Enhancing Efficiency of Hybrid Transactional Memory via Dynamic Data Partitioning Schemes

Pedro Raminhas
INESC-ID, Instituto Superior Técnico
University of Lisbon

Shady Issa
INESC-ID, Instituto Superior Técnico
University of Lisbon

Paolo Romano
INESC-ID, Instituto Superior Técnico
University of Lisbon

Abstract—**Transactional Memory (TM) is an emerging paradigm that promises to significantly ease the development of parallel programs. Hybrid TM (HyTM) is probably the most promising implementation of the TM abstraction, which seeks to combine the high efficiency of hardware implementations (HTM) with the robustness and flexibility of software-based ones (STM). Unfortunately, though, existing Hybrid TM systems are known to suffer from high overheads to guarantee correct synchronization between concurrent transactions executing in hardware and software.**

This article introduces DMP-TM (Dynamic Memory Partitioning-TM), a novel HyTM algorithm that exploits, to the best of our knowledge for the first time in the literature, the idea of leveraging operating system-level memory protection mechanisms to detect conflicts between HTM and STM transactions. This innovative design allows for employing highly scalable STM implementations, while avoiding instrumentation on the HTM path.

This allows DMP-TM to achieve up to $\sim 20\times$ speedups compared to state of the art Hybrid TM solutions in uncontended workloads. Further, thanks to the use of simple and lightweight self-tuning mechanisms, DMP-TM achieves robust performance even in unfavourable workload that exhibits high contention between the STM and HTM path.

I. Introduction

Multicore processors have become ubiquitous in today's computing systems, and the foreseeable future trends point towards increasingly large and complex parallel architectures. Developing applications capable of full untapping the potential of modern parallel architectures, though, is far from being a trivial task. The problem of regulating concurrent access to shared state, in a scalable and provably correct way, is a key source of complexity that developers of parallel programs need to face. The traditional approach to this problem is to rely on ad-hoc designed lock-based synchronization schemes. Unfortunately, though, the usage of lock-based synchronization, especially at a fine-grain level, is prone to subtle concurrency bugs and leads to developing non-composable [1] programs that are notoriously complex to debug and reason about [2].

Transactional Memory (TM) [3] answers the urge to simplify parallel programming by borrowing the familiar abstraction of transactions from the database domain: programmers only need to identify which code blocks should be executed atomically, delegating to the implementation of the TM run-time the problem of how to ensure atomicity.

The plethora of alternative systems that have been proposed in the TM literature, e.g., [4]–[7], can be coarsely classified depending on whether they rely on hardware implementations (HTM), on software ones (STM), or on combinations thereof, where the latter are often referred to as Hybrid TM (HyTM).

HTM and STM have complementary pros and cons. STM, due to its software nature, can accommodate transactions that access a virtually arbitrary large number of memory regions. Further, state of the art STM systems encapsulates sophisticated concurrency control schemes, which allow for a high degree of concurrency and have been shown to achieve high scalability. However, the STM's need for instrumenting in software read and write memory accesses represents a major source of overhead, which can impose a severe toll on performance [8].

Existing HTM implementations, conversely, leverage on the pre-existing cache coherency protocol to track conflicts among transactions and ensure atomicity. By avoiding the overhead of software instrumentations, HTM can achieve high efficiency. However, the cache-centric nature of current HTM implementations imposes also severe limitations. In particular, existing HTM systems provide no guarantees on the ability to commit transactions, even if these run solo for a variety of reasons, e.g., exceeding cache capacity or being subject to a context switch. Due to these limitations, HTMs require an alternative fall-back path, typically a single global lock that is activated in case a transaction fails too many times in hardware.

HyTMs seek to obtain the best of STM and HTM by allowing HTM transactions to use some STM implementation on their fallback path. Unfortunately, despite the number of proposals in this area [7], [9]–[11], existing HyTM implementations still suffer from large synchronization overheads to ensure correctness when HTM and STM run concurrently [12]. For example, HyTMs that fallback to the LSA's STM [13] require HTM to manipulate per-location metadata (often referred to as Ownership Records, or ORecs) used by STMs. This extends significantly the memory footprint of HTM transactions, making them prone to capacity aborts. The only HyTM solution we are aware of that avoids instrumenting read and write memory accesses is HyNORec [4], which uses the NORec STM on its fallback path. However, NORec is optimized for low thread counts and is less scalable than ORec-based approaches [8], thus being a suboptimal fallback path for large-scale parallel systems. Further, HyNOrec induces spurious aborts of HTM transactions in presence of concurrent commits of *non-conflicting* STM transactions.

This work addresses the shortcomings of existing HyTMs by presenting DMP-TM (Dynamic Memory Partitioning), the first HyTM system that can rely on highly scalable ORec-based STM implementations, while avoiding any instrumentation cost on the HTM path.

The key novel idea exploited in DMP-TM is to rely on operating system (OS) level memory protection mechanisms to detect conflicts between HTM and STM transactions. DMP-TM maps the heap of a TM application twice in the process virtual address space, one view being accessed by the HTM path and one by the STM back-end. It then relies on OS memory protection mechanisms to selectively prevent pages of one heap from being accessed by the opposite back-end.

This design brings two important benefits, but also non-trivial challenges. A first key advantage is that DMP-TM is agnostic to the actual STM implementation being used: this allows DMP-TM to be used in conjunction with highly scalable and efficient ORec-based STM systems, while avoiding the harsh instrumentation overheads imposed to the HTM path by existing HyTM systems. Another major benefit stemming from this design is that DMP-TM allows HTM and STM transactions that access disjoint memory pages to commit concurrently, sparing them from spurious aborts that would instead arise with state of the art HyTM systems [7].

The key challenge related to DMP-TM's design is that it relies on system calls to enforce memory partitions, which have a non-negligible cost. Indeed, DMP-TM investigates an interesting trade-off, which to the best of our knowledge, has not been currently explored in the literature: leveraging the data partitionability present in applications in order to reduce the runtime overheads of detecting conflicts among STM and HTM transactions, at the cost of a performance penalty in case conflicts between STM and HTM transactions do materialize.

The partitionability of memory access patterns of TM applications is a property already observed in some reference benchmarks by previous works [14] and also confirmed by our experiments. When TM applications do exhibit such a property, DMP-TM allows both HTM and STM transactions to execute avoiding mutual interference, thus minimizing the synchronization of STM transactions and completely removing the need of synchronization overheads for the HTM side. With workloads that generate excessive contention between HTM and STM transactions, the cost of migrating page protections from a heap to the other may outweigh the performance gains stemming from the avoidance of expensive synchronization mechanisms in non-contended runs.

In order to maximize the gains achievable in favourable workloads, while ensuring robust performance also with unfavourable ones, DMP-TM integrates two key self-tuning mechanisms that detect, in a transparent and automatic way: i) which back-end (STM or HTM) to employ for the different transactional blocks of a TM application; ii) whether the degree of partitionability of the accesses generated by the STM and HTM back-ends is too low, being thus preferable to use exclusively the most efficient of the two back-ends.

We evaluated DMP-TM via an extensive study based on both synthetic and realistic benchmarks, i.e., STAMP [15] and a porting of TPC-C [16] to the TM domain, and compared its performance with HTM, 2 STM and 3 HyTM systems. The results of our study show that, in favourable workloads, DMP-TM achieves up to 8.1×/20× speedups vs the best STM/HyTM implementation and up to 37× vs HTM. DMP-TM achieves significant performance gains even when faced with realistic applications: DMP-TM outperforms all the considered baselines in 2 out of the 3 benchmarks of the STAMP suite that are favourable for HyTM systems, achieving up to 2× speedups versus the best alternative. Analogous speedups are obtained even with TPC-C. Overall, our study shows that DMP-TM can achieve significant performance gains with realistic workloads that do not exhibit perfectly partitionable access patterns, providing experimental evidence in support of the practical viability of the proposed solution.

II. Related Work

As this work targets HyTM, it has relations with the body of literature that focused on enhancing the efficiency of HTM systems using software mechanisms.

A first branch of works aimed to enhance the efficiency of HTM systems that use as fall-back path a single global lock (SGL). Afek et. al [17] proposed using an auxiliary lock to avoid the avalanche effect, where cascading aborts happen when the SGL is acquired. Calciu et. al [18] investigated the idea of lazy subscription of the global lock to reduce conflicts between HTM transactions and the fallback path. Tuner [19] leverages online tuning mechanisms to decide when to acquire the pessimistic fallback lock. SEER [20] relies on probabilistic techniques to identify and schedule conflicting transactions executing in hardware. POWER8-TM [21] aims to increase the effective capacity of hardware transactions via *Rollback-only Transactions*, namely atomic, but non-serializable, transactions available on IBM's POWER8 processor [22].

DMP-TM has clearly strong relations with the research in the area of HyTM systems, which, like DMP-TM, aim to support the concurrent execution of both HTM and STM transactions. One such example is HyNOrec [7]. HyNOrec relies on a simple instrumentation mechanism on HTM side that only requires to increase the sequence lock used by NOrec. Although the instrumentation required by HyNOrec is relatively lightweight, it exposes HTM transactions to spurious aborts with non-conflicting STM transactions. Reduced HyNOrec [23] extended HyNOrec with a transactional chopping mechanism to further enhance its efficiency. These solutions are limited to employing NOrec, which is optimized for low thread counts [4]. Conversely, DMP-TM is STM agnostic and thus can be integrated with ORec-based STMs that were shown to achieve much higher scalability levels [6], [8]. Further, DMP-TM does not require instrumenting the HTM path and does not suffer from spurious aborts.

Existing HyTM algorithms that support more scalable ORec-based implementations do exist. Unfortunately, though, they either suffer from spurious aborts (e.g., HyTL2 [23]), analogous to HyNOrec, or require instrumenting the read, write and

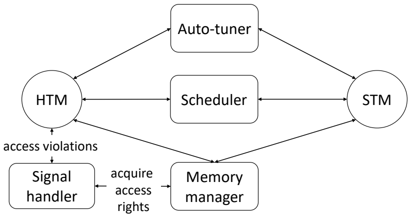

Fig. 1: Architecture of DMP-TM system

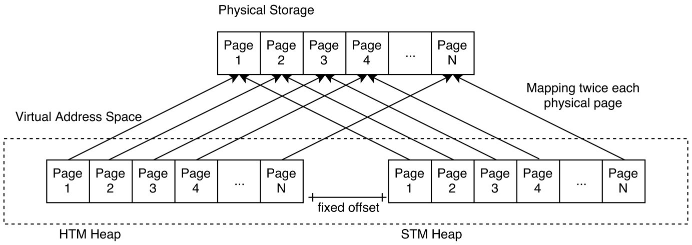

Fig. 2: Mapping of the address space in the HTM Heap and in the STM Heap

commit operations of HTM transactions (e.g., Invyswell [24] and Hybrid-LSA [25]), incurring significant overheads — which we will quantify via the study in Section V.

The Hybrid Cohorts [10] HyTM spares both HTM and STM transactions from extra instrumentations by only allowing HTM to commit when there are no active STM transactions. An analogous idea was explored by PhTM [9], which aims to reduce synchronization overheads between STM and HTM by executing them in alternate phases. Unlike DMP-TM, these solutions prohibit concurrency between HTM and STM.

The idea of exploiting memory protection mechanisms in the context of TM was previously used [26] to ensure correct synchronization between a STM system and non-transactional code, i.e., ensure strong atomicity [1]. DMP-TM builds on analogous base building blocks (hardware-/OS-based memory protection mechanisms) but applies them in a different context (HyTM) and to solve a different problem: enable concurrent, yet safe, execution between HTM and STM transactions.

Finally, DMP-TM is related with prior works that investigate the partitionability of workloads not only for TM applications [14], but also in relational [27] and NoSQL [28] domains. Particularly relevant for DMP-TM is the work by Riegel et al. [14], which has first shown that, even in irregular TM applications like the ones included in the STAMP suite, it is possible to encounter disjoint data partitions that can benefit from the adoption of distinct (software-based) synchronization schemes. It should be noted, though, that DMP-TM considers different synchronization mechanisms than the ones targeted by Riegel et al., and, hence, a different definition of ”partitionability”. Also, unlike the solution proposed by Riegel at al., DMP-TM is designed to operate (and can achieve significant performance gains) also in presence of workloads that are not perfectly partitionable.

III. DMP-TM: Architecture and overview

Figure 1 depicts DMP-TM's architecture, which is composed by four main components: (i) Memory Manager, (ii) Scheduler, (iii) Signal Handler and (iv) Auto-Tuner. It should be noted that these are 4 logical components, which, as we will detail in the next section, are physically implemented in a scalable and decentralized fashion in DMP-TM to enhance efficiency.

The Memory Manager is responsible for regulating the accesses by the HTM and STM paths to the shared transactional heap. This entails: mapping the transactional heap in the process address space twice (via the *mmap*() system call), granting and revoking access grants of the two paths to the shared heaps (via the *mprotect*() system call), as well as caching in user space the state of the per-page memory protections (an optimization that spares from executing system calls in absence of contention between HTM and STM).

The Scheduler classifies transactional blocks as either HTM-friendly or non-HTM-friendly, and accordingly establishes the execution paths to be used to support their execution.

The Signal Handler is activated when HTM transactions access a memory page that was last accessed in an incompatible mode by a STM transaction, triggering an access violation. In such a case, the Memory Manager is consulted to determine when it is safe to restore the needed protection for HTM.

Finally, the Auto-Tuner determines when it is beneficial to turn off one of the back-ends, as the excessively high frequency of system calls' activations outweighs the gains achievable by using concurrently the HTM and STM paths.

As already mentioned, DMP-TM is designed to be STM agnostic, i.e., it can enable concurrent execution of HTM with any STM algorithm. The flexibility enabled by this feature of DMP-TM is particularly relevant given that various works have shown the lack of a no-one-size-fits-all solution when it comes to STM implementations [29]. In fact, DMP-TM's current prototype integrates TinySTM, which was selected due to its high scalability and to the efficiency of its implementation [8]. However, it would be straightforward to develop variants of DMP-TM using alternative STM implementations.

As for the HTM implementation, DMP-TM assumes a conventional/plain interface for transaction demarcation. The only assumption that DMP-TM makes on the underlying HTM system is that, upon a page access violation, the HTM implementation aborts the transaction and makes available to the signal handler the information on the address that triggered the exception. This is an information that HTM implementations by Intel do not currently disclose, but that is instead provided by IBM POWER8's HTM (and by other IBM implementations, to the best of our knowledge).

A. Memory Manager: double heap approach

DMP-TM manipulates the process' virtual space in such a way that the heap is mapped twice (see figure 2) — one heap being used by STM transactions (STM heap) and the other by HTM transactions (HTM heap). Although both heaps point to the same data, with this arrangement, it is feasible to control the access rights of specific regions in one of the heaps, without affecting the other heap. It is worth noting here that this solution does not restrict a transaction type to be executed by a specific thread, i.e., any thread can execute any transaction, either using HTM or STM, at any moment of time. DMP-TM automatically maps the access to shared data to the correct heap according to the back-end being used to execute a given transaction.

Upon its initialization, DMP-TM creates a shared memory zone using the directive *shm_open()*. Then, using the Unix system call *mmap()* the shared memory zone is mapped twice to the HTM heap and STM heap. To control access rights, DMP-TM uses the *mprotect()* system call, which operates at the granularity of a single page. This system call changes the protection of memory pages contained in a given range of addresses. In order to allow an efficient way to calculate the page to be revoked in the opposite heap, both heaps are placed at a constant offset. Thus, calculating the address in the opposite heap is achieved by simply adding or subtracting a fixed offset. This implies basically two changes in the way applications should be developed to be used with DMP-TM:

1) Dynamic memory should only allocate memory from the shared memory region. To this end, we developed a simple, custom implementation of malloc that allocates memory exclusively from the range of addresses associated with the shared region. Analogously to other memory allocators, e.g., [30], DMP-TM's custom malloc implementation splits the range of virtual addresses in n equally sized, disjoint, page aligned splits, where n is the number of threads. This allows each thread to serve malloc/free requests from its ”private” split, avoiding any synchronization overheads with other threads.
2) Two code paths need to be produced, one for STM and one for HTM transactions, each targeting the corresponding heap. Given that the translation between the two address spaces is simple (just a plain translation), the generation of the code paths could be fully automated by a compiler — although the current prototype of DMP-TM does not provide compiler support for this task.

B. Memory Manager: enforcing dynamic partitions

DMP-TM spares HTM transactions from having to check or notify the STM path about possible conflicts, placing any required instrumentation on the STM path. Throughout the execution of an STM transaction, before any shared data access, DMP-TM checks if that data lies on a page accessible, in an incompatible mode, by HTM and, if needed, it accordingly removes the access permissions from the corresponding page in the HTM heap. The result of this design is that whenever a HTM transaction accesses a page for which it does not own adequate access rights, the OS (which, in its turn, exploits virtual memory hardware supports) triggers an access violation by raising a *SIGSEGV* signal. This causes the immediate abort of any HTM transaction that has already accessed that page, or that will access that page in the future. This signal is treated by the Signal Handler module, which is in charge of restoring access rights for HTM transactions.

In order to regulate access to the HTM and STM heaps, DMP-TM stores the following per page metadata:

- *Status field*, which tracks the access rights of a page in the HTM heap. Pages can be in one of three states: (i) Read/Write: HTM can update data on this page, (ii) Read: HTM can only read data from this page and (iii) None: HTM can not access this page in any form. This allows the STM to retrieve in an efficient way, i.e., without issuing system calls, the access permissions of the pages in the HTM heap.
- *Transition count*, which stores how many times the write access permissions to a page have been restored by the Signal Handler. This counter is monitored by STM transactions to detect if the write permissions to a previously read page have, in the meanwhile, been granted back to HTM. If this is the case, some HTM transaction may have overwritten a value previously read by the STM transaction, which is, thus, restarted.
- *Writers Count*, which tracks the number of active STM transactions that wrote to a page. This counter is atomically incremented by an STM transaction upon its first write to a page and atomically decremented upon its commit or abort. This variable is used to prevent restoring access rights to a page in the HTM heap, while there are active STM transactions that wrote to it, thus, preventing HTM transactions from observing inconsistent states.
- *Lock bit*, which acts as a mutex that is acquired whenever the protection and state of a page have to be altered, preventing transactions from concurrently altering the state and protection (via *mprotect()*) of the same page.

C. Transaction Scheduler and Auto-Tuner

As discussed earlier, the Scheduler module has the responsibility of determining the back-end (HTM or STM) to be used by each transaction. We utilize a simple heuristic to accomplish this task. The Scheduler tracks the number of aborts due to the exceeding of the cache capacity by HTM transactions. If the ratio (*capacity aborts/number of commits+capacity aborts*) is greater than 90%, this transaction type is labelled as STM.

In workloads with high degrees of contention between the HTM and STM back-ends, DMP-TM is likely to incur high costs due to the cost of handling access violations and issuing system calls to restore the access rights on the HTM heap. In order to detect when these costs outweigh the gains of executing HTM without instrumentation concurrently with STM, the Auto-Tuner module of DMP-TM employs a *bailout* mechanism based on the following heuristic: If DMP-TM is spending more than 20% of time issuing system calls, it resorts to using either of the back-ends. The decision of upon which back-end to fallback to is taken by sampling the throughput of both back-ends and choosing the back-end with higher throughput.

IV. Algorithm

Algorithm 1 shows the pseudocode for the STM path and the Signal Handler's logic. To simplify presentation, we present pseudocode for the core functionality that allows STM and

Algorithm 1 Pseudocode for STM and Signal Handler

```
1: Shared variables:
2:     $status[N] \leftarrow \{0, 0, \ldots, 0\}$        ▷ per page status field,
3:     $tc[N] \leftarrow \{0, 0, \ldots, 0\}$            ▷ per page transition count,
4:     $wc[N] \leftarrow \{0, 0, \ldots, 0\}$            ▷ per page writers count and
5:     $lb[N] \leftarrow \{0, 0, \ldots, 0\}$            ▷ per page lock bit
6: Local variables:                      ▷ initialized upon every transaction attempt
7:     $private\_tc[N] \leftarrow \{0, 0, \ldots, 0\}$   ▷ local version of tc
8:     $pages\_read \leftarrow \varnothing$              ▷ set of pages accessed as read
9:     $pages\_written \leftarrow \varnothing$           ▷ set of pages accessed as write
10: function STM_READ($addr$)
11:     $page \leftarrow$ GetPage($addr$)                ▷ get page where this address lies
12:     VALIDATE_READSET()
13:     if $page \notin pages\_read$ then                ▷ First time page is read
14:         $pages\_read \leftarrow pages\_read \cup page$
15:         $private\_tc[page] \leftarrow tc[page]$
16:     if $status[page] ==$ #READWRITE then
17:         ACQUIRE $lb[page]$
18:         MPROTECT(READ)                               ▷ issue *mprotect* with read-only
19:         $status[page] \leftarrow$#READ               ▷ set status to read
20:         RELEASE $lb[page]$
21:     $val \leftarrow$ TX_Read($addr$)                 ▷ call the STM API
22:     if $private\_tc[page] \neq tc[page]$ then
23:         STM_RESTART()
24:     return $val$
25: function STM_WRITE($addr$,$val$)
26:     if $page \notin pages\_written$ then             ▷ First time page is written
27:         ATOMIC_INCREMENT($wc[page]$)
28:         $pages\_written \leftarrow pages\_written \cup page$
29:     if $status[page] \neq$#NONE then
30:         ACQUIRE $lb[page]$
31:         MPROTECT(NONE)                               ▷ issue *mprotect* with none
32:         $status[page] \leftarrow$#NONE               ▷ set status to none
33:         RELEASE $lb[page]$
34:     TX_Write($addr, val$)                            ▷ call the STM API
35: function VALIDATE_READSET
36:     for $page \in pages\_read$ do
37:         if $tc[page] \neq private\_tc[page]$ then
38:             STM_RESTART()
39: function STM_RESTART
40:     for $page \in pages\_written$ do
41:         ATOMIC_DECREMENT($wc[page]$)
42:     TX_Abort
43: function STM_COMMIT
44:     VALIDATE_READSET()
45:     TX_Commit                                        ▷ ask the STM to commit
46:     for $page \in pages\_written$ do
47:         ATOMIC_DECREMENT($wc[page]$)
48: function HANDLE_AV($addr$,$isReadOnly$)
49:     wait until $wc[page] = 0$                        ▷ drain writing STM transactions
50:     ACQUIRE $lb[page]$
51:     if $\neg isReadOnly$ then
52:         $status[page] \leftarrow$#READWRITE
53:     else
54:         $status[page] \leftarrow$#READ
55:     MEM_FENCE
56:     if $wc[page] \neq 0$ then
57:         $status[page] \leftarrow$#NONE
58:         RELEASE $lb[page]$
59:         go to 49                                     ▷ wait for writers count to be 0
60:     if $\neg isReadOnly$ then
61:         $tc[page]++$
62:         MPROTECT(READ/WRITE)
63:     else
64:         MPROTECT(READ)
65:     RELEASE $lb[page]$
```

HTM transactions to correctly execute concurrently, omitting the logic of the Scheduler and Auto-Tuner (see Sec. III-C), as well as several optimizations that are discussed in Sec. IV-B).

STM reads. When a read is issued by a STM transaction, it is first checked if the *transition count* of previously read pages has changed since the last access. If any of them has changed in the meanwhile, it means that some HTM transaction may have updated that page (and possiby committed); thus, the STM transaction is aborted. Next, if it is the first read access to this page by this transaction, it stores a local copy of the *transition count* to use it for future checks. Then, it checks the metadata of the page to which it is issuing a read. If the page's access rights are Read/Write (line 16), which means that a HTM transaction can perform updates on it, then the *lock_bit* is acquired in order to change the protection of the page to Read, allowing concurrency with HTM transactions that read this page. After setting the metadata of the page and releasing the *lock_bit*, the transaction effectively reads the memory position, using the underlying STM's API, and checks again if the *transition count* has changed. If so, the performed read is not legal and consequently, the transaction is restarted. If not, the read is successful.

STM writes. Upon a write to shared data from within a STM transaction, if it is the first time the transaction writes to a page it atomically increases the *writers count* for that page (line 27). This blocks any attempt by concurrent Signal Handlers of changing the HTM access rights for that page. Then it is checked if the page corresponding to the location to be updated is accessible by HTM (line 29). If so, after acquiring the *lock_bit*, the access rights for HTM are revoked. This protects HTM from witnessing inconsistent states by observing values written by incomplete STM transactions. After changing the page access rights via *mprotect()* to None and updating the *status field* of the page, the *lock_bit* is released and the transactional update of the value is finally performed.

STM aborts. Before a STM transaction aborts, either due to data conflict or abort from DMP-TM, the *writers count* of all the pages previously written by the transaction is decreased (lines 40-41). This allows HTM transactions to regain accesses to those pages.

STM commits. After a STM transaction finishes its execution, it enters in the commit phase, in which it first checks if the *transition count* of the pages previously read have changed meanwhile (line 44). In the case they have changed, the transaction restarts as explained before. Otherwise, it continues to commit according to its implementation-dependent logic. Then, finally, it decrements the *writers count* atomically for all the pages it has written to (lines 46 - 47).

Signal Handler. The Signal Handler (function HANDLE_AV()) is activated whenever a HTM transaction accesses a page for which it does not have adequate access rights. In this case the OS generates a SIGSEGV, which is intercepted and managed in the same thread that generated the exception. After having extracted the target address of the memory operation that triggered the access violation[1], the Signal Handler waits for the *writers count* of the corresponding page to be zero. Next, it acquires the page's *lock_bit*, sets the metadata to

[1] This information is obtained via the *siginfo* struct that is passed by the OS to the Signal Handler. Existing Intel implementations of HTM reset the siginfo if the access violation occurs in a hardware transaction, which is the reason why DMP-TM does not currently support Intel's architecture.

Read/Write or Read (depending on whether the exception was generated in an update or a read-only transaction) and checks again for the *writers count*, to ensure that it did not change in the meanwhile. Otherwise, the Signal Handler defers to any active writing STM by resetting the access rights to None and going back to wait until the *writers count* is zero. After that, the Signal Handler can restore the HTM access rights to the desired page and, in case the exception occurred in an update transaction, it increments the *transition count* to notify STM transactions about the occurrence of possible conflicts with HTM transactions. It should be noted that *transition count* is increased before acquiring the Read/Write rights, in order to guarantee that if a HTM transaction is granted back permission to update and commit a page (via MPROTECT()), the STM path is guaranteed to detect the corresponding change of the transition count.

A. Correctness Argument

In this section, we provide a set of (informal) arguments on the correctness of the DMP-TM. We organize our analysis by discussing, separately, how DMP-TM enforces isolation of HTM and STM transactions.

Isolation of HTM transactions. The key invariant enforced by DMP-TM to ensure correctness of a HTM transaction T_{HTM}, despite the concurrent execution of any STM transaction T_{STM}, is to ensure that T_{HTM} has no permission to access any of the pages written by T_{STM} throughout its execution. To this end, STM transactions remove HTM's access rights to each page they write to, before issuing the actual write operation. As already mentioned, the removal of the access right causes the immediate abort of any HTM transaction that had already read/written that page, as well as future accesses by HTM transactions to that page.

It is however necessary to carefully synchronize the concurrent execution of the Signal Handler(s) and STM transactions that compete to restore/remove the access rights of the same page. In particular, it is necessary to ensure that the Signal Handler can restore HTM's access to a page only when there are no active STM transactions updating that page, i.e., when the page's *writers count* is set to zero. This is achieved by having the Signal Handler check for the *writers count* twice, while setting the metadata to Read/Write in between. For the second check to be valid, there can be no concurrent STM transaction that started a write operation on this page yet — recall that the *writers count* is atomically incremented as the first step of processing a STM write. In case of a STM transaction T starting a write operation after the second check of the Signal Handler, then T will notice that HTM transactions have access to the page, thanks to the memory barrier that precedes the second check (line 55); in this case, T will acquire the page's lock (synchronizing with any concurrent Signal Handler operating on the same page) and issue a MPROTECT that will abort any concurrent HTM transaction.

Isolation of STM transactions. In order to ensure correctness of STM transactions, DMP-TM guarantees that none of the pages accessed by STM transactions can be altered by HTM transactions, since the time in which each page is first read and until the end of the STM transaction. This is achieved via two key mechanisms: i) ensuring that a STM transaction accesses a page after having removed any non-compatible permission to the corresponding HTM heap's page; ii) checking the transition count of every read page, upon each read and before commit, letting the transaction proceed (without aborting) only if none of them changed since the first time in which that page was read. This implies that the page was not concurrently modified by a HTM transaction, since the *transition count* would be found different if protections had been changed in the meanwhile. It should also be noted that the atomicity of each individual STM read is guaranteed by reading the *transition count* of a page before and after performing the read via the API of the underlying STM implementation.

It should be noted that, in order to reduce overheads, STM transactions check the status of a page without first acquiring the corresponding lock. It is hence possible that a STM transaction finds, in line 16, a page as not writeable by the HTM path and that, before it completes the read, a Signal Handler restores write permission to HTM for the same page. In this case, HTM transactions may even commit and update to that page, before the STM completes executing its read. In such a case, though, the transition count would be found to have changed, when it is checked for the second time in line 22 by the STM transaction. If, instead, the values of the transition counts are not found to have changed, then the STM read is also guaranteed to observe the permissions set by the Signal Handler when it increased the value of transition count — as the memory fence in line 55 ensures that if the increase to the transition count is globally visible, so is the corresponding page's state. This causes the STM transaction to synchronize with concurrent Signal Handlers, by acquiring the page's lock, and to ensure that HTM write permissions are removed before performing the read.

B. Optimizations

The following are a set of optimizations that can be applied to the algorithm described above to further enhance its performance:

- instead of checking the *transition count* of every accessed page upon each access, one can use a global *transition count* that is incremented atomically from within the Signal Handler together with the increment of the page's *transition count*. Then instead of checking each *transition count* of every page upon each STM access, it suffices to only check if the global *transition count* has changed. If it has not (the common case in workloads that exhibit good partitionability), no further checks are required; else, we undergo the normal procedure and have the STM transaction check the *transition count* of each page it previously read.
- instead of maintaining a single *writers count* per page shared by all STM threads, which must be incremented or decremented atomically, one can use a set of, per thread,

local counters. This will spare STM transactions from the need to perform expensive atomic operations. However, the Signal Handler will need to ensure that all the flags are unset before it attempts to change the page's protection.
- certain STM algorithms need to re-validate their readset in order to ensure the safe execution of transactions, e.g., as in the case of TinySTM in presence of concurrent commits of update transactions. These STM implementations could be made aware of the execution of read-set validations performed by DMP-TM (if *transition count* is found to have increased), which may allow them to spare duplicate validations.

The current implementation of DMP-TM integrates the first two optimizations, but not the last one. Unlike the first two mechanisms, in fact, the last optimization comes at the cost of requiring to alter the inner logic of the STM implementation employed by DMP-TM, whereas one of the key design goals of DMP-TM is its STM-agnostic nature.

V. Evaluation

This section reports the results of an extensive experimental study, in which we evaluated both the static (DMP-TM) and the dynamic version (DMP-TUNE) of the proposed solution. In the static version, transactions are statically assigned to either one of the back-ends according to an exhaustive offline search for the best configuration, whereas in the dynamic version the scheduler module decides upon the transaction assignment during run-time. We compared DMP-TM and DMP-TUNE (whose implementation we made publicly available[2]) against: i) HTM with a single global lock as fallback (htm-sgl); ii) TinySTM with the same configuration as the one used for DMP-TM and DMP-TUNE; iii) HyNOrec using 2 counters to decouple subscribing from signaling; iv) NOrec with write back configuration; v) HyTinySTM, which we implemented by adapting the original algorithm [25] to replace the *prefetchw* instruction, which is not available in current HTM implementations, with a write; vi) HyTL2, based on the algorithm described in [23]. All hardware-based solutions try executing each transaction 10 times in hardware before resorting to the fallback path. DMP-TM and DMP-TUNE, however, try the transactions attributed to HTM 100 times before acquiring the global lock. This is done since the transactions attributed to HTM in DMP-TM and DMP-TUNE are hardware-friendly and likely to commit using HTM, if tried long enough. Conversely, using such a high retry count with other hardware-based solutions will dramatically degrade their performance, since the approaches try *all* transactions (including non-HTM-friendly ones) first in hardware.

We start by using synthetic benchmarks to generate diverse workloads. intended to test extreme scenarios regarding partitionability of HTM and STM access patterns. Then, we tested DMP-TM using real-world complex benchmarks, namely two benchmarks of the popular TM benchmark suite STAMP [15] and TPC-C [16]. All presented results were obtained by executing on an 80-way IBM Power8 8284-22A processor with 10 physical cores, where each core can execute 8 hardware threads. The OS installed is Fedora 24 with Linux 4.7.4 with page size of 64KB and the compiler used is GCC 6.2.1 with -O2. The reported results are the average of 5 runs.

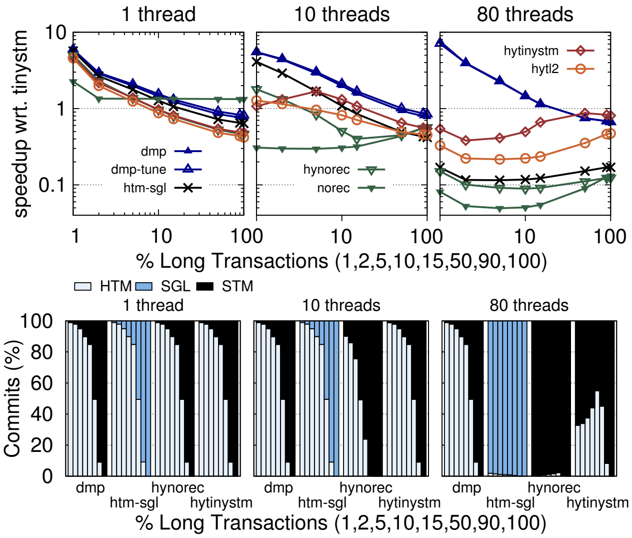

Fig. 3: Speedup and commits breakdown for disjoint data structures running 1, 10 and 80 threads

Thread pinning was used to pin a thread per core at the beginning of each run for all the solutions until exhausting the number of available cores, and then distributing them in round-robin fashion to minimize unbalances between cores.

A. Synthetic Benchmarks

In order to assess the effectiveness of DMP-TM in diverse, yet identifiable workload settings, we rely on a synthetic benchmark based on two different hashmaps, storing 128, resp. 1024, elements per bucket. This setting was motivated by the fact that if HTM transactions read all the elements of a bucket from the first hashmap, the size of the read-set fits in the cache. However, if HTM transactions read all the elements from the second hashmap, the read-set will exceed the cache size, thus causing a capacity abort.

Disjoint data structures. To demonstrate the potential of DMP-TM we consider a workload, composed of two transactions types, one amenable for execution in hardware and one not (as it exceeds deterministically HTM's capacity), that generate perfectly partitionable memory accesses, i.e., the sets of pages accessed by each transaction type are disjoint. To We populated the hashmaps to use 64 pages in total and used a workload with 10% lookups and 90% update transactions.

Figure 3 reports the throughput speedup normalized with respect to TinySTM and the breakdown of commits for three different thread configurations: 1, 10 and 80 threads. For each of these configurations, we varied the percentage of small transactions executed. The x-axis reports the probability of a thread to be executing a long transaction from 1% to 100% (only long transactions). The results show remarkable gains

[2] https://github.com/pedroraminhas/DMP-TM

either for DMP-TM and DMP-TUNE, since in this experiment both data structures are disjoint and the operations executed are so heterogeneous that HTM shines when executing short transactions, whereas STM excels with long ones. We normalize the throughput of all the solutions according to TinySTM (which we accordingly omit from the plot), and use log scale on both y and x-axis to enhance visualization.

With one thread, we can assess the overhead that DMP-TM and DMP-TUNE incur. When the workload constitutes mainly small transactions, both variants of DMP-TM achieve better performance than HTM-SGL, thanks to their ability to run HTM transactions without any instrumentation and executing large transactions in STM — which spares the cost of retrying them several times before using the fallback path. As the percentage of large transactions in the workload increases, DMP-TM variants start to be outperformed by TinySTM, paying a penalty of ∼20% in the 100% long transactions workload. This is the cost of the extra instrumentation that DMP-TM adds on top of TinySTM. Note that NOrec consistently outperforms TinySTM, thanks to its more lightweight instrumentation.

At 10 threads, DMP-TM becomes the best performing backend, achieving speedups of up to ∼2× compared to HTM-SGL, and more than 3× compared to TinySTM and ∼ 6× compared to NOrec-based solutions. This can be explained by the breakdown of commits shown in the middle row of Figure 3. DMP-TM is able to execute short transactions in HTM and long transactions as STM, unlike HTM-SGL that executes large transactions using the pessimistic single global lock. In this configuration, TinySTM starts to surpass the throughput of HTM-SGL when the workload is running operations in the smaller data structure with probability less than 20% but only at 50%, TinySTM's throughput equalizes DMP-TM. At 100% of large transactions, due to the fact that DMP-TM requires additional instrumentation on top of STM, it suffers ∼20% performance penalty w.r.t. TinySTM.

80 Threads continue showing the same trend as 10 threads, but with even greater speedups: more than 20× compared to HTM-SGL, ∼10× compared to HyTinySTM and ∼7× compared to TinySTM. The gains with respect to both HTM-SGL and HyTinySTM are due to DMP-TM's ability to execute more transactions in hardware, as shown in the commits breakdown plot. At high number of threads, capacity aborts become non-deterministic as more threads share hardware resources and it is, thus, beneficial to retry more times in hardware than reverting to the fallback path. However, without differentiating between deterministic and non-deterministic capacity aborts, high retry counts becomes harmful in terms of throughput. Again, TinySTM outperforms DMP-TM whenever the probability of executing large transactions is greater than 30% leading to a 30% overhead at 100% large transactions.

Throughout the entire experiment, the dynamic version of the algorithm (DMP-TUNE) worked as expected, since whenever transactions execute operations in the smaller hashmap the percentage of aborts due to exceeding transactions footprint is negligible. Nevertheless, when transactions execute operations in the larger hashmap, they deterministically abort due to exceeding the cache size. Thus, the scheduler module assigns the type of transactions accessing large hashmap as STM transactions, matching the offline assignment used by DMP-TM . Thanks to this assignment ability, DMP-TUNE is able to still benefit from using high retry counts unlike other hardware-based solutions.

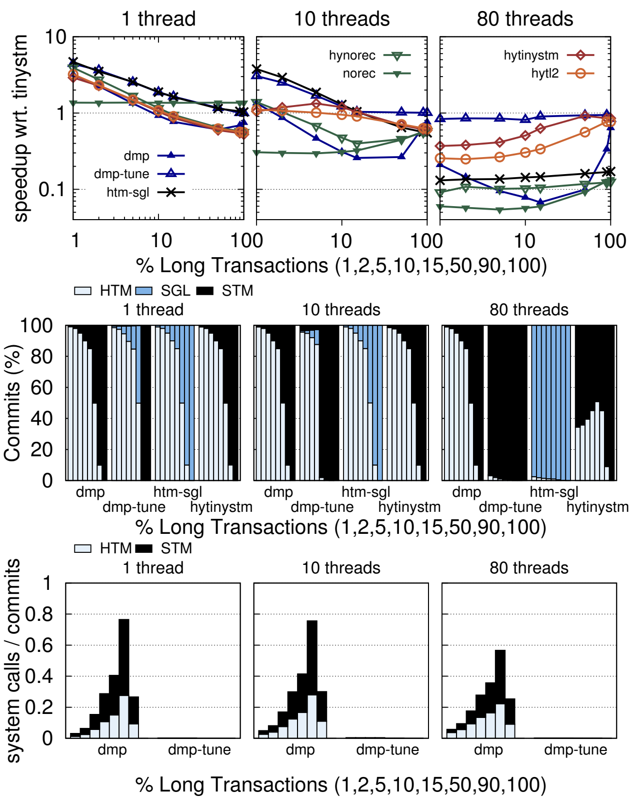

Fig. 4: Speedup, commits breakdown and system calls ratio for non-disjoint data structure running 1, 10 and 80 threads

Non-disjoint data structures. This experiment stresses the worst case scenario for DMP-TM, by allocating both hashmaps in the same memory region and interleaving the buckets of each hash map. With the granularity of DMP-TM being a single page, any access to either of the hashmaps is going to be considered a conflict. Therefore, DMP-TM will suffer from a very large number of system calls, as for either of the back-ends to commit a transaction, it is most likely that some page protection has to be restored, given that we are considering a 90% update workload.

Figure 4 depicts the results of running this workload with 1, 10 and 80 threads. It is clear that DMP-TM suffers large performance penalties, up to 20× compared with TinySTM in the worst case. This is true across all workloads except when the workload is dominated by either short or long transactions. At those extremes, there is no need for changing the pages access rights since DMP-TM is executing only one of the back-ends. Other than that, DMP-TM incurs significant overheads. This can be explained by looking at the ratio of system calls to commits, which shows that DMP-TM can pay up to more than 0.8 system calls per commit.

This is a typical workload where the Auto-Tuner module should decide to *bailout* and stick to either one of the back-ends. Inspecting the speedup plots in Figure 4, we can see that DMP-TUNE manages to match the performance of the best performing of both back-ends. At 1 thread, HTM-SGL performs better than TinySTM, except when the system is running only large transactions (right side of the 1 thread figure); so, as expected, DMP-TUNE falls back to HTM-SGL with percentage of long transaction less than 100%. At 10 threads, by looking at the commit breakdown is possible to see that DMP-TUNE falls back to HTM-SGL in workloads characterized by less than 50% of long transactions. However, after this mark, TinySTM begins to be the best performing back-end. Thus, DMP-TM falls back to TinySTM at this mark. At 80 threads mark, the best performing back-end is TinySTM, independently of the percentage of long transactions, and DMP-TUNE correctly adapts itself to employing TinySTM.

B. *STAMP*

STAMP [15] is a popular TM benchmark suite constituted of 8 different complex applications. Out of the 8 applications, Genome and Intruder are the ones that typically benefit from HyTM systems as they encompass both small and large transactions. Other applications generate either only small transactions (SSCA2 and KMeans) or large ones (Labyrinth and Yada). Thus, there is no room for improvement for any HyTM. The remaining two applications are Bayes and Vacation. Bayes is known to suffer of very high variance and yields unreliable results [31]. Vacation is another benchmark that can benefit from a HyTM system, however, it has low degree of partitionability. Hence, DMP-TUNE would perform as good as either HTM or STM. We omitted the results of Vacation for space constraints.

Genome represents the process of reconstructing the original source genome from a pool of DNA segments. We conducted an extensive brute-force experimental study in order to infer which of the 5 transaction types generated by Genome to run with HTM or STM. We found that this workload indeed presents partitionability, as we can define three disjoint transactional clusters according to their data access. DMP-TM successfully exploits this workload's property, achieving the maximum throughput and scaling to 64 threads, yielding $\sim$2× higher throughput than HyTinySTM, the second best baseline at 80 threads (Figure 5).

This can be explained by analyzing the commits breakdown, which shows DMP-TM's ability to execute >90% of transactions in hardware, $\sim$1% using the pessimistic fallback path and $\sim$5% as STM at all thread counts. Although HyTinySTM manages to demonstrate similar commit patterns up to 8 threads, it performed worse than DMP-TM due to the extra instrumentation it imposes to its HTM path. Beyond 8 threads, HyTinySTM could not commit as many transactions in hardware, since it incurs more frequent capacity aborts that consume its retry count and lead to more frequent activations of the fallback path.

Furthermore, the workload is characterized by low contention. Therefore, up to 4 threads, NOrec and TinySTM achieve slightly better throughput than DMP-TM as they do not impose any extra instrumentation to their STM path. Up to 2 threads, HyNOrec, shows similar throughput as DMP-TM, since as shown in the commit breakdown plots of Figure 5, it still manages to commit 95% and 38% of the times in hardware, respectively for 1 and 2 threads, and uses as fallback NOrec, which as stated before achieves higher throughput than DMP-TM in this workload for a low thread count. However, as the thread count increases, abort rate starts to increase. In these contention settings, DMP-TM benefits from executing transactions in software, which enables more concurrency than the SGL fallback used by HTM-SGL. For the case of HyNOrec, the fallback of only one thread makes all the threads fallback to NOrec, and this has an adverse impact on performance when the thread count is higher than 4. After 16 threads, HyTinySTM starts to incur overheads due to the fact that at this thread count, cores are shared by more than one hardware thread, which reduces the effective cache capacity available for each hardware thread. As HyTinySTM instruments hardware transactions to check for changes in the STM ORecs, it suffers from an increased abort rate compared to the solutions that do not instrument hardware transactions, namely HTM-SGL and DMP-TM. For the case of HyTL2, unlike the STM counterpart, TL2, not shown in the study, does not extend the snapshot used during STM reads, which leads to an increase of the transaction's abort rate.

Intruder is a signature-based network intrusion detection system that encompasses three parallel transactions. The right column of figure 5 shows DMP-TM being the only back-end to scale up to 16 threads achieving $\sim$1.5× higher throughput than TinySTM, the second best performing back-end. The lower speedups in Intruder, as compared with Genome, can be attributed to the lower percentage of transactions ($\sim$30%) that DMP-TM manages to execute in hardware. Again, HyTinySTM, which even commits more transactions in hardware, is outperformed by DMP-TM achieving 2× lower throughput at 16 threads, due to the costly instrumentation of its HTM path. NOrec's HyTM counterpart follows the same trend as NOrec. However, the commit breakdown shows that starting the execution in the HTM path and falling back to NOrec causes performance losses in the order of 0.73× compared to NOrec. It is worth noting that NOrec achieves the best throughput until 4 threads, thanks to its lower instrumentation costs. However, at 8 threads, its throughput deteriorates due to the increase of the contention in the workload. Further, HTM-SGL and HyTL2 are the worst back-ends, due to the pessimistic nature of the fallback of the former and to the inability of the STM fallback path of the latter to perform well in high contention workloads.

C. *TPC-C*

Finally, we move to evaluating DMP-TM using TPC-C [16]. TPC-C is a well-known benchmark for relational database management systems that emulates a workload of a whole-

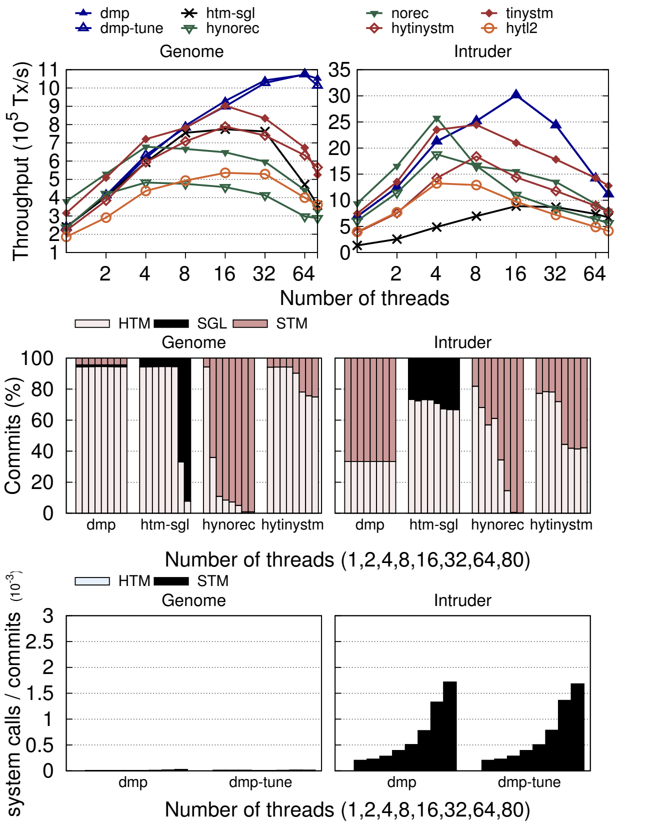

Fig. 5: Speedup, commits breakdown and system calls ratio for Genome and Intruder of STAMP benchmark suite.

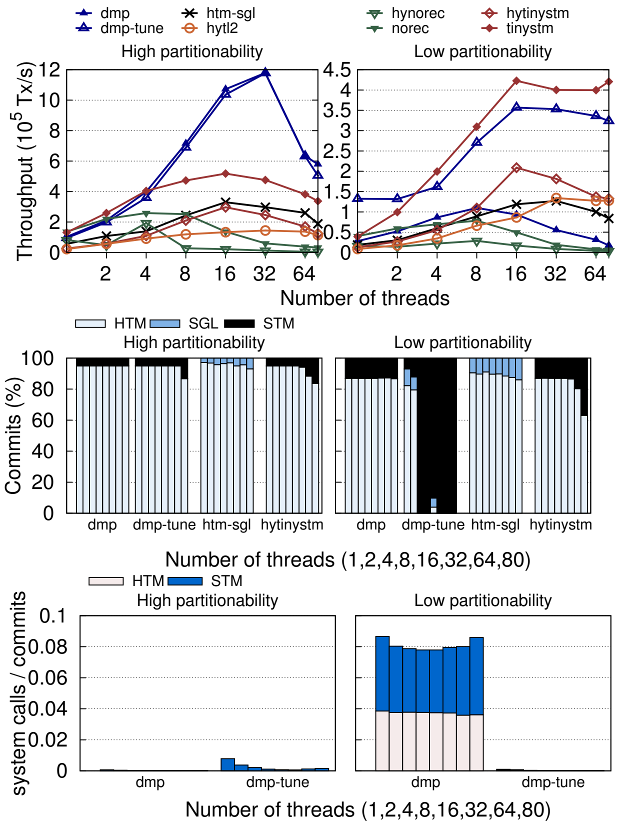

Fig. 6: Speedup, commits breakdown and system calls ratio for two workloads of TPC-C.

sale supplier. It consists of five transactions operating on a database. In this work, we use a port of TPC-C for inmemory databases[3] where transactions are executed by an underlying TM implementation. To promote partitionability, we performed vertical partitioning, according to the TPC-C standard, by moving attributes that are points of conflict to different memory regions to reduce false conflicts. We consider two different workloads that exhibit different degrees of partitionability between short and long transactions.

The left column of Figure 6 reports the results of a workload composed by 95% of payment, 1% stock level and 4% of delivery transactions. This workload has a high degree of partitionability, reflected in a very low system calls to commits ratio. DMP-TM achieves the best throughput showing up to 2.4× speedups compared with the second best contender, TinySTM. At a low thread count, namely up to 4 threads, TinySTM achieves slightly better throughput than DMP-TM due to the fact that it has no extra instrumentation. Although HTM-SGL and HyTinySTM execute more than 90% of the transactions in hardware, they yield ∼3× lower throughput than DMP-TM. For HTM-SGL, this is due to the fact that stock level and delivery operations, that do not meet HTM's capacity limitations, are much longer than payment operation. This hinders parallelism and thus limits throughput gains and scalability of HTM-SGL. While for HyTinySTM, also in this case the problem is rooted to the high instrumentation costs of the HTM path. NOrec, HyNOrec and HyTL2 incur performance losses due to the fact that payment operation has very high contention. After 32 threads, neither DMP-TM nor DMP-TUNE scale. This happens despite the commit breakdown plot (and the abort rate) incurred by these solutions do not show any significant spike. We have verified, though, that, above 32 threads, the Instruction Per Cycle drop severely, with a corresponding spike in the number of stalled cycles — which suggests that, increasing the thread count, the bottleneck for DMP-TM eventually becomes contention to some physical resource, probably memory or some micro-architectural resource of the processor.

Finally, the right column of Figure 6 shows the throughput results for a workload with low degree of partitionability, as reflect by the high systems calls to commits ratio. Due to the high number of system calls, DMP-TM incurs 16× lower throughput compared with TinySTM at 80 threads. Again thanks to its self-tuning ability, DMP-TUNE is able to fallback to TinySTM and achieve similar throughput.

VI. Conclusions and future work

This paper presented DMP-TM, a HyTM algorithm that exploits a key novel idea: leveraging operating system-level memory protection mechanisms to detect conflicts between HTM and STM transactions. This innovative design allows for employing highly scalable ORec-based STM implementa-

[3] https://github.com/evanj/tpccbench

tions, while avoiding any instrumentation on the HTM path. DMP-TM demonstrated robust performance in an extensive evaluation, achieving gains of up to ∼20× when compared to state of the art HyTM systems.

As a concluding remark, we note that the current implementation of DMP-TM cannot be used with Intel's HTM implementations, which, unlike IBM's, lack support for a simple feature: reporting information on the address that caused an access violation that occurred within a transaction. We hope that the performance benefits achievable by DMP-TM will motivate other CPU manufacturers, besides IBM, to integrate this feature in their future CPU generations.

In our future work, we plan to investigate two mechanisms to allow DMP-TM to cope with this limitation and allow interoperability with existing Intel's HTM implementations. The first technique consists in obtaining the address of the instruction that caused the exception via the Last Branch Records, which in recent Intel processors store the last branches executed by the CPU and can be used to pinpoint the address of the instruction that caused an access violation [32]. The key challenge with this approach lies in determining which memory address had the offending instruction targeted, based on the program control flow information stored in the LBRs.

A second approach we intend to investigate is the use of Processor Tracing (PT), namely a recent ISA extension that supports inbuilt tracing mechanism for Intel TSX, and provides extensive control not only on the control flow within transactions, but also precise timing analysis on asynchronous events (like interrupts and signals). While it appears that this information could be used to accurately estimate the memory addresses that triggered an access violation by a HTM transaction, the overheads incurred by tracing and analyzing this information in run-time are still unclear and can only be evaluated by building a realistic prototype.

Acknowledgments

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. We wish to thank the anonymous reviewers for their insightful comments and Prof. Pascal Felber for providing access to the POWER8 machine used in the experimental study.

References

[1] T. Harris, J. Larus, and R. Rajwar, *Transactional Memory*, 2nd ed. Morgan and Claypool Publishers, 2010.

[2] C. J. Rossbach, O. S. Hofmann, and E. Witchel, "Is transactional programming actually easier?" in *PPoPP'10*.

[3] M. Herlihy and J. E. B. Moss, "Transactional memory: Architectural support for lock-free data structures," in *ISCA'93*.

[4] L. Dalessandro, M. F. Spear, and M. L. Scott, "Norec: Streamlining stm by abolishing ownership records," in *PPoPP'10*.

[5] A. Dragojević, R. Guerraoui, and M. Kapalka, "Stretching transactional memory," in *PLDI'09*.

[6] P. Felber, C. Fetzer, and T. Riegel, "Dynamic performance tuning of word-based software transactional memory," in *PPoPP'08*.

[7] 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*.

[8] N. Diegues, P. Romano, and L. Rodrigues, "Virtues and limitations of commodity hardware transactional memory," in *PACT'14*.

[9] Y. Lev, M. Moir, and D. Nussbaum, "Phtm: Phased transactional memory," in *TRANSACT'07*.

[10] W. Ruan and M. Spear, "Hybrid transactional memory revisited," in *DISC'15*.

[11] A. Matveev and N. Shavit, "Reduced hardware norec: A safe and scalable hybrid transactional memory," in *ASPLOS'15*.

[12] T. Brown and S. Ravi, "Cost of Concurrency in Hybrid Transactional Memory," in *DISC'17*.

[13] T. Riegel, P. Felber, and C. Fetzer, "A lazy snapshot algorithm with eager validation," in *DISC'06*.

[14] T. Riegel, C. Fetzer, and P. Felber, "Automatic data partitioning in software transactional memories," in *SPAA'08*.

[15] C. Cao Minh, J. Chung, C. Kozyrakis, and K. Olukotun, "STAMP: Stanford transactional applications for multi-processing," in *IISWC'08*.

[16] "Transaction Processing Performance Council, TPC BENCHMARK™ C," Revision 5.11, February 2010.

[17] Y. Afek, A. Levy, and A. Morrison, "Software-improved hardware lock elision," in *PODC'14*.

[18] I. Calciu, T. Shpeisman, G. Pokam, and M. Herlihy, "Improved single global lock fallback for best-effort hardware transactional memory," in *TRANSACT'14*.

[19] N. Diegues and P. Romano, "Self-tuning intel transactional synchronization extensions," in *ICAC'14*.

[20] N. Diegues, P. Romano, and S. Garbatov, "Seer: Probabilistic scheduling for hardware transactional memory," in *SPAA'15*.

[21] S. Issa, P. Felber, A. Matveev, and P. Romano, "Extending Hardware Transactional Memory Capacity via Rollback-Only Transactions and Suspend/Resume," in *DISC'17*.

[22] H. Le, G. Guthrie, D. Williams, M. Michael, B. Frey, W. Starke, C. May, R. Odaira, and T. Nakaike, "Transactional memory support in the ibm power8 processor," *IBM Journal of Research and Development*, vol. 59, no. 1, 2015.

[23] A. Matveev and N. Shavit, "Reduced hardware transactions: A new approach to hybrid transactional memory," in *SPAA'13*.

[24] J. E. Gottschlich, M. Vachharajani, and J. G. Siek, "An efficient software transactional memory using commit-time invalidation," in *CGO'10*.

[25] T. Riegel, P. Marlier, M. Nowack, P. Felber, and C. Fetzer, "Optimizing hybrid transactional memory: The importance of nonspeculative operations," in *SPAA'11*.

[26] M. Abadi, T. Harris, and M. Mehrara, "Transactional memory with strong atomicity using off-the-shelf memory protection hardware," in *PPoPP'09*.

[27] C. Curino, E. Jones, Y. Zhang, and S. Madden, "Schism: A workload-driven approach to database replication and partitioning," *Proc. VLDB Endowment*, vol. 3, no. 1-2, 2010.

[28] J. Paiva, P. Ruivo, P. Romano, and L. Rodrigues, "AUTOPLACER: Scalable self-tuning data placement in distributed key-value stores," in *ICAC'13*.

[29] D. Didona, N. Diegues, A.-M. Kermarrec, R. Guerraoui, R. Neves, and P. Romano, "Proteustm: Abstraction meets performance in transactional memory," in *ASPLOS'16*.

[30] S. Ghemawat and P. Menage, "Tcmalloc: Thread- caching malloc," http://goog-perftools.sourceforge.net/doc/tcmalloc.html, Accessed: 23-02-2018.

[31] Q. Wang, S. Kulkarni, J. Cavazos, and M. Spear, "A transactional memory with automatic performance tuning," *ACM Transactions on Architecture and Code Optimization*, vol. 8, no. 4, 2012.

[32] Intel, "Intel64 and ia-32 architectures optimization reference manual," 2016.