Enhancing efficiency of Hybrid Transactional Memory via Dynamic Data Partitioning Schemes

Pedro Raminhas
Instituto Superior Técnico, Universidade de Lisboa
Lisbon, Portugal
Email: pedro.raminhas@tecnico.ulisboa.pt

***Abstract*—Multicore processors represent the *de-facto* standard architecture for a wide variety of devices, from high performance computers to mobile devices. However, developing concurrent programs to harness the capabilities of multicore architectures using traditional lock-based synchronization is known to be a complex and error prone task, as the fine-grained locking approach is prone to deadlocks, livelocks and it is difficult to debug, while coarse-grained locking does not provide performance gains. Transactional Memory (TM) is a powerful paradigm that promises to simplify the development of concurrent programs, while still achieving performance similar to locking. With TM, programmers only need to specify which parts of the code have to appear as executed atomically, and not how atomicity should be achieved. This document proposes a novel Hybrid TM algorithm, which we call DMP-TM, that aims to minimise synchronization overheads between HTM and STM by relying on a novel dynamic memory partitioning scheme. The key novel idea underlying DMP-TM is to leverage on operating system level memory protection mechanism to enforce dynamic memory partitions, in which HTM and STM can operate assuming no mutual interference. This design contrasts with current state of the Hybrid TM designs that incur extra-overheads by checking the metadata. We evaluated DMP-TM using both synthetic and standard benchmarks for Transactional Memory. Our experimental study highlights that DMP-TM can achieve up to 4x time speedups when compared to state of the art HTM and STM and up to 10x speedups compared with state of the art Hybrid TM.**

I. Introduction

Multicore processors have become the standard architecture in today's computing systems, ranging from high-end servers, to personal computers, as well as smartphones and embedded systems, all have the multicore technology and the trend is to increase the number of cores towards many core architectures. Unfortunately, the development of parallel applications that can fully exploit the advantages of the multicore technology is far from being a trivial task.

Indeed, when using traditional lock-based synchronization mechanisms, developers are faced with a major dilemma. They can either opt for using coarse-grained locks, which simplifies significantly reasoning on the correctness of concurrent applications, but can severely hinder parallelism or hamper performance. Or they can opt for a second alternative which is to use fine-grained locking. This approach allows for extracting the most parallelism possible, avoiding the inefficiency of coarse-grained locking, but it is harder to devise and more prone to deadlocks and livelocks, which are notoriously hard to debug [1]. In fact, fine grained locking is considered difficult to use for the average programmer [1], who may not be necessarily well trained for the development of concurrent applications.

Transactional Memory (TM) [2] precisely answers this urge by specifying a new programming paradigm that alleviates the complexity of classical locking mechanisms while ensuring performance similar or even better than complex fine-grained locking schemes. TM borrows the concept of transactions from databases and applies it to parallel programming: with TM, programmers devise code into blocks that will be ensured by the TM system to run atomically. The underlying system is responsible for either committing the transactions, and making its modifications globally visible, or to aborting the transactions and ensuring that no modifications are observed out of the context of the transaction. This approach leads to drastically simplifying the development of parallel applications and abate their time to market and costs.

Transactional Memory was initially proposed twenty years ago as an extension to multi-processors' cache coherence protocols [2]. Due to the difficulty of rapid prototyping in hardware environments, researchers resorted to Software Transactional Memory to advance the state of the art [3], [4], [5], [6]. Simultaneously, hardware-based implementations have also been proposed, whose designs were validated using simulators.

Hardware Transactional Memory (HTM) is currently supported by the mainstream Intel Haswell and the IBM Power8 processors. These implementations are *best-effort*, which means that they always ensure correctness, but do not provide any progress guarantee. This means that transactions executing in hardware may never commit (even in absence of concurrency) due to inherent limitations of hardware, such as transaction exceeding capacity of hardware or running prohibited instructions. This raises the need for a software fallback path to ensure forward progress [7].

This fallback path can be either as simple as a global lock

or as complex as a Software Transactional Memory, which is designated by Hybrid Transactional Memory (Hybrid TM) [8], [9], [10], [11], [12]. The goal of Hybrid TM is to fully exploit *best-effort* HTM as much as possible, and in the case of abort, fall back to the more costly Software Transactional Memory (STM), which provides progress guarantee.

A. Motivation

Despite the number of papers published in this area in recent years, Hybrid TM still suffer from large overheads [7]. These overheads are particularly exacerbated on Hybrid TMs which require HTM to manipulate the per-location metadata (often referred to as Ownership Records, ORecs) used by STM —and, as such, extend significantly the working set of HTM transactions making them prone to capacity exceptions; but also in Hybrid TMs that use a software fallback without per location metadata, as these Hybrid TMs suffer from a scalability bottleneck as every transaction must read a sequence lock. Further, they can induce spurious aborts of HTM transactions if any concurrent, non-conflicting HTM commits [7].

Benchmarks' tests indicate that the performance of state of the art Hybrid TM cannot surpass HTM performance for workloads characterized by short transactions with small read and write-set, and that it cannot surpass the scalability of STM for workloads characterized by long transactions with large read and write-set. The results show that Hybrid TM has costs incurred by synchronizing both the execution of HTM and STM, as the memory locations accessed by one TM has to be validated with the read and write-set of both TMs. Results also show that the performance of Hybrid TM is highly dependent on the workload.

B. Contributions

The goal of this document is to provide a new class of Hybrid TM without the main drawback of state of the art Hybrid TM: the overheads incurred by the synchronization of transactions running HTM and transactions running STM. To accomplish that, we rely on operating system's memory protection mechanism to dynamically establish a memory partitioning scheme that ensures that HTM and STM execute, at any time, on disjoint memory regions. By delegating to the OS memory protection mechanism the task of detecting violations of the currently established memory partitioning scheme, DMP-TM investigates an interesting trade-off, which has not currently explored in the literature: reducing the runtime overheads for detecting conflicts among STM and HTM transactions, at the cost of a large performance penalty in case conflicts between STM and HTM transactions do materialize.

As it will be shown via an extensive experimental study, this design choice allows DMP-TM to achieve up to 4x speedups compared to HTM and STM, and 10x speedups compared to Hybrid-NOrec [9], in workloads where the memory regions accessed by HTM and STM transactions are unlikely to overlap.

II. Background

A. Background

Next, background on TM is provided. First, is presented Software Transactional Memory which aims to deliver the capabilities of Transactional Memory (STM) via software library. Then, we present Hardware Transactional Memory (HTM), an implementation of TM that implements TM in hardware. Finally, we present Hybrid TM, which aims to use HTM and then fallback to STM whenever HTM cannot succeed.

The TM programming model relies on the assumption that programmers demarcate portions of code that must execute as atomic transactions. TM guarantees that safe concurrent operations succeed and that transactions that have conflicts with other, abort.

Many designs and solutions have been proposed in the last years. Next, they will be presented and discussed their advantages and trade-offs in comparison with Dynamic Memory Partitioning (DMP-TM).

TM implementations. The TM abstraction has been implemented in software (STM), hardware (HTM), or combinations thereof (Hybrid TM).

A wide variety of STMs have been proposed [3], [4], [5]. Mainly, because STM always grant correctness and progress, however they require costly code instrumentation in order to track transactional operations.

HTM [13], [**?**] does not need to add extra-instrumentation to the code, however it is *best-effort*, i.e., it does not guarantee that a transaction can succeed. A HTM transaction can abort due to system calls performed during their execution, exceeding cache capacities or even *spurious aborts*. In order to guarantee progress, researchers used a single global lock to provide synchronization. Whenever HTM transactions abort more than a fixed *threshold*, it acquires a global lock, that is subscribed by all transactions, that aborts all the running transactions and avoid the start of new ones. The single global lock is a mechanism that guarantees progress, but is slow, as every HTM transaction aborts when it is acquired.

The usage of a single global lock as a fallback mechanism in *best-effort* HTM motivated the researchers of TM to create a more complex fallback mechanism, called Hybrid Transactional Memory (Hybrid TM). The goal of Hybrid TM is to exploit *best-effort* HTM whenever possible due to its cheaper capabilities, and fallback to the more costly STM when a transaction cannot complete in hardware. This approach promises to scale well and incur low latency, with the worst-case overhead and scalability comparable to the underlying STM.

To perform as a coherent whole, the Hybrid TM system must be able to detect conflicts arising among concurrent

hardware and software transactions. This requirement is usually achieved by logging additional metadata while executing hardware transactions, so that the conflict detection system has access to both sides information. Certain STM algorithms [4], [5] require interaction with per-location metadata, and hybrid versions of these algorithms [12] wasted limited hardware capacity on this metadata. The interaction of HTM with STM metadata could lead to capacity aborts, or could lead to false sharing of cache lines that hold the metadata, resulting in additional HTM aborts, and increased fallback to the STM path.

III. Dynamic Memory Partitioning Hybrid TM

The analysis of the state of the art conducted in the subsection II-A highlighted that Hybrid TM still incurs an overly large overhead, due to the need of synchronization of concurrent transactions using HTM and STM.

The approach proposed is inspired by recent findings [14], which verified that, in several reference TM benchmarks, namely Genome, Vacation and K-Means of STAMP benchmark [15], the memory layout can be partitioned in a way that it is safe to use different concurrency control mechanisms, optimized for different workloads, on different partitions.

Based on that, this document proposes a new class of Hybrid TM that takes advantage of the ”partitionability” of workloads by minimizing the overheads due to the synchronization between HTM and STM.

Dynamic Memory Partitioning (DMP-TM), avoids to trace conflicts by forcing transactions to log additional information during their execution, but rather to delegate to the OS the function of detecting conflicts, which is done at page-level.

A. Memory Mapping

DMP-TM manipulates the process virtual space in such a way that the heap is mapped twice; one heap is used by transactions running STM and the another one is used by transactions HTM, this way is possible to either one of the TM implementations to revoke the access to the other by simply changing the protection of the corresponding page in its heap. At any point in time, DMP-TM ensures, via OS memory protection mechanism, that each memory page can only be updated by either HTM or STM or that it can be shared in read-only mode by both TM implementations.

This design was motivated by the goal of sparing HTM from any instrumentation overhead in case it accesses memory regions that belong to its own areas. In case HTM accesses a memory region that does not belong to its own areas, we detect conflicts using the signal handling of Unix and whenever a signal is detected we change the memory protection according to the access being done. This is fundamental to ensure that performance of HTM is not hampered, since, as already discussed, HTM's performance

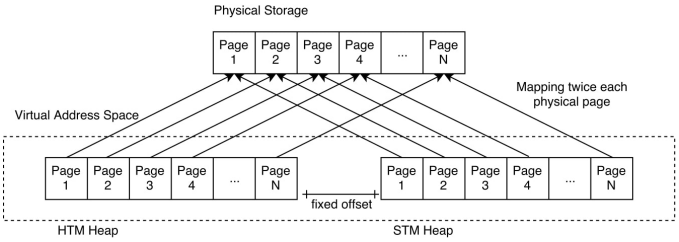


Figure 1: Map of the address space in the HTM Heap and in the STM Heap

is very sensitive to instrumentation, requiring additional memory accesses given its limited capacity.

The revoke of access rights is achieved by using the system call *mprotect* of Unix. This system call changes the protection of calling process memory pages contained in a given range of addresses. *Mprotect()* must be issued on a piece of memory obtained using *mmap()* and the given address must be aligned at the page boundary, which in DMP-TM is. Otherwise, according to the POSIX standard, its behavior is undefined.

The result is if a page's access rights is revoked, there is not a way to distinguish accesses performed by transactions running STM, which are always enabled by this concept from transactions running HTM, that must check the page protection first, because the page's access right is done at the process level, which makes pages inaccessible for the entire process.

In order to solve this, we organize the process virtual space in a way that the heap is mapped twice (figure 1): transactions running HTM access the corresponding pages in a mapping of the heap reserved for them, called HTM Heap, and in a similar way, transactions running STM access the corresponding pages in a mapping of the memory called STM Heap.

The result of this concept is that whenever transactions running HTM try to access a page for which they do not have the access, the operative system detects that the process does not have the access rights to do it and triggers an AV in the form of a SIGSEGV, which would be further treated by a Unix *signal handler*.

B. HTM component

The HTM implementation used in this solution is the one provided by IBM Power8. The leading cause that motivate us to choose this specific processor instead of the more common processor Intel's Haswell is that in Intel's Haswell when a transaction running HTM performs an access on a shared memory zone and triggers a *Segmentation Fault* (SIGSEGV) signal, we have experimentally verified that the signal handler is not executed and keeps retrying the transaction.

If aborted by a capacity abort, transactions running HTM immediately fallback to the one of the fallback mechanism described more in detail in subsection , i.e., either acquiring the fallback single global lock or run the transaction using STM. In case of a conflict, the transaction is repeated up to 5 times - a standard value that has been recommended in the study of Yoo et. al [16] - before activating the fallback path.

C. Signal Handler and Contention Management

When a transaction running HTM tries to access the data from a page to which it is not allowed, the operative system raises a SIGSEGV signal. This signal is handled by a Unix handler, where is possible to check the address that originated the SIGSEGV and consequently calculate the page that triggered it. The page protection is then restored accordingly to the type of access that HTM was going to do, this is an information passed specifically by the programmer, if the programmer does not pass any information then it is assumed that the transaction is an update one: if the transaction running HTM is read-only, then the protection of the page in the HTM Heap is changed to Read-mode, because it is guaranteed that there will not exist any update of memory in the future to cause a conflict with running STM transactions; if the transaction running HTM is an update transaction, then the memory protection is changed to Read/Write-mode. In order to notify transactions running STM that transactions running HTM now have Read/Write access to one of the pages previously read, and possibly have written to a page previously read, a special per-page counter, named *transition-count*, is incremented in the *signal handler*. If a transaction running STM transactions checks during its execution that the transition count has changed, then the transaction rollsback and restarts.

D. STM component

We opted to integrate in DMP-TM TinySTM, a very efficient word-based STM, which has been shown to have very robust performance across heterogeneous workloads [7].

As mentioned before, we want to avoid instrumentation on HTM, therefore in DMP-TM we opt for placing any additional run-time check aimed to enforce correct synchronization between HTM and STM, on the STM side. This is motivated by the fact that STMs need anyway to perform checks to detect conflicts, which allows amortizing the additional costs associated with detecting access violations to memory partitions. To this end, we associate with each memory page the following metadata: 1) *status field*, that encompasses the access rights that HTM currently has on that page (read, read-write, none), 2) *transition count* that encompasses how many times have the access rights for HTM have been restored to read/write, 3) *writer count* that

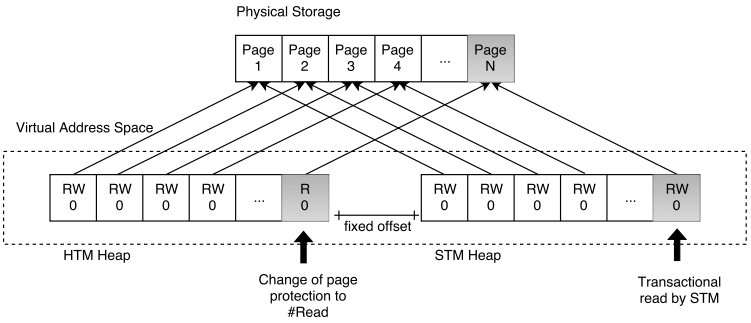


Figure 2: Software read

encompasses how many STM transactions have written the page.

E. Reads performed by STM transactions

When a transaction running STM issues a read (see figure 3). it first checks if the transition count of all the pages previously read have been changed since the last time they were read. If any of the transition count stored is different than the present one, then the transaction is restarted since an HTM transaction has restored the memory protection of the page and possibly issue a write to an address previously read by the STM transaction.

Then, it is checked the metadata of the page to which the transaction running STM issued a read. If the protection of the corresponding page in the HTM Heap is Read/Write, which means that HTM transactions can write a value whenever STM is reading, then the lock bit is grabbed using the directive *compare-and-swap*, which guarantees that only one transaction acquires the *lock bit* and issues a system call *mprotect* in order to change the protection of the page to Read.

We considered which change of page protection a read done by STM should trigger and it is preferable that instead of revoking, we opted for preserving read rights for HTM transactions. This choice allows in fact concurrency between STM and HTM transactions that read, without updating, memory positions in that page. After altering the metadata of the page, it is issued a transactional read recurring to the function *stm_load()* of TinySTM and it is checked again if the transition count has changed. If it does, it means that the read performed is not legal an consequently the transaction is restarted. If not, the read has been successful.

F. Writes performed by STM transactions

When a transaction running STM issues a write operation, it first checks the page's metadata. Then, it checks the *status field* of the page containing the address and if it finds that the page that wants to be written is readable or writable by transactions running HTM, it acquires the *lock bit* using the *compare-and-swap* directive and revokes the access rights, by issuing a *mprotect* call and changing the protection of

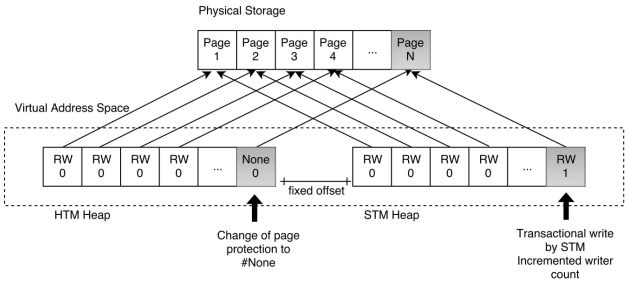


Figure 3: Software Write

the corresponding page in the HTM Heap to None, so that no HTM transaction can read it or write it in middle of an STM write. After changing the protection via *mprotect()* to None and furthermore the update of the *status field* of the page, the writer count for that page is increased. Finally, it is released the *lock bit* and is performed the transactional write of the value recurring to the function *stm_store()* of TinySTM.

G. Restart of an STM transaction

Whenever a software transaction finds the transition count of a page previously read different from the one read by the time the read is done, the transaction running software begins the process to restart itself. This operation is performed by first atomically decrease the number of writers that a page has in the case the aborted transaction had written the page and issue a restart of the transaction.

H. Commits performed by transactions running STM

By the time a *Software transaction* finishes its execution, it enters in the commit phase, in which it checks if the transition count of the pages previously read have changed meanwhile. Whenever a software transaction finds the transition count of a page different than the one read by the time the read is done, the software transaction begins the process to restart itself: First, recurring to the function *__sync_sub_and_fetch()*, it atomically decreases the number of writers for all the pages that the transaction has written before and finally issues a transactional restart recurring to the function *stm_abort()* provided by TinySTM.

IV. Variants of the algorithm

This section describes the variants of the algorithm developed in order to accommodate the cases when the solution reach the point that it cannot succeed using hardware transactions, either because of reaching the maximum number of tries in hardware or reaching the maximum number of page changes, thus having to fallback to other solutions. We developed two main variants of the algorithm:

- DMP-SGL - Resorting to executing the transaction sequentially, i.e., without any other transaction executing concurrently. This is the standard fall back approach for HTM. In this case, though, it has to be extended to ensure that neither HTM nor STM transactions are active concurrently to the transaction executing in the fallback path. As we will discuss next, we have developed three variants of this approach, which use different ways to block the execution of STM transactions.
- DMP-STM - Stopping the execution of HTM transactions (by acquiring the global lock they subscribe upon begin), and running the transaction using a plain version of TinySTM.

V. Memory Allocation

Recall that our solution relies on mapping the memory accessible via TM in specific memory regions of the address space. This implies that the mechanism in charge for allocating/deallocating memory dynamically (malloc/free in C) should be made aware of the range of addresses from which memory chunks should be allocated upon request of the application.

In order to address this issue, we developed a custom implementation of malloc that allocates memory from the mapped STM heap. In the beginning of its execution, the STM Heap is split in *n equal parts* aligned with the page's boundary, being *n* the number of threads. Then, when is needed to allocate a new piece of memory, the allocator returns one address of the split reserved for that thread and increments the *per-thread* counter that indicates the point to where, in the partition, memory is allocated.

The key benefit of this solution is that the memory allocator does not have to deal with deallocation of memory, as each thread consumes memory from different memory regions. This approach, although simple, comes with a cost: if a thread finishes its split, and even if the other splits are empty, it cannot access those memory regions. However, the development of an efficient malloc support is outside of the scope of this dissertation and that the current, simple approach was sufficient to carry out the evaluation of the system.

VI. Correctness Argument

In this section, we provide a set of (informal) arguments on the correctness of the proposed solution.

As for HTM transactions, they are effectively isolated by concurrent updates by STM transactions as a HTM transaction is aborted as soon as it attempts to access a memory address for which it does not hold adequate (read or Read/Write) permissions.

The cases where HTM transactions read a value from a page and prior to their commit, a STM transaction begins and writes to the same variable are straightforward: Because cache coherence mechanism uses physical addresses to detect conflicts, it ensures correctness even when the accesses are issued on different virtual addresses that point to the same physical address.

The case where pages have been written by software transactions are straightforward: each transaction increments the writer count atomically and revokes the access for HTM transactions before issuing the transactional write. On the signal handler side, if the transaction wants to revert to Read/Write protection, the AV waits until there are no writers active and then acquires the lock bit atomically which guarantees that its the only transaction that can execute system calls on that page.

The cases where pages have been read are handled in the following manner: Before STM's first read to a page, it has to store the page's transition count and also check if the access protection for transactions running HTM is either None or Read-only. In the negative case, it acquires *lock bit* and issues a system call in order to change that protection to Read. After it has performed the transactional read, the transaction must check that the transition count read before remained unchanged. This means that the page was never written by HTM, because if it has changed the protection meanwhile, the transition count would be different than the one before.

Finally, at commit time, all the transition count of the pages read are checked ,otherwise some HTM transaction could write meanwhile and invalidate the values read by STM transactions before. In order to ensure opacity, the transition count of the pages read, so far, by an active STM transaction are also checked. This ensures that no HTM can have conflicted with the STM transaction, by overwriting any of the values it read so far.

VII. Evaluation

A. Performance Evaluation

This section presents an extensive evaluation of DMP-TM. In subsection VII-B, we present the methodology and the evaluation metrics used in this study. Then, subsection VII-C presents the results obtained using synthetic benchmarks, .i.e., benchmarks created by the authors. Finally, subsection VII-D presents the results obtained in 3 benchmarks of the STAMP benchmark suite [15].

B. Methodology and Evaluation Metrics

In this experimental evaluation, we used multiple workloads representative of real-life applications, such as the ones contained STAMP benchmark suite, in order to stress the advantages of using DMP-TM in real life applications. In order to compare all the implementations, we used the following evaluation metrics: throughput, abort rate, breakdown of aborts, breakdown of commits and Number of System calls. The number of system calls is particularly important, since DMP-TM relies on *mprotect()* to synchronize the execution of transactions running HTM and transactions running STM, is important to track the number of *system calls* issued, as this is an evidence of how precise the initial partitioning is.

All the 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 4.5.5 and the compiler used is GCC 6.2.1 with -O2 optimization level. The reported results represent the average of 5 runs.

For the cases that the number of threads exceed the number of physical cores, we start to pin more than one thread to the same physical core, leading to have worse performance because different threads share the same processor, thus the same cache lines. If the memory is not aligned with the cache line size, two different variables could end up on the same cache line and if two threads read/write two variables on the same cache line, then one of them is invalidated, further aborting transactions running hardware and the processor is forced to use concurrency mechanisms to access those cache lines. By aligning the variables to the cache line size, we avoid *false sharing* as two variables are guaranteed to reside on different cache lines.

Also, we tuned the value of maximum changes of protection of 5. This way, the maximum partition's change that we allow is 5, after that, we use the fallback mechanism.

C. Experiments with 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 pool of pages, HTM and STM pool, where each page represents a set of buckets containing various elements. Each page, depending on the pool that it belongs, is composed by l buckets with each bucket containing e elements. A page belonging to the HTM pool is composed by $l = 10\ 240$ buckets with each bucket containing $e = 32$ elements not ordered, being each element a *long*. This way it is expected to be benefited the execution of transactions running HTM in deterioration of transactions running STM, since if a transaction have to go through all the elements of the bucket the size of each bucket will not exceed the cache capacity which is 8KB, i.e 1024, elements, because each element is a 8 byte long. On opposite side, if the page belongs to STM pool, then the page is composed by $l = 176$ buckets with each bucket containing $e = 1024$ elements not ordered, this hashmap is expected to benefit transactions running STM in deterioration of transactions running HTM, since if a transaction reads all the elements of the bucket will likely to exceed the cache's capacity and abort due to capacity aborts.

There are two main parameters that we use to tune the benchmark in order to exert precise control over the workload characteristics and gain deeper understanding over the behavior of DMP-TM over certain workload characteristics in comparison with the state of the art TM implementations. The parameters are the following:

- ***p_bias*** - indicates the percentage of operations issued on pages that belong to the STM pool, if *p_bias* equals

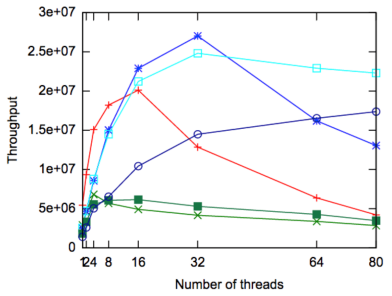
to 0% , then all operations will be issued to pages belonging to HTM pool and by opposition, if *p_bias* equals to 100% then all the operations will be issued to pages belonging to the STM pool. This way is possible to control the transaction length, since the upper limit is 32 elements for transactions issuing operations on pages belonging to the HTM pool and 1024 elements for transactions issuing operations on pages belonging to the STM pool, being the former case beneficial for transactions running HTM, because they are faster than transactions running STM by not incurring extra instrumentation, and the latter for transactions running STM because the number of elements will likely induce capacity aborts by overflowing the cache's capacity.

- ***p_op*** - indicates the percentage of the update operations issued, if *p_op* equals 0% then all the operations issued are reads and by opposition, if *p_op* equals to 100%, then all the operations issued are writes.

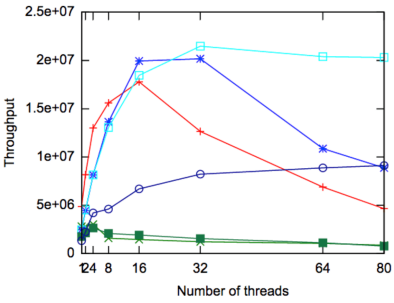
Then, according to the random numbers obtained and the *p_bias* and *p_op* defined at the beginning, the thread will pick a random page from either the HTM or STM pool and issue a read or write operation in it. If it is a read operation, it will choose a random number between the range of [0-Pool Population], being Pool Population the possible number of values present in the bucket. We use Pool Population 100 times the number of elements in a bucket. Then, it will calculate the bucket in which the element would be present, by simply using the formula *bucket_to_search = value_to_find % total_number_of_buckets_in_page* and go through all the elements in the bucket, if it finds the value that it is looking for, it stops, if not it continues until the end of the bucket.

If it is a write operation, it will generate two random numbers between the range of [0-Pool Population], then it will find the bucket which is supposed to be the first value by using the formula stated before and will read through all the elements, if it finds the element, it writes the second value instead of the first, if not, it continues until the end of the bucket. By reaching the end of the bucket, it writes the second value in a random position.

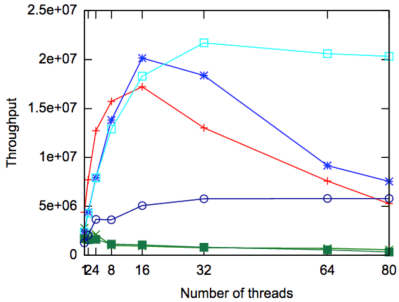
Figure 4 captures the results obtained by adjusting the parameter *p_bias* equals to 2%. In all the workloads obtained by varying the probability of writes, HTM-SGL scales up to 32 threads, being the best TM implementation in the comparison up to 8 threads, then after 8 threads it begins to have a decrease in the throughput. Behind the loss of throughput is that the number of transactions that active the fallback increases, in the case of HTM-SGL, after 8 threads. Surprisingly, Hybrid-NOrec achieves the worst throughput in comparison with all the TM implementations present in the experiment, this is explained by the fact that Hybrid-NOrec also has the highest abort rate of the experiment. The algorithms presented in this paper shined in this specific workload by using the correct TM implementation according to the pool of the page accessed, this granted average speedups of 1.7x after 16 threads for *p_op* = 10%, average speedups of 2x in *p_op* = 50%, and average speedups of 1.5x in the case of *p_op* = 90%. Following a similar trend as last experiment, DMP-SGL begins to have performance penalties after 32, due to the fact that some transactions begins to commit using the fallback mechanism, however DMP-STM uses this mechanism in its favor and can maintain the same throughput from 32 to 80 threads.



(a) Throughput of experiment on hashmap with *p_bias*=2% and *p_op*=10%



(b) Throughput of experiment on hashmap with *p_bias*=2% and *p_op*=50%



(c) Throughput of experiment on hashmap with *p_bias*=2% and *p_op*=90%

Figure 4: Comparison between state of the art TM implementations, DMP-STM and DMP-SGL using probability of accessing the STM pool = 2%

D. STAMP benchmark

In this subsection we present three different experiments conducted with STAMP benchmark. These experiments are intended to stress the benefits of using DMP-STM and DMP-SGL in real-world applications, first we analyzed three scenarios in Vacation, a online travel agency, then, we tested with Genome, a process of taking a large number of DNA segments and matching them to reconstruct the original source genome and Intruder, a Signature-based network intrusion detection systems (NIDS).

It should be noted that we incorporated the mechanism for allocating new memory described in subsection V in every TM implementation described in this subsection with the goal of having a fair comparison between all the implementations and also to stress the usability of the partition mechanism proposed in this dissertation.

1) Vacation: Vacation implements an online transaction processing system that intends to emulate a travel reservation system. The system is composed by a set of trees that keep track of customers and their reservations for various time travels items. There are three types of operations: reservations, cancellations and updates and each operation is enclosed in a transaction that guarantees the consistency of the database, since multiple threads can read and update reservations.

Minh et. al [15] proposed two different scenarios, by using different parameters that generates different workloads: vacation low, a workload with low contention, and vacation high, a workload with high contention. In addition to vacation low and vacation high workloads we generated a third one using the following parameters: `-n5 -q60 -u90 -r16384 -t4096`. The generated workload has higher number of queries per client, which tests the workloads with more contention than vacation high.

In this subsection we present three different experiments conducted with STAMP benchmark. This experiments are intended to stress the benefits of using DMP-STM and DMP-SGL in real-world applications, first we analyzed three scenarios in Vacation, a online travel agency, then, we tested with Genome, process of taking a large number of DNA segments and matching them to reconstruct the original source genome and Intruder, a Signature-based network intrusion detection systems (NIDS).

It should be noted that we incorporated the mechanism for allocating new memory described in subsection V in every TM implementation described in this subsection with the goal of having a fair comparison between all the implementations and also to stress the usability of the partition mechanism proposed in this dissertation.

2) Vacation: Vacation implements an online transaction processing system that intends to emulate a travel reservation system. The system is composed by a set of trees that keep track of customers and their reservations for various time travels items. There are three types of operations:

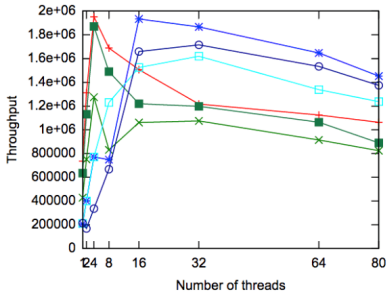


Figure 5: Vacation Low Throughput

reservations, cancellations and updates and each operation is enclosed in a transaction that guarantees the consistency of the database, since multiple threads can read and update reservations.

Minh et. al [15] proposed two different scenarios, by using different parameters that generates different workloads: vacation low, a workload with low contention, and vacation high, a workload with high contention. In addition to vacation low and vacation high workloads we generated a third one using the following parameters: `-n5 -q60 -u90 -r16384 -t4096`. The generated workload has higher number of queries per client, which tests the workloads with more contention than vacation high.

By analysing figure 5 is possible to infer that in vacation low, HTM-SGL scales up to 4 threads, closely followed by NOrec, that has higher throughput due to the low contention present in this workload, with average throughput difference of 5%. Then, after 4 threads, both HTM-SGL and NOrec have performance penalties due to the increase of the abort rate, which goes from roughly 0.017 with 8 threads to almost 0.025 with 8 threads, and continue to decrease their throughput up to 80 threads. At 16 threads, DMP-SGL, TinySTM and DMP-STM begin to outperform all the other TM implementations and remain that way for the rest of the thread counts; DMP-SGL achieves the best throughput of the test after 16 threads, having average 1.13x speedups in comparison with TinySTM, we achieve this by running software transactions whenever are done reservations or whenever the administrator add a new item to the database, all other operations, namely the delete of a customer and the calculation of a bill of a certain client are calculated by using transactions running HTM. DMP-STM achieves the third best throughput in the study, having 0.12x throughput penalty compared with the second best, TinySTM.

By analysing figure 6 is possible to infer that in vacation high, the workload is prone to benefit HTM-SGL and NOrec up to 4 threads, which after that thread count begin to have some performance penalties. DMP-SGL has the worst

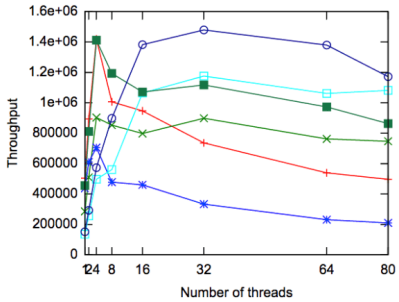


Figure 6: Vacation High Throughput

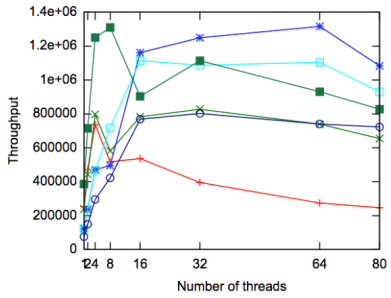


Figure 7: Vacation with more queries per-client Throughput

throughput in this experiment, since the average commit of 30% of transactions executing the fallback mechanism hinder the throughput. DMP-STM achieves the second best throughput in the experiment, because of the higher abort rate, transactions revert to the execution of TinySTM, which greatly improves the performance of this experiment. TinySTM achieves the best throughput in this experiment with 1.3x speedups in comparison with the second best, DMP-STM. Hybrid NOrec achieves a medium performance in this experiment, being better than HTM-SGL and DMP-SGL but being worse than TinySTM, DMP-STM and NOrec. DMP-STM and DMP-SGL issue an average of 200 system calls during their execution to synchronize the execution of transactions running STM and transactions running HTM.

By analysing figure 7 is possible to infer that in the proposed scenario of vacation with more queries per-client, NOrec achieves the best throughput up to 8 threads and after it, the workload is dominated by the proposed solutions that, in the case of DMP-SGL, achieve on average 1.3x speedups in comparison with TinySTM and Hybrid-NOrec and on average 3.25x speedups in comparison with HTM-SGL.

3) Genome: Genome represents the process of taking a large number of DNA segments and matching them to reconstruct the original source genome. This benchmark has two phases: the first phase uses a hash-set on the pool of duplicate DNA segments in order to create a set of unique segments. In the second phase, it tries to take a segment from the pool and try to add it to a set of currently matched segments. Transactions are used in both phases of the algorithm to guarantee that the changes to the pool are coherent.

By checking figure 8 is possible to check that HTM-SGL has higher throughput up to 2 threads and after that DMP-STM achieves the best throughput in all the study, achieving speedup gains in the order of 1.2x until 64 threads, where it begins to have similar throughput of TinySTM. DMP-SGL is not competitive, achieving the third best throughput due to the fact that it has roughly 0.3 abort rate and 1% of transactions using the fallback mechanism, which hinders the performance. Hybrid-NOrec has the worst throughput in this experiment, mainly because of its high abort rate, which causes falling back to NOrec and consequently the abort of other transactions running HTM. DMP-SGL commits using 97% of transactions running HTM, 2% of transactions running STM and 1% of transactions using the fallback mechanism. DMP-STM commits using 95% of transactions running HTM, 4% of transactions running STM and 1% of transactions using the fallback mechanism; we achieve this by running transactions executing STM in the first phase of the algorithm and running transactions executing HTM in the second phase of the algorithm.

VIII. Conclusion

This paper proposes a novel Hybrid TM implementation called Dynamic Memory Partitioning (DMP-TM), which relies on the hardware's memory protection mechanism to synchronize the shared accesses by HTM and STM. DMP-TM divides the memory layout in pages and those pages are either accessed by HTM with no instrumentation, which minimizes the probability of incurring aborts, or using STM, which incurs an extra-overhead by changing the protection via *system call* for transactions running HTM. That way whenever one implementation accesses data in a page to which its access is not allowed it receives an SIGSEGV that indicates that the page could be accessed by the other implementation and possibly causing a conflict and wrong behavior. We designed an algorithm intended to minimize the costs of having to issue system calls and evaluated several fallback mechanisms. In the light of this study, this paper proposes two variants of the algorithm: DMP-SGL and DMP-STM. DMP-TM has been evaluated in an extensive study recurring to a synthetic benchmark composed by two disjoint hashmaps, one more prone to benefit transactions running HTM and another one more to benefit transactions running STM. DMP-TM demonstrated to be competitive when accesses are done only to one of the hashmaps, a test

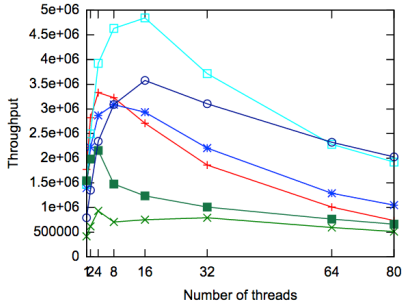


Figure 8: Genome Throughput

that is expected to be beneficial for respectively HTM and STM, depending on the hashmap accessed. Also DMP-TM demonstrated to achieve average performance gains of 1.5x (and up to 4x) in comparison to TinySTM whenever the larger hashmap is used with probability less than 10%. As for HTM, throughput gains in these workloads average 1.33x with peak gains that extend up to 4x. Finally, when compared with Hybrid NOrec, DMP-TM achieves throughput gains up to 10x. When considering workloads that are designed to be optimally managed by either STM or HTM, the proposed solution remains very competitive, achieving indistinguishable performance when compared to HTM, and averages overheads of 20 (and maximum of 30) vs TinySTM. This way, we proved that Hybrid TM is still a promising concept and that for workloads characterized by accesses performed on disjoint pages, Hybrid TM proves to be better than HTM and STM by using the best of both implementations.

REFERENCES

[1] V. Pankratius and A.-R. Adl-Tabatabai, "A study of transactional memory vs. locks in practice," in *Proceedings of the Twenty-third Annual ACM Symposium on Parallelism in Algorithms and Architectures*, ser. SPAA '11, 2011, pp. 43–52.

[2] M. Herlihy and J. E. B. Moss, "Transactional memory: Architectural support for lock-free data structures," in *Proceedings of the 20th Annual International Symposium on Computer Architecture*, ser. ISCA '93, 1993, pp. 289–300.

[3] L. Dalessandro, M. F. Spear, and M. L. Scott, "NOrec: streamlining STM by abolishing ownership records," *ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP)*, pp. 67–78, 2010.

[4] A. Dragojević, R. Guerraoui, and M. Kapalka, "Stretching transactional memory," *ACM SIGPLAN Notices*, vol. 44, p. 155, 2009.

[5] P. Felber, C. Fetzer, and T. Riegel, "Dynamic performance tuning of word-based software transactional memory," in *Proceedings of the 13th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, ser. PPoPP '08, 2008, pp. 237–246.

[6] J. E. Gottschlich, M. Vachharajani, and J. G. Siek, "An efficient software transactional memory using commit-time invalidation," in *Proceedings of the 8th Annual IEEE/ACM International Symposium on Code Generation and Optimization*, ser. CGO '10, 2010, pp. 101–110.

[7] N. Diegues, P. Romano, and L. Rodrigues, "Virtues and limitations of commodity hardware transactional memory," in *Proceedings of the 23rd International Conference on Parallel Architectures and Compilation*, ser. PACT '14, 2014, pp. 3–14.

[8] I. Calciu, J. Gottschlich, T. Shpeisman, G. Pokam, and M. Herlihy, "Invyswell: A hybrid transactional memory for haswell's restricted transactional memory," in *Proceedings of the 23rd International Conference on Parallel Architectures and Compilation*, ser. PACT '14, 2014, pp. 187–200.

[9] 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," *SIGPLAN Not.*, vol. 46, no. 3, pp. 39–52, Mar. 2011.

[10] P. Damron, A. Fedorova, Y. Lev, V. Luchangco, M. Moir, and D. Nussbaum, "Hybrid transactional memory," in *Proceedings of the 12th International Conference on Architectural Support for Programming Languages and Operating Systems*, ser. ASPLOS XII, 2006, pp. 336–346.

[11] A. Matveev and N. Shavit, "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*, ser. ASPLOS '15, 2015, pp. 59–71.

[12] T. Riegel, P. Marlier, M. Nowack, P. Felber, and C. Fetzer, "Optimizing hybrid transactional memory: The importance of nonspeculative operations," in *Proceedings of the Twenty-third Annual ACM Symposium on Parallelism in Algorithms and Architectures*, ser. SPAA '11, 2011, pp. 53–64.

[13] "Power ISA™ Version 2.07," 2013. [Online]. Available: https://www.power.org/wp-content/uploads/2013/05/PowerISA$_V$2.07$_P$UBLIC.pdf

[14] T. Riegel, C. Fetzer, and P. Felber, "Automatic data partitioning in software transactional memories," in *Proceedings of the Twentieth Annual Symposium on Parallelism in Algorithms and Architectures*, ser. SPAA '08, 2008, pp. 152–159.

[15] C. C. Minh, J. Chung, C. Kozyrakis, and K. Olukotun, "Stamp: Stanford transactional applications for multi-processing." in *IISWC*. IEEE Computer Society, 2008, pp. 35–46.

[16] R. M. Yoo, C. J. Hughes, K. Lai, and R. Rajwar, "Performance evaluation of intel® transactional synchronization extensions for high-performance computing," in *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*, ser. SC '13, 2013, pp. 19:1–19:11.