree: In the first insert-only stage, 16 worker threads insert 16384 keys into the tree. Then in the second stage, threads either update or query the tree. The first phase models the B+Tree index of an Online Transactional Processing (OLTP) table where new entries are concurrently created by assigning monotonically increasing entry IDs. The second phase resembles YCSB-A [38] read-update: 50% reads and 50% updates are performed on a "hotspot" that is gradually drifting in the key space. Compared with linked list, the B+Tree workload has fewer contention and lower read-write ratio, since conflict almost only happens on leaf levels due to the fact that node split is rare.

For insert-only, all other three HTMs outperform 2PL by 60% – 75%, with 70% – 80% less aborts. This is because 2PL transactions expose writes eagerly, which is detrimental: On average, half of the node data is moved around when inserting into a B+Tree leaf node, causing uncommitted write-read and write-write conflicts. In addition, SI-TM has the best performance, because SI-TM commits transactions even for committed WAR. For read-update, OverlayTM runs 20% – 90% faster than all other HTMS with negligible aborts, while SI-TM suffers from concurrent WAW on the same item and TCC suffers from committed WAR. The reason OverlayTM does not abort in the case of WAW is that we implicitly adopt the assumption that conflict detection is on word granularity. Blind writes (writes to memory addresses without reading first) can therefore be optimized by not adding them to the read set. If we take out this assumption, then OverlayTM performance will be slightly worse than SI-TM, but still much better than TCC and 2PL (not shown).

4) Array: The array workload models an Online Analytical Processing (OLAP) table, where worker threads update the table at the front end, and background threads run real-time auditing operations that scan the entire table. This workload features long read-only sequences and short updates. As expected, OverlayTM and SI-TM handle this case extremely well due to multiversioning: Performance improves by 10% – 25% with negligible aborts. Although cycle improvement is not as significant as abort rates, we argue that, in this scenario, latency (i.e. number of aborts for auditing transactions) is more critical than aggregated cycles because the background auditing threads might be used to support real-time decision making systems where the timeliness of data is the uttermost.

5) STAMP: OverlayTM, SI-TM and TCC improve performance by approximately 30% on `vacation` high and `intruder` out of the four STAMP workloads, with 57% – 77% less aborts. For `bayes` and `yada`, the performance is improved by 80% and 60% respectively, with 90% and 49% – 75% less aborts. On `genome`, `kmeans`, `ssca2` and `vacation` low, performance improvement is very limited (less than 10%) for all three HTMs. This can be explained by the fact that the absolute abort rate is already low in our implementation of lock elision: Only 3.3% of total 14749 transactions suffer aborts for 2PL in `genome`. Optimizing aborts is meaningless in this case because only a small fraction of execution cycles are wasted.

`labyrinth` implements the transactional version of Lee's algorithm [39], [40]. The algorithm runs BFS between a point pair on a maze after copying the maze to the local storage, and then attempts to establish a connecting path between the two points using backtracking. Without early release (Section IV-G), transactions are forced to serialize, because any writing transaction to the maze will cause committed WAR conflict with every concurrent transaction. As shown in the figure, only SI-TM maintains low abort rate and low cycle wastage, as it commits transactions despite committed WAR. With early release enabled on OverlayTM, however, only transactions that truly conflict (i.e. two transactions select the same grid during backtracking) will abort, ignoring committed WAR on irrelevent grids since they are not part of the read set. The overall performance is comparable to that of SI-TM, a significant improvement.

We also noticed that these numbers seem to deviate from what has been published [14]. Our explanation is that our lock elision algorithm upper bounds the number of retries a transaction may attempt using a retry counter. Once this upper limit is reached, the transaction will grab the global lock and execute non-transactionally. This simple technique avoids one or a few long transactions repeatedly aborting all other transactions and each other in eager systems (e.g. 2PL), and hence reduces aborts significantly [41].

TABLE III: **Commit Queue Overhead** – The first column lists the average number of pending requests in the buffer when a processor requests to commit. The second column shows the percentage of total execution cycles spent on waiting for OMC to process commit requests. We only model a single-issue, centralized commit queue.

Workload	**Avg. # Pending**	**% Cycles**
Linked List Write-Heavy	1.65	2.40
Linked List Read-Heavy	0.79	3.33
B+Tree Write-Heavy	1.42	6.39
B+Tree Insert-Only	5.00	5.73
Array Update-Heavy	0.12	0.012
Array Scan-Heavy	0.39	0.43
`genome`	3.62	4.54
`vacation` High Contention	4.23	0.63
`vacation` Low Contention	4.62	1.10
`intruder`	3.93	3.70
`bayes`	1.66	0.059
`kmeans`	2.46	5.75
`labyrinth`	1.90	0 (negligible)
`labyrinth (Early Release)`	1.67	0 (negligible)
`ssca2`	3.49	26
`yada`	2.64	0.091

B. Commit Queue Overhead

Recall from Section IV-A that when the OMC receives a commit request, all *gts* operations including transaction begin and commit will be blocked. Processors that have a pending request must stall to wait for the OMC to complete the commit sequence. In this section we evaluate the overhead of serialized commits.

We measure the serialization of transaction commit by modeling the FIFO buffer in front of the commit queue. Pending requests are inserted into the buffer in the order they are received by the OMC. We assume an average overhead of 20 cycles for processing one request. The simulator notes down the relative positions of requests in the buffer when they arrive (zero means the request is processed immediately). The relative position is treated as an approximation of the level of contention. We analyze the overhead of serialization using these numbers, and summarize the result in Table III.

Our analysis shows that, for most workloads that we use, the commit queue does not constitute a bottleneck. 12 out of 16 workloads spend less than 5% of total execution time waiting for pending requests to complete. For B+Tree write-heavy, B+Tree insert-only and `kmeans`, the percentage of wasted cycles is higher, but are still less than 7%. `ssca2` represents one extreme case where the commit queue overhead constitutes 26% of total execution cycles. After inspecting the statistics, we found out that the high commit overhead of `ssca2` is caused by unusually short transactions, the average size of which is around 20 cycles. In this case, the commit queue has become major bottleneck of the system's transaction throughput in our evaluation. We agree that a centralized commit queue is not capable of handling extremely short transactions very well, and would like to leave this to future work.

Table III also shows that the average number of pending requests that a processor will have to wait for is usually below five. This means that, in average, there are five other transactions waiting for validation when another validation request is received by the commit queue. This shows a great potential for parallel validation described in Section IV-H.

VIII. Conclusions

In this paper, we have presented a new framework for supporting *multiversioning* in modern memory systems (**Multiversioned Program Overlays** (**MPO**)) and a new *hardware transactional memory* design (**OverlayTM**) built on top of that framework. By cleanly separating the hardware necessary for general-purpose multiversioning (in MPO) from the additional hardware necessary for HTM (in OverlayTM), we believe that MPO is attractive in its own right because it enables N-way versions of the many use cases described in the original page overlays paper [19] (i.e. overlay-on-write, sparse data structures, fine-grained deduplication, checkpointing, fine-grained metadata management and flexible super-pages), among other things. MPO enables multiversioning at a cache line granularity without significantly altering existing virtual memory frameworks or introducing high overheads.

OverlayTM builds on top of MPO using a hardware *commit queue* to implement commit-time ordering with backward optimistic concurrency control to support unbounded transactions where read-only transactions are guaranteed to successfully commit. In contrast with SI-TM (another HTM design that uses multiversioning), OverlayTM achieves similar or better performance while scaling to multi-socket systems and providing *full serializability* By significantly reducing abort rates compared with other state-of-the-art HTM designs in a number of cases, OverlayTM successfully leverages multiversioning to improve both performance and functionality for transactional memory programmers. Given these results, we believe that OverlayTM (even on its own) presents a strong argument for including MPO support in future systems.

References

[1] M. Herlihy and J. E. B. Moss, *Transactional memory: Architectural support for lock-free data structures*. ACM, 1993, vol. 21, no. 2.

[2] P. Hammarlund, A. J. Martinez, A. A. Bajwa, D. L. Hill, E. Hallnor, H. Jiang, M. Dixon, M. Derr, M. Hunsaker, R. Kumar *et al.*, "Haswell: The fourth-generation intel core processor," *IEEE Micro*, vol. 34, no. 2, pp. 6–20, 2014.

[3] D. Christie, J.-W. Chung, S. Diestelhorst, M. Hohmuth, M. Pohlack, C. Fetzer, M. Nowack, T. Riegel, P. Felber, P. Marlier *et al.*, "Evaluation of amd's advanced synchronization facility within a complete transactional memory stack," in *Proceedings of the 5th European conference on Computer systems*. ACM, 2010, pp. 27–40.

[4] D. Dice, Y. Lev, M. Moir, and D. Nussbaum, "Early experience with a commercial hardware transactional memory implementation," in *Proceedings of the 14th International Conference on Architectural Support for Programming Languages and Operating Systems*, ser. ASPLOS XIV. New York, NY, USA: ACM, 2009, pp. 157–168. [Online]. Available: http://doi.acm.org/10.1145/1508244.1508263

[5] C. Click, "Azul's experiences with hardware transactional memory," in *HP Labs-Bay Area Workshop on Transactional Memory*, vol. 89, 2009.

[6] A. Wang, M. Gaudet, P. Wu, J. N. Amaral, M. Ohmacht, C. Barton, R. Silvera, and M. Michael, "Evaluation of blue gene/q hardware support for transactional memories," in *Proceedings of the 21st international conference on Parallel architectures and compilation techniques*. ACM, 2012, pp. 127–136.

[7] K. E. Moore, J. Bobba, M. J. Moravan, M. D. Hill, D. A. Wood *et al.*, "Logtm: log-based transactional memory." in *HPCA*, vol. 6, 2006, pp. 254–265.

[8] R. Rajwar, M. Herlihy, and K. Lai, "Virtualizing transactional memory," in *Proceedings of the 32Nd Annual International Symposium on Computer Architecture*, ser. ISCA '05. Washington, DC, USA: IEEE Computer Society, 2005, pp. 494–505. [Online]. Available: https://doi.org/10.1109/ISCA.2005.54

[9] W. Chuang, S. Narayanasamy, G. Venkatesh, J. Sampson, M. Van Biesbrouck, G. Pokam, B. Calder, and O. Colavin, "Unbounded page-based transactional memory," in *Proceedings of the 12th International Conference on Architectural Support for Programming Languages and Operating Systems*, ser. ASPLOS XII. New York, NY, USA: ACM, 2006, pp. 347–358. [Online]. Available: http://doi.acm.org/10.1145/1168857.1168901

[10] C. S. Ananian, K. Asanovic, B. C. Kuszmaul, C. E. Leiserson, and S. Lie, "Unbounded transactional memory," in *High-Performance Computer Architecture, 2005. HPCA-11. 11th International Symposium on*. IEEE, 2005, pp. 316–327.

[11] H. Chafi, J. Casper, B. D. Carlstrom, A. McDonald, C. C. Minh, W. Baek, C. Kozyrakis, and K. Olukotun, "A scalable, non-blocking approach to transactional memory," in *High Performance Computer Architecture, 2007. HPCA 2007. IEEE 13th International Symposium on*. IEEE, 2007, pp. 97–108.

[12] L. Ceze, J. Tuck, J. Torrellas, and C. Cascaval, "Bulk disambiguation of speculative threads in multiprocessors," in *ACM SIGARCH Computer Architecture News*, vol. 34, no. 2. IEEE Computer Society, 2006, pp. 227–238.

[13] A. Shriraman, S. Dwarkadas, and M. L. Scott, "Flexible decoupled transactional memory support," in *Proceedings of the 35th Annual International Symposium on Computer Architecture*, ser. ISCA '08. Washington, DC, USA: IEEE Computer Society, 2008, pp. 139–150. [Online]. Available: https://doi.org/10.1109/ISCA.2008.17

[14] H. Litz, D. Cheriton, A. Firoozshahian, O. Azizi, and J. P. Stevenson, "Si-tm: reducing transactional memory abort rates through snapshot isolation," *ACM SIGARCH Computer Architecture News*, vol. 42, no. 1, pp. 383–398, 2014.

[15] S. V. Adve and K. Gharachorloo, "Shared memory consistency models: A tutorial," *Computer*, vol. 29, no. 12, pp. 66–76, Dec. 1996. [Online]. Available: http://dx.doi.org/10.1109/2.546611

[16] C. 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*. Citeseer, 2008, pp. 35–46.

[17] 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 *ACM SIGARCH Computer Architecture News*, vol. 32, no. 2. IEEE Computer Society, 2004, p. 102.

[18] C. Gniady, B. Falsafi, and T. N. Vijaykumar, "Is sc + ilp = rc?" in *Proceedings of the 26th Annual International Symposium on Computer Architecture*, ser. ISCA '99. Washington, DC, USA: IEEE Computer Society, 1999, pp. 162–171. [Online]. Available: http://dx.doi.org/10.1145/300979.300993

[19] V. Seshadri, G. Pekhimenko, O. Ruwase, O. Mutlu, P. B. Gibbons, M. A. Kozuch, T. C. Mowry, and T. Chilimbi, "Page overlays: An enhanced virtual memory framework to enable fine-grained memory management," *ACM SIGARCH Computer Architecture News*, vol. 43, no. 3, pp. 79–91, 2016.

[20] 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*. IEEE, 2007, pp. 261–272.

[21] L. Ceze, J. Tuck, P. Montesinos, and J. Torrellas, "Bulksc: bulk enforcement of sequential consistency," in *ACM SIGARCH Computer Architecture News*, vol. 35, no. 2. ACM, 2007, pp. 278–289.

[22] H.-T. Kung and J. T. Robinson, "On optimistic methods for concurrency control," *ACM Transactions on Database Systems (TODS)*, vol. 6, no. 2, pp. 213–226, 1981.

[23] T. Härder, "Observations on optimistic concurrency control schemes," *Information Systems*, vol. 9, no. 2, pp. 111–120, 1984.

[24] S. Tu, W. Zheng, E. Kohler, B. Liskov, and S. Madden, "Speedy transactions in multicore in-memory databases," in *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*. ACM, 2013, pp. 18–32.

[25] D. Dice, T. L. Harris, A. Kogan, Y. Lev, and M. Moir, "Hardware extensions to make lazy subscription safe," *arXiv preprint arXiv:1407.6968*, 2014.

[26] H. E. Ramadan, C. J. Rossbach, and E. Witchel, "Dependence-aware transactional memory for increased concurrency," in *Proceedings of the 41st annual IEEE/ACM International Symposium on Microarchitecture*. IEEE Computer Society, 2008, pp. 246–257.

[27] J. Fix, N. P. Nagendra, S. Apostolakis, H. Zhang, S. Qiu, and D. I. August, "Hardware multithreaded transactions," in *Proceedings of the Twenty-Third International Conference on Architectural Support for Programming Languages and Operating Systems*. ACM, 2018, pp. 15–29.

[28] X. Qian, B. Sahelices, and J. Torrellas, "Omniorder: Directory-based conflict serialization of transactions," in *2014 ACM/IEEE 41st International Symposium on Computer Architecture (ISCA)*. IEEE, 2014, pp. 421–432.

[29] S. A. R. Jafri, G. Voskuilen, and T. Vijaykumar, "Wait-n-gotm: improving htm performance by serializing cyclic dependencies," in *ACM SIGPLAN Notices*, vol. 48, no. 4. ACM, 2013, pp. 521–534.

[30] S. Tomić, C. Perfumo, C. Kulkarni, A. Armejach, A. Cristal, O. Unsal, T. Harris, and M. Valero, "Eazyhtm: eager-lazy hardware transactional memory," in *Proceedings of the 42nd Annual IEEE/ACM International Symposium on Microarchitecture*. ACM, 2009, pp. 145–155.

[31] U. Aydonat and T. S. Abdelrahman, "Hardware support for relaxed concurrency control in transactional memory," in *Microarchitecture (MICRO), 2010 43rd Annual IEEE/ACM International Symposium on*. IEEE, 2010, pp. 15–26.

[32] M. Gorman, *Understanding the Linux virtual memory manager*. Prentice Hall Upper Saddle River, 2004.

[33] D. Cheriton, A. Firoozshahian, A. Solomatnikov, J. P. Stevenson, and O. Azizi, "Hicamp: architectural support for efficient concurrency-safe shared structured data access," in *ACM SIGPLAN Notices*, vol. 47, no. 4. ACM, 2012, pp. 287–300.

[34] Intel Corporation, *Intel® 64 and IA-32 Software Developer's Manual*, May 2018, no. 325462-067US.

[35] M. Herlihy, V. Luchangco, M. Moir, and W. N. Scherer III, "Software transactional memory for dynamic-sized data structures," in *Proceedings of the twenty-second annual symposium on Principles of distributed computing*. ACM, 2003, pp. 92–101.

[36] D. Sanchez and C. Kozyrakis, "Zsim: Fast and accurate microarchitectural simulation of thousand-core systems," in *ACM SIGARCH Computer architecture news*, vol. 41, no. 3. ACM, 2013, pp. 475–486.

[37] Intel Corporation, *Intel® 64 and IA-32 Architectures Optimization Reference Manual*, April 2018, no. 248966-040.

[38] 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*. ACM, 2010, pp. 143–154.

[39] I. Watson, C. Kirkham, and M. Luján, “A study of a transactional parallel routing algorithm,” in *Proceedings of the 16th International Conference on Parallel Architecture and Compilation Techniques*. IEEE Computer Society, 2007, pp. 388–398.

[40] C. Y. Lee, “An algorithm for path connections and its applications,” *IRE transactions on electronic computers*, no. 3, pp. 346–365, 1961.

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