Safety Hints for HTM Capacity Abort Mitigation

Anirudh Jain†
School of Computer Science
Georgia Institute of Technology
Atlanta, Georgia, USA
Email: anirudh.j@gatech.edu

Divya Kiran Kadiyala†
School of Electrical and Computer Engineering
Georgia Institute of Technology
Atlanta, Georgia, USA
Email: dkadiyala3@gatech.edu

Alexandros Daglis
School of Computer Science
Georgia Institute of Technology
Atlanta, Georgia, USA
Email: alexandros.daglis@gatech.edu

***Abstract*—Hardware Transactional Memory (HTM) is a high-performance instantiation of the powerful programming abstraction of transactional memory, which simplifies the daunting—yet critically important—task of parallel programming. While many HTM implementations with variable complexity exist in the literature, commercially available HTMs impose rigid restrictions to transaction and system behavior, limiting their practical use. A key constraint is the limited size of supported transactions, implicitly capped by hardware buffering capacity. We identify the opportunity to expand the effective capacity of these limited hardware structures by being more selective in memory accesses that need to be tracked. We leverage compiler and virtual memory support to identify *safe* memory accesses, which can never cause a transaction abort, subsequently passed as *safety hints* to the underlying HTM. With minor extensions over a conventional HTM implementation, HinTM uses these hints to selectively allocate transactional state tracking resources to unsafe accesses only, thus expanding the HTM's effective capacity, and conversely reducing capacity aborts. We demonstrate that HinTM effectively augments the performance of a range of baseline HTM configurations. When coupled with a POWER8 HTM implementation, HinTM eliminates 64% of transactional capacity aborts, achieving 1.4× average speedup, and up to 8.7×.**

I. Introduction

Hardware Transactional Memory (HTM) is among the few hardware technologies that have been influential enough to mature from research ideas into commercial implementations. Since its emergence as a research topic in the late 70s, we have witnessed major CPU vendors develop specifications [2], [7], [16], [20] and offer concrete HTM instances in their commercial CPU product lines [13], [37], [39], [41] in the last few years. Despite its presence in modern processors, HTM usage is hardly mainstream. However, as the future of computing is undoubtedly parallel, aiming efforts at improving HTM—a technology targeting the programmability and performance of parallel code—by addressing shortcomings of current implementations is a worthy investment.

A utility obstacle to existing HTMs is their limited transactional capacity, capped by the size of hardware structures, whether these are dedicated buffers coupled with the CPU's caches, or these caches themselves [12]. When a transaction (TX) exceeds the HTM's tracking resources, it aborts. Capacity aborts are particularly detrimental to performance for two reasons. First, as a capacity abort only occurs after exceeding the hardware's tracking resources, it mostly impacts longer-running TXs, hence more work is lost. Second, the abort's nature precludes retrying the aborted TX: if a TX exceeded the HTM's capacity once, it will do so again. Thus, the system falls back to a software handler, which usually relies on a coarse-grained lock to guarantee atomicity, eliminating all parallelism until the TX commits.

Despite prior research proposals for HTMs with sophisticated mechanisms providing resilience to capacity limitations ("large" HTMs [31]), existing HTM implementations opt for practicality and follow more conservative designs of simpler early HTM proposals [34] to constrain hardware complexity. Our goal is to alleviate the capacity limitations of conventional HTMs, while preserving the relative hardware simplicity that facilitates their commercial availability. We pursue a software-hardware co-design approach that introduces *auxiliary* compiler and runtime mechanisms to expand the *effective capacity* of limited resources for transactional state buffering.

Our key observation is that conventional HTMs exert high pressure on their limited resources by being overly conservative in resource allocation for memory access tracking. HTMs commonly implement implicitly transactional semantics—i.e., they track *all* memory accesses within an annotated transactional block. However, many of these accesses are thread-private and thus cannot be involved in race conditions. An HTM aware of such *safe* accesses could omit tracking them to increase its effective transactional capacity and hence reduce capacity aborts.

The basic idea of memory access differentiation has been previously employed in similar contexts. Instances of Software Transactional Memory (STM) used compiler techniques to avoid costly logging and synchronization software overheads for thread-private memory accesses [32], [56]. While such software overhead concerns do not apply to HTM, we identify that similar techniques can be leveraged to mitigate *capacity* restrictions that are unique to HTM. A second relevant category of prior work leverages memory access differentiation by employing explicit page-level privatization via direct programmer annotations [71]. In contrast, our goal is to alleviate HTM capacity pressure via *automatic* memory access classification mechanisms that are *transparent* to programmers and can be naturally coupled with conventional HTM implementations.

We introduce *Hinted HTM* (HinTM), an *auxiliary* mechanism that boosts the effective transactional capacity of conventional HTMs by enabling them to distinguish safe from unsafe memory accesses. HinTM comprises two complementary memory access classification mechanisms. The first mech-

† Equal contribution.

anism is fine-grained annotation performed by a static safety analysis compiler pass. The second mechanism is dynamic but coarse-grained, extending the translation subsystem to keep track of temporal memory-sharing behavior at page granularity and to classify the safety of individual memory accesses at runtime. Through a new software-hardware interface, HinTM enables a conventional HTM to take a *safety hint* with each memory access, produced by the two classification mechanisms. That hint allows the HTM to reserve allocation of tracking resources exclusively for *unsafe* TX memory accesses.

We make the following contributions:

- We are the first to show *automatic* memory access classification as a mechanism capable of increasing the effective transactional capacity of HTMs, particularly the most capacity-constrained ones, and show this opportunity can be harnessed with modest hardware additions.
- We present HinTM, a set of two memory access classification mechanisms—at the compiler and runtime level—and a new interface to pass memory safety hints to the underlying HTM controller. Both mechanisms enable the effective expansion of the HTM's limited transactional capacity automatically and transparently to programmers.
- We evaluate HinTM's efficacy using a set of transactional workloads. Coupled with a POWER8-style HTM, HinTM reduces capacity aborts by 62% on average, resulting in performance improvements of up to 8.7× (1.4× on average). We also demonstrate that HinTM is an effective auxiliary mechanism for larger HTMs as well, using two concrete examples. For instance, compared to an HTM equipped with hardware signatures for readset expansion, HinTM yields an average 1.3× speedup by reducing capacity and false conflict aborts.

Paper outline: §II provides HTM background and motivates the promise of increased effective HTM capacity via memory access classification. §III presents HinTM's memory access classification techniques. We describe a concrete application of HinTM's mechanisms to an HTM implementation in §IV. We detail our methodology in §V and evaluate HinTM in §VI, discuss related work in §VII, and conclude in §VIII.

II. Background and Motivation

A. Hardware Transactional Memory

Transactional memory (TM) appeared in the late 70s as a programming abstraction to facilitate the daunting task of shared memory parallel programming. TM shifts the burden of coordinating shared memory accesses of different threads from the programmer to the underlying system. As it became increasingly evident that the future of computing is parallel, the promise of increased parallel programming productivity led to significant TM research activity.

TM implementations fall under two broad categories: software (STM) and hardware (HTM). STMs rely purely on software mechanisms, without any special hardware support. In contrast, HTMs accumulate a TX's state in hardware structures and rely on the cache coherence protocol to detect conflicts with other concurrent TXs. HTMs thus avoid prime performance overheads of STMs associated with software-based bookkeeping, including explicit data-copying and conflict detection, but their reliance on inherently bounded hardware structures sets rigid usability bounds and narrows the scope of HTM applicability. *This work proposes techniques to improve the utility of HTMs by relaxing their hardware-imposed transactional capacity constraints.*

Over the past decade, a few HTM implementations have become available by CPU vendors, most notably by Intel and IBM. Despite differences in their implementation details, all commercial HTMs impose an—often implicit—upper bound to supported TX size in direct relation to the capacity of hardware structures. When a TX's buffered state exceeds that hardware structure's capacity, the TX aborts and falls back to a software handler. Transactional state buffering is implemented as a memory hierarchy extension, either within the caches, or by a supplementary structure. For example, Intel's implementation [72] and IBM zEC12 [41] track transactional state within each core's private L1 cache, while IBM POWER8 employs a dedicated 64-entry fully associative buffer, linked with the general-purpose cache hierarchy [33], [46].

While some HTM implementations, like POWER8, use the same structure for transactional readset and writeset, HTMs often support asymmetric readset and writeset capacities. Readsets are cheaper to scale, because only memory addresses—and no updated data values—must be tracked. Hardware signatures are a common space-efficient readset expansion mechanism [14], [62], [71]. With signatures, addresses of the readset that are evicted from the primary tracking structure are hashed and stored as bits in one or more bitvectors. Read-write conflicts are checked by hashing the address of coherence invalidation messages and checking for collision in the bitvector(s). Signatures offer a coarse-grained conflict detection mechanism that significantly expands an HTM's readset capacity, but have two limitations. First, they do not improve an HTM's writeset capacity. Second, conflict checks in the address-summarizing bitvectors are susceptible to unnecessary aborts due to false positives.

In this work, we devise an auxiliary mechanism to boost the effective transactional capacity of practical HTM implementations via memory access classification, enabled through a hardware-software co-design. We show that our approach is most effective in HTMs with small transactional capacity, like POWER8, but also effectively complements larger HTMs, including implementations that support asymmetric readset/writeset sizes.

B. TX Capacity Expansion via Memory Access Classification

HTM capacity pressure is partially an artifact of its programming interface. To use a modern HTM, programmers annotate transactional block boundaries with special instructions (`begin` and `end`). The marked code block bears *implicitly* transactional semantics: *all* memory accesses within the annotated block are treated as transactional, and the underlying hardware ensures that all memory accesses within it are

performed atomically. If the hardware detects any atomicity violation, or the running TX exceeds the HTM's available tracking capacity, the TX aborts and all of its changes are discarded. The architectural state is restored to the point right before the aborted TX's start and then a software handler determines whether to retry the TX in HTM mode or to trigger a software fallback locking mechanism to guarantee success and forward progress. In addition to conflicts and capacity overflows, there are other reasons for a TX abort [31], which our work accounts for but does not tackle.

We explore the opportunity of boosting the effective capacity of HTMs via a software-hardware co-design approach requiring only minor hardware additions. Our key observation is that the typical implicitly transactional HTM interface results in conservative tracking of *all* memory locations touched within a TX, and hence unnecessary volume of tracked state. Memory locations that are only accessible by a single thread or are shared by multiple threads but are read-only cannot be involved in a race condition, and could therefore be treated as *safe* from a concurrency control standpoint. The HTM can potentially omit tracking accesses to memory locations that are a priori guaranteed to be *safe*. By distinguishing between safe and unsafe memory locations and allocating its tracking resources only to the latter, an HTM can make more effective use of its limited hardware resources to support larger TXs and reduce the occurrence of capacity overflows.

To derive a first-order estimation of the proposed approach's potential, we use a set of transactional benchmarks and study the fraction of memory accesses within each TX that could be potentially treated as safe. We use two metrics to approximately gauge the existing opportunity. First, the fraction of safe memory regions over total memory regions accessed by the program. A memory region is safe if there is no read-write sharing (at memory region granularity) between two or more threads throughout the program's execution. Second, the fraction of *read* accesses to safe memory regions over the total number of memory accesses within TX code blocks. We collect these two metrics at two granularities: cache block (64B) and page (4KB).

Fig. 1 shows the fraction of runtime spent on capacity aborts, as well as the two aforementioned metrics for a range of transactional workloads running 8 threads (4 for genome and yada) on a POWER8-like HTM configuration (methodology details in §V). While some applications like kmeans and ssca2 only use tiny transactions and never exceed the HTM's transactional capacity, capacity aborts can account for up to 89% of an application's runtime, and 22% on average. At the same time, we find that a remarkable subset of an application's touched pages is safe (62% on average). In turn, the number of transactional read accesses to such safe pages represents 40% of the total transactional memory accesses, on average. When tracked at the finer cache-block granularity, the fraction of such safe read accesses approaches 60%.

Although these results do not directly correspond to a commensurate reduction in tracking requirements—and, by extension, capacity aborts—they do indicate significant potential in leveraging memory access classification toward that goal. In addition, we observe that most of the opportunity can be captured even via tracking at a page granularity, hinting at an opportunity of developing a practical coarse-grained mechanism for memory region safety tracking.

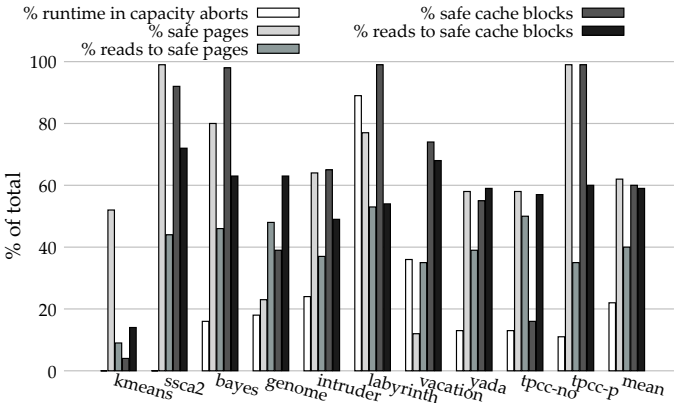


Fig. 1: Opportunity study for transactional capacity pressure reduction via memory access classification.

C. Memory Classification in Prior TM Work

The general concept of memory access classification in the broad context of TM is not new. Here, we highlight the key distinctions between our proposed approach and closely related prior work. We discuss additional related work in §VII.

STMs employ compiler techniques to identify thread-private memory accesses and eschew performance overheads for tracking those. Harris et al. [32] use inter-procedural analysis to identify and skip logging objects that are newly allocated within a TX, thus unreachable if the TX aborts. Shpeisman et al. [56] leverage dynamic escape analysis to distinguish between private and public objects, and skip costly synchronization on read/write barriers for the former. Inspired by such prior STM approaches, HinTM's static mechanism for distinguishing between safe and unsafe memory accesses is based on similar compiler support. We do not introduce novel compiler techniques, but rather *introduce an innovative application of these techniques in a distinctly different context.*

STMs employ classification to mitigate the *software* overheads of explicit data copies and version comparisons for conflict detection. We reuse the idea of memory classification, but to alleviate capacity constraints of HTMs—a concern that does not exist in the context of STM. Instead of using such information exclusively in software, we introduce a *hardware-software co-design*, where memory safety information is derived in software (i.e., the compiler) and is explicitly passed down to hardware via a new interface. We thus borrow an idea previously used by STMs, to alleviate a weakness inherent to HTM and without compromising HTM's key strength of being a fully hardware solution, as we do not introduce any software overheads at runtime.

Notary [71] is the only prior work we know leveraging memory access classification on an HTM rather than on an STM. Notary provides an interface to annotate safe pages

```
1  THREAD_REGION_BEGIN()
2  while (!isempty(globalTaskListPtr)) {
3    learner_task_t *taskPtr;
4    TRANSACTION_BEGIN(tid);
5      taskPtr = popTask(globalTaskListPtr);
6    TRANSACTION_END(tid);
7    struct operation_t op = taskPtr->op;
8    long fromID = taskPtr->fromId;
9    long toID = taskPtr->toId;
10
11   TRANSACTION_BEGIN(tid);
12     switch (op) {
13       net_applyOp(globalNetPtr, op, fromId, toID);
14       // ... More Code Follows in Transaction ....
15     }
16   TRANSACTION_END(tid);
17 }
18 THREAD_REGION_END()
```

Listing 1: Pseudocode snippet - Bayes application [48].

```
1  THREAD_REGION_BEGIN()
2  grid_t* myGridPtr =
3  Pgrid_alloc(gridPtr->width, gridPtr->height,
                  gridPtr->depth);
4  vector_t* myPathVectorPtr = Pvector_alloc(capacity);
5  while(!isempty(workQueuePtr)) {
6    TRANSACTION_BEGIN(tid);
7    grid_copy(dst: myGridPtr, src: globalGridPtr);
8    // ... More Code Follows in Transaction ....
9    Pvector_insert(src: globalPathVectorListPtr,
                    dst: myPathVectorPtr);
10   TRANSACTION_END(tid);
11 }
12 Pgrid_free(myGridPtr); // Release memory
13 THREAD_REGION_END()
```

Listing 2: Pseudocode snippet - Labyrinth application [48].

that can skip HTM tracking, but annotation is coarse-grained and programmer-driven, and no concrete implementation is described, as we further elaborate in §VII. Manual annotation of thread-private memory has also been used in the broader context of concurrency control and STM [3], [8], [38]. In contrast, HinTM employs *fine-grained* (instruction-level) *compiler-driven annotation* and *automatic coarse-grained* (page-level) *annotation at runtime*, which are less error-prone and more scalable than manual programmer annotation.

III. HinTM's Memory Classification

We propose enhancing existing HTMs with auxiliary memory access classification mechanisms, enabling the hardware to differentiate between *safe* and *unsafe* memory accesses within a TX. We start with a more precise definition for access safety than §II-B's intuitive introduction. From a transactional standpoint, a *memory location* is safe if:

(i) it is exclusively accessible by a single thread during a TX's lifetime; *or*
(ii) it is shared across threads but is read-only.

Similarly, a *memory access* is safe if:

(i) it targets a safe memory location; *and*
(ii) it does not have side-effects that would leave the application's state corrupted if the TX aborts.

By these definitions, all load operations to safe memory locations are safe. Safety of store operations is more nuanced, as even stores to thread-private (i.e., safe) memory locations may leave an application's state corrupted if the TX aborts and the original value prior to the transactional write is not restored. Hence, a store can be classified as safe only if it is also *initializing*—i.e., it is the first memory operation to a given memory location used in the transaction. Examples of such naturally occurring initializing writes are local variables initialized within a TX for local use, newly created objects about to be entered into a shared data structure, or arguments pushed into the stack before a function call.

To demonstrate safe memory locations and accesses we use pseudocode snippets from two STAMP applications, shown in Listings 1 and 2. In Listing 1, the initializing write to the stack-based memory location `taskPtr` in the first TX, and subsequent reads from memory locations `op`, `fromID` and `toID`, being passed as function arguments can be considered safe since only a single thread accesses these memory locations. Listing 2, on the other hand, demonstrates a case of a thread-private data structure being allocated on the heap. `myGridPtr` and `myPathVectorPtr` are both allocated per thread on the heap via a shared memory allocation call. Since all writes to `myGridPtr` (in the sub-function `grid_copy`, Line 7) are initializing and `myGridPtr` is never assigned to a memory location shared among threads, all accesses to it are safe. In contrast, accesses to `myPathVectorPtr` are unsafe because that pointer is assigned to `globalPathVectorListPtr`, a thread-shared data structure, resulting in the memory region allocated for `myPathVectorPtr` to potentially have multiple reader/writer threads with possible conflicts.

Assuming that every load and store operation in a TX is marked as safe or unsafe, the HTM can use such metadata for differentiated handling of the two memory access types. The memory locations touched by safe accesses need not be tracked, thus reserving state tracking resources for unsafe accesses only. The net effect of memory access classification in the two code-snippet examples is that the underlying HTM hardware must only track accesses associated with `globalNetPtr` in Listing 1, and with `myPathVectorPtr` and `globalPathVectorListPtr` in Listing 2. In contrast, a conventional HTM indiscriminately tracks all memory locations touched within the TXs: `globalNetPtr`, `taskPtr`, `op`, `fromId`, `toId` in Listing 1 and `myGridPtr`, `myPathVectorPtr`, and `globalPathVectorListPtr` in Listing 2.

Next, we describe HinTM's two memory access classification mechanisms: static fine-grained code annotation at compile time and dynamic coarse-grained annotation at runtime.

A. Fine-grained Static Classification

Static compile-time analysis can be used to mark loads and stores to memory locations statically determined to be thread-private (e.g., most stack-allocated variables and parameters passed to a function by value), and loads to shared read-only locations. If static analysis determines that all memory locations accessed by a given load/store instruction are safe, that instruction is marked as safe. If an instruction accesses memory locations of indeterminable safety, or accesses both safe and unsafe memory locations, it is conservatively classified as

unsafe. §IV-A elaborates on the static analysis techniques we leverage for such compiler-based instruction classification.

Instruction-marking by the compiler requires *ISA extensions* in the form of an additional "safety flag" bit for each load/store instruction, to differentiate between safe and unsafe memory accesses. Safety information is conveyed from the processor to the HTM controller, which now only needs to track memory accesses performed by unsafe instructions.

B. Coarse-grained Dynamic Classification

Static instruction classification is accurate but inherently conservative, as the compiler is unaware of the dynamic memory access sequence from different threads at runtime. For instance, a shared heap-allocated dataset that is accessible by all threads, but is partitioned by the application across threads and thus is not actively shared in practice, could be treated as safe. We therefore leverage a secondary, dynamic memory access classification mechanism to complement the conservative compiler-based static classification.

We extend the address translation mechanism to track page-level inter-thread sharing patterns at runtime. Read accesses to safe pages can be classified as safe, and can thus be ignored by the HTM controller. The criteria for page safety are the same as for memory location safety (§III):

(i) the page is only accessible by a single thread; *or*
(ii) the page is read-only (possibly shared across threads).

Unlike static classification, dynamic classification never marks write accesses as safe, because determining the essential *initializing* quality at runtime is challenging.

To identify inter-thread page sharing, we extend conventional process-level page protection to thread-level page protection. If a translated address belongs to a safe page, the memory read is marked safe. Similar to compiler-annotated instructions, that information propagates to the HTM controller. Unlike compiler-annotated safe instructions, a memory access' safety can be revoked at any time, when a page's status transitions from safe to unsafe. For example, a thread may request write permissions to a shared read-only page, or a second thread may request access to a read-write page previously exclusively accessed by another thread.

Page state transitions may happen amid an active TX, raising an implication: As the HTM has not been tracking memory accesses that were considered to be—at the time—safe, any potential atomicity violations on addresses falling within the safe-turning-unsafe page could go unnoticed. Therefore, any active TX that has touched such an affected page must conservatively abort. When the aborted TX is retried, the page—and hence all memory accesses to it—are marked as unsafe, thus reverting to normal tracking by the HTM.

Distinction from Page-based TMs. Some TM instances track transactional state at page granularity. IBM's 801 [15] extends the page tables to keep track of coarse-grained transaction-based locking, triggering an abort if a TX touches a page already held by another concurrent TX. PTM [18] and XTM [19] use page-based tracking to support TXs exceeding cache capacity. HinTM's approach differs from such page-based TMs, as it doesn't accumulate transactional state at page granularity and thus does not require additional hardware structures over a conventional HTM design that tracks transactional state at cache-block granularity. Instead, HinTM leverages page-level metadata as a first *exclude* filter, informing the underlying HTM's bookkeeping mechanism (that operates at cache-block granularity) whether it can *omit* tracking certain cache blocks.

IV. HinTM Implementation

We now describe the implementation of HinTM's memory access classification mechanisms introduced in §III. Our implementation builds on top of a baseline HTM that tracks transactional state by leveraging either a dedicated hardware structure or additional metadata within the data cache. With minimal hardware additions, HinTM relaxes the transactional capacity constraint of *any bounded HTM*. HinTM involves three components: static memory access classification, dynamic memory access classification, and a new interface for safety hint propagation to the underlying HTM controller.

A. Compiler Support for Static Classification

Our end-to-end compilation pipeline involves identifying *safe* memory locations and memory operations to these locations, annotating the identified safe operations in the generated binary, and propagating that information to the CPU during runtime via special `safe load/store` instructions. We employ static analysis to identify *safe* memory operations, implemented as a series of LLVM compiler passes [45]. The compiler marks a memory operation within a TX as *safe* if it can conclusively determine that the operation:

- is a load to a thread-private memory location—i.e., no other thread can access that memory location—or to a shared *read-only* memory location. We employ escape analysis to assess thread-private access.
- is an *initializing* store operation to a thread-private memory location. Our compiler pass determines this quality if the store's target thread-private memory location is defined before being used within a TX.

We employ three well-known techniques to identify safe memory operations. First, we extend LLVM's Capture Tracking to identify stack-allocated local objects and the call instructions passing these local objects by reference. The called function is then replicated to the thread-local object, with all qualifying load/store instructions replaced with their safe counterparts. The `call` instruction itself is modified to point to the newly replicated function. Given the potential existence of multiple callers passing different arguments in terms of safety, function replication is required to ensure that calls to the original function with non-transactional and *unsafe* arguments remain unaffected.

Second, we target thread-private heap-allocated data structures, often used as scratchpad memory in TM programs. Algorithm 1 provides a high-level description of our analysis that identifies potentially safe, thread-private heap-allocated

Algorithm 1: Inter-procedural analysis identifying potentially thread-private heap data structures.

```
   Input: AST/IR Source Code
   Output: Set of potentially thread-private data structure objects
1  Set_of_Thread_Private = {All heap allocations within Thread Begin
     and Thread End segments with corresponding memory
     de-allocation operation within the same region}
2  Set_of_Shared = {External allocations as arguments to Thread Begin
     function and Global Objects}
3  Worklist = {Thread Spawn function}
4  Visited = {}
5  while !Worklist.empty() do
6      function, argPos = Worklist.pop()
7      if Visited.contains((function, argPos)) then
8          continue
9      for Instructions Inst ∈ function do
10         for Uses u ∈ to Set_of_Thread_Private do
11             for Defs d ∈ Set_of_Shared do
12                 if d = Deref(u) then
13                     Set_of_Thread_Private -= u
14                     Set_of_Shared += u
15         if Inst is Call and u ∈ Set_of_Thread_Private then
16             Worklist.push((Inst.calledFunction, argPos))
17     Visited.add((function, argPos))
```

data structures. Algorithm 1's compiler pass performs an inter-procedural analysis that identifies local allocations, which are deemed potentially safe as long as they:

(i) are not assigned to global or shared objects; and
(ii) are de-allocated within the multi-threaded region.

An escape analysis pass (using the results of a pointer alias analysis) is then run on this set of potentially safe objects to verify that they are truly non-escaping and private to a single thread. This escape analysis is required to ensure that partially shared objects are not incorrectly marked safe. Loads and initializing stores to these locations, including those in functions where these objects are passed by reference, are then marked safe (including function replication, if necessary).

Third, we identify read-only shared memory locations in parallel regions. We utilize a pointer alias analysis pass [59] to create a set of read-only shared memory locations and mark load operations associated with these regions as safe.

Optimization Opportunity. The automated compiler passes we use for HinTM are neither novel nor intend to represent the state of the art in static safety analysis. Our goal is a proof-of-concept prototype to demonstrate the potential of such memory access classification in the novel context of our study. Use of more advanced compiler techniques (e.g., [27], [61]) and explicit programmer-driven annotations would only help increase the fraction of identified safe accesses, and improve HinTM's potential.

ISA Extensions. Conveying compiler-generated instruction safety information to the CPU, and ultimately to the HTM controller, requires ISA support: either in the form of an additional bit in existing memory operations (loads and stores), or by introducing new instructions. Our MIPS-based implementation uses two unused opcodes to encode a `load_word_safe` and a `store_word_safe` instruction. The instructions are

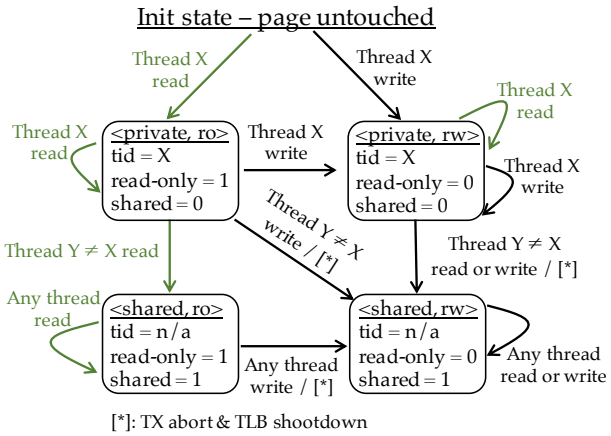


Fig. 2: State transition diagram for a page's lifetime. Green memory accesses are dynamically marked safe.

functionally equivalent to their unsafe counterparts, except that the HTM controller does not track the addresses they operate on. All optimizations such as load coalescing and merging are still performed for the newly added instructions, resulting in no performance loss. The transformation of loads/stores to their safe counterparts is automatically performed by our compiler and is therefore transparent to the programmer.

While introducing new instructions just to enable HinTM may seem costly, we note that modern ISAs already provide a number of instructions to allow direct CPU communication with co-processors (e.g., ARM [6]). By treating the underlying HTM as a co-processor, a CPU can leverage such co-processor instructions to convey memory access safety hints.

B. System Support for Dynamic Classification

HinTM's dynamic classification mechanism leverages address translation to track inter-thread sharing patterns at page granularity. All of a process' threads by default share the same page table (PT), which we extend to track each page's state on a per-thread basis. We add three fields in each PT entry's software structure: (i) a thread id (*tid*), (b) a *read-only (ro)* bit, and (c) a *shared* bit. *tid* records the first thread that accessed the page. Read accesses to a $\langle private, * \rangle$ or $\langle shared, ro \rangle$ page are safe; all other accesses are unsafe. We also add these two bits per data TLB entry, from which the page's safety state can be derived. The dynamic classification mechanism does not apply to load/store instructions already statically marked as safe by the compiler (§IV-A). Fig. 2 shows a page's state transition diagram as threads access it over time. When thread X initially accesses a page, the page walk following the TLB miss finds the page untouched, marks the page private, sets $tid = X$ and the access mode to *ro* or *rw*, depending on the access type. The translation is installed in the TLB in $\langle private, * \rangle$ state. When thread X attempts to write a page in $\langle private, ro \rangle$ state, a minor page fault triggers a PT entry $\langle private, ro \rangle \rightarrow \langle private, rw \rangle$ state transition. While a page's state is $\langle private, * \rangle$, all of thread X's read accesses to that page are marked as safe. When thread Y accesses the same page, it misses in the TLB and page walks to read the corresponding PT entry. If thread Y's access is a read and the page's state

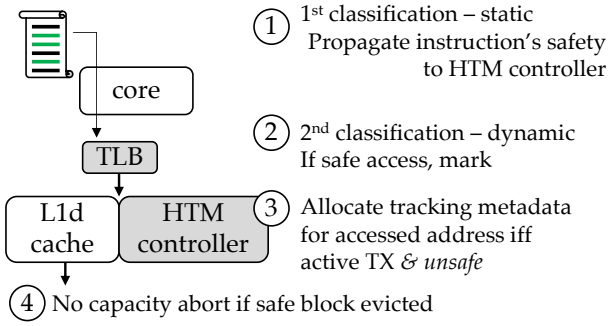


Fig. 3: HinTM end-to-end operation example.

is $\langle private, ro \rangle$, the page transitions to $\langle shared, ro \rangle$ and the translation is installed in the TLB.

When a thread attempts to write a $\langle shared, ro \rangle$ page, the cached translations of that page in other TLBs are invalidated via a conventional TLB shootdown [68]. The core that initiates the $\langle shared, ro \rangle \rightarrow \langle shared, rw \rangle$ page transition (i) aborts its current TX (if applicable), (ii) executes the OS TLB shootdown handler that collects the list of cores that have accessed the target page in the past (i.e., the slave cores), and (iii) sends an IPI to each slave core. Each slave core invalidates the TLB entry indicated in the IPI, while the IPI reception itself causes any running TX on the slave core to abort. Finally, the initiator core updates the PT entry's state as soon as all IPI receptions are acknowledged, thus ensuring that the page's unsafe state is recorded and visible to all cores in the future.

Optimization Opportunity. As shown in Fig. 2, $\langle shared, rw \rangle$ is a terminal page state in our implementation. This simple approach may leave performance opportunities on the table for long-running programs with phased behavior that would favor a periodic page state reset. Developing such a policy would require balancing the tradeoff between costly periodic TLB shootdowns and the benefit of effective HTM transactional capacity boost afforded from temporary page safety.

C. Putting it All Together

Fig. 3 shows HinTM's high-level architecture, which harnesses both statically and dynamically set memory access safety hints to shrink state tracking requirements per TX. HinTM is equally applicable to HTM baseline systems where transactional cache block tracking leverages an enhanced tag array in the L1d cache (e.g., Intel RTM), or dedicated buffers coupled with the cache (e.g., IBM POWER8).

A program compiled using HinTM's static analysis (§IV-A) has a subset of its memory instructions marked as safe (abstractly represented by green lines in Fig. 3's "code snippet"). Safety information is encoded in the ISA and flows through the core's pipeline down to the HTM controller as an additional address bit (step ①). A memory access that is not statically marked as safe may still be marked as safe by the dynamic classification mechanism, which is embedded in the translation process (step ②). HinTM leaves the baseline system's HTM controller mostly unmodified: memory accesses performed within a TX's boundaries are recorded and monitored in hardware structures for potential atomicity violations. However, the HTM controller can ignore memory accesses marked as safe by HinTM's classification mechanisms, by skipping any transactional bookkeeping for them (step ③). Safe cache blocks touched by a TX can spill from the L1d cache (or transactional buffer) without causing a TX abort (step ④). The net result is that the limited hardware resources available to the HTM controller for address monitoring are dedicated to exclusively tracking unsafe accesses, thus supporting larger TXs and mitigating capacity aborts.

Overall, HinTM is a software-hardware co-design, where most changes are introduced in software (compiler and page tables) and are supported by modest hardware extensions. Table I summarizes all of HinTM's hardware additions.

V. Methodology

System Organization. We use the SESC [54] cycle-accurate simulator used in prior work to model conventional HTMs with eager conflict detection [52]. Table II summarizes the used parameters for our simulated 8-core SMP. Cache latencies are obtained from CACTI 7 [9] at 22nm.

Compiler. We extend LLVM 9.0.1 [1] with the memory access classification passes detailed in §IV-A, using a MIPS backend to match the ISA supported by our simulator.

HTM Configurations. We evaluate four baseline HTMs:

- **Dedicated transactional buffering.** Modeled after IBM's POWER8 HTM implementation [46], this configuration provisions an external 64-entry fully associative buffer coupled with the L1 data cache to track cache blocks that belong to the running transaction's readset and writeset. We refer to this HTM configuration as `P8`.
- **Hardware signatures.** We extend the `P8` baseline with hardware signatures, modeled after the state-of-the-art PBX hashing with a 1kb bitvector [71]. Signatures prevent capacity aborts when a TX's readset spills from the transactional buffer. We refer to this configuration as `P8S`. `P8S` increases readset but not writeset capacity, and introduces the possibility of false conflict aborts due to aliasing.
- **In-L1-cache transactional buffering.** Instead of provisioning dedicated buffers, a transaction's state is tracked in the L1 data cache, offering larger transactional capacity than `P8`. We refer to this configuration as `L1TM`.
- **Infinite buffering.** This ideal—from a capacity perspective—HTM configuration never aborts due to capacity overflows, but otherwise remains functionally identical to the previous two HTMs in terms of detecting and reacting to conflict aborts, etc. We refer to this configuration as `InfCap` and use it as an upper bound for the maximum

TABLE I: HinTM's required hardware modifications.

Core	a) two new instructions for safe loads/stores and b) one extra safety bit from the CPU to the TLB and L1 cache (<2% address bus width increase)
TLB	two bits per TLB entry to mark whether a given page is shared or thread-private, and read-only or read-write (<2% TLB area overhead)
HTM controller	∼ an extra mux to skip tracking safe loads/stores as transactional

gains achievable by completely eliminating capacity aborts. Fig. 1's fraction of runtime wasted on capacity aborts is derived as a comparison between `InfCap` and `P8`.

We extend the `P8`, `P8S`, and `L1TM` baselines with HinTM's memory classification techniques and hardware extensions, introducing three additional configurations for each of them:

- **HinTM-st** employs static memory classification (§IV-A).
- **HinTM-dyn** employs dynamic memory classification (§IV-B). Safe to unsafe page mode transitions incur a TX abort and TLB shootdown, which involves OS handler executions and IPI latencies. We model a cost of 6600 and 1450 cycles for the initiator and involved slave cores, respectively, derived from an extensive TLB shootdown cost study [68]. $\langle private, ro \rangle$ to $\langle private, rw \rangle$ page mode transitions incur a 1450-cycle minor page fault cost [10].
- **HinTM** employs both static and dynamic memory classification (HinTM-st + HinTM-dyn).

Workloads. Similar to recent HTM work [40], [50], [51], [52], we evaluate HinTM using the STAMP transactional benchmark suite [48], as well as TPCC's two most prevalent queries: `new_order` (named *tpcc-no*) and `payment` (named *tpcc-p*) [43], [63]. We deploy genome and yada on four threads because we observed poor scalability for higher thread counts. Every other application runs on eight threads.

Why Capacity-limited HTM? We base our study on HTM implementations with small capacity, for practical reasons. As the latter has larger transactional capacity, creating sufficient capacity pressure requires significantly larger application inputs, resulting in impractical simulation times, as indicated by the fact that transactional benchmarks often provide smaller inputs for simulation-based studies [48].

Although HTMs with larger transactional capacity will suffer fewer capacity aborts, capacity limitations of even the larger Intel-style HTMs are known to impose real-world performance overheads [12]. Ultimately, evaluating HTM capacity limitations using existing transactional benchmarks poses a chicken-and-egg problem as benchmarks tune TXs to mostly fit in the available transactional capacity. Larger HTM capacity will reduce capacity aborts for existing benchmarks, but also enable new workloads with larger TXs; existing benchmarks won't demonstrate the latter gain. Our evaluated benchmarks are not artificially dominated by capacity constraints. For instance, as seen next in our evaluation, 85% of aborts in TPCC-p are conflict aborts, both with and without HinTM, but still, reducing the small fraction of existing capacity aborts results in 16% speedup.

TABLE II: Simulation parameters.

CPU	8 OoO cores, 2GHz, 176-entry ROB 4-wide dispatch/retirement, MIPS ISA
L1 Cache	32KB 8-way L1d/L1i (split) 64B blocks, 3-cycle latency
L2 Cache	shared non-inclusive, 8MB, 16-way 64B blocks, 12-cycle latency
Coherence	Snoopy MESI
Memory	100-cycle latency

In summary, increasing the underlying hardware's transactional capacity only delays the onset of capacity limitations on transaction sizes. Although our evaluation directly quantifies HinTM's benefits using a few different capacity-constrained HTMs, we posit that HinTM is a mechanism *complementary* to any baseline HTM configuration. The `P8S` configuration demonstrates HinTM's effect on HTMs with asymmetric readset/writeset capacities, a characteristic of some commercial HTM implementations such as Intel's. Finally, `L1TM` represents HTMs with larger transactional capacity than `P8`.

VI. Evaluation

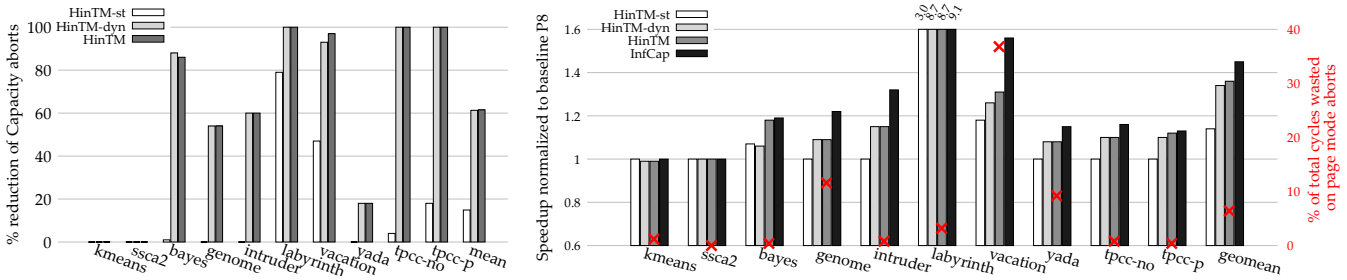
We start our evaluation by demonstrating that HinTM is an effective auxiliary mechanism for `P8`, where HinTM's greatest benefits stem from effective readset size reduction, directly translating into capacity abort reduction (§VI-A). §VI-B drills down on the costs incurred by HinTM's newly introduced page mode aborts and §VI-C analyzes the breakdown of each application's memory accesses to better illustrate how HinTM increases effective HTM capacity. We conclude our evaluation in §VI-D where we demonstrate that HinTM yields solid performance gains on HTM configurations with larger effective capacity than `P8` as well, by achieving writeset reduction, mitigating false conflict aborts, and/or reducing the occurrence of conservative capacity aborts due to conflict misses in the cache.

A. Capacity Abort Reduction and Speedup with `P8` HTM

Fig. 4a illustrates the achieved capacity abort reduction for each of HinTM-st, HinTM-dyn, and HinTM as compared to the baseline `P8` HTM configuration. Fig. 4b shows how Fig. 4a's capacity abort reductions reflect into performance improvement, and also includes the performance effect of the hypothetical `InfCap`, which eliminates *all* capacity aborts.

As evidenced in Fig. 4a, static classification (HinTM-st) alone in most cases is not sufficient to reduce capacity aborts and, by extension, improve performance. As we later demonstrate, while static classification does identify safe accesses, they are not enough to reduce a TX's size enough to prevent a capacity abort from occurring. There are two notable exceptions: labyrinth and vacation, for which HinTM-st prevents ∼80% and ∼48% of capacity aborts, which results in 2.98× and 1.18× speedup, respectively. The resulting speedup is not a direct function of the achieved reduction in capacity aborts: while HinTM-st prevents some TXs from capacity-aborting by implicitly increasing the underlying `P8`'s effective transactional capacity, these same TXs may end up aborting later due to other reasons (e.g., a conflict). `InfCap`'s speedup in Fig. 4b is indicative of each application's potential benefit from capacity abort reduction. For instance, `InfCap` demonstrates that labyrinth's improvement potential is significantly higher than vacation's (9.1× versus 1.6×).

HinTM-dyn is more effective than HinTM-st, eliminating 61% of capacity aborts on average. This drastic reduction yields an average speedup of 1.34× (1.45× if applications



(a) Capacity abort reduction.

(b) Speedup and cycles wasted on page mode transitions.

Fig. 4: Performance impact of HinTM on the P8 HTM configuration.

with zero capacity aborts—kmeans and tcca2—are disregarded), with labyrinth achieving the highest speedup of 8.7×. Because HinTM-dyn as a standalone mechanism is much more effective than HinTM-st, the achieved capacity abort reductions of HinTM (which is a combination of dyn and st) are mostly subsumed by HinTM-dyn. As a result, HinTM's speedup is marginally higher than HinTM-dyn's (2% higher on average). Overall, HinTM achieves an average speedup of 1.36× over baseline P8 (1.14× excluding the extreme cases of labyrinth, kmeans, and ssca2), and is within 6% of InfCap's performance. Importantly, for applications like ssca2 and kmeans that exclusively employ very small TXs and thus never trigger a capacity abort, HinTM has a performance-neutral effect: naturally, HinTM cannot improve performance, but is not detrimental either. HinTM-st has no effect, while the page mode transition aborts introduced by HinTM-dyn do not incur a noticeable slowdown. This marginal slowdown is avoidable by proactively disabling HinTM's mechanisms for applications that only use tiny TXs.

B. Effect of Page Mode Aborts

Fig. 4b's secondary y axis shows the cost incurred by our newly introduced page mode aborts, quantified as the aggregate cycle count spent across cores on page mode abort actions (including initiator/slave core overheads as per §V) divided by the total number of cycles spent for the application's execution. The net cost of page mode aborts depends on two factors: the frequency of such aborts and the resulting cost per abort. The former is a function of the fraction of safe pages and the application's total runtime; the longer the runtime, the better page mode transition costs are amortized, as each page may transition at most once in our HinTM implementation. The latter is a function of the number of threads affected by the ensuing TLB shootdown, and the amount of TX work that ends up being lost upon such shootdown.

Fig. 4b shows that the fraction of cycles spent on page mode transitions is modest, with vacation as the only outlier because of a combination of three factors: vacation exhibits a high fraction of read-write pages (>85%), the highest frequency of page-mode aborts, and the highest cost (number of execution cycles lost) per page-mode-transition abort. To further study vacation's outlier behavior and investigate opportunities for further optimization, we conduct an additional experiment with

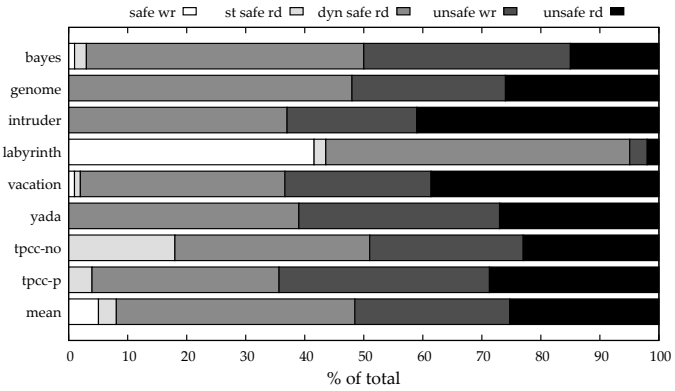


Fig. 5: Memory access breakdown within transactions.

an ideal zero-cost TLB shootdown. Surprisingly, we find that vacation's achieved speedup only increases by 4%. Despite that vacation on HinTM spends almost 40% of total cycles in page mode aborts, eliminating these costs yields minimal gains, as the increased effective concurrency converts into more conflict aborts that consume most of the achieved overhead savings. Mechanisms for page safety transition prediction or for faster TLB shootdowns (like DiDi [68]) can alleviate page mode transition costs.

C. Transactional Capacity Pressure Study

We henceforth *omit ssca2 and kmeans for brevity*. To identify HinTM's source of benefits, Fig. 5 shows the dynamic breakdown of memory accesses performed within each application's TXs by type, distinguishing between compiler- and runtime-annotated safe accesses (collected using HinTM +P8).

The two memory access classification mechanisms combined identify ∼50% of the total memory accesses on average as safe. In labyrinth's extreme case where most memory accesses are to thread-private buffers, our combined classification mechanisms identify 95% of the accesses as safe. The vast majority of safe accesses are identified by the dynamic classification mechanism, indicating why HinTM-dyn is notably more effective than HinTM-st in Fig. 4.

The fraction of compiler-annotated safe accesses indicates why HinTM-st only achieves small performance benefits. Our static analysis identifies no safe accesses for genome, intruder, and yada, but classifies 18% of tpcc-no's loads as safe. The best case for static classification is labyrinth, marking 44%

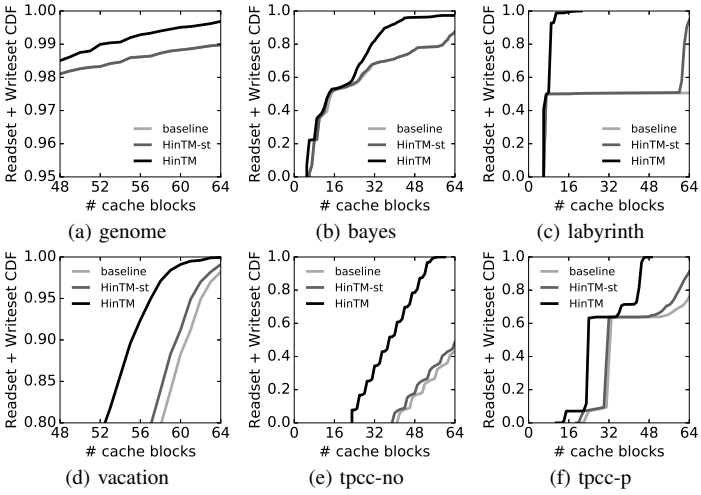


Fig. 6: Transaction size CDFs for P8 configurations: baseline, HinTM-st, and HinTM. HinTM-st and baseline fully overlap in (a) and (b). Note the different y-axis scales.

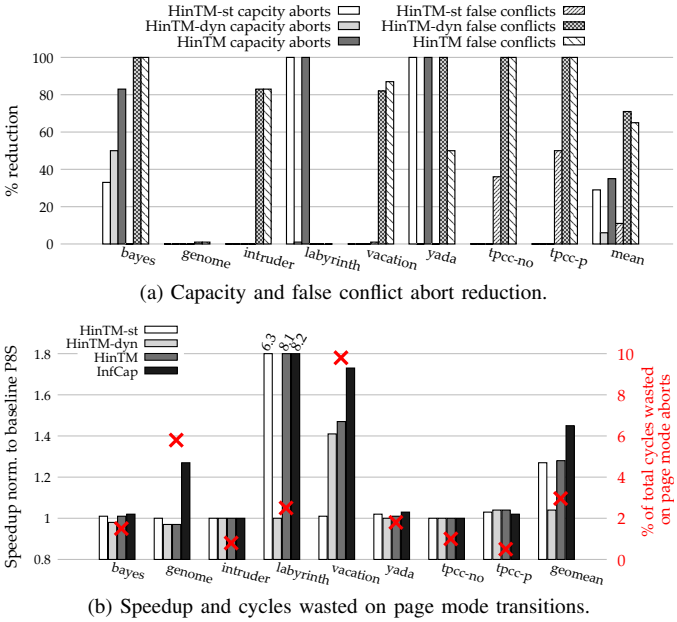


Fig. 7: Impact of HinTM on the P8S HTM configuration.

of the memory accesses performed within TXs as safe. This high percentage of statically identified safety is inherent to the application's structure, where every TX starts by making a thread-private copy of the grid it operates on to perform optimistic source-destination routing. Finally, vacation and bayes have a small fraction (3% and 2% respectively) of their transactional memory accesses statically identified as safe.

The fraction of safe accesses alone does not explain the observed capacity abort reduction and performance gains. For example, while HinTM-st identifies just 2% of transactional memory accesses as safe in vacation, it significantly reduces capacity aborts and achieves a sizeable speedup of 18% (Fig. 4). In contrast, the 18% of loads statically marked as safe in tpcc-no only reduce capacity aborts by 4%, resulting in virtually no speedup. Interestingly, the opposite trend holds for tpcc-p: only 4% of loads are statically marked as safe, allowing HinTM-st to reduce capacity aborts by 18%. This trend is attributed to tpcc-no's safe loads exhibiting higher spatiotemporal locality. To shed more light into these behaviors, we next analyze each application's TX size distribution when HinTM-st and HinTM are used.

Fig. 6 shows the readset+writeset Cumulative Distribution Function (CDF) of each application's TXs, omitting yada and intruder for brevity. We cap the x-axis at 64 cache blocks, matching our modeled P8 configuration's transactional capacity. TX with sizes beyond the x-axis range are TXs that surely abort due to capacity constraints. We collect TX sizes by running InfCap and recording each committed TX's readset+writeset size as recorded by:

1) baseline HTM—i.e., every cache block touched in a TX.
2) HinTM-st—i.e., every cache block touched by a memory operation that is not statically marked as safe.
3) HinTM—i.e., every cache block touched by a memory operation that is not marked as safe by either of our two classification mechanisms.

The gap between HinTM's and baseline's CDFs shows how memory access classification (primarily the dynamic mechanism) shrinks effective TX sizes. Specifically the gap between different HTM configurations on the graphs' far right end indicates the reduced fraction of TXs exceeding P8's transactional capacity and explains the capacity abort reductions reported in Fig. 4a.

To illustrate, Fig. 6d shows that 2% of vacation's TXs exceed P8's transactional capacity, causing baseline P8 to perform 56% worse than InfCap (Fig. 4b). HinTM-st's safety marking enables half of those TXs to fit in P8's transactional buffer, leading to a 47% reduction in capacity aborts, and recouping about half of the performance gap between baseline and InfCap. We hypothesize that the effect is so pronounced, despite only 2% of vacation's runtime memory accesses being statically identified as safe, because these safe accesses are to unique cache blocks, while unsafe accesses have high spatio-temporal locality in the cache blocks they touch.

D. HinTM Effect on Larger HTMs

We now evaluate HinTM's impact combined with larger HTM baselines. We use §VI-A's applications with larger inputs to generate sufficient transactional capacity pressure.

1) P8S: P8S alleviates P8's capacity pressure by employing signatures to support an unbounded readset. In this case, the opportunity for HinTM's memory access classification is narrower, with benefits stemming only from false conflict and modest writeset size reduction. Fig. 7a illustrates the achieved capacity and false conflict abort reduction for HinTM-st, HinTM-dyn, and HinTM as compared to the P8S baseline for a subset of our benchmark applications. The effect of these reductions on performance is shown in Fig. 7b.

Compared to the P8 configuration, HinTM's opportunity for capacity abort reduction is limited, because any such reduction must stem from static classification's identification of safe writes for writeset size reduction, as P8S's readset is

effectively unbounded. HinTM only reduces capacity aborts for bayes, labyrinth and yada.

Due to the application's nature, HinTM-st identifies a large fraction of labyrinth's writes as safe, eliminating all capacity aborts. For bayes and yada, the small fraction of writes HinTM identifies as safe results in large relative capacity abort reduction, yet performance gains are minimal, because the absolute number of capacity aborts for both applications is low, accounting for a negligible fraction of their runtime.

Safe read identification does not reduce capacity aborts in `P8S`, but does mitigate false conflict aborts, which sometimes constitute a significant fraction of the application's total conflict aborts. For instance, false conflicts cause 39% and 37% of total conflict aborts in vacation and genome, respectively. However, reducing these false conflicts does not always result in performance improvements. For vacation, HinTM's reduction of false conflicts by 87% improves performance by 1.47× over baseline `P8S`, while genome's performance remains unaffected. In tpcc-no's case, ∼2% of TXs result in false conflicts, which HinTM completely eliminates. However, HinTM's overhead due to page mode transitions offsets this benefit, resulting in a small net performance loss.

Overall, when combined with `P8S`, HinTM's benefit wanes, as signatures eliminate readset capacity constraints. However, HinTM remains beneficial, benefiting most applications modestly and some (like labyrinth and vacation) significantly, for an average speedup of 1.28×.

2) `L1TM`*:* `L1TM` reduces capacity pressure by tracking read and write sets in the larger (32KB 8-way) private L1 cache. This tracking style can suffer from capacity aborts due to both capacity and set-conflict misses. In order to generate capacity pressure in `L1TM` while using practical workload sizes in our simulation environment, we employ 2-way SMT on each core. Fig. 8 shows the performance results. Despite `L1TM`'s increased transactional capacity, HinTM yields significant performance gains—1.7× on average and up to 7.1×—by reducing capacity aborts by 29–100%. Due to SMT-incurred capacity pressure, HinTM delivers the best performance improvements of all baseline HTM configurations evaluated.

The best gains are achieved for labyrinth, followed by genome. Page mode aborts are, on average, of no concern, with one significant outlier. As indicated by `InfCap`, transactional capacity expansion holds significant promise for vacation, which however is not fulfilled due to exorbitant page mode abort costs. Vacation's case motivates investigating improved mechanisms for page-mode classification with reduced page mode transition penalties, as previously mentioned in §IV-B.

E. Evaluation Summary

Our evaluation shows HinTM's effectively extends a baseline HTM's limited transactional capacity, improving performance by alleviating capacity aborts. Achieving the same effect solely with hardware requires larger buffering capacity or increased complexity for sophisticated overflow mechanisms. Even when such overflow mechanisms exist (e.g., signatures), HinTM is a beneficial auxiliary mechanism.

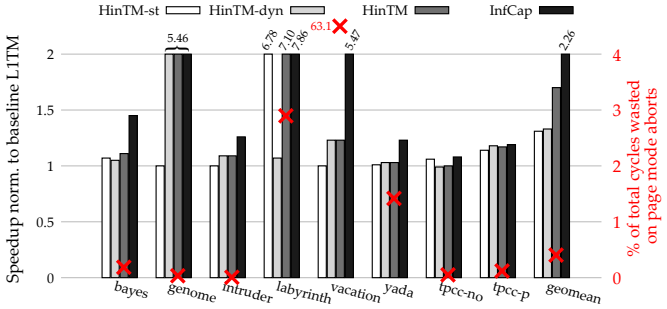


Fig. 8: HinTM's impact on `L1TM`. Speedup and cycles wasted on page mode transitions.

HinTM-st's average benefits are underwhelming. Although only noticeably boosting few workloads, HinTM-st is worth considering, as its implementation cost is predominantly on the compiler, with minimal hardware requirements. Binary size increase is also modest; our function replication (§IV-A) leads to an average and maximum increase of 3% and 5.8%, respectively. We reiterate that advanced compiler techniques can potentially be used to identify more safe accesses than our simple static analysis techniques and improve HinTM-st's potential. Furthermore, all our evaluated workloads are written in C, resulting in very conservative compiler safety classifications. We expect increased static classification opportunity when using languages with stronger in-built memory safety guarantees (e.g., Rust).

VII. Related Work

We discussed prior work most closely related to HinTM's mechanisms—compiler-driven classification in STM and user-annotated privatization—in §II-C. §III-B highlighted the distinction between HinTM's page-based memory safety classification and prior proposals on page-based TMs.

User-annotated privatization. Notary is the only work we know that uses a privatization method to boost an HTM's effective TX capacity. It provides a programmer interface for *manual* coarse-grained marking of thread-private data structures, and mentions automatic marking of thread-private stack pages. However, Notary's proposed design is more conceptual than algorithmic, lacking concrete implementation details of how the hardware maintains and retrieves that information to use it. In contrast, we meticulously detail the methods HinTM employs to identify safe memory and pass that information to hardware. Although we focused on *automation*, HinTM can trivially support both coarse-grained (i.e., Notary-style) and fine-grained programmer annotations as well.

ISA escape actions support for HTM. Intel and IBM HTMs offer suspend/resume operations that can be used to mark windows of instructions within a TX to skip HTM controller tracking [13], [21]. LogTM [49] uses similar escape actions to skip tracking non-TM code blocks during TXs (traps, interrupts, etc.). Such coarse-grained pause/resume approach is typically used to execute sizeable code blocks of secondary software synchronization mechanisms within a TX without aborting [28], [40], [42], [47], [74] and therefore concep-

tually differs from HinTM's automated identification and fine-grained differentiation of memory accesses that actually belong to a TX, with the goal of conserving the HTM's transactional capacity. Instead of our proposed *safe load/store* instructions, a compiler could use suspend/resume to wrap each load/store identified by HinTM's static classification as safe. IBM's System Z features a non-transactional store for debugging purposes [35]. Rock [16] and AMD's ASF [17] are HTMs that featured non-transactional load instructions. Prior work [4] discussed different ways of using such load instructions to improve HTM performance, and seminal HTM work [34], [47] hinted at selectively using non-transactional instructions, but only at a conceptual level. TCC [29] marked stack references as local to avoid broadcasts and improve scalability. We are the first to comprehensively study and evaluate automatic generation (by the compiler or the hardware) of non-transactional memory operations within transactions to mitigate HTM capacity limitations. HinTM can use these instructions on ISAs that feature them, as instances of our proposed safe load/store instructions. Finally, neither suspend/resume nor non-transactional instructions can be used to mark dynamically identified safe loads/stores, which yield most of HinTM's performance benefits.

Other approaches addressing HTM capacity limitations. Pre-abort handlers [51] provide a software fallback path to attempt to save TXs before aborting. For capacity aborts, a pre-abort handler would convert the aborting TX into a critical section, preventing work loss, but still resulting in serialization. The technique can be used in conjunction with HinTM, which reduces capacity overflows. Cai et al. [12] study the effect of cache replacement policy on Intel TSX capacity aborts. SI-HTM [28] builds on the semantics of snapshot isolation to expand POWER8 HTM's limited transactional capacity by only buffering a TX's writeset. SI-HTM combines roll-back only TXs [36] (ROTs—TXs that don't track loads) with a software mechanism that delays all TXs with a non-empty writeset to commit after all concurrent TXs that started earlier have already committed. Rather than snapshot isolation, HinTM targets the stricter transactional model of 2-phase locking (2PL). Issa et al. [40] combine ROTs with a software technique to prevent the readset from occupying transactional capacity, but still achieve strict 2PL semantics, at the cost of tracking the TX's whole readset in software and touching it again before a TX commits. HinTM does not require an additional TX validation phase with extra instructions and memory accesses, and in general leaves TX structure unmodified.

Dynamic page-based classification in non-TM contexts. Singh et al. [57] utilize static and dynamic memory access classification mechanisms to relax the ordering of thread-private memory accesses on sequentially consistent CPUs. OS page classification has been used by Cuesta et al. [23] to eliminate directory coherence tracking for data in private pages; in VIPS to construct a simple low-cost coherence mechanism [55]; and to inform intelligent data placement decisions in distributed memories [11], [30], [64], [65].

Beyond conventional HTMs. Our work targets "conventional" HTMs [31], [34] that impose rigid, hardware-bound limits on transaction sizes [50]. While all commercial implementations are instances of such conventional HTMs, the research literature is rich in advanced techniques to allow spilling transactional state from the limited hardware structures without aborting. "Large" HTMs (LTM [5], LogTM [49]) modify the cache hierarchy or coherence to track overflown state. "Unbounded" HTMs (UTM [5], VTM [53]) enhance hardware with software mechanisms to enable TXs to not only exceed hardware structure capacities, but also survive context switches. Hybrid TMs [24], [26], [44] are founded on an underlying STM implementation and leverage HTM as an auxiliary mechanism to improve performance, whenever possible. HinTM maintains the relative hardware simplicity of conventional HTMs and proposes minimally intrusive extensions that alleviate their rigid capacity constraints without sacrificing HTM's performance advantages over STM.

Other metadata tracking and hint-based approaches. Memory access differentiation lies at the core of HinTM's principle of operation. Such differentiation information could be encoded by leveraging systems featuring tagged memory [22], [25], [58], [60], [73] and capability-based approaches like CHERI [70] or Mondrian [69], using flexible metadata management frameworks like XMem [66] or MetaSys [67].

VIII. Conclusion

We presented HinTM, a lightweight extension to conventional HTMs leveraging a compiler-based and a runtime-based memory access classification mechanism to alleviate transactional capacity pressure and, by extension, mitigate capacity aborts. The classification mechanisms annotate memory accesses that cannot be involved in race conditions and the underlying HTM takes these hints to reduce memory access tracking requirements. Our evaluation of HinTM coupled with a range of baseline HTM configurations showed promising capacity abort reduction and performance gains. Indicatively, coupling HinTM with a POWER8 HTM implementation eliminates 64% of capacity aborts on average, resulting in average performance improvements of 1.5× and up to 8.7×. In addition to its performance gains, HinTM's approach has practical value as it is applicable to all existing HTM instances with only modest hardware modifications and no additional hardware structures. HinTM also facilitates HTM adoption in simpler processor designs—for example, RISC-V cores that do not yet feature such support but already have a planned HTM feature on the horizon [2].

Acknowledgments

We thank the anonymous reviewers for their valuable feedback. We are grateful to Sunjae Park for his help in the baseline HTM simulation model's setup and to Jisheng Zhao for his guidance on formulating the inter-procedural thread-private data structure analysis pass. We thank Hamed Seyedroudbari, Marina Vemmou, Albert Cho, and Anirudh Sarma for their constructive feedback that helped improve the paper.

REFERENCES

[1] "LLVM Project Github Source," https://github.com/llvm/llvm-project/tree/llvmorg-9.0.1.

[2] "RISC-V Instruction Set Manual: Standard Extension for Transactional Memory," http://five-embeddev.com/riscv-isa-manual/latest/t.html#sec:tm.

[3] M. Abadi, T. Harris, and K. F. Moore, "A model of dynamic separation for transactional memory," *Inf. Comput.*, vol. 208, no. 10, pp. 1093–1117, 2010.

[4] Y. Afek and H. Avni, "Evaluating the Addition of Non-Transactional Loads to HTM," *6th Workshop on the Theory of Transactional Memory (WTTM)*, 2014.

[5] C. S. Ananian, K. Asanovic, B. C. Kuszmaul, C. E. Leiserson, and S. Lie, "Unbounded Transactional Memory," in *Proc. 11th IEEE Symp. High-Perf. Comp. Architecture (HPCA)*, 2005, pp. 316–327.

[6] Arm Limited, "ARM Architecture Reference Manual: Coprocessor Instructions," https://developer.arm.com/documentation/ddi0406/c/Application-Level-Architecture/The-Instruction-Sets/Coprocessor-instructions.

[7] Arm Limited, "New Technologies for the Arm A-Profile Architecture," https://community.arm.com/developer/ip-products/processors/b/processors-ip-blog/posts/new-technologies-for-the-arm-a-profile-architecture.

[8] W. Baek, C. C. Minh, M. Trautmann, C. Kozyrakis, and K. Olukotun, "The OpenTM Transactional Application Programming Interface," in *Proc. 16th Int. Conf. Parallel Architecture and Compilation Techniques (PACT)*, 2007, pp. 376–387.

[9] R. Balasubramonian, A. B. Kahng, N. Muralimanohar, A. Shafiee, and V. Srinivas, "CACTI 7: New Tools for Interconnect Exploration in Innovative Off-Chip Memories," *ACM Trans. Archit. Code Optim.*, vol. 14, no. 2, pp. 14:1–14:25, 2017.

[10] M. Becker and S. Chakraborty, "Measuring Software Performance on Linux," *CoRR*, vol. abs/1811.01412, 2018.

[11] N. Beckmann and D. Sánchez, "Jigsaw: Scalable software-defined caches," in *Proc. 22nd Int. Conf. Parallel Architecture and Compilation Techniques (PACT)*, 2013, pp. 213–224.

[12] Z. Cai, S. M. Blackburn, and M. D. Bond, "Understanding and Utilizing Hardware Transactional Memory Capacity," in *Proc. 2021 ACM SIGPLAN Int. Symp. on Memory Management (STMM)*, 2021.

[13] H. W. Cain, M. M. Michael, B. Frey, C. May, D. Williams, and H. Q. Le, "Robust architectural support for transactional memory in the power architecture," in *Proc. 40th Int. Symp. Comp. Architecture (ISCA)*, 2013, pp. 225–236.

[14] L. Ceze, J. Tuck, J. Torrellas, and C. Cascaval, "Bulk Disambiguation of Speculative Threads in Multiprocessors," in *Proc. 33rd Int. Symp. Comp. Architecture (ISCA)*, 2006, pp. 227–238.

[15] A. Chang and M. F. Mergen, "801 Storage: Architecture and Programming," *ACM Trans. Comput. Syst.*, vol. 6, no. 1, pp. 28–50, 1988.

[16] S. Chaudhry, R. Cypher, M. Ekman, M. Karlsson, A. Landin, S. Yip, H. Zeffer, and M. Tremblay, "Rock: A High-Performance Sparc CMT Processor," *IEEE Micro*, vol. 29, no. 2, pp. 6–16, 2009.

[17] D. Christie, J.-W. Chung, S. Diestelhorst, M. Hohmuth, M. Pohlack, C. Fetzer, M. Nowack, T. Riegel, P. Felber, P. Marlier, and E. Riviere, "Evaluation of AMD's advanced synchronization facility within a complete transactional memory stack," in *Proc. 2010 EuroSys Conf.*, 2010, pp. 27–40.

[18] W. Chuang, S. Narayanasamy, G. Venkatesh, J. Sampson, M. V. Biesbrouck, G. Pokam, B. Calder, and O. Colavin, "Unbounded page-based transactional memory," in *Proc. 12th Int. Conf. Architectural Support for Prog. Languages and Operating Sys. (ASPLOS-XII)*, 2006, pp. 347–358.

[19] J. Chung, C. C. Minh, A. McDonald, T. Skare, H. Chafi, B. D. Carlstrom, C. Kozyrakis, and K. Olukotun, "Tradeoffs in transactional memory virtualization," in *Proc. 12th Int. Conf. Architectural Support for Prog. Languages and Operating Sys. (ASPLOS-XII)*, 2006, pp. 371–381.

[20] C. Click, "The Azul Hardware Transactional Memory Experience," Hydra Distributed Computing Conference talk. https://2019.hydraconf.com/2019/talks/2jix5mst7iduyp9linqhfj/, 2019.

[21] I. Corporation, "Intel Architecture Instruction Set Extensions and Future Features Programming Reference," 2020.

[22] J. R. Crandall and F. T. Chong, "Minos: Control Data Attack Prevention Orthogonal to Memory Model," in *Proc. 37th Annu. IEEE/ACM Int. Symp. Microarchitecture (MICRO)*, 2004, pp. 221–232.

[23] B. Cuesta, A. Ros, M. E. Gómez, A. Robles, and J. Duato, "Increasing the effectiveness of directory caches by deactivating coherence for private memory blocks," in *Proc. 38th Int. Symp. Comp. Architecture (ISCA)*, 2011, pp. 93–104.

[24] L. Dalessandro, M. F. Spear, and M. L. Scott, "NOrec: streamlining STM by abolishing ownership records," in *Proc. 15th ACM SIGPLAN Symp. Princ. and Practice of Parallel Prog. (PPoPP)*, 2010, pp. 67–78.

[25] M. Dalton, H. Kannan, and C. Kozyrakis, "Raksha: a flexible information flow architecture for software security," in *Proc. 34th Int. Symp. Comp. Architecture (ISCA)*, 2007, pp. 482–493.

[26] P. Damron, A. Fedorova, Y. Lev, V. Luchangco, M. Moir, and D. Nussbaum, "Hybrid transactional memory," in *Proc. 12th Int. Conf. Architectural Support for Prog. Languages and Operating Sys. (ASPLOS-XII)*, 2006, pp. 336–346.

[27] M. M. Das, G. Southern, and J. Renau, "Section based program analysis to reduce overhead of detecting unsynchronized thread communication," in *Proc. 20th ACM SIGPLAN Symp. Princ. and Practice of Parallel Prog. (PPoPP)*, 2015, pp. 283–284.

[28] R. Filipe, S. Issa, P. Romano, and J. Barreto, "Stretching the capacity of hardware transactional memory in IBM POWER architectures," in *Proc. 24th ACM SIGPLAN Symp. Princ. and Practice of Parallel Prog. (PPoPP)*, 2019, pp. 107–119.

[29] L. Hammond, V. Wong, M. K. 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 *Proc. 31st Int. Symp. Comp. Architecture (ISCA)*, 2004, pp. 102–113.

[30] N. Hardavellas, M. Ferdman, B. Falsafi, and A. Ailamaki, "Reactive NUCA: near-optimal block placement and replication in distributed caches," in *Proc. 36th Int. Symp. Comp. Architecture (ISCA)*, 2009, pp. 184–195.

[31] T. Harris, J. R. Larus, and R. Rajwar, *Transactional Memory, 2nd edition*, ser. Synthesis Lectures on Computer Architecture. Morgan & Claypool Publishers, 2010.

[32] T. Harris, M. Plesko, A. Shinnar, and D. Tarditi, "Optimizing memory transactions," in *Proc. ACM SIGPLAN 2006 Conf. Prog. Language Design and Implementation (PLDI)*, 2006, pp. 14–25.

[33] W. Hasenplaugh, A. Nguyen, and N. Shavit, "Quantifying the Capacity Limitations of Hardware Transactional Memory," *7th Workshop on the Theory of Transactional Memory (WTTM)*, 2015.

[34] M. Herlihy and J. E. B. Moss, "Transactional Memory: Architectural Support for Lock-Free Data Structures," in *Proc. 20th Int. Symp. Comp. Architecture (ISCA)*, 1993, pp. 289–300.

[35] IBM, "Transactional execution debugging," https://www.ibm.com/docs/en/zos/2.1.0?topic=execution-transactional-debugging.

[36] IBM, "POWER9 Processor User's Manual (version 2.0)," 2018.

[37] IBM Blue Gene team, "Design of the IBM Blue Gene/Q Compute Chip," *IBM J. Res. Dev.*, vol. 57, no. 1/2, p. 1, 2013.

[38] S. Imam, J. Zhao, and V. Sarkar, "A Composable Deadlock-Free Approach to Object-Based Isolation," in *21st Int. European Conf. on Parallel and Distributed Computing (Euro-Par)*, 2015, pp. 426–437.

[39] Intel Corporation, "Restricted transactional memory overview," sep 2022. [Online]. Available: https://www.intel.com/content/www/us/en/develop/documentation/cpp-compiler-developer-guide-and-reference/top/compiler-reference/intrinsics/intrinsics-for-avx2/intrinsics-for-tsx/intrinsics-for-restrict-transactional-mem-ops/restricted-transactional-memory-overview.html

[40] S. Issa, P. Felber, A. Matveev, and P. Romano, "Extending Hardware Transactional Memory Capacity via Rollback-Only Transactions and Suspend/Resume," in *Proc. 31st Int. Symp. on Distributed Computing (DISC)*, 2017, pp. 28:1–28:16.

[41] C. Jacobi, T. J. Slegel, and D. F. Greiner, "Transactional Memory Architecture and Implementation for IBM System Z," in *Proc. 45th Annu. IEEE/ACM Int. Symp. Microarchitecture (MICRO)*, 2012, pp. 25–36.

[42] M. C. Jeffrey, V. A. Ying, S. Subramanian, H. R. Lee, J. S. Emer, and D. Sánchez, "Harmonizing Speculative and Non-Speculative Execution in Architectures for Ordered Parallelism," in *Proc. 51st Annu. IEEE/ACM Int. Symp. Microarchitecture (MICRO)*, 2018, pp. 217–230.

[43] A. Joshi, V. Nagarajan, M. Cintra, and S. Viglas, "DHTM: Durable Hardware Transactional Memory," in *Proc. 45th Int. Symp. Comp. Architecture (ISCA)*, 2018, pp. 452–465.

[44] S. Kumar, M. Chu, C. J. Hughes, P. Kundu, and A. D. Nguyen, "Hybrid transactional memory," in *Proc. 11th ACM SIGPLAN Symp. Princ. and Practice of Parallel Prog. (PPoPP)*, 2006, pp. 209–220.

[45] C. Lattner and V. S. Adve, "LLVM: A Compilation Framework for Lifelong Program Analysis & Transformation," in *Proc. 2nd IEEE/ACM Int. Symp. on Code Generation and Optimization (CGO)*, 2004, pp. 75–88.

[46] H. Q. Le, G. L. Guthrie, D. Williams, M. M. Michael, B. Frey, W. J. Starke, C. May, R. Odaira, and T. Nakaike, "Transactional memory support in the IBM POWER8 processor," *IBM J. Res. Dev.*, vol. 59, no. 1, 2015.

[47] A. McDonald, J. Chung, B. D. Carlstrom, C. C. Minh, H. Chafi, C. Kozyrakis, and K. Olukotun, "Architectural Semantics for Practical Transactional Memory," in *Proc. 33rd Int. Symp. Comp. Architecture (ISCA)*, 2006, pp. 53–65.

[48] C. C. Minh, J. Chung, C. Kozyrakis, and K. Olukotun, "STAMP: Stanford Transactional Applications for Multi-Processing," in *Proc. 2008 IEEE Int. Symp. Workload Characterization (IISWC)*, 2008, pp. 35–46.

[49] K. E. Moore, J. Bobba, M. J. Moravan, M. D. Hill, and D. A. Wood, "LogTM: log-based transactional memory," in *Proc. 12th IEEE Symp. High-Perf. Comp. Architecture (HPCA)*, 2006, pp. 254–265.

[50] T. Nakaike, R. Odaira, M. Gaudet, M. M. Michael, and H. Tomari, "Quantitative comparison of hardware transactional memory for Blue Gene/Q, zEnterprise EC12, Intel Core, and POWER8," in *Proc. 42nd Int. Symp. Comp. Architecture (ISCA)*, 2015, pp. 144–157.

[51] S. Park, C. J. Hughes, and M. Prvulovic, "Transactional pre-abort handlers in hardware transactional memory," in *Proc. 27th Int. Conf. Parallel Architecture and Compilation Techniques (PACT)*, 2018, pp. 33:1–33:11.

[52] S. Park, M. Prvulovic, and C. J. Hughes, "PleaseTM: Enabling transaction conflict management in requester-wins hardware transactional memory," in *Proc. 22nd IEEE Symp. High-Perf. Comp. Architecture (HPCA)*, 2016, pp. 285–296.

[53] R. Rajwar, M. Herlihy, and K. K. Lai, "Virtualizing Transactional Memory," in *Proc. 32nd Int. Symp. Comp. Architecture (ISCA)*, 2005, pp. 494–505.

[54] J. Renau, B. Fraguela, J. Tuck, W. Liu, M. Prvulovic, L. Ceze, S. Sarangi, P. Sack, K. Strauss, and P. Montesinos, "SESC simulator," January 2005, http://sesc.sourceforge.net.

[55] A. Ros and S. Kaxiras, "Complexity-effective multicore coherence," in *Proc. 21st Int. Conf. Parallel Architecture and Compilation Techniques (PACT)*, 2012, pp. 241–252.

[56] T. Shpeisman, V. Menon, A.-R. Adl-Tabatabai, S. Balensiefer, D. Grossman, R. L. Hudson, K. F. Moore, and B. Saha, "Enforcing isolation and ordering in STM," in *Proc. ACM SIGPLAN 2007 Conf. Prog. Language Design and Implementation (PLDI)*, 2007, pp. 78–88.

[57] A. Singh, S. Narayanasamy, D. Marino, T. D. Millstein, and M. Musuvathi, "End-to-end sequential consistency," in *Proc. 39th Int. Symp. Comp. Architecture (ISCA)*, 2012, pp. 524–535.

[58] E. Spertus, S. C. Goldstein, K. E. Schauser, T. von Eicken, D. E. Culler, and W. J. Dally, "Evaluation of Mechanisms for Fine-Grained Parallel Programs in the J-Machine and the CM-5," in *Proc. 20th Int. Symp. Comp. Architecture (ISCA)*, 1993, pp. 302–313.

[59] B. Steensgaard, "Points-to Analysis in Almost Linear Time," in *Proc. 23rd ACM SIGPLAN Symp. Princ. of Prog. Languages (POPL)*, 1996, pp. 32–41.

[60] G. E. Suh, J. W. Lee, D. Zhang, and S. Devadas, "Secure program execution via dynamic information flow tracking," in *Proc. 11st Int. Conf. Architectural Support for Prog. Languages and Operating Sys. (ASPLOS-XI)*, 2004, pp. 85–96.

[61] Y. Sui and J. Xue, "SVF: Pointer Analysis for C and C++," https://svf-tools.github.io/SVF/.

[62] D. Sánchez, L. Yen, M. D. Hill, and K. Sankaralingam, "Implementing Signatures for Transactional Memory," in *Proc. 40th Annu. IEEE/ACM Int. Symp. Microarchitecture (MICRO)*, 2007, pp. 123–133.

[63] Transaction Processing Performance Council, "TPC Benchmark C," http://www.tpc.org/tpc_documents_current_versions/pdf/tpc-c_v5.11.0.pdf, 2010.

[64] P.-A. Tsai, N. Beckmann, and D. Sánchez, "Jenga: Software-Defined Cache Hierarchies," in *Proc. 44th Int. Symp. Comp. Architecture (ISCA)*, 2017, pp. 652–665.

[65] P.-A. Tsai, C. Chen, and D. Sánchez, "Adaptive Scheduling for Systems with Asymmetric Memory Hierarchies," in *Proc. 51st Annu. IEEE/ACM Int. Symp. Microarchitecture (MICRO)*, 2018, pp. 641–654.

[66] N. Vijaykumar, A. Jain, D. Majumdar, K. Hsieh, G. Pekhimenko, E. Ebrahimi, N. Hajinazar, P. B. Gibbons, and O. Mutlu, "A Case for Richer Cross-Layer Abstractions: Bridging the Semantic Gap with Expressive Memory," in *Proc. 45th Int. Symp. Comp. Architecture (ISCA)*, 2018, pp. 207–220.

[67] N. Vijaykumar, A. Olgun, K. Kanellopoulos, F. N. Bostanci, H. Hassan, M. Lotfi, P. B. Gibbons, and O. Mutlu, "MetaSys: A Practical Open-source Metadata Management System to Implement and Evaluate Cross-layer Optimizations," *ACM Trans. Archit. Code Optim.*, vol. 19, no. 2, pp. 26:1–26:29, 2022.

[68] C. Villavieja, V. Karakostas, L. Vilanova, Y. Etsion, A. Ramírez, A. Mendelson, N. Navarro, A. Cristal, and O. S. Unsal, "DiDi: Mitigating the Performance Impact of TLB Shootdowns Using a Shared TLB Directory," in *Proc. 20th Int. Conf. Parallel Architecture and Compilation Techniques (PACT)*, 2011, pp. 340–349.

[69] E. Witchel, J. Cates, and K. Asanovic, "Mondrian memory protection," in *Proc. 10th Int. Conf. Architectural Support for Prog. Languages and Operating Sys. (ASPLOS-X)*, 2002, pp. 304–316.

[70] J. Woodruff, R. N. M. Watson, D. Chisnall, S. W. Moore, J. Anderson, B. Davis, B. Laurie, P. G. Neumann, R. M. Norton, and M. Roe, "The CHERI capability model: Revisiting RISC in an age of risk," in *Proc. 41st Int. Symp. Comp. Architecture (ISCA)*, 2014, pp. 457–468.

[71] L. Yen, S. C. Draper, and M. D. Hill, "Notary: Hardware techniques to enhance signatures," in *Proc. 41st Annu. IEEE/ACM Int. Symp. Microarchitecture (MICRO)*, 2008, pp. 234–245.

[72] R. M. Yoo, C. J. Hughes, K. Lai, and R. Rajwar, "Performance evaluation of Intel® transactional synchronization extensions for high-performance computing," in *Proc. 2013 ACM/IEEE Conf. Supercomputing (SC)*, 2013, pp. 19:1–19:11.

[73] N. Zeldovich, H. Kannan, M. Dalton, and C. Kozyrakis, "Hardware Enforcement of Application Security Policies Using Tagged Memory," in *Proc. 8th Symp. Operating Sys. Design and Implementation (OSDI)*, 2008, pp. 225–240.

[74] C. B. Zilles and L. Baugh, "Extending hardware transactional memory to support non-busy waiting and non-transactional actions," in *Proc. of the First ACM SIGPLAN Workshop on Languages, Compilers, and Hardware Support for Transactional Computing (TRANSACT)*, 2006.