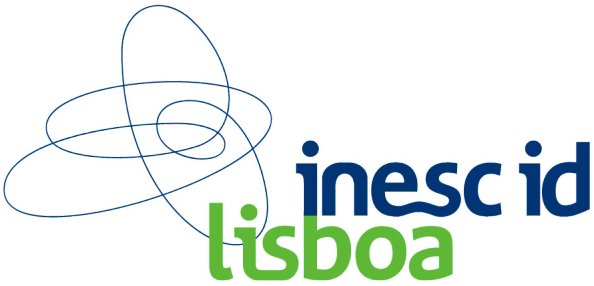

Technical Report RT/02/2017

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

Shady Issa
INESC-ID/IST

Pascal Felber
University of Neuchâtel

Alexander Matveev
MIT

Paolo Romano
INESC-ID/IST

February 10, 2017

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

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

[1]INESC-ID / Instituto Superior Técnico, University of Lisbon, Portugal
[2]University of Neuchatel, Switzerland
[3]MIT, USA

Abstract

Transactional memory, which aims at simplifying concurrent programming by bringing the familiar abstraction of transactions to parallel computing, has grown from a "research toy" to a mature technology integrated in mainstream programming language and CPU architectures. Yet, despite being supported in commodity processors from Intel and IBM, hardware transactional memory (HTM) suffers from some limitations that hamper its wide adoption. One such notable limitation is the inability to execute transactions whose working sets exceed the capacity of CPU caches. In this paper we propose a novel approach to mitigating this limitation on IBM's POWER8 architecture by leveraging a key combination of techniques: uninstrumented read-only transactions, ROT-based update transactions, HTM-friendly (software-based) read-set tracking, and self-tuning. Our algorithm, 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.

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 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 [6].

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 [19, 21, 27].

Existing hardware implementations share various architectural choices, although they do come in different flavours [22, 25, 32]. The key common treat 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 [7]:

- *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

[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.

of transaction that has lower overhead than regular transactions but also weaker semantics. In particular ROTs avoid tracing 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 as a unit 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 the spurious aborts and capacity limitations that affect HTM, while still allowing them to execute concurrency 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.

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 [16]. 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 is to develop "HTM-friendly" software-based read-set tracking. In fact, all the memory writes issued from within a ROT, including those needed to track in software the ROT 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 issued from within a ROT. P8TM integrates two read-set tracking mechanisms that explore different trade-offs between space and time efficiency.

Self-tuning. In order to ensure robust performance in a broad range of workloads, P8TM integrates a lightweight reinforcement learning mechanism (based on the UCB algorithm [24]) 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; or *(iii)* to even completely switch off ROTs/UROs, using only HTM.

We evaluated P8TM by means of extensive study that encompasses synthetic micro-benchmarks, the benchmarks in the Stamp suite [9], as well as a porting to TM of the popular TPC-C benchmark [30]. 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 [19, 21, 27]. 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 [11, 23] (HyTM) attempts to address this issue by falling back to software-based TM (STM) implementations when transactions cannot successfully execute in hardware. Hybrid NoREC is probably one of the most popular and effective HyTM designs proposed in the literature. Hybrid-NoRec [10] 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 [26] proposed to decompose a transaction running on the fallback path into multiple hardware transactions: a read-only prefix 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 Hy-NoRec. 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 conse-

quently 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 actually manage to 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* [14] 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. [8] 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 [12].

P8TM integrates a self-tuning approach that shares a common theoretical framework (the UCB reinforcement learning algorithm [24]) with Tuner [17]. 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 [18]. 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 [20], 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 is devoted to providing background information on POWER8's HTM system, which will be relevant to understand the operation of P8TM. Analogously to other HTM implementations, POWER8 provides APIs 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, i.e., 128 bytes. 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—which occupy the same space within the TMCAM.

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 the 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 force hardware speculation on memory writes and hide them from concurrent reads. This

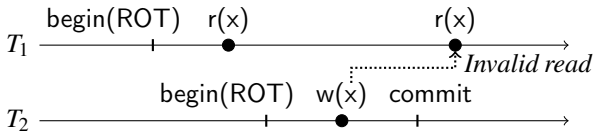

Figure 1: ROTs do not track reads and may observe different values when reading the same variable multiple times, even if all updates to the variable were performed in the context of transactions.

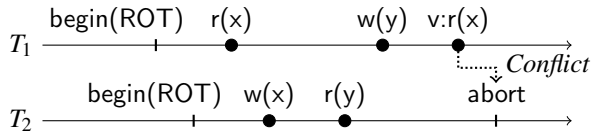

Figure 2: By re-reading x during *rot-rset* validation at commit time (denoted by v:r), T_1 forces an abort of T_2 that has updated x in the meantime, hence preventing an inconsistent execution.

allows us to cover 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 transactions commits, either as a transaction or a ROT, it first suspends the transaction and then executes the quiescence mechanism that waits for the completion of currently executing ROTs/URO transactions. In addition to that, in case of ROTs, it then executes an original *touch-based validation* step, which is described next, before resuming and committing. This process of "suspending and waiting" ensures that the writes of an update transaction will not overwrite and corrupt any other ROT/URO transaction that previously read any of these written memory locations.

4.2 Touch-based Validation

One core mechanism of our algorithm, which we call *Touch-To-Validate* (T2V), aims at enabling safe and concurrent execution of ROTs. Indeed, unlike regular transactions, ROTs do not track read accesses within the transaction. It is therefore unsafe to execute them concurrently as they are not serializable.

Consider the example shown in Figure 1. 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 2, 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 and abort concurrent yet uncommitted writers. This also means that the values read during *rot-rset* validation are irrelevant and ignored by the algorithm.

4.3 Basic Algorithm

We first present below the basic version of the P8TM algorithm assuming we only have ROTs and we blindly retry to execute failed ROTs irrespective of the abort cause. The pseudo-code is shown in Algorithm 1.

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 keep 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 from 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

Since 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

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] ← ⊥
```

read tracking and validation upon commit. Therefore, we extend the algorithm so that if first tries to use regular transactions, then upon failure 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 shown in Algorithm 2.

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 (Lines 11–15). Note that a simple quiescence, without a validation step afterwards, is sufficient in this case. 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 (Lines 35 and 50). The GL fallback uses a basic spin lock, which is acquired upon transaction begin (Lines 54–55) and released upon commit (Line 76). Observe 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 (Line 58), and that the GL transaction must actually wait for ROTs to fully complete, not just enter the commit phase as for the the other execution modes (Line 14). The rest of the algorithm is relatively straightforward.

Read-only Transactions

We finally describe the URO path that optimized for read-only (RO) transactions in which reads do not need to be tracked, hence significantly decreasing runtime overheads. This would also allow to execute large RO transactions that do not fit in hardware, and are doomed to execute after acquiring the global lock. The pseudo-code is shown in Algorithm 3.

To understand the intuition behind URO path, consider first that transactions that do not modify shared data cannot cause the abort of a transaction or a ROT . Furthermore, because transactions and ROTs buffer their writes and quiesce before committing, they cannot propagate inconsistent updates to RO transactions. Finally, GL and RO transactions cannot conflict with each other as long as they do not run concurrently. This is ensured by the quiescence phase after acquiring the global lock, and the fact that RO transactions do not start executing until the lock is free (Line 6).

Note that, if the lock is taken, RO transactions defer to the writer by resetting their status (Line 5) before waiting for the lock to be free and retrying the whole procedure. Otherwise we could run into a deadlock situation with an RO transaction waiting for the lock held by a GL transaction, while the latter is blocked in quiescence waiting for the former to complete its execution.

4.5 Correctness Argument.

When the GL path is active, concurrency is disabled. This is guaranteed by transactions in HTM path subscribing eagerly to the GL, and are thus aborted upon the activation of this path, and the quiescence call that is executed after the GL is acquired to wait for active ROTs or UROs. Atomicity of transaction in the HTM path is provided by the hardware against concurrent transactions/ROTs and by eager subscription to the GL.

As for the UROs, the quiescence mechanism guarantees two properties: *(i)* UROs activated after the start of an update transaction T, and before the end 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 hardware transaction/ROT commits; *(ii)* 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.

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. Nevertheless, since ROTs do modify the shared state, they can still produce non-serializable histories; such as the scenario in Figure 2. Since ROTs perform T2V before they commit and after the quiescence phase, and since updates of ROTs are performed atomically, then at least one ROT (among the ROTs that reached the commit stage together—concurrent ROTs) will re-read all its read-set again before any of the other concurrent ROTs do commit. Now, these

Algorithm 2 — P8TM: complete algorithm

```
1:  Shared variables:
2:      status[N] ← {⊥,⊥,...,⊥}                    ▷ One per thread
3:      glock ← FREE                                ▷ Spin lock to serialize transactions
4:  Local variables:
5:      tid ∈ [0..N]                                ▷ Identifier of current thread
6:      mode ∈ {HTM,ROT,GL}                         ▷ Transaction mode
7:      rot-rset ← ∅                                ▷ Transaction's read-set
8:  function READ(addr)                             ▷ Read shared variable
9:      if mode = ROT then
10:         rot-rset ← rot-rset ∪ {addr}            ▷ Track ROT reads
11: function SYNCHRONIZE
12:     s[N] ← status                               ▷ Read and copy all status variables
13:     for i ← 0 to N−1 do                         ▷ Wait until all threads...
14:         if s[i] = ACTIVE                        ▷ ...that are active...
            ..... ∨ (mode = GL ∧ s[i] = ROT-COMMITTING) then
15:             wait until status[i] ≠ s[i]         ▷ ...cross barrier
16: function TOUCH_VALIDATE
17:     for addr ∈ rot-rset do                      ▷ Re-read all elements...
18:         read addr                               ▷ ...from read-set
19: function BEGIN
20:     wait until glock = FREE                     ▷ Global lock must be free
21:     BEGIN_HTM                                   ▷ Try HTM first
22:     if mode ≠ HTM then                          ▷ If HTM fails...
23:         BEGIN_ROT                               ▷ ...fall back to ROT
24:         if mode ≠ ROT then                      ▷ If ROT also fails...
25:             BEGIN_GL                            ▷ ...default to global lock
26: function BEGIN_HTM
27:     trials ← 0
28:     repeat                                      ▷ Retry HTM a few times
29:         trials ← trials+1
30:         tx ← TX_BEGIN                           ▷ HTM begin
31:         if tx = STARTED then                    ▷ Success?
32:             if glock ≠ FREE then                ▷ Add lock to read-set
33:                 TX_ABORT                        ▷ Abort if lock taken
34:             mode ← HTM                          ▷ Run in HTM mode
35:     until mode = HTM                            ▷ Repeat until success...
        ..... ∨ tx = CAPACITY-ABORT                 ▷ ...or capacity abort...
        ..... ∨ trials > MAX-HTM-TRIALS             ▷ ...or too many trial
36: function BEGIN_ROT
37:     trials ← 0
38:     repeat                                      ▷ Retry ROT a few times
39:         trials ← trials+1
40:         status[tid] ← ACTIVE                    ▷ Indicate we are in ROT
41:         MEM_FENCE                               ▷ Make sure others know
42:         if glock ≠ FREE then                    ▷ Global lock taken?
43:             status[tid] ← ⊥                     ▷ Yes: defer to writer...
44:             wait until glock = FREE             ▷ ...wait...
45:             go to 40                            ▷ ...and retry
46:         rot-rset ← ∅                            ▷ Clear read-set
47:         tx ← TX_BEGIN_ROT                       ▷ HTM ROT begin
48:         if tx = STARTED then                    ▷ Success?
49:             mode ← ROT                          ▷ Run in ROT mode
50:     until mode = ROT                            ▷ Repeat until success...
        ..... ∨ tx = CAPACITY-ABORT                 ▷ ...or capacity abort...
        ..... ∨ trials > MAX-HTM-TRIALS             ▷ ...or too many trial
51: function BEGIN_GL
52:     status[tid] ← ⊥                             ▷ Not using TM
53:     MEM_FENCE                                   ▷ Make sure others know
54:     repeat                                      ▷ Acquire global lock
55:         wait until glock = FREE                 ▷ Test and...
56:     until CAS(glock,FREE,LOCKED)                ▷ ...test and set
57:     mode ← GL                                   ▷ Run in GL mode
58:     SYNCHRONIZE                                 ▷ Perform quiescence phase
59: function COMMIT
60:     switch mode do
61:         case HTM
62:             TX_SUSPEND                          ▷ Suspend transaction
63:             SYNCHRONIZE                         ▷ Perform quiescence phase
64:             TX_RESUME                           ▷ Resume transaction
65:             TX_COMMIT                           ▷ End transaction
66:         case ROT
67:             TX_SUSPEND                          ▷ Suspend transaction
68:             status[tid] ← ROT-COMMITTING        ▷ Tell others...
69:             MEM_FENCE                           ▷ ...we are committing
70:             TX_RESUME                           ▷ Resume transaction
71:             SYNCHRONIZE                         ▷ Quiescence inside ROT
72:             TOUCH_VALIDATE                      ▷ Touch to validate
73:             TX_COMMIT                           ▷ End transaction
74:             status[tid] ← ⊥
75:         case GL
76:             glock ← FREE                        ▷ Release global lock
```

Algorithm 3 — P8TM: URO path

```
1:  function BEGIN_RO
2:      status[tid] ← ACTIVE                        ▷ Indicate we are reader
3:      MEM_FENCE                                   ▷ Make sure writers see reader
4:      if glock ≠ FREE then                        ▷ Global lock taken?
5:          status[tid] ← ⊥                         ▷ Yes: defer to writer...
6:          wait until glock = FREE                 ▷ ...wait...
7:          go to 2                                 ▷ ...and retry
8:  function COMMIT_RO
9:      status[tid] ← ⊥                             ▷ Publish new status
```

reads are guaranteed to have taken place after the other concurrent ROTs have finished all their write operations, therefore the hardware will abort the conflicting due to read-after-write conflict. This guarantees that for any ROT to commit, there will be no other concurrent ROT that will commit and can not be serialized either before or after.

5 Read-set Tracking

The T2V mechanism requires to track the read-sets of update transactions 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. This 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. This approach 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 Self-tuning

In workloads where transactions fit the HTM's capacity restrictions, P8TM still 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 is workload dependent and is hard to determine via static analysis in applications that make use of pointers.

We address this issue by integrating into P8TM a self-tuning mechanism based on a lightweight reinforcement learning technique, UCB [24], which we shall describe shortly. 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

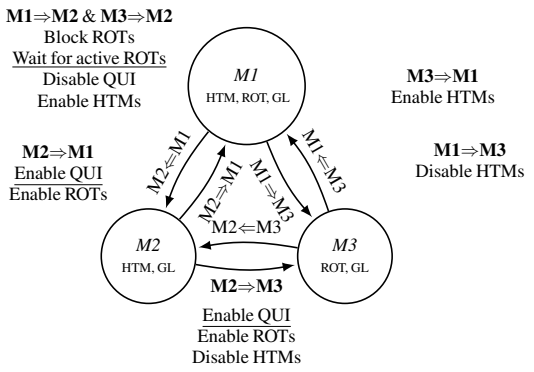

Figure 3: Different execution paths that can be used by transactions and rules for switching between them. Lines within rules represent necessary memory barriers.

to reduce the search space to be explored by the self-tuning mechanism, whenever we disable ROTs, we also disable UROs. In such cases read-only transactions are treated as update transactions.

Figure 3 shows the three paths and the rules for switching between them. When ROTs are disabled (M1⇒M2), the quiescence call within transactions can be skipped only once there are no more active ROTs. When switching from mode 2 to mode 3, instead, it is enough to enable ROTs after having ensure, via a memory fence, that all threads are informed about the need to enable quiescence. This force any active HTM transaction to perform the quiescence once it reaches its commit phase, while it will abort any active transaction that has reached commit stage but has not yet committed (since the flag is already part of the read-set). The other rules are straightforward.

UCB. Upper confidence bounds (UCB) [4] is a provably optimal solution to the multiarmed bandit problem [5], i.e., a classical reinforcement-learning problem that embodies the trade-off between exploration and exploitation. In the multiarmed bandit, a gambler tries to maximize the reward obtained from playing different levers of a multiarm slot machine, where the levers' rewards are random variables with a priori unknown distributions. We opted for using the UCB technique given that it provides strong theoretical guarantees[2] while imposing negligible computational overheads. After an initial phase, in which every lever is sampled once, UCB estimates the expected reward for lever i as $\bar{x}_i + \sqrt{(2\log n)/n_i}$, where: $\bar{x}_i$ is the average reward for lever i; n is the number of the current trial; and n_i is the number of times the lever i has been tried.

In order to use UCB in P8TM, we associate each execution mode to a different lever, and its reward to the throughput obtained by using that mode during a sampling interval of 100 microseconds.

[2]Logarithmic bounds on the cumulative error, called regret, from playing non-optimal levers even in finite time horizons [4].

7 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 manipulates a hashmap via lookup, insert, and delete transactons. Finally, we test P8TM using two complex, realistic workloads: the popular STAMP benchmark suite [9] and a port of the TPCC benchmark for in-memory databases [30].

We compare our solution with the following baselines: *(1)* plain HTM with a global lock fallback (HTM-SGL), *(2)* NoRec with write back configuration, *(3)* the Hybrid NoRec algorithm with three variables to synchronize transactions and NoRec fallback, and finally *(4)* 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 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 doing an extensive offline experiment and selecting the best on average with different number of retries and different capacity aborts handling policies (e.g., fallback immediately vs treating it as a normal 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 IBM Power8 8284-22A processor that has 10 physical cores, with 8 hardware threads each. The source code, which is publicly available here [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.

7.1 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)* 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 version 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 transaction length across orders of magnitude to stress the ROT-path.

First, we start with an almost contention-free workload to highlight the effect of capacity aborts alone. The speedup with respect to HTM-SGL, breakdown of

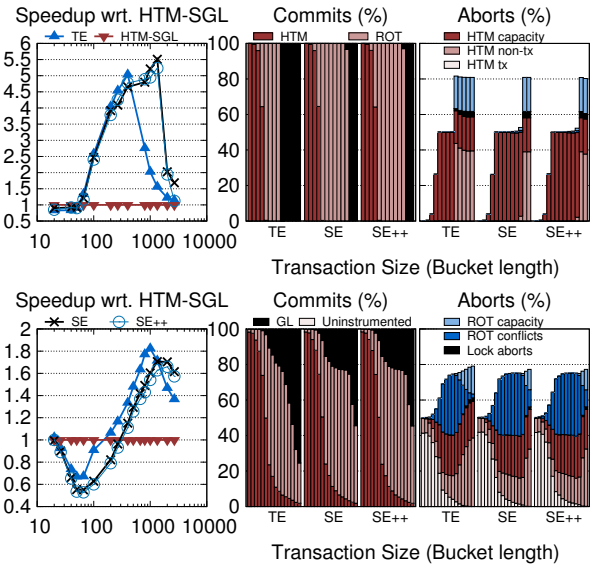

Figure 4: Evaluation of different implementations of read-set tracking.

commits and aborts for this workload is shown in the first row of Figure 4. 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. However, when moving to larger transactions, the three variants start outperforming HTM-SGL by up to 5.5× due to their ability to fit transactions within ROTs. By looking at the aborts breakdown in this region, 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 even larger transactions as ROTs. Nevertheless, they incur an extra overhead, which is reflected as a slightly lower speedup than TE, before this starts to experience ROT capacity aborts; only then their ability to further compress the read-set within TMCAM 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 forced to execute using the GL. Finally, when comparing SE and SE++, as in this workload there is a very low probability that the accessed addresses share 6 bytes long prefixes.

The second row in Figure 4 shows the results for a workload that exhibits a higher degree of contention. 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. 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.

7.2 Sensitivity analysis

We now report the results of a sensitivity analysis that aimed to assess the impact of the following factor on P8TM's performance: *(1)* the size of transactions, *(2)* the degree of contention, and *(3)* the percentage of read-only transactions. We explored these three dimensions using the following configurations: *(i)* high capacity, low contention, *(ii)* high capacity, high contention, and *(iii)* low capacity, low contention, with 10%, 50%, and 90% update transactions. 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. Figure 5 shows the throughput, commits and abort breakdown for the high capacity, low contention configuration. We observe that for the read dominated workload, both variants of P8TM are able to outperform all the other TM solutions by up to 7×. This can be easily explained by looking at the commits breakdown, where both P8TM and P8TM$_{ucb}$ commit 90% of their transactions as UROs while the other 10% are committed as ROTs. 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 is designed for such workloads, P8TM was able to achieve ∼2.4× higher throughput, thanks to its ability of executing ROTs concurrently. Another interesting point is that P8TM$_{ucb}$ can outperform P8TM thanks to 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).

We can see similar trend when moving to the workloads with more update transactions: P8TM and P8TM$_{ucb}$ outperform HTM-SGL by ∼2.2× and ∼1.4× in the 50% and 90% workloads, respectively. They also achieve the highest throughput in all workloads among all the considered baselines. By looking at the breakdown of

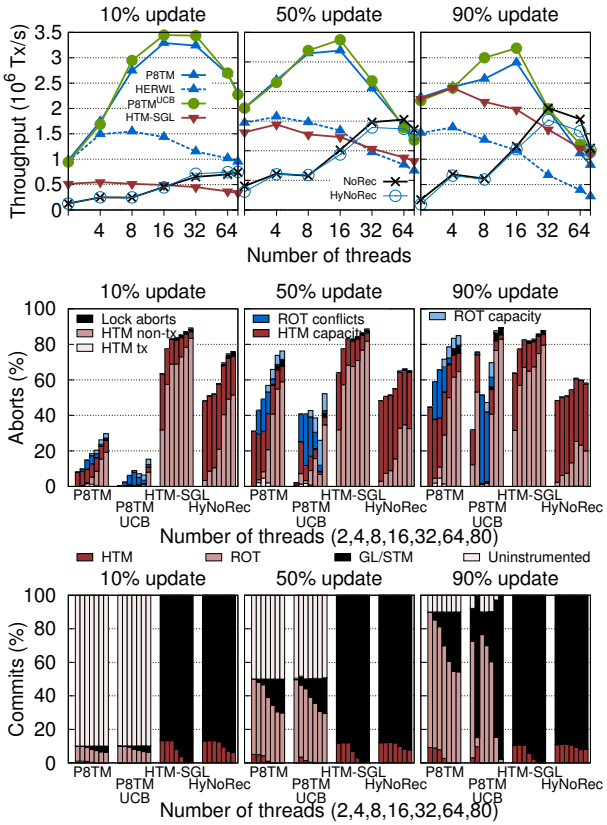

Figure 5: High capacity-low contention configuration: throughput, abort rate, and breakdown of commit modes at 10%, 50% and 90% update ratios.

commits, we can see that P8TM executes almost all update transactions using either HTMs or ROTs up to 8 threads, unlike HTM-SGL that only executes 10% of transactions in hardware. At high thread count we notice that NoRec and Hy-NoRec start to outperform both P8TM and P8TM$_{ucb}$, especially in the 90% workload. This can be explained by two reasons: *(1)* 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 *(2)* the cost of quiescence becomes more significant as threads have to wait longer. Despite that, P8TM variants achieve 2× and ∼1.4× higher throughput than NoRec and Hy-NoRec, when comparing their maximum throughputs regardless of the thread count.

High capacity, high contention. Figure 6 reports the results for the high capacity, high contention configuration. Trends for read dominated workloads are similar to the case of lower contention degree. However, scalability is much lower here due to the higher conflict rate. When considering the workloads with 50% and 90% update transactions, we notice that P8TM still achieves the highest throughput. Moreover, unlike in the low contention scenario, P8TM outperforms NoRec nor Hy-NoRec even at high thread count. Although HERWL uses URO to execute RO transactions, it was unable to scale even in the

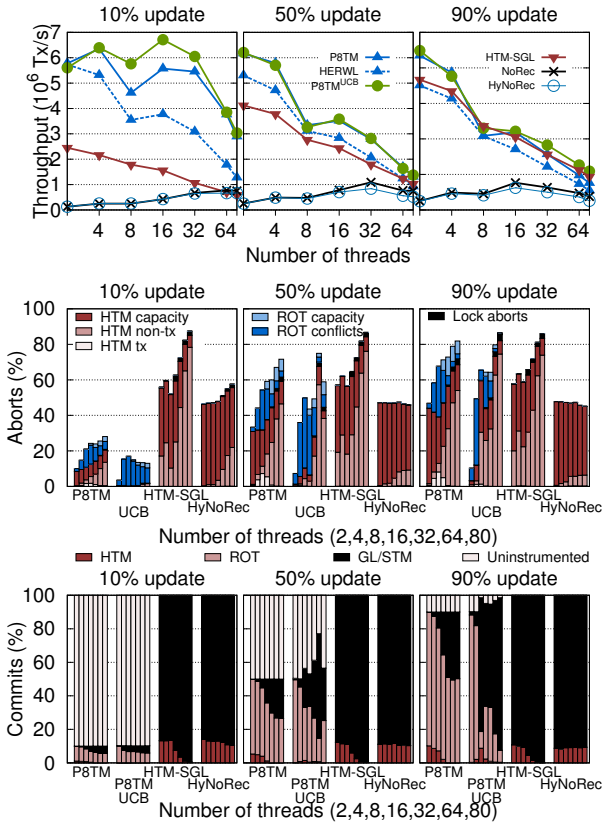

Figure 6: High capacity, high contention configuration: throughput, abort rate, and breakdown of commit modes at 10%, 50% and 90% update ratios.

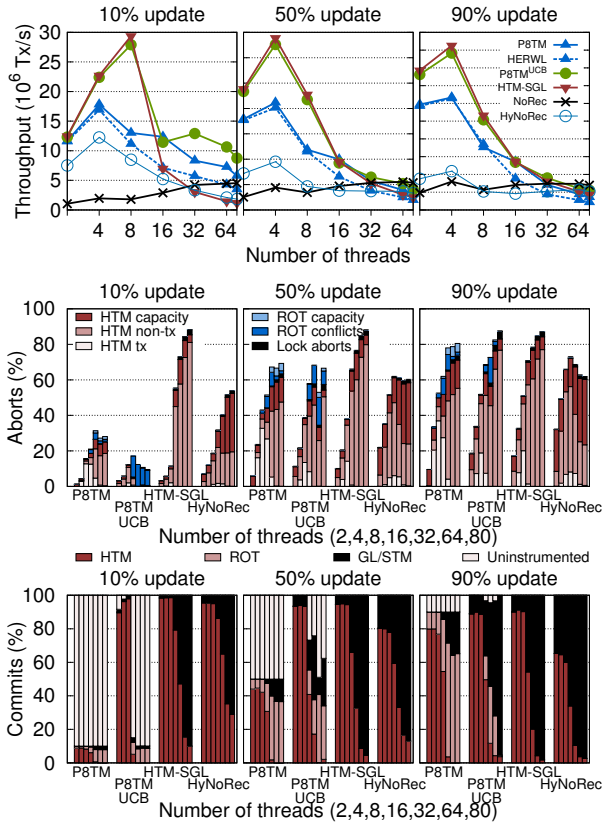

Figure 7: Low capacity, low contention configuration: throughput, abort rate, and breakdown of commit modes at 10%, 50% and 90% update ratios.

90% read-only workload, where its throughput is 2.5× lower than P8TM's. Again this is due to its inability to execute concurrent writers. This clearly indicates that T2V is beneficial even in read-dominated workloads.

Low capacity, low-contention. In workloads where transactions fit in HTM, it is expected that HTM-SGL will outperform all other TM solutions and that the overheads of P8TM will prevail. Results in Figure 7 confirm this expectation: HTM-SGL outperforms all other solutions, regardless of the ratio of read-only transactions, achieving up to 2.5× 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 all other approaches 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.

It is worth noting, though, that P8TM, with 90% read-only transactions, does outperform HTM-SGL beyond 16 threads. By inspecting the breakdown of aborts and commits we notice that when hardware multithreading is enabled the performance of HTM-SGL deteriorates dramatically, due to the increased contention on hardware resources. Conversely, P8TM can still execute transactions in ROTs, hence achieving higher throughput.

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.

7.3 STAMP benchmark suite

STAMP is a popular benchmark suite in the TM domain, 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 ROTs in parallel. We omitted the results of the Bayes benchmark due to its high variance [31]. We omit also Labyrinth, due to space restrictions, as its transactions do not fit in neither HTM nor ROT—hence, exhibiting very similar performance trend to Yada.

Genome and **Vacation** are two applications with medium sized transactions and low contention; hence, they behave similarly to the previously analyzed high capacity, low contention workloads. When looking at Figure 8, we can see trends very similar to the workloads with high up-

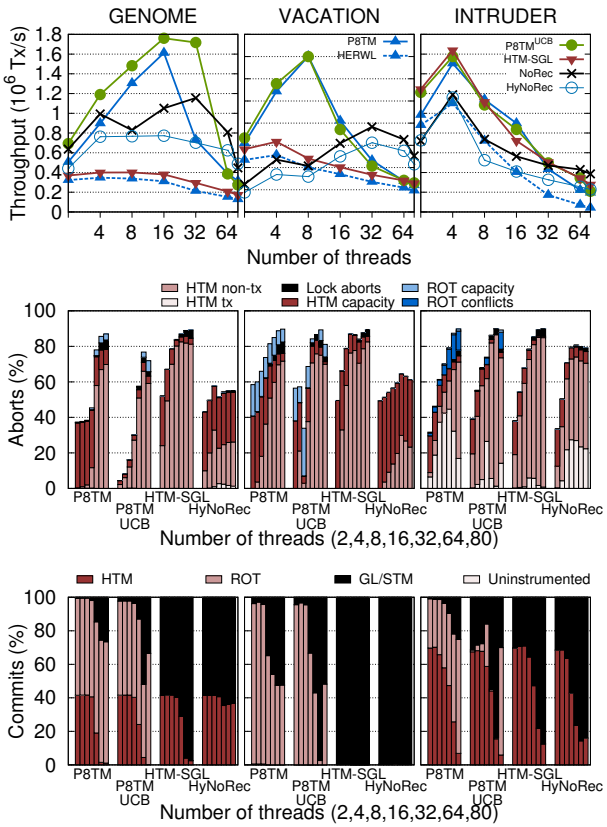

Figure 8: Throughput, abort rate, and breakdown of commit modes of STAMP benchmarks (1).

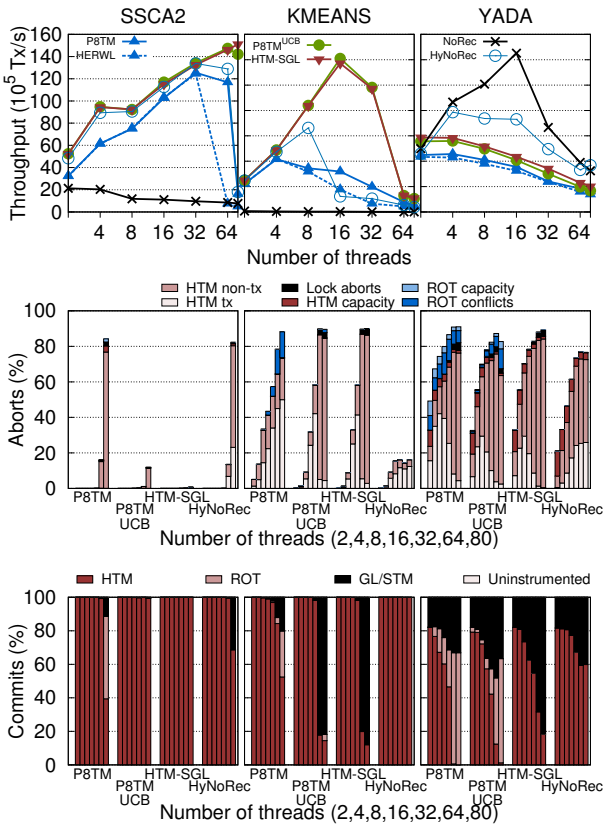

Figure 9: Throughput, abort rate, and breakdown of commit modes of STAMP benchmarks (2).

date ratios in Figure 5. P8TM is capable of achieving the highest throughput and outperforming HTM-SGL by up to 4.5× in case of Genome and ∼3.2× in the case of Vacation. Again P8TM$_{ucb}$ is even able to achieve higher throughput than P8TM due to deactivating the HTM path when capacity aborts are encountered, thus decreasing the abort rate. When looking at the breakdown of commits, we notice also the ability of P8TM to execute most of transactions in either HTM or ROT at low thread counts. One difference between Genome and Vacation is that, in Vacation, HTM-SGL never manages to commit transactions in hardware.

We also notice the same drawback at high number of threads when comparing P8TM to NoRec and Hy-NoRec. Nevertheless, it is worth noting that the maximum throughput achieved by P8TM (at 16 threads) is 1.5× and 2× higher than NoRec (at 32 threads) in Genome and Vacation, respectively. This is due to instrumentation overheads of these solutions. These overheads are completely eliminated in case of write accesses within P8TM and are much lower for read accesses.

Intruder generates transactions with medium read-/write sets and high contention. This results in a similar performance for both P8TM and HTM-SGL: they achieve almost the same peak throughput at 8 threads and follow the same pattern with increasing number of threads. Although P8TM manages to execute all transactions as either HTM transactions or ROTs at low numbers of threads, given the low level of parallelism, the synchronization overheads incurred by P8TM are not outweighed by its ability to run ROTs concurrently. Nevertheless, P8TM$_{ucb}$ manages to overcome this limitation by disabling the ROT path and avoid these overheads. Both NoRec and HyNoRec were outperformed, which is again simply due to their high instrumentation costs.

SSCA2 and **Kmeans** generate transactions with small read/write sets and low contention. These are HTM friendly characteristics, and by looking at the throughput results in Figure 9 we see that HTM-SGL is able to outperform all the other baselines and scale up to 80 threads in case of SSCA2 and up to 16 threads in case of Kmeans. Although Hy-NoRec was able to achieve performance similar to HTM up to 32 threads in SSCA2 and 8 threads in KMeans, it was then outperformed due to the extra overheads it incurs to synchronize with the NoRec fallback. These overheads lead to increased capacity aborts as seen in the aborts breakdown.

Although P8TM commits almost all transactions using HTM up to 64 threads, it performed worse than both HTM-SGL and Hy-NoRec in SSCA2 due to the costs of synchronization. An interesting observation is that the overhead is almost constant up to 32 threads, since up to 64 threads there are no ROTs running and the overhead of

the quiescence call 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. Similar trends can be seen for KMeans, however with different threads counts and with lower adverse effects for P8TM. Again, these are workloads where P8TM$_{ucb}$ comes in handy as it manages to disable the ROT path and thus tends to employ HTM-SGL, which is the most suitable solution for these workloads.

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 9 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. Moreover, unlike Intruder where HTM-SGL was able to commit a smaller percentage of transactions in hardware, HTM-SGL is unable to scale with Yada. This can be related to the difference in the nature of workloads, where the transactions that trigger capacity abort form the critical path of execution; hence with such workloads it is not preferable to use hardware-based solutions.

7.4 TPC-C benchmark

TPC-C represents a wholesale supplier benchmark for relational databases [30]. In this work we use a version ported to work on an in-memory database [29], which we adapted to support TM. TPCC has 5 different types of transactions, two of which are read-only. Figure 10 shows the results for workloads with 10%, 50%, and 90% update transactions that consists of a mix of the five types of transactions.

Throughput results show clear advantage of P8TM over all the other baselines in all workloads, regardless of the number of active threads. When compared with software based solutions, P8TM is able to achieve up to 5× higher throughput than both NoRec and Hy-NoRec at 16 threads in the 90% update workload. Although both NoRec and Hy-NoRec can scale up to 16 threads, their lower performance can be explained by the much lower instrumentation overheads that P8TM incurs when compared to software-based solutions. When compared to HTM-SGL, P8TM achieves 5.5× higher throughput with workloads that have a high percentage of read-only transactions, thanks to the URO path. When moving to workloads with higher percentages of update transactions, P8TM still outperforms HTM-SGL by 2× and 1.25× on the 50% and 90% update workloads, respectively. Again, looking at the breakdown plots, we can notice that P8TM is able to commit all update transactions either as HTM or ROTs up to 8 threads. We notice that P8TM$_{ucb}$ manages to achieve even further improvement in throughput by disabling the HTM path, hence decreasing the abort rate significantly.

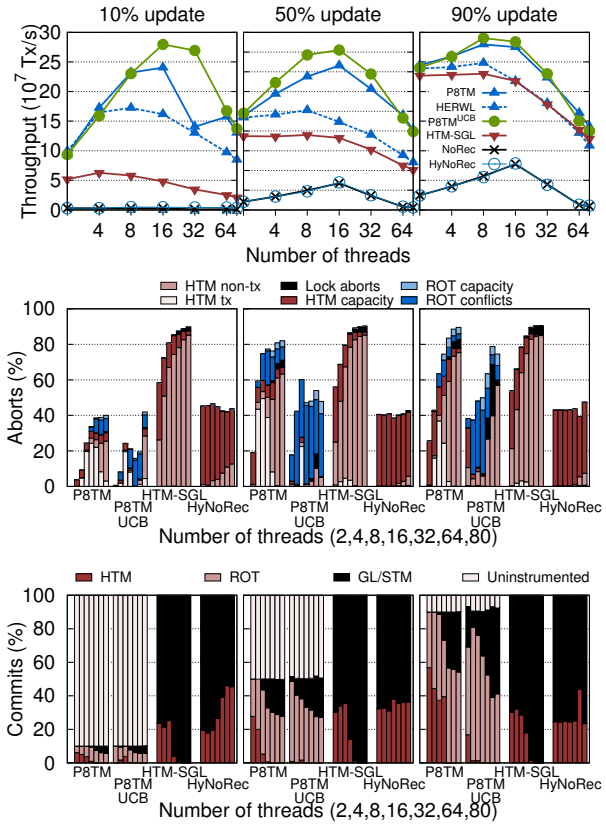

Figure 10: Throughput, abort rate, and breakdown of commit modes of TPCC at 10%, 50% and 90% update ratios.

8 Conclusion

In this paper we presented P8TM, a transactional memory system that tackles the capacity limitation of HTM, which is a main obstacle for its adoption in real-life applications. This was achieved by novel techniques that exploit hardware capabilities available in commodity processors. Together with self-tuning mechanisms, P8TM is capable of providing efficient concurrent execution paths that can accommodate up to 10× larger transactions in hardware. Via an extensive experimental evaluation, we have shown that P8TM provides robust perforamnce across a wide range of benchmarks, ranging from simple data structures to complex applications, and achieves remarkable speedups.

The importance of P8TM rises from the fact that the characteristics of HTM are not expected to change in the near future, due to the high costs of hardware modifications. Therefore, techniques that mitigate the intrinsic limitations of HTM can broaden its applicability to a wider range of realistic settings. Furthermore, they highlight the relevance of hardware features such as ROTs and suspend/resume, which are available in the IBM POWER8 processor but not part of Intel TSX.

References

[1] https://github.com/shadyalaa/POWER8TM, 2017.

[2] ADL-TABATABAI, A.-R., , SHPEISMAN, T., AND GOTTSCHLICH, J. "draft specification of transactional language constructs for c++. *Intel* (2012).

[3] AFEK, Y., LEVY, A., AND MORRISON, A. Programming with hardware lock elision. In *Proceedings of the 18th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming* (New York, NY, USA, 2013), PPoPP '13, ACM, pp. 295–296.

[4] AUER, P. Using confidence bounds for exploitation-exploration trade-offs. *Journal of Machine Learning Research 3*, Nov (2002), 397–422.

[5] BERRY, D., AND FRISTEDT, B. Bandit problems. *London: Chapman and Hall* (1985).

[6] BOEHM, H., GOTTSCHLICH, J., LUCHANGCO, V., MICHAEL, M., MOIR, M., NELSON, C., RIEGEL, T., SHPEISMAN, T., AND WONG, M. Transactional language constructs for c++. *ISO/IEC JTC1/SC22 (Programming languages and operating systems) WG21 (C++)* (2012).

[7] CAIN, H. W., MICHAEL, M. M., FREY, B., MAY, C., WILLIAMS, D., AND LE, H. Robust architectural support for transactional memory in the power architecture. *SIGARCH Comput. Archit. News 41*, 3 (June 2013), 225–236.

[8] CALCIU, I., SHPEISMAN, T., POKAM, G., AND HERLIHY, M. Improved single global lock fallback for best-effort hardware transactional memory. *9th ACM SIGPLAN Wkshp. on Transactional Computing* (2014).

[9] CAO MINH, C., CHUNG, J., KOZYRAKIS, C., AND OLUKOTUN, K. STAMP: Stanford transactional applications for multi-processing. In *IISWC '08* (2008).

[10] DALESSANDRO, L., CAROUGE, F., WHITE, S., LEV, Y., MOIR, M., SCOTT, M. L., AND SPEAR, M. F. Hybrid norec: A case study in the effectiveness of best effort hardware transactional memory. In *Proceedings of the Sixteenth International Conference on Architectural Support for Programming Languages and Operating Systems* (New York, NY, USA, 2011), ASPLOS XVI, ACM, pp. 39–52.

[11] DAMRON, P., FEDOROVA, A., LEV, Y., LUCHANGCO, V., MOIR, M., AND NUSSBAUM, D. Hybrid transactional memory. In *Proceedings of the 12th International Conference on Architectural Support for Programming Languages and Operating Systems* (New York, NY, USA, 2006), ASPLOS XII, ACM, pp. 336–346.

[12] DICE, D., HARRIS, T. L., KOGAN, A., LEV, Y., AND MOIR, M. Hardware extensions to make lazy subscription safe. *CoRR abs/1407.6968* (2014).

[13] DICE, D., KOGAN, A., LEV, Y., MERRIFIELD, T., AND MOIR, M. Adaptive integration of hardware and software lock elision techniques. In *Proceedings of the 26th ACM Symposium on Parallelism in Algorithms and Architectures* (New York, NY, USA, 2014), SPAA '14, ACM, pp. 188–197.

[14] DICE, D., LEV, Y., MOIR, M., AND NUSSBAUM, D. Early experience with a commercial hardware transactional memory implementation. In *Proceedings of the 14th International Conference on Architectural Support for Programming Languages and Operating Systems* (New York, NY, USA, 2009), ASPLOS XIV, ACM, pp. 157–168.

[15] DICE, D., SHALEV, O., AND SHAVIT, N. Transactional locking ii. In *Proceedings of the 20th International Conference on Distributed Computing* (Berlin, Heidelberg, 2006), DISC'06, Springer-Verlag, pp. 194–208.

[16] DICE, D., AND SHAVIT, N. Understanding tradeoffs in software transactional memory. In *International Symposium on Code Generation and Optimization (CGO'07)* (March 2007), pp. 21–33.

[17] DIEGUES, N., AND ROMANO, P. Self-tuning intel transactional synchronization extensions. In *11th International Conference on Autonomic Computing, ICAC '14, Philadelphia, PA, USA, June 18-20, 2014.* (2014), X. Zhu, G. Casale, and X. Gu, Eds., USENIX Association, pp. 209–219.

[18] DIEGUES, N., ROMANO, P., AND GARBATOV, S. Seer: Probabilistic scheduling for hardware transactional memory. In *Proceedings of the 27th ACM Symposium on Parallelism in Algorithms and Architectures* (New York, NY, USA, 2015), SPAA '15, ACM, pp. 224–233.

[19] DIEGUES, N., ROMANO, P., AND RODRIGUES, L. Virtues and limitations of commodity hardware transactional memory. In *Proceedings of the 23rd International Conference on Parallel Architectures and Compilation* (New York, NY, USA, 2014), PACT '14, ACM, pp. 3–14.

[20] FELBER, P., ISSA, S., MATVEEV, A., AND ROMANO, P. Hardware read-write lock elision. In *Proceedings of the Eleventh European Conference on Computer Systems* (New York, NY, USA, 2016), EuroSys '16, ACM, pp. 34:1–34:15.

[21] GOEL, B., TITOS-GIL, R., NEGI, A., MCKEE, S. A., AND STENSTROM, P. Performance and energy analysis of the restricted transactional memory implementation on haswell. In *2014 IEEE 28th International Parallel and Distributed Processing Symposium* (May 2014), pp. 615–624.

[22] JACOBI, C., SLEGEL, T., AND GREINER, D. Transactional memory architecture and implementation for ibm system z. In *Proceedings of the 2012 45th Annual IEEE/ACM International Symposium on Microarchitecture* (Washington, DC, USA, 2012), MICRO-45, IEEE Computer Society, pp. 25–36.

[23] KUMAR, S., CHU, M., HUGHES, C. J., KUNDU, P., AND NGUYEN, A. Hybrid transactional memory. In *Proceedings of the Eleventh ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming* (New York, NY, USA, 2006), PPoPP '06, ACM, pp. 209–220.

[24] LAI, T., AND ROBBINS, H. Asymptotically efficient adaptive allocation rules. *Advances in Applied Mathematics* (1985).

[25] LE, H., GUTHRIE, G., WILLIAMS, D., MICHAEL, M., FREY, B., STARKE, W., MAY, C., ODAIRA, R., AND NAKAIKE, T. Transactional memory support in the ibm power8 processor. *IBM Journal of Research and Development 59*, 1 (Jan 2015), 8:1–8:14.

[26] MATVEEV, A., AND SHAVIT, N. Reduced hardware norec: A safe and scalable hybrid transactional memory. In *Proceedings of the Twentieth International Conference on Architectural Support for Programming Languages and Operating Systems* (New York, NY, USA, 2015), ASPLOS '15, ACM, pp. 59–71.

[27] NAKAIKE, T., ODAIRA, R., GAUDET, M., MICHAEL, M. M., AND TOMARI, H. Quantitative comparison of hardware transactional memory for blue gene/q, zenterprise ec12, intel core, and power8. In *2015 ACM/IEEE 42nd Annual International Symposium on Computer Architecture (ISCA)* (June 2015), pp. 144–157.

[28] RIEGEL, T., MARLIER, P., NOWACK, M., FELBER, P., AND FETZER, C. Optimizing hybrid transactional memory: The importance of nonspeculative operations. In *Proceedings of the Twenty-third Annual ACM Symposium on Parallelism in Algorithms and Architectures* (New York, NY, USA, 2011), SPAA '11, ACM, pp. 53–64.

[29] STONEBRAKER, M., MADDEN, S., ABADI, D. J., HARIZOPOULOS, S., HACHEM, N., AND HELLAND, P. The end of an architectural era: (it's time for a complete rewrite). In *Proceedings of the 33rd International Conference on Very Large Data Bases* (2007), VLDB '07, VLDB Endowment, pp. 1150–1160.

[30] TPC COUNCIL. TPC-C Benchmark. http://www.tpc.org/tpcc, 2011.

[31] WANG, Q., KULKARNI, S., CAVAZOS, J., AND SPEAR, M. A transactional memory with automatic performance tuning. *ACM Trans. Archit. Code Optim. 8*, 4 (Jan. 2012), 54:1–54:23.

[32] YOO, R. M., HUGHES, C. J., LAI, K., AND RAJWAR, R. Performance evaluation of intel transactional synchronization extensions for high-performance computing. In *International Conference for High Performance Computing, Networking, Storage and Analysis, SC'13, Denver, CO, USA - November 17 - 21, 2013* (2013), W. Gropp and S. Matsuoka, Eds., ACM, pp. 19:1–19:11.