Extending Hardware Transactional Memory Capacity via Rollback-Only Transactions and Suspend/Resume*†

Shady Issa[1], Pascal Felber[2], Alexander Matveev[‡3], and Paolo Romano[4]

1 **INESC-ID / Instituto Superior Técnico, University of Lisbon, Portugal**
2 **University of Neuchatel, Switzerland**
3 **MIT, Cambridge, USA**
4 **INESC-ID / Instituto Superior Técnico, University of Lisbon, Portugal**

Abstract

Transactional memory (TM) aims at simplifying concurrent programming via the familiar abstraction of atomic transactions. Recently, Intel and IBM have integrated hardware based TM (HTM) implementations in commodity processors, paving the way for the mainstream adoption of the TM paradigm. Yet, existing HTM implementations suffer from a crucial limitation, which hampers the adoption of HTM as a general technique for regulating concurrent access to shared memory: the inability to execute transactions whose working sets exceed the capacity of CPU caches. In this paper we propose P8TM, a novel approach that mitigates this limitation on IBM's POWER8 architecture by leveraging a key combination of techniques: uninstrumented read-only transactions, Rollback Only Transaction-based update transactions, HTM-friendly (software-based) read-set tracking, and self-tuning. P8TM can dynamically switch between different execution modes to best adapt to the nature of the transactions and the experienced abort patterns. In-depth evaluation with several benchmarks indicates that P8TM can achieve striking performance gains in workloads that stress the capacity limitations of HTM, while achieving performance on par with HTM even in unfavourable workloads.

1998 ACM Subject Classification D.1.3 Concurrent Programming

Keywords and phrases hardware transactional memory, self tuning

Digital Object Identifier 10.4230/LIPIcs.DISC.2017.28

1 Introduction

Transactional memory (TM) has emerged as a promising paradigm that aims at simplifying concurrent programming by bringing the familiar abstraction of atomic and isolated transactions to the domain of parallel computing. Unlike when using locks to synchronize access to shared data or code portions, with TM programmers need only to specify *what* is synchronized and not *how* synchronization should be performed. This results in simpler designs that are easier to write, reason about, maintain, and compose [4].

* This work was supported by Portuguese funds through Fundação para a Ciência e Tecnologia via projects UID/CEC/50021/2013 and PTDC/EEISCR/1743/2014.
† The extended version [18] can be found at `http://www.inesc-id.pt/ficheiros/publicacoes/12747.pdf`
‡ Alexander Matveev was supported by the NSF under grants IIS-1447786 and CCF- 1563880

Over the last years, the relevance of TM has been growing along with the maturity of available supports for this new paradigm, both in terms of integration at the programming language as well as at the architectural level. On the front of integration with programming languages, a recent milestone has been the official integration of TM in mainstream languages, such as C/C++ [2]. On the architecture's side, the integration of hardware supports in Intel's and IBM's processors, a technology that goes under the name of hardware transactional memory (HTM), has represented a major breakthrough, thanks to enticing performance gains that such an approach can, at least potentially, enable [15, 17, 24].

Existing hardware implementations share various architectural choices, although they do come in different flavours [19, 22, 25]. The key common trait of current HTM systems is their best effort nature: current implementations maintain transactional metadata (e.g., memory addresses read/written by a transaction) in the processor's cache and rely on relatively non-intrusive modification to the pre-existing cache coherency protocol to detect conflict among concurrent transactions. Due to the inherently limited nature of processor caches, current HTM implementations impose stringent limitations on the number of memory accesses that can be performed within a transaction,[1] hence providing no progress guarantee even for transactions that run in absence of concurrency. As such, HTM requires a fallback synchronization mechanism (also called *fallback path*), which is typically implemented via a pessimistic scheme based on a single global lock.

Despite these common grounds, current HTM implementations have also several relevant differences. Besides internal architectural choices (e.g., where and how in the cache hierarchy transactional metadata are maintained), Intel's and IBM's implementations differ notably by the programming interfaces they expose. In particular, IBM POWER8's HTM implementation extends the conventional transactional demarcation API (to start, commit and abort transactions) with two additional, unique features [5]:

- *Suspend/resume*: the ability to suspend and resume a transaction, allowing, between the suspend and resume calls, for the execution of instructions/memory accesses that escape from the transactional context.
- *Rollback-only transaction (ROT)*: a lightweight form of transaction that has lower overhead than regular transactions but also weaker semantics. In particular ROTs avoid tracking load operations, i.e., they are not isolated, but still ensure the atomicity of the stores issued by a transaction, which appear to be all executed or not executed at all.

In this work we present POWER8 TM (P8TM), a novel TM that exploits these two specific features of POWER8's HTM implementation in order to overcome (or at least mitigate) what is, arguably, the key limitation stemming from the best-effort nature of existing HTM systems: the inability to execute transactions whose working sets exceed the capacity of CPU caches. P8TM pursues this objective via an innovative hardware-software co-design that leverages several novel techniques, which we overview in the following:

- **Uninstrumented read-only transactions (UROs).** P8TM executes read-only transactions outside of the scope of hardware transactions, hence sparing them from spurious aborts and capacity limitations, while still allowing them to execute concurrently with update transactions. This result is achieved by exploiting the POWER8's suspend/resume mechanism to implement a RCU-like quiescence scheme that shelters UROs from observing inconsistent snapshots that reflect the commit events of concurrent update transactions.

[1] The list of restrictions is actually longer, including the lack of support for system calls and other non-undoable instructions, context switches and ring transitions.

- **ROT-based update transactions.** In typical TM workloads the read/write ratio tends to follow the 80/20 rule, i.e., transactified methods tend to have large read-sets and much smaller write sets [12]. This observation led us to develop a novel concurrency control scheme based on a novel hardware-software co-design: it combines the hardware-based ROT abstraction—which tracks only transactions' write sets, but not their read-sets, and, as such, does not guarantee isolation—with software based techniques aimed to preserve correctness in presence of concurrently executing ROTs, UROs, and plain HTM transactions. Specifically, P8TM relies on a novel mechanism, which we called Touch-To-Validate (T2V), to execute concurrent ROTs safely. T2V relies on a lightweight software instrumentation of reads within ROTs' and a hardware aided validation mechanism of the read-set during the commit phase.
- **HTM-friendly (software-based) read-set tracking.** A key challenge that we had to tackle while implementing P8TM was to develop a "HTM-friendly" software-based read-set tracking mechanism. In fact, all the memory writes issued from within a ROT, including those needed to track the read-set, are transparently tracked in hardware. As such, the read-set tracking mechanism can consume cache capacity that could be otherwise used to accommodate application-level writes. P8TM integrates two read-set tracking mechanisms that explore different trade-offs between space and time efficiency.
- **Self-tuning.** To ensure robust performance in a broad range of workloads, P8TM integrates a lightweight reinforcement learning mechanism (based on the UCB algorithm [21]) that automates the decision of whether: i) to use upfront ROTs and UROs, avoiding at all to use HTM; ii) to first attempt transactions in HTM, and then fallback to ROTs/UROs in case of capacity exceptions; iii) to completely switch off ROTs/UROs, and use only HTM.

We evaluated P8TM by means of an extensive study that encompasses synthetic micro-benchmarks and the benchmarks in the Stamp suite [7]. The results of our study show that P8TM can achieve up $\sim$5$\times$ throughput gains with respect to plain HTM and extend its capacity by more than one order of magnitude, while remaining competitive even in unfavourable workloads.

2 Related Work

Since the introduction of HTM support in mainstream commercial processors by Intel and IBM, several experimental studies have aimed to characterize their performance and limitations [15, 17, 24]. An important conclusion reached by these studies is that HTM's performance excels with workloads that fit the hardware capacity limitations. Unfortunately, though, HTM's performance and scalability can be severely hampered in workloads that contain even a small percentage of transactions that do exceed the hardware's capacity. This is due to the need to execute such transactions using a sequential fallback mechanism based on a single global lock (SGL), which causes the immediate abort of any concurrent hardware transactions and prevents any form of parallelism.

Hybrid TM [9, 20] (HyTM) attempts to address this issue by falling back to software-based TM (STM) implementations when transactions cannot successfully execute in hardware. Hybrid NoRec (Hy-NoRec) is probably one of the most popular and effective HyTM designs proposed in the literature. Hy-NoRec [8] falls back on using the NoRec STM, which lends itself naturally to serve as fallback for HTM. In fact, NoRec uses a single versioned lock for synchronizing (software) transactions. Synchronization between HTM and STM can hence be attained easily, by having HTM transactions update the versioned lock used by NoRec.

Unfortunately, the coupling via the versioned lock introduces additional overheads on both the HTM and STM side, and can induce spurious aborts of HTM transactions.

Recently, RHyNoRec [23] proposed to decompose a transaction running on the fallback path into multiple hardware transactions: a read-only prefix and a single post-fix that encompasses all the transaction's writes, with regular NoRec shared operations in between. This can reduce the false aborts that would otherwise affect hardware transactions in HyNoRec. Unfortunately, though, this approach is only viable if the transaction's postfix, which may potentially encompass a large number of reads, does fit in hardware. Further, the technique used to enforce atomicity between the read-only and the remaining reads relies on fully instrumenting every read within the prefix hardware transaction, this utterly limits the capacity—and consequently the practicality—of these transactions. Unlike RHyNoRec, P8TM can execute read-only transactions of arbitrary length in a fully uninstrumented way. Further, the T2V mechanism employed by P8TM to validate update transactions relies on a much lighter and efficient read-set tracking and validation schemes that can even further increase the capacity of transactions.

Our work is also related to the literature aimed to enhance HTM's performance by optimizing the management of the SGL fallback path. A simple, yet effective optimization, which we include in P8TM, is to avoid the, so called, *lemming effect* [11] by ensuring that the SGL is free before starting a hardware transaction. An alternative solution to the same problem is the use of an auxiliary lock [3]. In our experience, these two solutions provide equivalent performance, so we opted to integrate in P8TM the former, simpler, approach. Herihly et al. [6] suggested lazy subscription of the SGL in order to decrease the vulnerability window of HTM transactions. However, this approach was shown to be unsafe in subtle scenarios that are hard to fix using automatic compiler-based techniques [10].

P8TM integrates a self-tuning approach that shares a common theoretical framework (the UCB reinforcement learning algorithm [21]) with Tuner [13]. However, Tuner addresses an orthogonal self-tuning problem to the one we tackled in P8TM: Tuner exploits UCB to identify the optimal retry policy before falling back to the SGL path upon a capacity exception; in P8TM, conversely, UCB is to determine which synchronization to use (e.g., ROTs/UROs vs. plain HTM). Another recent work that makes extensive use of self-techniques to optimize HTM's performance is SEER [14]. Just like Tuner, SEER addresses an orthogonal problem—defining a scheduling policy that seeks an optimal trade-off between throughput and contention probability—and could, indeed, be combined with P8TM.

Finally, P8TM builds on and extends on HERWL[16], where we introduced the idea of using POWER8's suspend-resume and ROT facilities to elide read-write locks. Besides targeting a different application domain (transactional programs vs. lock elision), P8TM integrates a set of novel techniques. Unlike HERWL, P8TM supports the concurrent execution of update transactions in ROTs. Achieving this result implied introducing a novel concurrency control mechanism (which we named Touch-To-Validate). Additionally, P8TM integrates self-tuning techniques that ensure robust performance also in unfavourable workloads.

3 Background on POWER8's HTM

This section provides background on POWER8's HTM system, which is relevant to the operation of P8TM. Analogously to other HTM implementations, POWER8 provides an API to begin, commit and abort transactions. When programs request to start a transaction, a *started* code is placed in the, so called, status buffer. If, later, the transaction aborts, the program counter jumps back to just after the instruction used to begin the transaction.

Hence, in order to distinguish whether a transaction has just started, or has undergone an abort, programs must test the status code returned after beginning the transaction.

POWER8 detects conflicts with granularity of a cache line. The transaction capacity (64 cache lines) in POWER8 is bound by a 8KB cache, called TMCAM, which stores the addresses of the cache lines read or written within the transaction.

As mentioned, in addition to HTM transactions, POWER8 also supports Rollback-Only Transactions (ROT). The main difference being that in ROTs, only the writes are tracked in the TMCAM, giving virtually infinite read-set capacity. Reads performed by ROTs are essentially treated as non transactional reads. From this point on, whenever we use the term *transaction*, we refer to a plain HTM transaction.

Both transactions and ROTs detect conflict eagerly, i.e., they are aborted as soon as they incur a conflict. The only exception is when they incur a conflict while in suspend mode: in this case, they abort only once they resume. Finally, P8TM exploits how POWER8 manages conflicts that arise between non-transactional code and transactions/ROTs, i.e., if a transaction/ROT issues a write on X and, before it commits, a non-transactional read/write is issued on X, the transaction/ROT is immediately aborted by the hardware.

4 The P8TM Algorithm

This section describes P8TM (*POWER8 Transactional Memory*). We start by overviewing the algorithm. Next, we detail its operation and present several optimizations.

4.1 Overview

The key challenge in designing execution paths that can run concurrently with HTM is efficiency: it is hard to provide a software-based path that executes concurrently with the HTM path, while preserving correctness and speed. The main problem is that the protocol must make the hardware aware of concurrent software memory reads and writes, which requires to introduce expensive tracking mechanisms in the HTM path.

P8TM tackles this issue by exploiting two unique features of the IBM POWER8 architecture:

(1) suspend/resume for hardware transactions, and

(2) ROTs.

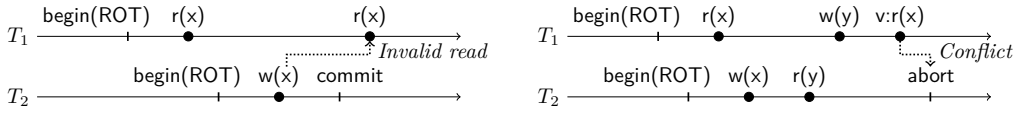
P8TM combines these new hardware features with an RCU-like quiescence scheme in a way that avoids the need to track reads in hardware. This can in particular reduce the likelihood of capacity aborts that would otherwise affect transactions that perform a large number of reads.

The key idea is to provide two novel execution paths alongside the HTM path:

(i) a, so called, *ROT path*, which executes write transactions that do not fit in HTM as ROTs, and

(ii) a, so called, *URO path*, which executes read-only transactions without any instrumentation.

Transactions and ROTs exploit the speculative hardware supports to hide writes from concurrent reads. This allows to cope with read-write conflicts that occur during ROTs/UROs, but it does not cover read-write conflicts that occur after the commit of an update transaction. For this purpose, before a write transaction commits, either as a transaction or a ROT, it first suspends itself and then executes a quiescence mechanism that waits for the completion of currently executing ROTs/URO transactions. In addition to that, in case of ROTs, it further executes an original *touch-based validation* step, which is described next, before



(a) ROTs do not track reads and may observe different values when reading the same variable multiple times.

(b) By re-reading x during *rot-rset* validation at commit time (denoted by v:r), T_1 forces an abort of T_2 that has updated x in the meantime.

Figure 1 Operation scenarios.

resuming and committing. This process of "suspending and waiting" ensures that the writes of an update transaction will be committed only if they do not target/overwrite any memory location that was previously read by any concurrent ROT/URO transaction.

4.2 Touch-based Validation

Touch-To-Validate (T2V) is a core mechanism of our algorithm that enables safe and concurrent execution of ROTs. Indeed, ROTs do not track read accesses within the transaction, therefore it is unsafe to execute them concurrently, as they are not serializable.

Consider the example shown in Figure 1a. Thread T_1 starts a ROT and reads x. At this time, thread T_2 starts a concurrent ROT, writes a new value to x, and commits. As ROTs do not track reads, the ROT of T_1 does not get aborted and can read inconsistent values (e.g., the new value of x), hence yielding non-serializable histories. To avoid such scenarios T2V leverages two key mechanisms that couple:

(i) software-based tracking of read accesses; and
(ii) hardware- and software-based read-set validation during the commit phase.

For the sake of clarity, let us assume that threads only execute ROTs—we will consider other execution modes later. A thread can be in one of three states: *inactive*, *active*, and *committing*. A thread that executes non-transactional code is inactive. When the thread starts a ROT, it enters the active phase and starts tracking, in software, each read access to shared variables by logging the associated memory address in a special data structure called *rot-rset*. Finally, when the thread finishes executing its transaction, it enters the committing phase. At this point, it has to wait for concurrent threads that are in the active phase to either enter the commit phase or become inactive (upon abort). Thereafter, the committing thread traverses its *rot-rset* and re-reads each address before eventually committing.

The goal of this validation step is to "touch" each previously read memory location in order to abort any concurrent ROT that might have written to the same address. For example, in Figure 1b, T_1 re-reads x during *rot-rset* validation. At that time, T_2 has concurrently updated x but has not yet committed, and it will therefore abort (remember that ROTs track and detect conflicts for writes). This allows T_1 to proceed without breaking consistency: indeed, ROTs buffer their updates until commit and hence the new value of x written by T_2 is not visible to T_1. Note that adding a simple quiescence phase before commit, without performing the *rot-rset* validation, cannot solve the problem in this scenario.

The originality of the T2V mechanism is that the ROT does not use read-set validation for verifying that its read-set is consistent, as many STM algorithms do, but to trigger hardware conflicts detection mechanisms. This also means that the values read during *rot-rset* validation are irrelevant and ignored by the algorithm.

Algorithm 1 P8TM: ROT path only algorithm.

```
1:  Shared variables:
2:      status[N] ← {⊥, ⊥, ..., ⊥}                          ▷ One per thread
3:  Local variables:
4:      tid ∈ [0..N]                                        ▷ Identifier of current thread
5:      rot-rset ← ∅                                        ▷ Transaction's read-set
6:  function READ(addr)                                     ▷ Read shared variable
7:      rot-rset ← rot-rset ∪{addr}                         ▷ Track ROT reads
8:  function SYNCHRONIZE
9:      s[N] ← status                                       ▷ Read and copy all status variables
10:     for i ← 0 to N−1 do                                 ▷ Wait until all threads...
11:         if s[i] = ACTIVE then                           ▷ ...that are active...
12:             wait until status[i] ≠ s[i]                 ▷ ...cross barrier
13: function TOUCH_VALIDATE
14:     for addr ∈ rot-rset do                              ▷ Re-read all elements...
15:         read addr                                       ▷ ...from read-set
16: function BEGIN_ROT
17:     repeat                                              ▷ Retry ROT forever
18:         status[tid] ← ACTIVE                            ▷ Indicate we are active
19:         MEM_FENCE                                       ▷ Make sure others know
20:         rot-rset ← ∅                                    ▷ Clear read-set
21:         tx ← TX_BEGIN_ROT                               ▷ HTM ROT begin
22:     until tx = STARTED                                  ▷ Repeat until success...
23: function COMMIT
24:     TX_SUSPEND                                          ▷ Suspend transaction
25:     status[tid] ← ROT-COMMITTING                        ▷ Tell others...
26:     MEM_FENCE                                           ▷ ...we are committing
27:     TX_RESUME                                           ▷ Resume transaction
28:     SYNCHRONIZE                                         ▷ Quiescence inside ROT
29:     TOUCH_VALIDATE                                      ▷ Touch to validate
30:     TX_COMMIT                                           ▷ End transaction
31:     status[tid] ← ⊥
```

4.3 Basic Algorithm

We first present below the basic version of the P8TM algorithm (Algorithm 1) assuming we only have ROTs and we blindly retry to execute failed ROTs irrespective of the abort cause.

To start a transaction, a thread first lets others know that it is *active* and initializes its data structures before actually starting a ROT (Lines 18–21). Then, during ROT execution, it just keeps track of reads to shared data by adding them to the thread-local *rot-rset* (Line 7). To complete the ROT, the thread first announces that it is *committing* by setting its shared *status* variable. Note that this is performed while the ROT is suspended (Lines 24–27) because otherwise the write would be buffered and invisible to other threads.

Next, the algorithm quiesces by waiting for all threads that are in a ROT to at least reach their commit phase (Lines 8–12). It then executes the touch-based validation mechanism, which simply consists in re-reading all address in the *rot-rset* (Lines 13–15), before finally committing the ROT (Line 30) and resetting the *status*.

4.4 Complete Algorithm

The naive approach of the basic algorithm to only use ROTs is unfortunately not practical nor efficient in real-world settings for two main reasons: (1) ROTs only provide "best effort" properties and thus a fallback is needed to guarantee liveness; and (2) using ROTs for short critical sections set that fit in a regular transaction is inefficient, because of the overhead of software-hardware read-set tracking and validation upon commit. Therefore, we extend the previous algorithm so that it first tries to use regular transactions, then switches to ROTs, and finally falls back to a global lock (GL) in order to guarantee progress. The pseudo-code of the complete algorithm is available in the extended version [18].

For transactions and ROTs to execute concurrently, the former must delay their commit until completion of all active ROTs. This is implemented using an RCU-like quiescence mechanism as in the basic algorithm. Transactions try to run in HTM and ROT modes a limited number of times, switching immediately if the cause of the failure is a capacity abort. The GL fallback uses a basic spin lock, which is acquired upon transaction begin and released upon commit. Note that the quiescence mechanism must also be called after acquiring the lock to wait for completion of ROTs that are in progress and might otherwise see inconsistent updates. Further, the GL fallback must also wait for ROTs to fully complete.

Read-only transactions. We finally describe the URO path, i.e., the execution mode optimized for read-only (RO) transactions in which reads are not tracked, hence significantly decreasing runtime overheads. This would also allow to execute large RO transactions that do not fit in hardware, and would otherwise be doomed to execute in the GL path.

To understand the intuition behind the URO path, note that whenever a URO develops a read-after-write with any concurrent transaction/ROT T, T is immediately aborted by the hardware. As for write-after-read conflicts, since transactions and ROTs buffer their writes and quiesce before committing, they cannot propagate inconsistent updates to RO transactions: this feature allows P8TM to achieve concurrency between UROs and transactions/ROTs, even when they encounter write-after-read conflicts (by serializing the URO before the T).

Finally, GL and RO transactions cannot conflict with each other as long as they do not run concurrently. This is ensured by performing a quiescence phase after acquiring the global lock, and executing RO transactions only when the lock is free. Note that, if the lock is taken, RO transactions defer to the writer by resetting their status before waiting for the lock to be free and retrying the whole procedure; otherwise we could run into a deadlock.

Correctness argument. When the GL path is active, concurrency is disabled. This is guaranteed since:

- **(i)** transactions in HTM path subscribe eagerly to the GL, and are thus aborted upon the activation of this path;
- **(ii)** after the GL is acquired, a quiescence phase is performed to wait for active ROTs or UROs.

Atomicity of a transaction in the HTM path is provided by the hardware against concurrent transactions/ROTs and by GL subscription.

As for the UROs, the quiescence mechanism guarantees two properties:

- UROs activated after the start of an update transaction T, and before the start of T's quiescence phase, can be safely serialized before T because they are guaranteed not to see any of T's updates, which are only made atomically visible when the corresponding transaction/ROT commits;

- UROs activated after the start of the quiescence phase of an update transaction T can be safely serialized after T because they are guaranteed to either abort T, in case they read a value written by T before T commits, or see all the updates produced by T's commit. It is worth noting here though that this is only relevant when UROs may conflict with T, in case of disjoint operation both serialization orders are equivalent.

Now we are only left with transactions running on the ROT path. The same properties of quiescence for UROs apply here and avoid ROTs reading inconsistent states produced by concurrent HTM transactions. Nevertheless, since ROTs do modify the shared state, they can still produce non-serializable histories; such as the scenario in Figure 2. Assume a ROT, say T_1, issued a read on X, developing a read-write conflict, with some concurrently active ROT, say T_2. There are two cases to consider: T_1 commits before T_2, or vice-versa.

If T_1 commits first, then if it reads X after T_2 (which is still active) wrote to it, then T_2 is aborted by the hardware conflict detection mechanism. Else, we are in presence of a write-after-read conflict. T_1 finds $status[T_2] := ACTIVE$ (because T_2 issues a fence before starting) and waits for T_2 to enter its commit phase (or abort). Then T_1 executes its T2V, during which, by re-reading X, would cause T_2 to abort.

Consider now the case in which T_2 commits before T_1. If T_1 reads X, as well as any other memory position updated by T_2, before T_2 writes to it, then T_1 can be safely serialized before T_2 (as T_1 observed none of T_2's updates). If T_1 reads X, or any other memory position updated by T_2, after T_2 writes to it and before T_2 commits, then T_2 is aborted by the hardware conflict detection mechanism; a contradiction. Finally, it is impossible for T_1 to read X after T_2 commits: in fact, during T_2's commit phase, T_2 must wait for T_1 to complete its execution; hence, T_1 must read X after T_2 writes to it and before T_2 commits, falling in the above case and yielding another contradiction.

Self-tuning. In workloads where transactions fit the HTM's capacity restrictions, P8TM forces HTM transactions to incur the overhead of suspend/resume, in order to synchronize them with possible concurrent ROTs. In these workloads, the ideal decision would be to just disable the ROT path, so to spare the HTM path from any overhead. However, it is not trivial to determine when it is beneficial to do so; this choice is workload dependent and is not trivial to determine via static code analysis techniques.

We address this issue by integrating into P8TM a self-tuning mechanism based on a lightweight reinforcement learning technique, UCB [21]. UCB determines, in an automatic fashion, which of the following modes to use:

(M1) HTM falling back to ROT, and then to GL;
(M2) HTM falling back directly to the GL;
(M3) starting directly in ROT before falling back to the GL.

Note that UROs and ROTs impose analogous overheads to HTM transactions. Thus, in order to reduce the search space to be explored by the self-tuning mechanism, whenever we disable ROTs (i.e., case (M2)), we also disable UROs (and treat RO transactions as update ones).

5 Read-set Tracking

The T2V mechanism requires to track the read-sets of ROTs for later replaying them at commit time. The implementation of the read-set tracking scheme is crucial for the performance of P8TM. In fact, as discussed in Section 3, ROTs do not track loads at the TMCAM level, but they do track stores and the read-set tracking mechanism must issue

stores in order to log the addresses read by a ROT. The challenge, hence, lies in designing a software mechanism that can exploit the TMCAM's capacity in a more efficient way than the hardware would do. In the following we describe two alternative mechanisms that tackle this challenge by exploring different trade-offs between computational and space efficiency.

Time-efficient implementation uses a thread local, cache aligned array, where each entry is used to track a 64-bit address. Since the cache lines of the POWER8 CPU are 128 bytes long, this means that 16 consecutive entries of the array, each storing an arbitrary address, will be mapped to the same cache line and occupy a single TMCAM entry. Therefore, this approach allows for fitting up to 16× larger read-sets within the TMCAM as compared to the case of HTM transactions. Given that they track 64 cache lines, each thread-local array is statically sized to store exactly 1024 addresses. It is worth noting here that since conflicts are detected at the cache line level granularity, it is not necessary to store the 7 least significant bits, as addresses point to the same cache line. However, we omit this optimization as this will add extra computational overhead, yielding a space saving of less than 10%.

Space-efficient implementation seeks to exploit the spatial data locality in the application's memory access patterns to compress the amount of information stored by the read-set tracking mechanism. This is achieved by detecting a common prefix between the previously tracked address and the current one, and by storing only the differing suffix and the size (in bytes) of the common prefix. The latter can be conveniently stored using the 7 least significant bits of the suffix, which, as discussed, are unnecessary. With applications that exhibit high spatial locality (e.g., that sequentially scan memory), this approach can achieve significant compression factors with respect to the time-efficient implementation. However, it introduces additional computational costs, both during the logging phase (to identify the common prefix) and in the replay phase (as addresses need to be reconstructed).

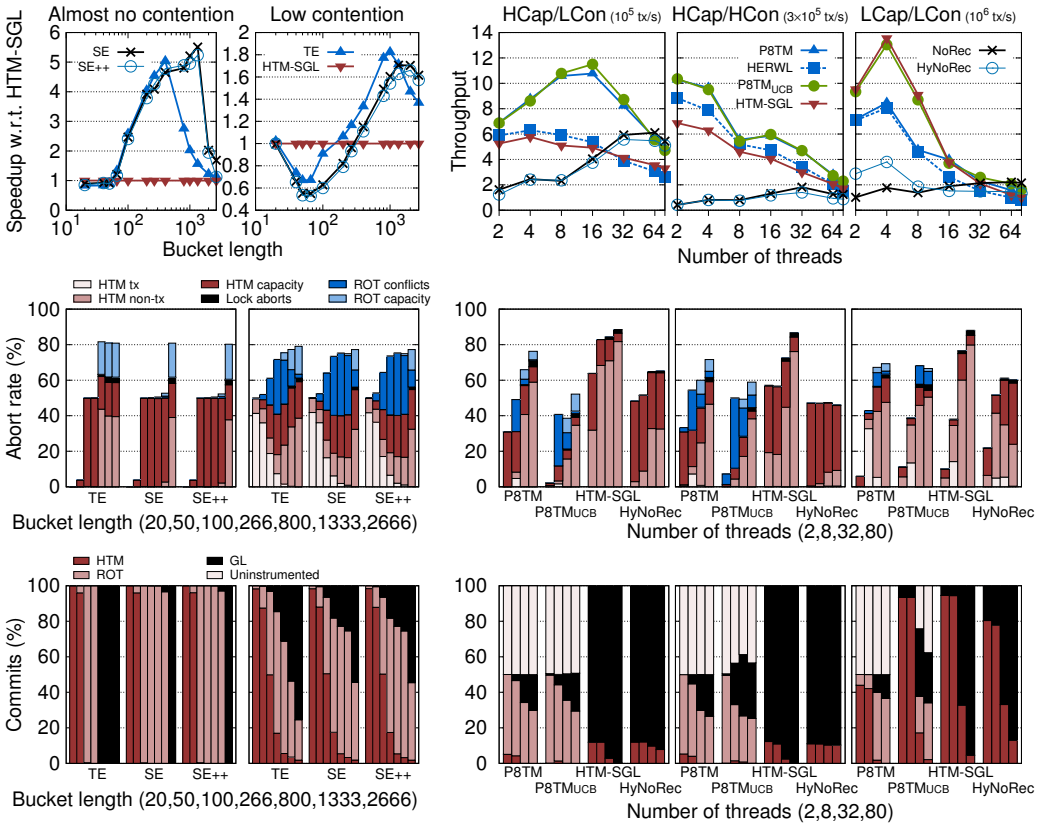
6 Evaluation

In this section we evaluate P8TM against state-of-the-art TM systems using a set of synthetic micro-benchmarks and complex, real-life applications. First, we start by evaluating both variants of read-set tracking to show how they are affected by the size of transactions and degree of contention. Then we conduct a sensitivity analysis aimed to investigate various factors that affect the performance of P8TM. To this end, we used a micro-benchmark that emulates a hashmap via lookup, insert, and delete transactions that accesses locations uniformly at random. This is a synthetic data structure composed of b buckets, where each bucket points to a linked-list, with an average length of l. By varying b and l we can control the degree of contention and probability of triggering capacity aborts respectively, which allows us to precisely stress different design aspects. Finally, we test P8TM using the popular STAMP benchmark suite [7].

We compare our solution with the following baselines:

(i) plain HTM with a global lock fallback (HTM-SGL),
(ii) NoRec with write back configuration,
(iii) the Hy-NoRec algorithm with three variables to synchronize transactions and NoRec fallback, and, finally,
(iv) the reduced hardware read-write lock elision algorithm HERWL (in this case, update transactions acquire the write lock while read-only transactions acquire the read lock).

Regarding the retry policy, we execute the HTM path 10 times and the ROT path 5 times before falling back to the next path, except upon a capacity abort when the next path is directly activated. These values and strategies were chosen after an extensive offline



(a) Evaluation of different implementations of read-set tracking.

(b) Sensitivity analysis: throughput and breakdown of abort rate and commits.

Figure 2 Micro-benchmarks (H=high, L=low, Cap=capacity, Con=contention).

experimentation and selecting the best configuration on average, regarding the number of retries and policies for capacity aborts (e.g., fallback immediately vs treating it as a conflict-induced abort). All results presented in this section represent the mean value of at least 5 runs. The experiments were conducted on a machine equipped with an IBM Power8 8284-22A processor that has 10 physical cores, with 8 hardware threads each, summing up to a total of 80 hardware threads. The source code, which is publicly available [1], was compiled with GCC 6.2.1 using `-O2` flag on top of Fedora 24 with Linux 4.5.5. Thread pinning was use to pin a thread per core at the beginning of each run for all the solutions, and threads were distributed evenly across the cores.

6.1 Read-set Tracking

The goal of this section is to understand the trade-off between the time-efficient and the space-efficient implementations of read-set tracking that were explained earlier in Section 5. We compare three variants of P8TM: i) a version using the time-efficient read-set tracking (TE), ii) a variant of space-efficient read-set tracking that only checks for prefixes of length 4 bytes, and otherwise stores the whole address (SE), and, finally, iii) a more aggressive variant of space-efficient read-set tracking that looks for prefixes of either 6 or 4 bytes (SE++).

Throughout this section, we fixed the number of threads to 10 (number of physical cores) and the percentage of update transactions at 100%, disabled the self-tuning module, and varied the buckets' lengths(l) across orders of magnitude to stress the ROT-path. First, we start with an almost contention-free workload (using $b = 10k$) to highlight the effect of capacity aborts alone. The speedup with respect to HTM-SGL, breakdown of abort rate (calculated as the aborts divided by the sum of aborted and committed transactions) and commits for this workload are shown in the left column of Figure 2a. As we can notice, the three variants of P8TM achieve almost the same performance as HTM-SGL with small transaction sizes that fit inside regular HTM transactions, as seen from the commits breakdown (l =∼20–50). However, when moving to larger transactions, the three variants start outperforming HTM-SGL achieving up to 5.5× higher throughput due to their ability to fit transactions within ROTs. By looking at the aborts breakdown in this region (l =∼100–266), we see that all P8TM variants suffer from almost 50% capacity aborts when first executing in HTM, and almost no capacity aborts when using the ROT path. This shows the clear advantage of the T2V mechanism and how it can fit more than 10× larger transactions in hardware.

Comparing TE with SE and SE++, we see that both space-efficient variants are able to execute larger transactions as ROTs: they do not suffer ROT capacity up to buckets of length ∼1333 items. Nevertheless, they incur an extra overhead, which is reflected as a slightly lower speedup than TE, before TE starts to experience ROT capacity aborts; only then their ability to further compress the *rot-rset* pays off. Again, by looking at the commits and aborts breakdown, we see that both space-efficient variants manage to commit all transactions as ROTs when TE is already using the GL (l =∼800–1333). Finally, when comparing SE and SE++, we notice that trying harder to find longer prefixes is not useful, due to the much lower probability of addresses sharing longer prefixes.

The right column of Figure 2a shows the results for a workload that exhibits a higher degree of contention ($b = 1k$). In this case, with transactions that fit inside regular HTM transactions, we see that HTM-SGL can outperform both SE and SE++ by up to 2× and TE by up to ∼30%. Since P8TM tries to execute transactions as ROTs after failing 10 times with HTM due to conflicts, the ROT path may be activated even in absence of capacity aborts; hence, the overhead of synchronizing ROTs and transaction becomes relevant also with small transactions. With larger transactions, we notice that the computational costs of SE and SE++ are more noticeable in this workload where they are always outperformed by TE, as long as this is able to fit at least 50% of transactions inside ROTs (up to l =∼800 items). Furthermore, the gains of SE and SE++ w.r.t. TE are much lower when compared to the contention-free workload. From this, we deduce that TE is more robust to contention. This was also confirmed with the other workloads that we will discuss next.

6.2 Sensitivity analysis

We now report the results of a sensitivity analysis that aimed to assess the impact of the following factors on P8TM's performance:

(i) the size of transactions,
(ii) the degree of contention, and
(iii) the percentage of read-only transactions.

We explored these three dimensions using the following configurations:

(i) high capacity, low contention, ($b = 1k$ and $l = 800$),
(ii) high capacity, high contention, ($b = 10$ and $l = 800$), and
(iii) low capacity, low contention ($b = 1k$ and $l = 40$).

We omitted showing the results for low capacity, high contention workload due to space restrictions, especially since they do not convey any extra information with respect to the low capacity, low contention scenario (which is actually even more favourable for HTM).

In these experiments we show two variants of P8TM, both equipped with the TE read-set tracking: with (P8TM$_{ucb}$) and without (P8TM) the self-tuning module enabled.

High capacity, low contention. The left most column of Figure 2b shows the throughput, abort rate and commits breakdown for the high capacity, low contention configuration with 50% update transactions.. We observe that, both variants of P8TM are able to outperform all the other TM solutions by up to 2×. This can be easily explained by looking at the commits breakdown, where both P8TM and P8TM$_{ucb}$ commit 50% of their transactions as UROs while the other 50% are committed mainly as ROTs up to 8 threads. On the contrary, HTM-SGL commits only 10% of the transactions in hardware and falls back to GL in the rest, due to the high capacity aborts it incurs. It is worth noting that the decrease in the percentage of capacity aborts, along with the increase of number of threads, is due to the activation of the fallback path, which forces other concurrent transactions to abort.

Although HERWL benefits from the URO path, P8TM was able to achieve ∼2× higher throughput, thanks to its ability of executing ROTs concurrently. Another interesting point is that P8TM$_{ucb}$ can outperform P8TM due to its ability to decrease the abort rate, as shown in the aborts breakdown. This is achieved by deactivating the HTM path, which spares from the cost of trying once in HTM before falling back to ROT (upon a capacity abort).

High capacity, high contention. The middle column of Figure 2b reports the results for the high capacity, high contention configuration with 50% update transactions. We can notice that although this workload is not scalable due to the high conflict rate, P8TM manages to achieve the highest throughput. Again this is due to P8TM's ability to fit large transactions into ROTs, almost all update transactions are executed in hardware up to 8 threads as can be seen from the commits breakdown. P8TM$_{ucb}$ also achieves higher throughput than P8TM after it disables the HTM path, which decreases the abort rate.

Low capacity, low-contention. In workloads where transactions fit inside HTM, it is expected that HTM-SGL will outperform all other TM solutions and that the overheads of P8TM will prevail. The results in the right most column of Figure 2b confirm this expectation: HTM-SGL outperforms all other solutions, achieving up to ∼1.75× higher throughput than P8TM. However, P8TM$_{ucb}$, thanks to its self-tuning ability, is the, overall, best performing solution, achieving performance comparable to HTM-SGL at low thread count, and outperforming it at high thread count. By inspecting the commits breakdown plots we see that P8TM$_{ucb}$ does not commit any transaction using ROTs up to 8 threads, avoiding the synchronization overheads that, instead, affect P8TM.

We note that, even though Hy-NoRec commits the same or higher percentage of HTM transactions than HTM-SGL, it is consistently outperformed by P8TM. This can be explained by looking at the performance of NoRec, which fails to scale due to the high instrumentation overheads it incurs with such short transactions. As for Hy-NoRec, its poor performance is a consequence of the inefficiency inherited by its NoRec fallback.

6.3 STAMP benchmark suite

STAMP is a popular benchmark suite that encompasses applications with different characteristics that share a common trait: they do not have any read-only transactions. Therefore, P8TM will not utilize the URO path and any gain it can achieve stems solely from executing

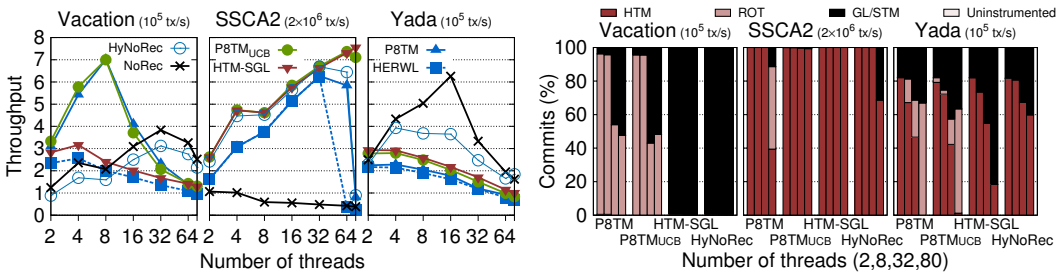


Figure 3 Throughput and breakdown of commits for the STAMP benchmarks.

ROTs in parallel. For space constraints we can only report the results for a subset of the STAMP benchmarks. The remaining benchmarks exhibit analogous trends and are available in an extended technical report [18].

Vacation is an application with medium sized transactions and low contention; hence, it behaves similarly to the previously analyzed high capacity, low contention workload. When looking at Figure 3, we can see trends very similar to the left most column of Figure 2b. P8TM is capable of achieving the highest throughput and outperforming HTM-SGL by up to ∼3.2× in this case. When looking at the breakdown of commits, we notice also the ability of P8TM to execute most of transactions as ROT up to 8 threads, while HTM-SGL never manages to commit transactions in hardware.

At high thread count we notice that NoRec and Hy-NoRec start to outperform both P8TM and P8TM$_{ucb}$. his can be explained by two reasons:

- with larger numbers of threads there is higher contention on hardware resources (note that starting from 32 threads ROT capacity aborts start to become frequent) and
- the cost of quiescence becomes more significant as threads have to wait longer.

Nevertheless, it is worth noting that the maximum throughput achieved by P8TM (at 8 threads) is ∼2× higher than NoRec (at 32 threads). This is due to the instrumentation overheads of these solutions. These overheads are completely eliminated in case of write accesses within P8TM and are much lower for read accesses—recall we only need to log the addresses and read them during validation.

SSCA2 generates transactions with small read/write sets and low contention. These are HTM friendly characteristics, and by looking at the throughput results in Figure 3 we see that HTM-SGL is able to outperform all the other baselines and scale up to 80 threads. This is also reflected in its ability to commit almost all transactions in hardware as shown in the commits breakdown. Although Hy-NoRec is able to achieve performance similar to HTM up to 32 threads, it is then outperformed due to the extra overheads it incurs to synchronize with the NoRec fallback.

Although P8TM commits almost all transactions using HTM up to 64 threads, it performed worse than both HTM-SGL and Hy-NoRec due to the costs of synchronization. An interesting observation is that the overhead is almost constant up to 32 threads. In fact, up to 64 threads there are no ROTs running and the overhead is is dominated by the cost of suspending and resuming the transaction. At 64 and 80 threads P8TM started to suffer also from capacity aborts similarly to Hy-NoRec. This led to a degradation of performance, with HTM-SGL achieving 7× higher throughput at 80 threads. This is a workload where P8TM$_{ucb}$ comes in handy as it manages to disable the ROT path and thus tends to employ HTM-SGL.

Yada has long transactions, large read/write set and medium contention. This is an example of a workload that is not hardware friendly and where hardware solutions are expected to be outperformed by software based ones. Figure 3 shows the clear advantage

of NoRec over any other solution, achieving up to 3× higher throughput than hardware based solutions. When looking at the commits and abort break down, one can see that up to 8 threads P8TM commits ∼80% of the transactions as either HTM or ROTs. Yet, despite P8TM manages to reduce the frequency of acquisition of the GL path with respect to HTM-SGL, it incurs overheads that end up outweighing the benefits provided by P8TM in terms of increased concurrency.

7 Conclusion

We presented P8TM, a TM system that tackles what is, arguably, the key limitation of existing HTM systems: the inability to execute transactions whose working sets exceed the capacity of CPU caches. This is achieved by novel techniques that exploit hardware capabilities available in POWER8 processors. Via an extensive experimental evaluation, we have shown that P8TM provides robust performance across a wide range of benchmarks, ranging from simple data structures to complex applications, and achieves remarkable speedups.

The importance of P8TM stems from the consideration that the best-effort nature of current HTM implementations is not expected to change in the near future. Therefore, techniques that mitigate the intrinsic limitations of HTM can broaden its applicability to a wider range of real-life workloads. We conclude by arguing that the performance benefits achievable by P8TM thanks to the use of the ROT and suspend/resume mechanisms represent a relevant motivation for integrating these features in future generations of HTM-enabled processors (like Intel's ones).

References

1 `https://github.com/shadyalaa/POWER8TM`, 2017.

2 A. Adl-Tabatabai, , T. Shpeisman, and J. Gottschlich. "draft specification of transactional language constructs for c++. *Intel*, 2012.

3 Y. Afek, A. Levy, and A. Morrison. Programming with hardware lock elision. In *Proceedings of the 18th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP'13)*, pages 295–296, 2013. `doi:10.1145/2442516.2442552`.

4 H. Boehm, J. Gottschlich, V. Luchangco, M. Michael, M. Moir, C. Nelson, T. Riegel, T. Shpeisman, and M. Wong. Transactional language constructs for c++. *ISO/IEC JTC1/SC22 WG21 (C++)*, 2012.

5 H. W. Cain, M. M. Michael, B. Frey, C. May, D. Williams, and H. Le. Robust architectural support for transactional memory in the power architecture. *SIGARCH Comput. Archit. News*, 41(3), 2013.

6 I. Calciu, T. Shpeisman, G. Pokam, and M. Herlihy. Improved single global lock fallback for best-effort hardware transactional memory. *9th ACM SIGPLAN Wkshp. on Transactional Computing*, 2014.

7 Chi Cao Minh, JaeWoong Chung, Christos Kozyrakis, and Kunle Olukotun. STAMP: Stanford transactional applications for multi-processing. In *IISWC'08*, 2008.

8 L. Dalessandro, F. Carouge, S. White, Y. Lev, M. Moir, M. L. Scott, and M. F. Spear. Hybrid norec: A case study in the effectiveness of best effort hardware transactional memory. In *ASPLOS'11*, 2011.

9 P. Damron, A. Fedorova, Y. Lev, V. Luchangco, M. Moir, and D. Nussbaum. Hybrid transactional memory. In *ASPLOS'06*, 2006.

10 D. Dice, T. L. Harris, A. Kogan, Y. Lev, and M. Moir. Hardware extensions to make lazy subscription safe. *CoRR*, abs/1407.6968, 2014.

11 D. Dice, Y. Lev, M. Moir, and D. Nussbaum. Early experience with a commercial hardware transactional memory implementation. In *ASPLOS'09*, 2009.

12 D. Dice and N. Shavit. Understanding tradeoffs in software transactional memory. In *CGO'07*, 2007.

13 N. Diegues and P. Romano. Self-tuning intel transactional synchronization extensions. In *ICAC'14*, 2014.

14 N. Diegues, P. Romano, and S. Garbatov. Seer: Probabilistic scheduling for hardware transactional memory. In *SPAA'15*, 2015.

15 N. Diegues, P. Romano, and L. Rodrigues. Virtues and limitations of commodity hardware transactional memory. In *PACT'14*, 2014.

16 P. Felber, S. Issa, A. Matveev, and P. Romano. Hardware read-write lock elision. In *EuroSys'16*, 2016.

17 B. Goel, R. Titos-Gil, A. Negi, S. A. McKee, and P. Stenstrom. Performance and energy analysis of the restricted transactional memory implementation on haswell. In *IPDPS'14*, 2014.

18 S. Issa, P. Felber, A. Matveev, and P. Romano. Extending hardware transactional memory capacity via rollback-only transactions and suspend/resume. Technical report, INESC-ID'17, 2017.

19 C.C Jacobi, T. Slegel, and D. Greiner. Transactional memory architecture and implementation for ibm system z. In *MICRO-45*, 2012.

20 S. Kumar, M. Chu, C. J. Hughes, P. Kundu, and A. Nguyen. Hybrid transactional memory. In *PPoPP'06*, 2006.

21 T. Lai and H. Robbins. Asymptotically efficient adaptive allocation rules. *Advances in Applied Mathematics*, 1985.

22 H.Q. Le, G.L. Guthrie, D.E. Williams, M.M. Michael, B.G. Frey, W.J. Starke, C. May, R. Odaira, and T. Nakaike. Transactional memory support in the ibm power8 processor. *IBM Journal of Research and Development*, 59(1), 2015.

23 A. Matveev and N. Shavit. Reduced hardware norec: A safe and scalable hybrid transactional memory. In *ASPLOS'15*, 2015.

24 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 *ISCA'15*, 2015.

25 R. M. Yoo, C. J. Hughes, K. Lai, and R. Rajwar. Performance evaluation of intel transactional synchronization extensions for high-performance computing. In *SC'13*, 2013.