Understanding and Utilizing Hardware Transactional Memory Capacity

Zixian Cai
zixian.cai@anu.edu.au
Australian National University
Canberra, ACT, Australia

Stephen M. Blackburn
steve.blackburn@anu.edu.au
Australian National University
Canberra, ACT, Australia

Michael D. Bond
mikebond@cse.ohio-state.edu
Ohio State University
Columbus, OH, USA

Abstract

Hardware transactional memory (HTM) provides a simpler programming model than lock-based synchronization. However, HTM has limits that mean that transactions may suffer costly capacity aborts. Understanding HTM capacity is therefore critical. Unfortunately, crucial implementation details are undisclosed. In practice HTM capacity can manifest in puzzling ways. It is therefore unsurprising that the literature reports results that appear to be highly contradictory, reporting capacities that vary by nearly three orders of magnitude. We conduct an in-depth study into the causes of HTM capacity aborts using four generations of Intel's Transactional Synchronization Extensions (TSX). We identify the apparent contradictions among prior work, and shed new light on the causes of HTM capacity aborts. In doing so, we reconcile the apparent contradictions. We focus on how replacement policies and the status of the cache can affect HTM capacity.

One source of surprising behavior appears to be the cache replacement policies used by the processors we evaluated. Both invalidating the cache and warming it up with the transactional working set can significantly improve the read capacity of transactions across the microarchitectures we tested. A further complication is that a physically indexed LLC will typically yield only half the total LLC capacity. We found that methodological differences in the prior work led to different warmup states and thus to their apparently contradictory findings. This paper deepens our understanding of how the underlying implementation and cache behavior affect the apparent capacity of HTM. Our insights on how to increase the read capacity of transactions can be used to optimize HTM applications, particularly those with large read-mostly transactions, which are common in the context of optimistic parallelization.

ISMM '21, June 22, 2021, Virtual, Canada

ACM ISBN 978-1-4503-8448-3/21/06.
https://doi.org/10.1145/3459898.3463901

CCS Concepts: • **Computer systems organization** → **Parallel architectures**; • **Software and its engineering** → *Memory management*; *Concurrency control.*

Keywords: hardware transactional memory, CPU caches, Intel Transactional Synchronization Extensions

ACM Reference Format:
Zixian Cai, Stephen M. Blackburn, and Michael D. Bond. 2021. Understanding and Utilizing Hardware Transactional Memory Capacity. In *Proceedings of the 2021 ACM SIGPLAN International Symposium on Memory Management (ISMM '21), June 22, 2021, Virtual, Canada.* ACM, New York, NY, USA, 14 pages. https://doi.org/10.1145/3459898.3463901

1 Introduction

Transactional memory (TM) is a concurrency control mechanism that executes a sequence of instructions atomically [22, 24].[1] To achieve this behavior, a TM implementation executes transactions speculatively and detects conflicts, which it resolves by undoing the side effects of aborted transactions.

Much of the prior work on TM is arguably impractical. Software transactional memory (STM) implementations incur high run-time overhead to maintain read and write sets in order to detect conflicts [10, 21, 22, 41]. Hardware transactional memory (HTM), on the other hand, minimizes performance costs by implementing conflict detection and resolution mechanisms in hardware [22, 24]. However, hardware support for unbounded transactions requires substantial changes to cache and memory subsystems [6, 8], and such support has not been commercially implemented.

In recent years, so-called *commodity* HTM has arisen as a compelling alternative to impractical STM and unbounded HTM implementations. Commodity HTM, which the rest of the paper refers to simply as "HTM," is now a pervasive hardware feature, available notably as Intel Transactional Synchronization Extensions (TSX) [43].

Traditionally, HTM provides atomic execution of programmer-specified transactions. While programmers can use HTM directly—with a fallback that ensures progress such as synchronization on a global lock [43]—programmers are more

[1] Transactional memory provides the same semantics as non-durable transactions in the database literature: atomic, consistent, and isolated (ACI) semantics [31].

likely to utilize HTM indirectly: as part of hybrid hardware–software TM [13, 30] or to execute critical sections on the same lock speculatively in parallel [32, 37], for example.

Additionally, HTM provides a variety of potential benefits to software. Prior work has employed HTM for a variety of uses, including implementing concurrent garbage collectors [5, 7, 9, 25, 33, 39], strong memory model enforcement [40], data race detection [45], cache side-channel defense [20], and persistent memory store ordering [17]. In such "nontraditional" contexts, transactions may experience relatively few conflicts, and the system developer can craft transactions to avoid writes as much as possible. In other words, the capability of many such transactions is bounded by the read set size. Compared with the traditional use of providing atomicity of programmer-specified regions, maximizing a transaction's read set size is arguably even more important in these nontraditional contexts.

Considering these diverse uses, it is especially critical to understand how to utilize HTM fully. However, practical limits on the size of the supporting hardware structures bound the size of transactions. Transactions that cannot be run within those bounds are aborted. To make effective use of HTM, transactions must avoid exceeding the capacity of the underlying hardware. Unfortunately, the details of how the underlying hardware mechanisms work are both proprietary and subtle, making it difficult to reason about the capacity of the hardware and therefore difficult to make use of HTM. The behavior of HTM is subtle because its implementations exploit existing cache structures whose state is dependent not only on reads and writes that occurred prior to the transaction, but on the cache replacement policy, which is proprietary and not exactly LRU.

These factors have no doubt contributed to confusion in the literature. For example, studies of the maximum transaction *read capacity* on the Intel Core i7-4770 vary by nearly three orders of magnitude, from 16 KB [38] to 7.5 MB [14]. In Section 2.4 we note nine different studies of Intel's TSX, none of which appear to be consistent. This lack of clarity undermines the ability of developers using HTM to reason about the performance impact of their design choices, particularly in the aforementioned nontraditional contexts.

This paper makes the following key contributions:

- We identify significant contradictions among existing empirical evaluations of HTM capacity.
- We conduct an in-depth study of HTM behavior across four generations of Intel's Transactional Synchronization Extension (TSX) hardware: Haswell, Broadwell, Skylake, and Coffee Lake. We use and extend prior methodologies for evaluating transaction capacity, building on Ritson and Barnes [38].
- We reproduce prior findings, and in doing so, reconcile their vastly different observations, shedding new light on HTM behavior.
- We confirm and refine prior understanding of how TSX is implemented [20] and how this relates to Intel's cache replacement policies, and make actionable findings on how to maximize HTM capacity.

We find that TSX write capacity is closely tied to L1 residency, consistent with prior studies. We find that read capacity is closely tied to LLC *residency* (not LLC size). We observe that LLC cache replacement policies can result in LLC evictions for surprisingly small read sets, depending on the prior state of the cache. This insight reconciles prior work that suggested that TSX read capacity was as small as the L1 size with work that suggested that it was as large as the LLC size. We find that read set capacity can be maximized by mitigating LLC evictions, either by flushing the cache or warming the cache prior to starting the transaction.

These findings will help programmers develop performant HTM code on modern hardware. Furthermore, by reconciling, clarifying, and extending prior methodologies, they create a solid foundation for future investigations of other HTM implementations.

The code we used in this work is publicly available: https://github.com/caizixian/rtm-bench

2 Background and Related Work

This section motivates both the challenge and importance of understanding how transactional memory is implemented. We first overview transactional memory and related processor cache mechanisms, before describing contradictory results reported by prior work, showing the need for a deeper understanding of how transactional memory is implemented.

2.1 Hardware Transactional Memory

Transactional memory (TM) executes code sections automatically using optimistic parallelism: Marked sections of code execute as *transactions*, and the TM system detects conflicts between transactions, which it resolves by aborting one or more transactions and rolling back their effects [22, 24]. Software TM (STM) detects and resolves conflicts in software, permitting support for unbounded transactions and sophisticated conflict resolution policies, but STM adds high run-time overhead [10, 21, 41]. Hardware TM (HTM), on the other hand, detects and resolves conflicts in hardware, which achieves low overhead but either requires complex hardware or places limits on transactional execution [6, 8, 24, 43].

HTM implementations typically track a transaction's read and write sets in processor caches, and piggyback on pervasive eager-invalidation-based cache coherence protocols (e.g., MESI and its variants) to detect conflicts. HTM similarly uses caches to maintain a transaction's modified state; it aborts a transaction by invalidating cache lines modified by the transaction without writing them back.

Unbounded HTM requires substantial hardware structures and logic to maintain read and write sets and to detect conflicts even for data that has been evicted from caches [6, 8]. In contrast, so-called *commodity* HTM performs *best-effort* tracking of transactional state, aborting transactions that overflow the cache(s) that track transactional state.

2.2 Intel Transactional Synchronization Extensions

Intel added HTM to its processors starting in 2013, accessible through ISA extensions called *Transactional Synchronization Extensions* (TSX). TSX consists of two programming models: *Hardware Lock Elision* (HLE) for speculative lock elision [37] and *Restricted Transactional Memory* (RTM) for general-purpose transactions. While we focus on RTM in this paper, our results should apply to HLE as well since they use the same microarchitectural implementation.

RTM provides `XBEGIN` and `XEND` instructions to demarcate transactions.[2] `XBEGIN` takes a parameter that is the address of an abort handler. In the event of an abort, all changes to the memory and architectural registers are reversed, control is transferred to the specified abort handler, and `eax` is set to a code that indicates the cause of the abort.

RTM is best effort and provides no progress guarantee, so programmers or runtime developers must provide a fallback mechanism to handle transactions that repeatedly abort. Two common fallbacks are synchronization on a single global lock (e.g., [43]) and STM execution in the context of hybrid software–hardware TM [11, 13, 30]. RTM transactions abort for the following reasons [28, Vol. 1: Chapters 16.3.5, 16.3.8]:

- Conflicts with other transactions or non-transactional instructions. These aborts are essential for ensuring atomicity of transactions.
- Synchronous conditions such as instructions that are illegal in transactions (e.g., system calls), debug breakpoints, and page faults. Programmers or language/runtime implementers should avoid these conditions.
- Asynchronous events such as timer interrupts. While longer transactions run an increasing risk of such an abort, in practice internal buffer overflows are the bigger threat to transaction progress.
- An internal buffer overflow, including eviction of a cache line accessed by the transaction.

We focus on the last abort cause because it is critical for executing large transactions (transactions with large read or write sets), and prior work is ambiguous about the maximum size of transactions that can commit, as well as guidance for *how* to execute large transactions successfully. Next we explain the connection between cache evictions and aborts.

[2]RTM also provides instructions for aborting a transaction explicitly and testing whether the core is executing a transaction, but we do not use them in this paper.

2.3 Cache Capacity and Transaction Aborts

HTM conflict detection typically piggybacks on existing processor cache mechanisms (Section 2.1), and cannot track data once it departs the cache. For example, suppose commodity HTM tracks the read set in the L1 and L2 caches and the write set in the L1 cache. Then a transaction must abort if a line written by the transaction is evicted from the L1, or if a line read by the transaction is evicted from the L2.

Hasenplaugh et al. showed experimentally that transaction capacity is limited not only by cache capacity but by the capacity of each associative set [23]. That is, if a line is evicted from the last cache level that tracks transactional state, the transaction will abort, regardless of whether the eviction was due to a capacity miss or a conflict miss.

A complicating factor is cache replacement policies, which select lines to evict in set-associative caches. Since perfect *least recently used* (LRU) is computationally expensive, modern processors use a *pseudo-LRU* algorithm. Reverse engineering [1–3] indicates that the processors we evaluate use tree-based pseudo-LRU [4] in the L1 and L2 caches, and variations on re-reference interval prediction (RRIP) [29] including quad-age LRU (QLRU) and Bimodal RRIP (BRRIP) [29] in their LLCs. Because the LLC replacement policies are so important to HTM read set capacity, we describe the policies used in the hardware we evaluate in more detail.

In QLRU, each cache line has a two-bit re-reference prediction value (RRPV) where a value of 0 indicates imminent reuse, and a value of 3 indicates distant reuse. When a miss occurs, QLRU chooses a line with an RRPV of 3 as the victim; to break a tie, it chooses the leftmost line with an RRPV of 3. If no line has an RRPV of 3, all lines' RRPVs are incremented until one reaches 3. QLRU instantiates new lines with an RRPV of 1, and on a hit, sets the RRPV to 0 for lines with an RRPV of 1, or to 1 if the RRPV was 2 or 3. The left half of Figure 1 shows the operation of QLRU on an 8-way set with a sequence of ten reads to eight unique lines. The red boxes indicate the line that is due to be evicted next. In this small example, QLRU gives LRU behavior.

BRRIP is a variation on QLRU that gives new lines an RRPV of either 3 or 1. The rationale for using an RRPV of 3 is that it protects the cache from displacement caused by streaming workloads [36]. The right half of Figure 1 shows BRRIP with one out of every four instantiations receiving an RRPV of 1 and the remainder receiving 3. Unlike QLRU, BRRIP will frequently evict recently accessed lines in streaming workloads. Notice that for this workload the most recently used cache line is evicted 7 out of 10 times.

The processors we evaluate use different combinations of QLRU and BRRIP [2]. Haswell and Broadwell both use an adaptive technique called set dueling, where 64 sets each are dedicated to QLRU and BRRIP, respectively, and the remainder of the sets (followers) adaptively switch between the two policies according to which one is working most

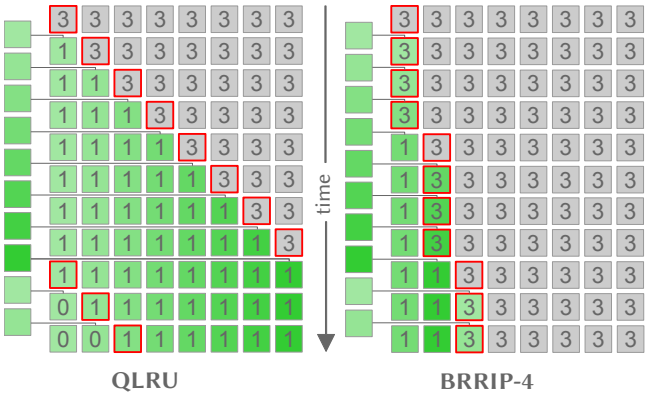


Figure 1. QLRU and BRRIP-4 replacement policies [29, 36] used in the LLC of every microarchitecture we evaluate. We show an 8-way set with a sequence of 10 accesses to 8 distinct cache lines. Numbers indicate the two-bit RRPV for each line. The illustration shows the moment prior to each access. The red box shows the line due to be evicted next—the leftmost line with the highest RRPV. QLRU sets the RRPV to 1 for new lines, while BRRIP-4 only sets it to 1 for one in four new lines, and to 3 otherwise. In this example, QLRU does not evict any new lines while BRRIP-4 evicts 7/10 new lines.

effectively. Haswell and Broadwell appear to use BRRIP-16 (1/16 instantiations receive an RRPV of 1) [3]. Skylake and Coffee Lake have 64 and 128 of their sets respectively implementing QLRU and the remainder appear to use an adaptive variation of BRRIP that only occasionally places lines in the most-recently-used (MRU) position.

A further complication is the hash function used by the hardware to map addresses to sets. In the simple case, a cache with 2^L lines arranged into 2^S sets uses S bits in the address as the hash. In the best case, a process can have a footprint the same as the cache capacity and see no capacity misses because the data maps uniformly to the 2^S sets. In the worst case, the data maps to a single set, leading to an effective cache capacity of just $1/2^S$ the actual capacity. When a cache is virtually addressed and the program has good spatial locality, the cache will be well utilized. However, when the cache is physically addressed (and small pages are used) the mapping of addresses to sets will tend to be effectively random. As Hasenplaugh et al. note, this setup, together with the theoretical result by Gonnet [19], tells us that for a given process we should expect the effective capacity of a physically indexed cache to be about half of its actual capacity [23].

2.4 Related Work

The rest of this section surveys related work that has reported on the transaction capacity of Intel TSX implementations. The point of this comparison is to identify significant inconsistencies in the results and develop hypotheses based on the methodologies and configurations used by prior work that might explain the apparent contradictions.

Table 1 summarizes the transaction capacities reported by prior work. The prior work agrees that write set capacity is bounded by the L1 data cache size: The reported write set capacities range from 22 KB [34] to 31 KB [42]. However, there is wild disagreement about the effective read set capacity of Intel TSX transactions: The reported read set capacities range from 22 **KB** [34] to 7.5 **MB**—both on the Core i7-4770 (Haswell microarchitecture).

Based on their results, Ritson and Barnes [38] and Diegues et al. [15] surmised that transactional work tracking is performed only in the L1 data cache. In contrast, Dice et al. [14] stated that the CPU tracks the read set in the L3 cache, but the write set is tracked in the L1 cache, while Gruss et al. [20] speculated that read set tracking uses a probabilistic structure, such as a Bloom filter.

Yoo et al. of Intel make it clear that write state is tracked (only) in the L1 cache:

> The first implementation of Intel TSX on the 4th Generation Core™ microarchitecture uses the first level (L1) data cache to track transactional states. All tracking and data conflict detection are done at the granularity of a cache line, using physical addresses and the cache coherence protocol. Eviction of a transactionally written line from the data cache will cause a transactional abort. However, evictions of lines that are only transactionally read do not cause an abort; they are moved into a secondary structure for tracking, and may result in an abort at some later time. [43, §2]

Their description about the handling of lines read during a transaction is opaque, but not inconsistent with findings that the read set is bounded by the LLC size—it appears to refer to L1 evictions and is ambiguous about what happens after that. Relatedly, the Intel optimization manual states that "[n]ewer microarchitectures are expected to have an improved second-level structure that tracks evicted read set addresses" [27, Chapter 16.2.4.2].

Most of the prior work constructed microbenchmarks that reused the same memory region for testing transactions of different sizes [20, 23, 34, 42].[3] While that work reports relatively high read capacities, we show that these results are likely due to reusing memory across transactions and retrying failed transactions many times—and thus high read capacity may not translate automatically to real workloads. In particular, Ritson and Barnes [38] explicitly chose distinct memory areas for different transactions, aiming to minimize the effects of L2 and LLC processor caches, and reported

[3] Some prior work does not describe execution configuration details [14, 18], while other work used transactional memory benchmarks instead of microbenchmarks [15, 35].

Table 1. Maximum observed read and write capacities (last two columns) reported in prior work, relative to the L1 or LLC size. For Diegues et al. [15] and Pereira et al. [35], the reported maximum capacity is for transactions that perform a mix of reads and writes, since their evaluations used transactional benchmark suites (STAMP [12] and Eigenbench [26]).

	Year	Architecture	Model	L1	LLC	Write	Read
Ritson and Barnes [38]	2013	Haswell	Core i7-4770	32 KB	8 MB	0.81 × L1	0.81 × L1
Diegues et al. [15]	2014	Haswell	Xeon E3-1275 v3	32 KB	8 MB	1.0 × L1	
Goel et al. [18]	2014	Haswell	Core i7-4770	32 KB	8 MB	0.50 × L1	0.50 × LLC
Pereira et al. [35]	2014	Haswell	Xeon E3-12?? v3	32 KB	8 MB	1.0 × L2	
Wang et al. [42]	2014	Haswell	Core i7-4770	32 KB	8 MB	0.97 × L1	0.50 × LLC
Dice et al. [14]	2015	Haswell	Core i7-4770	32 KB	8 MB	—	0.94 × LLC
Hasenplaugh et al. [23]	2015	Haswell	Core i7-4770	32 KB	8 MB	0.78 × L1	0.58 × LLC
Nakaike et al. [34]	2015	Haswell	Core i7-4770	32 KB	8 MB	0.69 × L1	0.50 × LLC
Gruss et al. [20]	2017	Skylake	Core i7-6600U	32 KB	4 MB	1.00 × L1	1.00 × LLC

dramatically lower read capacity. We use configurations that represent both approaches, and also introduce new configurations for executing transactions (Section 3).

3 Analyzing HTM Capacity

To investigate the discrepancies in HTM capacity reported by prior work, we set out to reproduce their results, make a hypothesis about the causes of the discrepancies, and identify *configurations* (i.e., run-time techniques) for maximizing HTM capacity. Prior work has agreed that write set capacity is bounded by the L1 size, so we focus on read set capacity, which prior work reports as varying between the L1 and LLC sizes. (We also evaluated write set capacity using our new configurations for maximizing HTM capacity and, consistent with prior work, found that write set capacity is bounded by L1 size. For brevity we omit these results from the paper.)

By reproducing the results of prior work, we find that a key issue is whether transactions access memory that was accessed by prior attempted transactions (regardless of whether they aborted or committed). From these results, we hypothesize that the LLC replacement policy evicts lines read by the transaction, in some cases, *before* cached lines that were last accessed earlier. We test this hypothesis by developing two new configurations—invalidating the entire cache and "warming up" the cache before each transaction—and find that they both increase apparent HTM read capacity up to 91–97% of LLC capacity. These results both validate our hypothesis and show that the configurations are promising avenues for maximizing HTM capacity in real-world settings.

Execution methodology. We extended `rtm-bench`, developed by Ritson and Barnes [38].[4,5] The benchmark evaluates the success rates of simple read- and write-only transactions. A read-only transaction of a given size, for example, accesses every 64-bit word in address order consecutively in a cache-line-aligned contiguous region of memory of the target size, all inside of an RTM transaction. To establish HTM capacity, each execution of the benchmark explores the success rate of a range of sizes, yielding a success rate curve (see Section 4). The HTM capacity is evaluated to be the largest transaction size for which `htm-bench` reports a nonzero success rate.

Each time `rtm-bench` is invoked, it acquires a 512 MB region of memory via `mmap`. As we will see later, it is noteworthy that while `mmap` assures that this region will be virtually contiguous, the region will generally be physically discontiguous (assuming pages are small, i.e., 4 KB). For each transaction size, `rtm-bench` runs N consecutive transactions, where N is the size of the region divided by the transaction size (e.g., at size 2 MB, $N = \frac{512\,\text{MB}}{2\,\text{MB}} = 256$), with an upper bound of $N = 2^{17}$. Each transaction uses memory that *has not been used* by any previous transaction of that size. We refer to this procedure as the **Baseline** configuration.

We extended `rtm-bench` to support three other configurations that affect the potential HTM capacity. **Reuse** reuses the same memory for all transactions of a given size and treats N as an independent variable. **Invalidation** invalidates all levels of the cache[6] before each transaction. **Warmup** repeatedly reads the memory that will be read by the transaction, 128 times for Haswell and 5 times for the other architectures,[7] prior to the transaction being executed.

We also extended `rtm-bench` to step the transaction sizes logarithmically rather than linearly. This was necessary as we explored transaction sizes that were orders of magnitude larger than those explored by Ritson and Barnes [38].

[4]The original `rtm-bench`: https://github.com/perlfu/rtm-bench
[5]Our code that extends `rtm-bench`: https://github.com/caizixian/rtm-bench

[6]We invalidate all lines in all caches using the x86 `wbinvd` instruction. Since `wbinvd` is privileged, we use a kernel module to invoke it (https://github.com/batmac/wbinvd). For the **Invalidation** configuration, we use an upper bound of $N = 2^{12}$ because the time taken to invalidate the caches would make larger bounds impractical to evaluate.
[7]We empirically determined the number of iterations at which more iterations provides diminishing returns (Section 4.2).

Table 2. Machines used in the evaluation. Each column describes an evaluation machine, which we refer to using its microarchitecture name (column header).

	Haswell	Broadwell	Skylake	Coffee Lake
Model	Core i7-4770	Xeon D-1540	Core i7-6700K	Core i9-9900K
Year	2013	2015	2015	2018
Technology	22 nm	14 nm	14 nm	14 nm
Clock	3.4 GHz	2.0 GHz	4.0 GHz	3.6 GHz
SMT × Cores	2 × 4	2 × 8	2 × 4	2 × 8
L1 Data Cache	32 KB × 4	32 KB × 8	32 KB × 4	32 KB × 8
L2 Cache	256 KB × 4	256 KB × 8	256 KB × 4	256 KB × 8
LLC	8 MB	12 MB	8 MB	16 MB
Sets	8 K	16 K	8 K	16 K
Memory Size	16 GB	16 GB	16 GB	32 GB
Memory Type	DDR3-1600	DDR4-2133	DDR3-1600	DDR4-2133

The **Baseline** configuration represents the methodology of Ritson and Barnes [38], while **Reuse** represents that used by Wang et al. [42], Hasenplaugh et al. [23], Nakaike et al. [34], Gruss et al. [20] (Section 2.4).

We retain all other elements of `rtm-bench`, including pinning of software threads to hardware threads. Ritson and Barnes published results for the Core i7-4770 [38], which we also use, allowing us to validate our measurements.

In each of our experiments, we invoke `rtm-bench` 50 times. For each transaction size, we plot the average success rate and 95% confidence intervals from the 50 trials. We initially found that rebooting each machine prior to each set of experiments led to more consistent and higher apparent capacity. We hypothesized that this was due to the greater likelihood of physical contiguity in the region returned by `mmap` after a reboot, the physical indexing of the LLC, and the consequent impact on effective LLC capacity (Section 2.3). To test our hypothesis, we ran experiments with 1 GB huge page support enabled, so that the virtually contiguous 512 MB region is also physically contiguous (Section 4.3). This removed variability and consistently yielded higher capacities for each configuration, supporting the hypothesis.

Platforms. Table 2 describes the machines used in our evaluation. We use a diverse set of microarchitectures because we expect the implementation of TSX to have evolved over time. On all machines, we enable simultaneous multithreading (i.e., Intel Hyper-Threading) and disable frequency scaling (i.e., Intel Turbo Boost Technology). To minimize experimental noise, we run the experiments in isolation, with as many background daemons disabled as possible.

All machines used in the evaluation run the same disk image. Each machine runs Ubuntu 18.04.5 with kernel version `5.4.0-64-generic` and microcode version `20201110`. We compiled `rtm-bench` with GCC version 7.5.0 with the `-O2` flag.

4 Evaluation

This section presents experimental results using the methodology described in Section 3. Our results using the **Baseline** and **Reuse** configurations reproduce results from prior work. The **Invalidation** and **Warmup** configurations both validate our hypothesis about the LLC replacement policy and serve as techniques for executing transactions with large read sets. Table 3 summarizes our results.

4.1 Reproducing Prior Work's Results

Using the **Baseline** and **Reuse** configurations, we reproduced the prior work's results.

Baseline configuration. Figure 2 shows the results for the **Baseline** configuration. For each microarchitecture, the figure plots a *success rate curve* (a representation introduced by Ritson and Barnes [38]). The success rate curve shows, for each attempted transaction size, the fraction of all transactions that successfully *Committed* and the fraction that experienced an *Overflow abort*. Any remaining fraction (not shown in the plot) is due to transactions that aborted for other reasons—generally asynchronous aborts such as timer interrupts. Each fraction is computed based on the mean of 50 trials of extended `rtm-bench`, each of which executes $\frac{512\,\text{MB}}{\text{txn\_size}}$ transactions (Section 3). The plot shows the variability across the 50 trials using a 95% confidence interval.

Figure 2 and the summary in Table 3 show that for Haswell, transactions larger than 32 KB cannot complete successfully. This reproduces the results of Ritson and Barnes [38]. Note that 32 KB happens to be the size of the L1 cache on all of the machines. However, the other microarchitectures show roughly double (Broadwell) and quadruple (Skylake and Coffee Lake) the read capacity with this methodology, even though all four microarchitectures share the same L1 size, associativity, and replacement policy [3].

These results on Haswell and later microarchitectures are consistent with our hypothesis that the LLC replacement

Table 3. Maximum observed read capacity from 50 trials for each microarchitecture–configuration combination. For each reported range $[l, u)$, l is the size of the maximum-sized committed transaction, and u is the next-highest size attempted (which always aborted). For the **Reuse** configuration, the result reports maximum capacity across all N. The log of the commit frequency for each achieved capacity is shown in a small gray font. For example, -4.9 means that $10^{-4.9}$ of the attempts at that capacity successfully committed. For the **Reuse** configuration, if multiple Ns can achieve the same maximum capacity, we report the frequency for the N with the highest frequency. This table summarizes the detailed results shown in Figures 2–5.

Architecture LLC	Haswell 8 MB	Broadwell 12 MB	Skylake 8 MB	Coffee Lake 16 MB
Baseline/L1	[1.00, 1.04) −4.9	[2.00, 2.07) −5.3	[3.73, 3.86) −5.0	[3.86, 4.00) −5.3
Baseline/LLC	[0.00, 0.00) −4.9	[0.01, 0.01) −5.3	[0.01, 0.02) −5.0	[0.01, 0.01) −5.3
Reuse/LLC	[0.76, 0.78) −1.8	[0.74, 0.77) −2.2	[0.76, 0.78) −1.8	[0.84, 0.87) −2.6
Invalidation/LLC	[0.87, 0.90) −0.7	[0.88, 0.91) −3.1	[0.87, 0.90) −1.5	[0.87, 0.90) −0.9
Warmup/LLC	[0.93, 0.97) −0.8	[0.91, 0.94) −2.1	[0.97, 1.00) −2.7	[0.93, 0.97) −2.8

Table 4. The impact of huge page support. These numbers reflect the same methodology as Table 3, but with huge page support enabled. Improvements relative to Table 3 are shown in green, and the one case of degradation is shown in red. Improvements are most pronounced with **Reuse**.

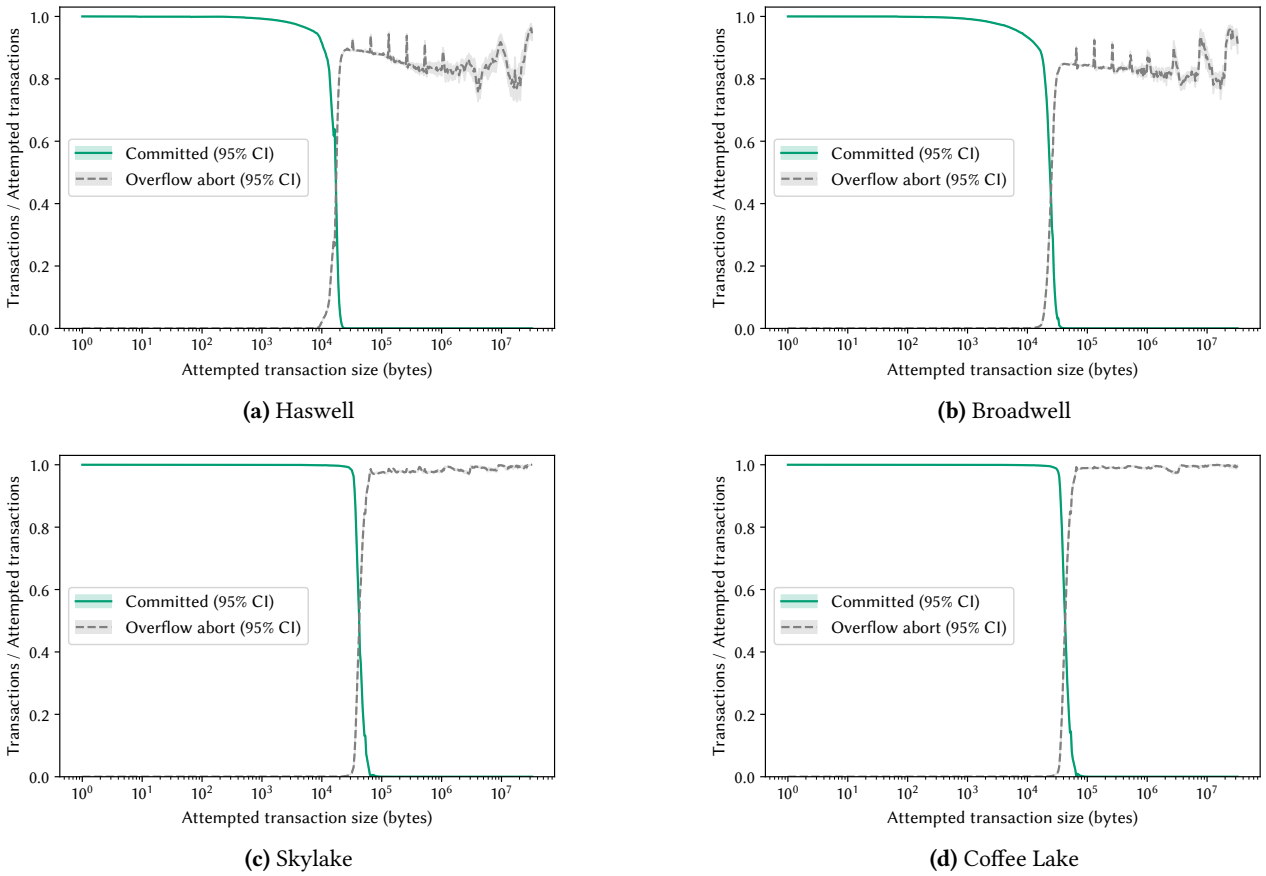
Architecture LLC	Haswell 8 MB	Broadwell 12 MB	Skylake 8 MB	Coffee Lake 16 MB
Baseline/L1	[1.00, 1.04) −4.4	[2.00, 2.07) −5.0	[3.73, 3.86) −5.3	[4.29, 4.44) −4.8
Baseline/LLC	[0.00, 0.00) −4.4	[0.01, 0.01) −5.0	[0.01, 0.02) −5.3	[0.01, 0.01) −4.8
Reuse/LLC	[0.93, 0.97) −0.9	[0.88, 0.91) −2.5	[0.93, 0.97) −1.1	[0.90, 0.93) −1.5
Invalidation/LLC	[0.90, 0.93) −2.2	[0.85, 0.88) −1.2	[0.90, 0.93) −2.2	[0.90, 0.93) −3.2
Warmup/LLC	[0.97, 1.00) −1.6	[0.91, 0.94) −1.4	[0.97, 1.00) −1.3	[0.93, 0.97) −1.2

policy sometimes evicts lines read by the transaction instead of lines in the same associative set that were last accessed before the transaction. A plausible cause for the different transaction read sizes achieved on the microarchitectures is differences in the replacement policy and number of LLC sets—and *not* because of inherent limitations in HTM read capacity, as results in the rest of this section show. Broadwell and Haswell share the same LLC replacement policy, each with 64 dedicated BRRIP sets, which will evict the MRU line with high probability (see Section 2.3). However, our Broadwell is 12-way associative with 16 K sets, while our Haswell is 16-way with 8 K sets [3]. We do not have a concrete explanation for Broadwell seeing twice the effective read set size of Haswell under the **Baseline** configuration. However, we hypothesize that this is due to the fraction of sets dedicated to BRRIP on Broadwell being half that of Haswell. For this explanation to make sense, the follower sets, which account for the majority, would have to be adopting the QLRU policy, which is plausible since although the workload is streaming, it is run in isolation so BRRIP will not lead to a lower miss rate than QLRU. Skylake and Coffee Lake share a slightly different LLC design, where none of the sets are dedicated to BRRIP but most will occasionally instantiate a new line into the MRU state (RRPV=3). This uniformity helps explain why they yield very close to the same capacity despite Coffee Lake having twice as many sets as Skylake.

Reuse configuration. Figure 3 shows success rate plots using the **Reuse** configuration, which corresponds to prior work that repeatedly executed transactions of a given size in the same memory region (Section 3). The **Reuse** configuration's plots differ from the other configurations' plots because **Reuse** treats N (the number of transactions attempted consecutively in the same memory region) as an independent variable. The plots show only the fraction of committed transactions (omitting the fraction of transactions aborted for overflowing capacity) for visual simplicity.

As the figure shows, at $N = 1$, when there is no reuse, the maximum transaction size is not much different than for the **Baseline** configuration. At $N = 4096$, the maximum transaction size achievable is orders of magnitude larger: between 74% and 84% of the LLC size.

These results support our hypothesis about the replacement policy. Repeated accesses to a BRRIP cache set will eventually displace stale data with the transactional data as shown in the right half of Figure 1, where the stale (grey) lines are gradually displaced from the set. This allows the transaction read set to eventually be resident in the cache



(a) Haswell **(b)** Broadwell

(c) Skylake **(d)** Coffee Lake

Figure 2. Success rate curves on different microarchitectures when using our **Baseline** configuration, which corresponds to `rtm-bench` with default parameters [38]. We conduct 50 trials and plot the mean with 95% confidence intervals. Transaction sizes are shown on a log scale. Haswell and Broadwell show capacities of 32 KB and 64 KB, respectively, while Skylake and Coffee Lake show capacities of about 120 KB.

with no misses (or evictions). Each prior aborted transaction makes it less likely that the memory it accessed will be evicted by a future attempted transaction. Even so, it takes many—thousands of—repeated attempted transactions to get close to maximizing HTM capacity. This behavior makes sense because each attempted transaction presumably only accesses one or a few lines more than the prior aborted transaction before being aborted for evicting a line that the transaction already accessed.

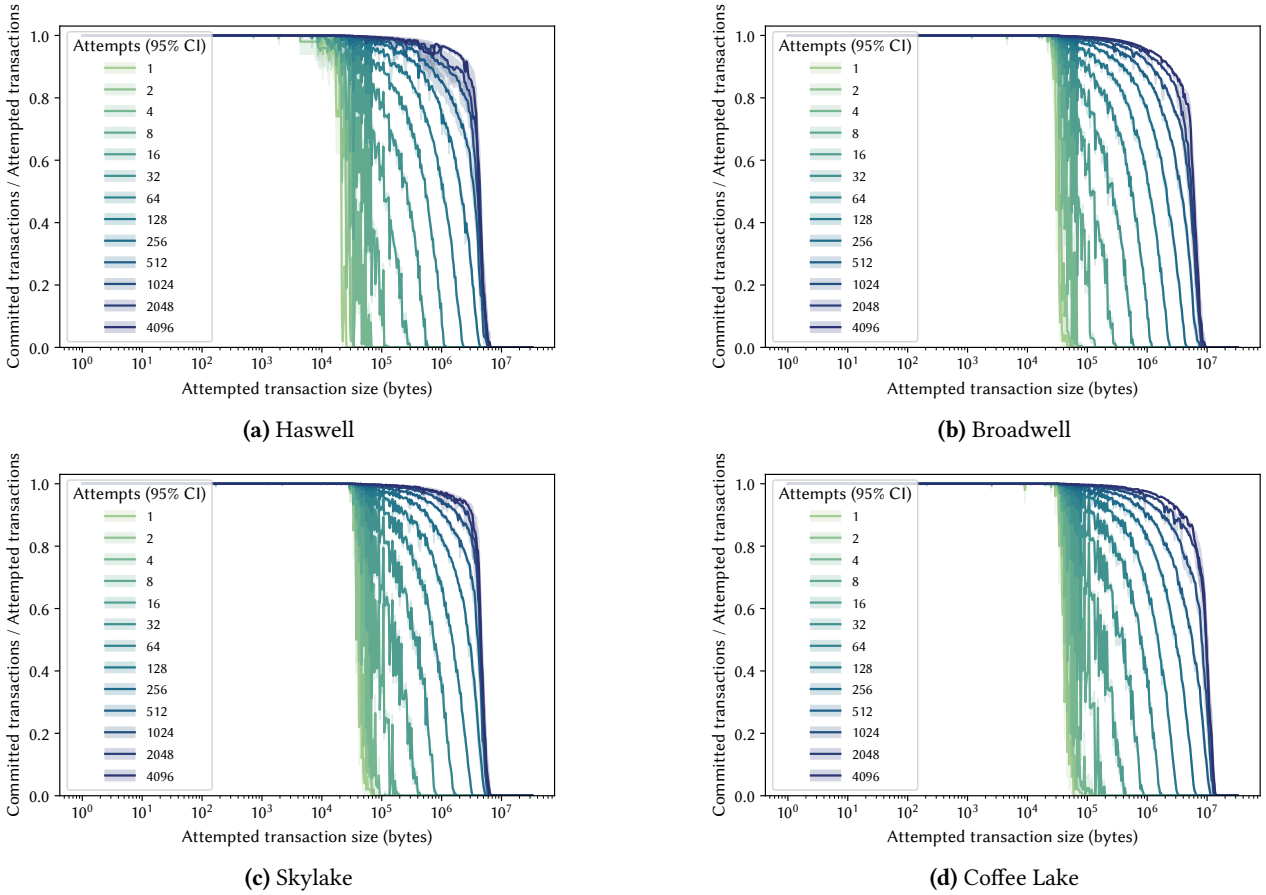
In summary, the results for the **Baseline** and **Reuse** configurations help reproduce and clarify inconsistent results from prior work. Furthermore, these results are consistent with our hypothesis about the impact of cache replacement policies on effective HTM capacity. Next we show results for the **Invalidation** and **Warmup** configurations, which further validate our hypothesis and, in essence, show how to get the benefits of the **Reuse** configuration more fully and efficiently.

4.2 Configurations for Maximizing HTM Capacity

This section evaluates our new configurations that aim to validate our hypothesis about the LLC replacement policy and to provide a technique for maximizing HTM capacity.

Figure 4 shows results for the **Invalidation** configuration, and Figure 5 shows results for the **Warmup** configuration. Table 3 summarizes the capacity results. Both configurations yield transaction read sizes that approach the LLC size.

The ability of both configurations to enable large transactions—larger than the transactions of the **Baseline** configuration or even the **Reuse** configuration with many iterations—is interesting, especially since invalidating and warming the cache are essentially opposite actions. However, both make sense given the behavior of BRRIP. Cache invalidation removes all prior data, allowing the transaction read set to fill the set without evictions until the set is full. Warmup will pre-fill the set, albeit gradually, given the slow fill behavior of BRRIP under such workloads.



(a) Haswell

(b) Broadwell

(c) Skylake

(d) Coffee Lake

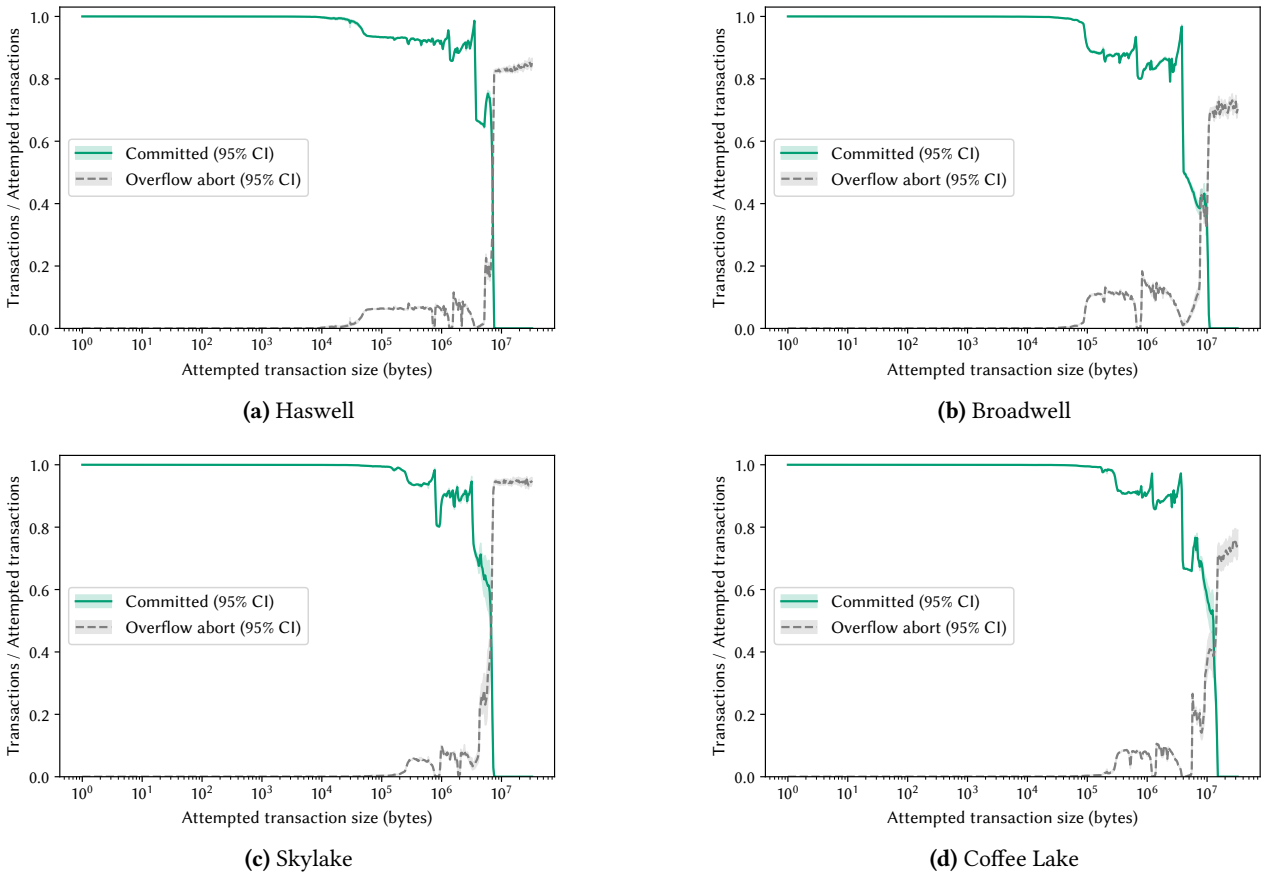
Figure 3. Success rate curves on different microarchitectures when using our **Reuse** configuration, which differs from Baseline by performing all of the N transaction attempts of a given size on the same memory region. We conduct 50 trials and plot the mean with 95% confidence intervals for each N. Transaction sizes are shown on a log scale. Haswell, Broadwell, and Skylake show capacities of about 75% of their LLC (6 MB, 9 MB, and 6 MB respectively) while Coffee Lake shows a capacity of 85% of its LLC (13.4 MB).

Figure 6 presents the results of an experiment to determine how many warmup iterations the **Warmup** configuration uses. On Haswell, diminishing returns are only achieved after 128 warmup iterations, while for the other architectures, returns tailed off after 2 iterations. For the **Warmup** results we present in Figure 5 and Table 2, we use 128 warmup iterations for Haswell and 5 for the other architectures.

4.3 Effects of Physical Indexing of LLC

The results presented so far use the default page size (4 KB). Under these circumstances, the 512 MB virtually contiguous region used by the experiments is unlikely to be be physically contiguous (Section 3), and furthermore, the degree of contiguity is not easy to measure or control since it is a function of the underlying state of the machine at the start of the experiment. Since the LLC is physically indexed, page allocation patterns will affect the probability of conflict misses in the LLC. This is easiest to understand by considering virtually contiguous accesses using an initially empty LLC. If the accesses are physically contiguous, a region as large as the LLC can be accessed before any of the sets in the LLC suffers a conflict miss. However, if the accesses are essentially random, some sets will fill faster than others, approximately halving the number of accesses before one of the sets incurs a conflict miss, thus halving the effective LLC capacity. This is particularly subtle on recent architectures where the LLC is divided into slices, and a hash function is used to map physical addresses to different slices [16].

To explore the effect of physical indexing of the LLC and its interaction with the page allocator, we repeat all four configurations, but with huge page support enabled. Table 4 shows the capacity findings reported in this setting. **Reuse** benefits substantially from larger page size, with capacities increased to more than or on par with **Invalidation** across



(a) Haswell

(b) Broadwell

(c) Skylake

(d) Coffee Lake

Figure 4. Success rate curves on different microarchitectures when using our **Invalidation** configuration, which differs from Baseline by invalidating all cache levels prior to every transaction. We conduct 50 trials and plot the mean with 95% confidence intervals. Transaction sizes are shown on a log scale. All processors show capacities between 85% and 90% of their LLC (7 MB, 10.2 MB, 7 MB, and 14.4 MB).

all platforms. Both **Invalidation** and **Warmup** benefit from huge page support on all microarchitectures as well, albeit less pronounced, with the exception of **Invalidation** on Broadwell. We believe that this is because of the noise in the experiment, noting the extremely low frequency ($\sim 10^{-3}$) of 0.88 LLC capacity of **Invalidation** on Broadwell with small pages. Interestingly, Coffee Lake is the only microarchitecture where **Baseline** benefits from larger pages, for unknown reasons. This result confirms our hypothesis that, with small pages, virtually contiguous memory is generally discontiguous and reduces effective transaction read set sizes.

5 Discussion

This section overviews our findings and discusses their meaning, and then makes recommendations based on the findings.

5.1 Findings and Meaning

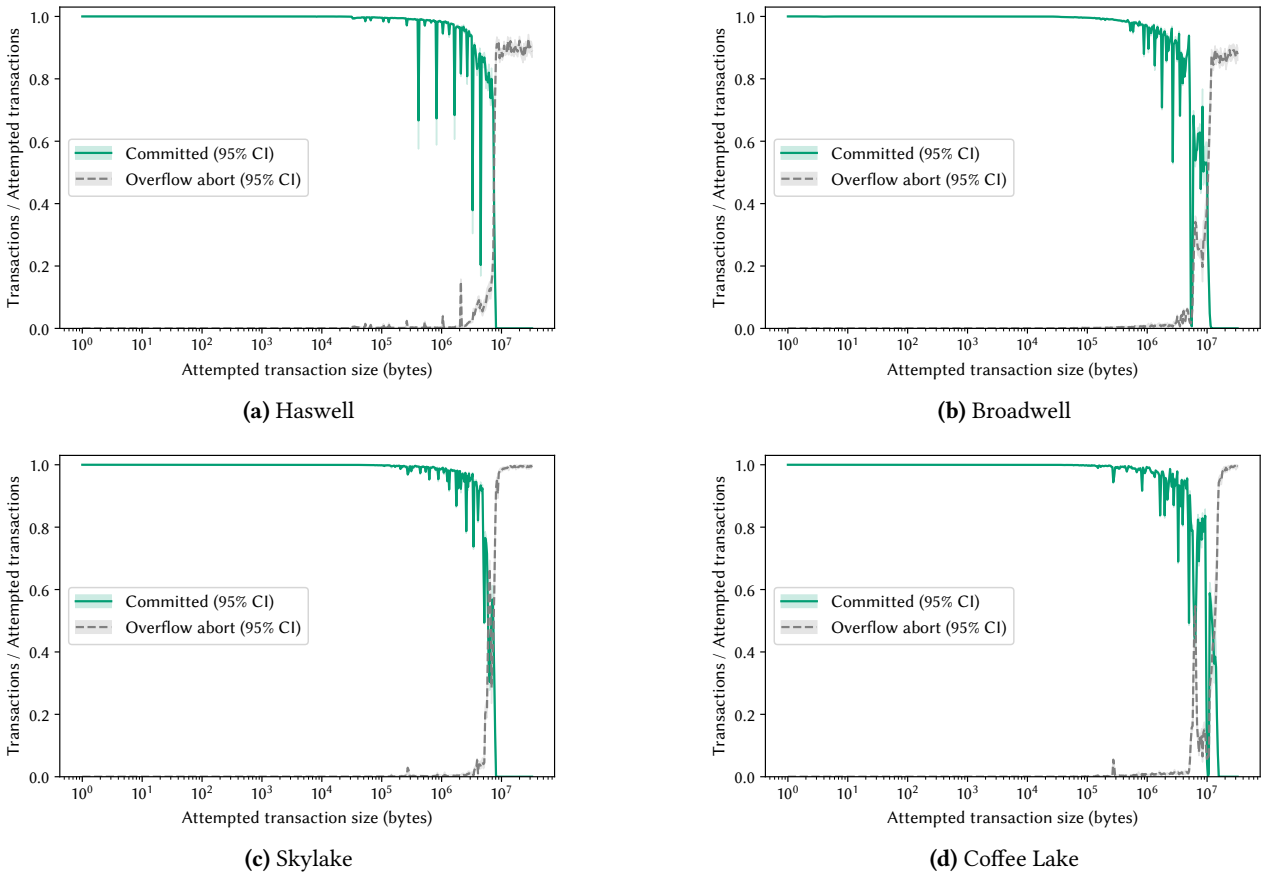
The prior work all pointed to write capacity on Intel's TSX being limited by the L1 size, and we confirm that here.

The prior work painted a very confusing picture of TSX read capacity, varying from 16 KB to 7.5 MB on the same i7-4770 processor, which leads to contradictory conclusions about how HTM works and how to use it (Section 2.4).

In contrast, we found that TSX read capacity is linked to LLC size, and the LLC replacement policy is a key determiner of the effective read capacity of a transaction. Prior work did not take the replacement policy into account.

Furthermore, physical indexing in caches is important because it affects effective capacity in ways that are difficult to control or predict. We empirically showed its effects by using huge pages (Section 4.3).

A broad point is that researchers generally view cache behavior through a performance lens, where *misses* are the



(a) Haswell **(b)** Broadwell

(c) Skylake **(d)** Coffee Lake

Figure 5. Success rate curves on different microarchitectures when using our **Warmup** configuration, which differs from Baseline by warming up the cache 128 times (Haswell) or 5 times (other microarchitectures) with a non-transactional execution of the workload. We conduct 50 trials and plot the mean with 95% confidence intervals. Transaction sizes are shown on a log scale. Haswell, Broadwell, and Coffee Lake show capacities of just over 90% of their LLCs (7.4 MB, 10.9 MB, and 14.9 MB), while Skylake shows a capacity of 97% of its LLC (7.7 MB).

focus, and *common case* behavior is what matters. Here, *evictions* are what matter, and the average or common case is irrelevant: A single eviction aborts the transaction. This is why the arcane cache replacement policies matter and why the physical indexing of the cache is relevant.

Bloom filter or other secondary structure. We did not find any evidence for a Bloom filter or secondary probabilistic structure that was mentioned by some prior work (Section 2.4). However, our results do not imply that such a structure does not exist. HTM conflict detection typically piggybacks on existing processor cache mechanisms (Section 2.1), so if any such secondary structure did exist, it could not extend the read capacity beyond the size of the LLC.

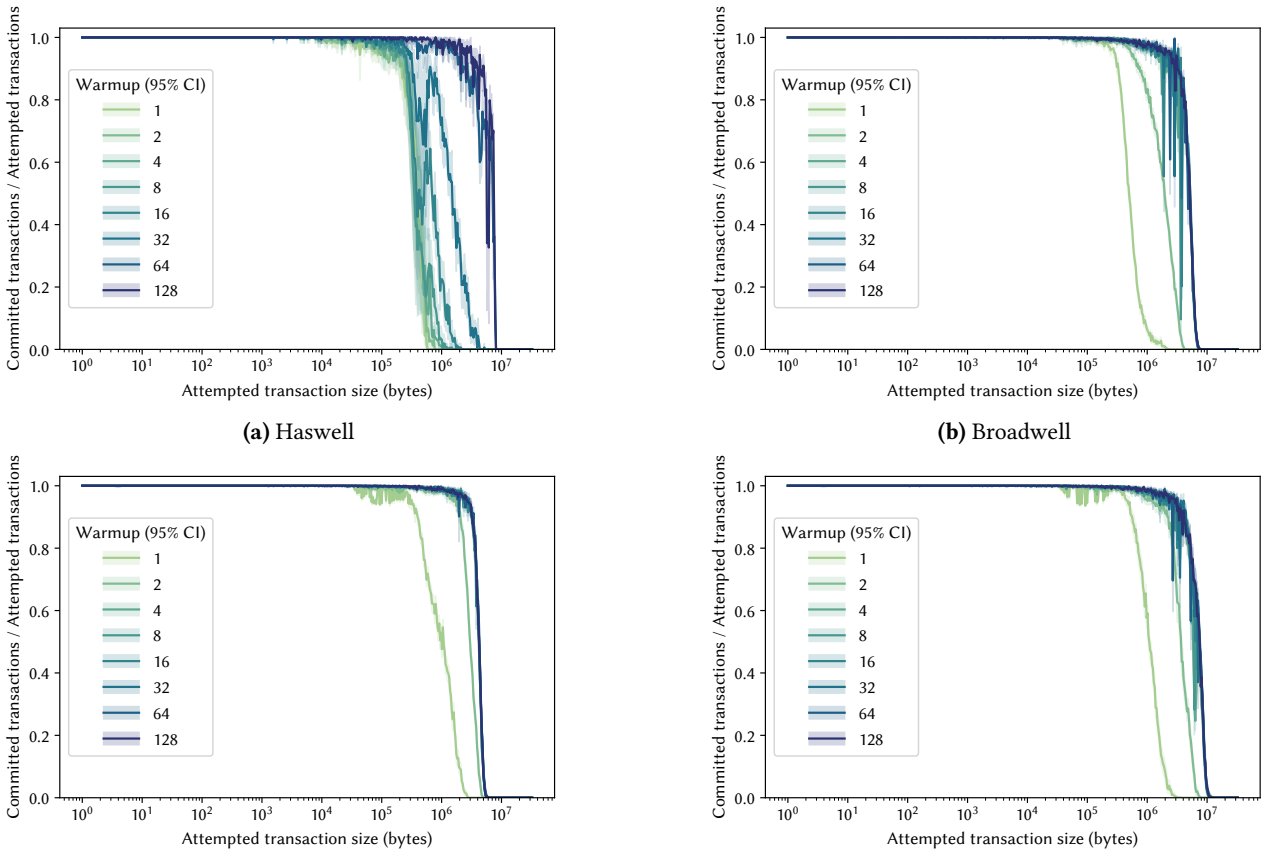
5.2 Recommendations

Our results not only shed light on how TSX is implemented, but also suggest how to make best use of TSX.

As we motivated in Section 1, many diverse uses for TSX exist, including nontraditional uses that are likely to result in large, low-conflict, mostly-read transactions. For such transactions, to maximize effective read capacity, systems should warm up the cache before executing the transaction. It is important to note that the goal is not to *speed up* the transaction, but to enable the transaction to *commit* instead of aborting due to a cache eviction. As such, we expect warmup to be an effective mechanism for improving HTM effectiveness in practice, but we leave evaluation of warmup in the context of a real application to future work.

In contrast, invalidating all cache lines before a transaction is probably impractical, although invalidating periodically might be worthwhile in some cases.

At the system configuration level, our results show that large pages are likely to help with the capacity challenges posed by a physically indexed LLC. Even medium-sized pages



(a) Haswell

(b) Broadwell

(c) Skylake

(d) Coffee Lake

Figure 6. Success rate curves on different microarchitectures for various numbers of non-transactional warmup iterations. These results were used to determine iteration count for our **Warmup** configuration. Transaction sizes are shown on a log scale, with 95% confidence intervals indicated.

such as 2 MB pages (the other option on Intel x86-64 besides 4 KB and 1 GB pages) should give more consistent and higher capacity compared with the default 4 KB page size.

We can also make recommendations for microprocessor designers. Future Intel microarchitectures could extend cache replacement policies to preferentially evict lines *not* accessed by an ongoing transaction. In the related context of region conflict detection, Zhang et al. showed how a modest extension to a simple pseudo-LRU policy can avoid eviction of lines accessed by an executing code region [44].

6 Conclusion

The practicality of applications relying on HTM is predicated on successfully exploiting the capacity of HTM transactions. In this paper, we explored how different factors can affect the capacity of transactions. In particular, we demonstrated how the cache status affects HTM capacity, and how both warming up or invalidating the cache—two seemingly opposite operations—can help large read-only transactions commit. The results resolve the apparent contradiction in the capacity numbers reported by prior work. The insights can help HTM programmers avoid transactional capacity aborts and maximize read capacity, and influence future studies of HTM behavior and future implementations of HTM.

Acknowledgments

We thank the anonymous reviewers for their detailed feedback and insightful suggestions for improving the paper. Thanks to Roman Dementiev, Kaan Genç, Tim Harris, Konrad Lai, Carl Ritson, Tomoharu Ugawa, and Rui Zhang for help understanding TSX capacity. This material is based upon work supported by the Australian Research Council under Grant No. DP190103367 and National Science Foundation under Grant No. XPS-1629126. Zixian Cai is supported by an Australian Government Research Training Program Scholarship.

References

[1] Andreas Abel and Jan Reineke. 2014. Reverse engineering of cache replacement policies in Intel microprocessors and their evaluation. In *2014 IEEE International Symposium on Performance Analysis of Systems and Software, ISPASS 2014, Monterey, CA, USA, March 23-25, 2014*. IEEE Computer Society, 141–142. https://doi.org/10.1109/ISPASS.2014.6844475

[2] Andreas Abel and Jan Reineke. 2020. nanoBench: A Low-Overhead Tool for Running Microbenchmarks on x86 Systems. In *2020 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*.

[3] Andreas Abel and Jan Reineke. 2021. uops.info. https://uops.info.

[4] Hussein Al-Zoubi, Aleksandar Milenkovic, and Milena Milenkovic. 2004. Performance evaluation of cache replacement policies for the SPEC CPU2000 benchmark suite. In *Proceedings of the 42nd Annual Southeast Regional Conference, 2004, Huntsville, Alabama, USA, April 2-3, 2004*, Seong-Moo Yoo and Letha H. Etzkorn (Eds.). ACM, 267–272. https://doi.org/10.1145/986537.986601

[5] Maria Carpen Amarie, Yaroslav Hayduk, Pascal Felber, Christof Fetzer, Gaël Thomas, and Dave Dice. 2017. Towards an Efficient Pauseless Java GC with Selective HTM-Based Access Barriers. In *Proceedings of the 14th International Conference on Managed Languages and Runtimes, ManLang 2017, Prague, Czech Republic, September 27 - 29, 2017*. ACM, 85–91. https://doi.org/10.1145/3132190.3132208

[6] C. Scott Ananian, Krste Asanovic, Bradley C. Kuszmaul, Charles E. Leiserson, and Sean Lie. 2005. Unbounded Transactional Memory. In *Proceedings of the 11th International Symposium on High-Performance Computer Architecture (HPCA '05)*. IEEE Computer Society, 316–327. https://doi.org/10.1109/HPCA.2005.41

[7] Todd Anderson, Melissa O'Neill, and John Sarracino. 2015. Chihuahua: A Concurrent, Moving, Garbage Collector using Transactional Memory. In *10th ACM SIGPLAN Workshop on Transactional Computing, TRANSACT 2015, June 15, 2015, Portland, OR, USA*, Victor Luchangco (Ed.).

[8] Colin Blundell, Joe Devietti, E. Christopher Lewis, and Milo M. K. Martin. 2007. Making the Fast Case Common and the Uncommon Case Simple in Unbounded Transactional Memory. In *Proceedings of the 34th Annual International Symposium on Computer Architecture* (San Diego, California, USA) *(ISCA '07)*. ACM, 24–34. https://doi.org/10.1145/1250662.1250667

[9] Maria Carpen-Amarie. 2017. *Efficient Memory Management with Hardware Transactional Memory: A Focus on Java Garbage Collectors and C++ Smart Pointers*. Ph.D. Dissertation. Université de Neuchâtel.

[10] Calin Cascaval, Colin Blundell, Maged M. Michael, Harold W. Cain, Peng Wu, Stefanie Chiras, and Siddhartha Chatterjee. 2008. Software Transactional Memory: Why Is It Only a Research Toy? *ACM Queue* 6, 5 (2008), 46–58. https://doi.org/10.1145/1454456.1454466

[11] Keith Chapman, Antony L. Hosking, and J. Eliot B. Moss. 2016. Extending OpenJDK to support hybrid STM/HTM: preliminary design. In *Proceedings of the 8th International Workshop on Virtual Machines and Intermediate Languages, VMIL@SPLASH 2016, Amsterdam, Netherlands, October 31, 2016*, Antony L. Hosking and Witawas Srisa-an (Eds.). ACM, 1–5. https://doi.org/10.1145/2998415.2998417

[12] Cao Minh Chí, JaeWoong Chung, Christos Kozyrakis, and Kunle Olukotun. 2008. STAMP: Stanford Transactional Applications for Multi-Processing. In *2008 IEEE International Symposium on Workload Characterization*. 35–46. https://doi.org/10.1109/IISWC.2008.4636089

[13] Peter Damron, Alexandra Fedorova, Yossi Lev, Victor Luchangco, Mark Moir, and Daniel Nussbaum. 2006. Hybrid transactional memory. In *Proceedings of the 12th International Conference on Architectural Support for Programming Languages and Operating Systems, ASPLOS 2006, San Jose, CA, USA, October 21-25, 2006*, John Paul Shen and Margaret Martonosi (Eds.). ACM, 336–346. https://doi.org/10.1145/1168857.1168900

[14] Dave Dice, Tim Harris, Alex Kogan, and Yossi Lev. 2015. The Influence of Malloc Placement on TSX Hardware Transactional Memory. *arXiv e-prints* (April 2015). arXiv:1504.04640 [cs.OS] https://arxiv.org/abs/1504.04640v2

[15] Nuno Diegues, Paolo Romano, and Luís Rodrigues. 2014. Virtues and Limitations of Commodity Hardware Transactional Memory. In *Proceedings of the 23rd International Conference on Parallel Architectures and Compilation* (Edmonton, AB, Canada) *(PACT '14)*. Association for Computing Machinery, New York, NY, USA, 3–14. https://doi.org/10.1145/2628071.2628080

[16] Alireza Farshin, Amir Roozbeh, Gerald Q. Maguire Jr., and Dejan Kostic. 2019. Make the Most out of Last Level Cache in Intel Processors. In *Proceedings of the Fourteenth EuroSys Conference 2019, Dresden, Germany, March 25-28, 2019*, George Candea, Robbert van Renesse, and Christof Fetzer (Eds.). ACM, 8:1–8:17. https://doi.org/10.1145/3302424.3303977

[17] Kaan Genç, Michael D. Bond, and Guoqing Harry Xu. 2020. Crafty: Efficient, HTM-Compatible Persistent Transactions. In *Proceedings of the 41st ACM SIGPLAN Conference on Programming Language Design and Implementation* (London, UK) *(PLDI 2020)*. ACM, 59–74. https://doi.org/10.1145/3385412.3385991

[18] B. Goel, R. Titos-Gil, A. Negi, S. A. McKee, and P. Stenstrom. 2014. Performance and Energy Analysis of the Restricted Transactional Memory Implementation on Haswell. In *2014 IEEE 28th International Parallel and Distributed Processing Symposium*. 615–624. https://doi.org/10.1109/IPDPS.2014.70

[19] Gaston H. Gonnet. 1981. Expected Length of the Longest Probe Sequence in Hash Code Searching. *J. ACM* 28, 2 (1981), 289–304. https://doi.org/10.1145/322248.322254

[20] Daniel Gruss, Julian Lettner, Felix Schuster, Olya Ohrimenko, Istvan Haller, and Manuel Costa. 2017. Strong and Efficient Cache Side-Channel Protection using Hardware Transactional Memory. In *26th USENIX Security Symposium (USENIX Security 17)*. USENIX Association, Vancouver, BC, 217–233. https://www.usenix.org/conference/usenixsecurity17/technical-sessions/presentation/gruss

[21] Tim Harris and Keir Fraser. 2003. Language Support for Lightweight Transactions. In *Proceedings of the 18th Annual ACM SIGPLAN Conference on Object-Oriented Programing, Systems, Languages, and Applications* (Anaheim, California, USA) *(OOPSLA '03)*. ACM, 388–402. https://doi.org/10.1145/949305.949340

[22] Tim Harris, James Larus, and Ravi Rajwar. 2010. *Transactional Memory, 2nd Edition* (2nd ed.). Morgan and Claypool Publishers.

[23] William Hasenplaugh, Andrew Nguyen, and Nir Shavit. 2015. Quantifying the Capacity Limitations of Hardware Transactional Memory. In *7th Workshop on the Theory of Transactional Memory* (Donostia-San Sebastián, Spain) *(WTTM 2015)*. http://www.gsd.inesc-id.pt/~salaa/wttm2015/html/abstracts/Hasenplaugh.pdf

[24] Maurice Herlihy and J. Eliot B. Moss. 1993. Transactional Memory: Architectural Support for Lock-Free Data Structures. In *Proceedings of the 20th Annual International Symposium on Computer Architecture* (San Diego, California, USA) *(ISCA '93)*. Association for Computing Machinery, New York, NY, USA, 289–300. https://doi.org/10.1145/165123.165164

[25] M. Teresa Higuera-Toledano. 2011. Using Transactional Memory to Synchronize an Adaptive Garbage Collector in Real-Time Java. In *14th IEEE International Symposium on Object/Component/Service-Oriented Real-Time Distributed Computing Workshops, ISORC Workshops 2011, Newport Beach, CA, USA, March 28-31, 2011*. IEEE Computer Society, 152–161. https://doi.org/10.1109/ISORCW.2011.24

[26] Sungpack Hong, Tayo Oguntebi, Jared Casper, Nathan Bronson, Christos Kozyrakis, and Kunle Olukotun. 2010. Eigenbench: A simple exploration tool for orthogonal TM characteristics. In *IEEE International Symposium on Workload Characterization (IISWC'10)*. 1–11. https://doi.org/10.1109/IISWC.2010.5648812

[27] Intel Corporation. 2020. *Intel® 64 and IA-32 Architectures Optimization Reference Manual*.

[28] Intel Corporation. 2020. *Intel® 64 and IA-32 Architectures Software Developer's Manual*.

[29] Aamer Jaleel, Kevin B. Theobald, Simon C. Steely Jr., and Joel S. Emer. 2010. High performance cache replacement using re-reference interval prediction (RRIP). In *37th International Symposium on Computer Architecture (ISCA 2010), June 19-23, 2010, Saint-Malo, France*, André Seznec, Uri C. Weiser, and Ronny Ronen (Eds.). ACM, 60–71. https://doi.org/10.1145/1815961.1815971

[30] Sanjeev Kumar, Michael Chu, Christopher J. Hughes, Partha Kundu, and Anthony D. Nguyen. 2006. Hybrid transactional memory. In *Proceedings of the ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, PPOPP 2006, New York, New York, USA, March 29-31, 2006*, Josep Torrellas and Siddhartha Chatterjee (Eds.). ACM, 209–220. https://doi.org/10.1145/1122971.1123003

[31] D. B. Lomet. 1977. Process Structuring, Synchronization, and Recovery Using Atomic Actions. In *Proceedings of an ACM Conference on Language Design for Reliable Software* (Raleigh, North Carolina). Association for Computing Machinery, New York, NY, USA, 128–137. https://doi.org/10.1145/800022.808319

[32] Hassan Salehe Matar, Ismail Kuru, Serdar Tasiran, and Roman Dementiev. 2014. Accelerating Precise Race Detection Using Commercially-Available Hardware Transactional Memory Support. In *Workshop on Determinism and Correctness in Parallel Programming*.

[33] Phil McGachey, Ali-Reza Adl-Tabatabai, Richard L. Hudson, Vijay Menon, Bratin Saha, and Tatiana Shpeisman. 2008. Concurrent GC leveraging transactional memory. In *Proceedings of the 13th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, PPOPP 2008, Salt Lake City, UT, USA, February 20-23, 2008*, Siddhartha Chatterjee and Michael L. Scott (Eds.). ACM, 217–226. https://doi.org/10.1145/1345206.1345238

[34] Takuya Nakaike, Rei Odaira, Matthew Gaudet, Maged M. Michael, and Hisanobu Tomari. 2015. Quantitative Comparison of Hardware Transactional Memory for Blue Gene/Q, ZEnterprise EC12, Intel Core, and POWER8. In *Proceedings of the 42nd Annual International Symposium on Computer Architecture* (Portland, Oregon) *(ISCA '15)*. Association for Computing Machinery, New York, NY, USA, 144–157. https://doi.org/10.1145/2749469.2750403

[35] M. M. Pereira, M. Gaudet, J. N. Amaral, and G. Araújo. 2014. Multi-dimensional Evaluation of Haswell's Transactional Memory Performance. In *2014 IEEE 26th International Symposium on Computer Architecture and High Performance Computing*. 144–151. https://doi.org/10.1109/SBAC-PAD.2014.33

[36] Moinuddin K. Qureshi, Aamer Jaleel, Yale N. Patt, Simon C. Steely Jr., and Joel S. Emer. 2007. Adaptive insertion policies for high performance caching. In *34th International Symposium on Computer Architecture (ISCA 2007), June 9-13, 2007, San Diego, California, USA*, Dean M. Tullsen and Brad Calder (Eds.). ACM, 381–391. https://doi.org/10.1145/1250662.1250709

[37] Ravi Rajwar and James R. Goodman. 2001. Speculative lock elision: enabling highly concurrent multithreaded execution. In *Proceedings of the 34th Annual International Symposium on Microarchitecture, Austin, Texas, USA, December 1-5, 2001*, Yale N. Patt, Josh Fisher, Paolo Faraboschi, and Kevin Skadron (Eds.). ACM/IEEE Computer Society, 294–305. https://doi.org/10.1109/MICRO.2001.991127

[38] Carl G. Ritson and Frederick R.M. Barnes. 2013. An Evaluation of Intel's Restricted Transactional Memory for CPAs. In *Communicating Process Architectures 2013 Proceedings of the 35th WoTUG Technical Meeting*, Peter H. Welch, Frederick R.M. Barnes, Jan F. Broenink, Kevin Chalmers, Jan B. Pedersen, and Adam T. Sampson (Eds.). Open Channel Publishing, 271–291. https://kar.kent.ac.uk/36939/

[39] Carl G. Ritson, Tomoharu Ugawa, and Richard E. Jones. 2014. Exploring garbage collection with Haswell hardware transactional memory. In *International Symposium on Memory Management, ISMM '14, Edinburgh, United Kingdom, June 12, 2014*, David Grove and Samuel Z. Guyer (Eds.). ACM, 105–115. https://doi.org/10.1145/2602988.2602992

[40] Aritra Sengupta, Man Cao, Michael D. Bond, and Milind Kulkarni. 2017. Legato: End-to-End Bounded Region Serializability Using Commodity Hardware Transactional Memory. In *Proceedings of the 2017 International Symposium on Code Generation and Optimization* (Austin, USA) *(CGO '17)*. IEEE Press, 1–13.

[41] Nir Shavit and Dan Touitou. 1995. Software Transactional Memory. In *Proceedings of the Fourteenth Annual ACM Symposium on Principles of Distributed Computing, Ottawa, Ontario, Canada, August 20-23, 1995*, James H. Anderson (Ed.). ACM, 204–213. https://doi.org/10.1145/224964.224987

[42] Zhaoguo Wang, Hao Qian, Jinyang Li, and Haibo Chen. 2014. Using Restricted Transactional Memory to Build a Scalable In-Memory Database. In *Proceedings of the Ninth European Conference on Computer Systems* (Amsterdam, The Netherlands) *(EuroSys '14)*. Association for Computing Machinery, New York, NY, USA, Article 26, 15 pages. https://doi.org/10.1145/2592798.2592815

[43] Richard M. Yoo, Christopher J. Hughes, Konrad Lai, and Ravi Rajwar. 2013. Performance Evaluation of Intel® Transactional Synchronization Extensions for High-Performance Computing. In *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis* (Denver, Colorado) *(SC '13)*. Association for Computing Machinery, New York, NY, USA, Article 19, 11 pages. https://doi.org/10.1145/2503210.2503232

[44] Rui Zhang, Swarnendu Biswas, Vignesh Balaji, Michael D. Bond, and Brandon Lucia. 2020. Peacenik: Architecture Support for Not Failing under Fail-Stop Memory Consistency. In *ASPLOS '20: Architectural Support for Programming Languages and Operating Systems, Lausanne, Switzerland, March 16-20, 2020*, James R. Larus, Luis Ceze, and Karin Strauss (Eds.). ACM, 317–333. https://doi.org/10.1145/3373376.3378485

[45] Tong Zhang, Dongyoon Lee, and Changhee Jung. 2016. TxRace: Efficient Data Race Detection Using Commodity Hardware Transactional Memory. In *Proceedings of the Twenty-First International Conference on Architectural Support for Programming Languages and Operating Systems* (Atlanta, Georgia, USA) *(ASPLOS '16)*. Association for Computing Machinery, New York, NY, USA, 159–173. https://doi.org/10.1145/2872362.2872384