SCISPACE
formerly Typeset

Open access • Journal Article • DOI:10.1109/TPDS.2018.2861712

The Case for Phase-Based Transactional Memory — Source link

Joao P. L. de Carvalho, Guido Araujo, Alexandro Baldassin

Institutions: State University of Campinas, Sao Paulo State University

Published on: 01 Feb 2019 - IEEE Transactions on Parallel and Distributed Systems (Institute of Electrical and Electronics Engineers (IEEE))

Topics: Transactional memory, Hybrid system, Serialization, POWER8 and Benchmark (computing)

Related papers:

- An effective hybrid transactional memory system with strong isolation guarantees
- Hybrid Transactional Memory Revisited
- Architectural support for high-performing hardware transactional memory systems
- PhTM: Phased Transactional Memory
- Optimizing hybrid transactional memory: the importance of nonspeculative operations

Share this paper:

View more about this paper here: https://typeset.io/papers/the-case-for-phase-based-transactional-memory-2mf04go2bc

The Case for Phase-Based Transactional Memory

João P. L. de Carvalho, Guido Araujo, and Alexandro Baldassin

Abstract—In recent years, Hybrid TM (HyTM) has been proposed as a transactional memory approach that leverages on the advantages of both hardware (HTM) and software (STM) execution modes. HyTM assumes that concurrent transactions can have very different phases and thus should run under different execution modes. Although HyTM has shown to improve performance, the overall solution can be complicated to manage, both in terms of correctness and performance. On the other hand, Phased Transactional Memory (PhTM) considers that concurrent transactions have similar phases, and thus all transactions could run under the same mode. As a result, PhTM does not require coordination between transactions on distinct modes making its implementation simpler and more flexible. In this article we make the case for phase-based transactional systems using PhTM*, the first implementation of PhTM on modern HTM-ready processors. PhTM* novelty relies on avoiding unnecessary transitions to software mode by: (i) taking into account the categories of hardware aborts; (ii) adding a new serialization mode. Experimental results using Broadwell's TSX reveal that, for the STAMP benchmark suite, PhTM* performs on average 1.68x better than PhTM, a previous phase-based TM, 2.08x better than HyTM-NOrec, a state-of-the-art HyTM, and 2.28x better than HyCO, the most recent hybrid system in the literature. In addition, PhTM* also showed to be effective when running on a Power8 machine by performing over 1.18x, 1.36x and 1.81x better than PhTM, HyTM-NOrec and HyCO, respectively. We also show that STAMP applications do not exhibit hybrid behavior to justify the use of conventional hybrid systems, thus making PhTM* a better solution to those type of programs. Finally, we show for the first time that conventional hybrid systems do not perform better than phased-based system in a scenario with hybrid-behaved transactions.

Index Terms—Transactional memory, phase-based execution, performance evaluation

1 Introduction

The idea of using *memory transactions* as an abstraction for exposing parallelism is an approach that has been extensively studied in the last ten years [1]. In a nutshell, a transaction is a block of instructions that are executed in an all-or-nothing fashion: either the updates to memory take place immediately once the block is finished and the transaction is *committed*, or changes to memory are discarded and the transaction is *aborted*, due to conflicting updates by different transactions. Herlihy and Moss [2] proposed a hardware implementation to allow the execution of memory transactions and coined the name *Transactional Memory (TM)*.

Although hardware implementations of TM (HTM) enable fast mechanisms for commit, abort and conflict detection, they suffer from *capacity* aborts when executing transactions that overflow the speculative storage capacity of the underlying hardware. Chip manufacturers such as IBM and Intel have recently added support for hardware transactions in their processors [3], [4], [5]. Such support is provided as *best-effort*, in the sense that there is no guarantee that a transaction will ever succeed executing in hardware. In such case, the HTM runtime must assure that the transaction will eventually commit. Contrary to HTM, software approaches for TM (STM) have been investigated [1] and became very popular due to their capacity flexibility, although the cost of such flexibility can sometimes impact program performance.

In order to leverage on the advantages of both HTM and STM, Hybrid TM (HyTM) schemes have also been proposed [6], [7], [8], [9], [10], [11], [12], [13]. In HyTM, a dynamic mechanism is provided to assign each transaction to HTM or STM according to its demand for speculative storage. The concurrent execution of both hardware and software transactions in HyTM might become quite complicated to manage, both in terms of correctness and performance. For instance, detecting conflicts between hardware and software transactions usually requires some kind of instrumentation on the fast path (hardware), possibly making the common case slower.

Another approach for TMs, *Phased Transactional Memory* (PhTM), was proposed in 2007 by Lev et al. [14] to allow the execution of transactions in *phases*. In a phase, all transactions execute in the same mode (hardware or software). As a result, PhTM does not require coordination between transactions running in different modes (as is the case with HyTM), making its implementation simpler and more flexible. For example, in PhTM different STMs can be used while in software mode. As described in [14], the main challenges of PhTM lies in: (i) identifying efficient modes;

- *J.P.L. de Carvalho and G. Araujo are with UNICAMP – Institute of Computing, Campinas 13083-970, Brazil. E-mail: {joao.carvalho, guido}@ic.unicamp.br.*
- *A. Baldassin is with UNESP – Univ Estadual Paulista, São Paulo 01049-010, Brazil. E-mail: alex@rc.unesp.br.*

Manuscript received 30 Jan. 2018; revised 17 July 2018; accepted 27 July 2018. Date of publication 31 July 2018; date of current version 16 Jan. 2019. (Corresponding author: João P. L. de Carvalho). Recommended for acceptance by A. C. M. Melo. For information on obtaining reprints of this article, please send e-mail to: reprints@ieee.org, and reference the Digital Object Identifier below. Digital Object Identifier no. 10.1109/TPDS.2018.2861712

1045-9219 © 2018 IEEE. Personal use is permitted, but republication/redistribution requires IEEE permission. See http://www.ieee.org/publications_standards/publications/rights/index.html for more information.

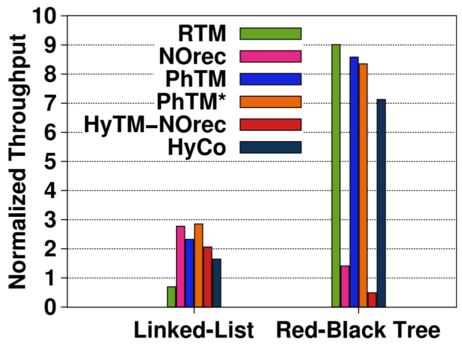


Fig. 1. Intset – 14 threads, 20 percent updates and 80 percent lookups.

(ii) managing correct (and fast) transitions between phases; and (iii) deciding when to switch phases.

At the time it was proposed, PhTM was evaluated in a simulated environment. The approach was not further investigated mostly because faster HyTM designs started to appear. Some of the arguments used against PhTM revolve around: (i) the need for system-wide synchronization barriers (which might incur in a high overhead), (ii) difficulty in determining when to change phases, and (iii) reduced performance due to a single transaction requiring execution in a slower mode (e.g., software) [8]. Although these arguments have been taken for granted, we are not aware of any work that has investigated the performance of PhTM on contemporary HTM-enabled processors.

In this article we shed some light on the effectiveness of phase-based transacional systems by making use of both the original version of PhTM, as described by Lev et al. [14], and its improved version (PhTM*) previously proposed by ourselves in [15]. We show that they are an efficient and appropriate approach for the execution of transactional programs on modern HTM-ready processors. To achieve that we show that PhTM* outperforms conventional hybrid systems for all STAMP applications. We also show that STAMP applications do not exhibit hybrid behavior to justify the use of conventional hybrid systems (HyTM-NOrec [8] and HyCO [13]), therefore making PhTM* a better suited solution. Finally, we show for the first time that conventional hybrid systems do not perform better than phased-based system in a scenario with hybrid-behaved transactions.

1.1 Motivating Example

To illustrate the potential of PhTM*, please consider Fig. 1, where performance evaluation numbers have been measured for two programs from `Intset`, a microbenchmark that implements basic data structure operations (search, insert, remove) over a sorted integer set. Fig. 1 presents (x axis) two `Intset` programs: a linked-list and a red-black tree, both containing 4096 elements. The configuration shown in the figure performs 20 percent updates (insert and remove) on four threads, and was run in six transactional systems: a hardware implementation (Intel's RTM), an STM (NOrec), the original (PhTM) and our phased approach (PhTM*), Dalessandro et al. hybrid system (HyTM-NOrec) [8] and Spear et al. hybrid system (HyCo) [13]. The y axis shows the normalized throughput with regard to the sequential execution (higher is better).

As shown in Fig. 1, RTM does not perform well when executing linked-list operations because of capacity aborts: most of the transactions cannot complete due to resource limitations and are serialized. On the other hand, NOrec performs much better (close to 3x). PhTM* is able to detect the best mode (software in this case) and also performs well, close to NOrec. Note that both hybrid systems (HyTM-NOrec and HyCO) perform better than RTM but still not as good as the phased systems.

On the other hand, for the red-black tree the behavior is the opposite: RTM performs very well (9x), since there are no capacity aborts in this scenario, while NOrec presents no speedup. Again, PhTM* is capable of detecting the best mode (hardware in this case) and has similar performance (close to 8.5x), whereas HyTM-NOrec does not provide any speedup. HyCO achives good speedup, but falls significantly behind RTM, PhTM and PhTM*.

In Section 4 a set of experiments is performed on larger benchmarks which support the claim that PhTM* can dynamically detect and get the best of both HTM/STM modes outperforming even more complex HyTM approaches.

1.2 Contributions

This article makes the following contributions:

- It confirms that PhTM* [15] is an efficient phase-based transactional system that provides the following novelties: (i) it avoids unnecessary switches to software mode by considering not only the number, but also the categories of hardware aborts; (ii) it reduces the frequency of spurious software mode transitions by using a new serialization mode (Section 3);
- The experimental results on two different HTM processors, Intel Broadwell (4.2) and IBM Power8 (4.3), using the STAMP [16] benchmark suite, reveal that PhTM* is able to detect the best execution mode and provides superior performance compared to both PhTM [14], a previous phase-based TM, and HyTM-NOrec [8] and HyCO [13], two state-of-the-art hybrid implementations. In particular, PhTM* performs on average 1.64x better than PhTM, 2.08x better than HyTM-NOrec and 2.28x better tahn HyCO on the Broadwell machine, and 1.18x, 1.36x and 1.81x on the Power8 machine, respectively (Sections 4.2 and 4.3);
- It shows that STAMP applications do not exhibit hybrid behavior to justify the use of conventional hybrid systems (HyTM-NOrec [8] and HyCO [13]), therefore making PhTM* a better suited solution. And, for the first time, it shows that conventional hybrid systems do not perform better than phased-based system in a scenario with hybrid-behaved transactions (Section 4.4);
- It presents the first experimental results showing the inherent limitations of conventional hybrid systems, as demonstrated by Shavit et al. [17] (Section 4.5);
- It discusses the virtues and limitations of phase-based transactional systems and reports our experience when designing PhTM* (Section 4.6).

This article is divided as follows. Section 2 presents the background and related work. Section 3 describes our PhTM* design. Section 4 presents the evaluation of PhTM*, comparing it against other state-of-the-art approaches. Finally, we conclude the article in Section 5.

2 Background

Transactional Memory (TM) is a parallel programming model which uses transactions to abstract the synchronization of shared memory accesses [1]. Instead of using a traditional approach such as *locks*, which require programmers to think about *how* to isolate multiple accesses to the same data, transactions only require programmers to think about *what* to synchronize: the TM runtime automatically provides atomicity, isolation and consistency. As a consequence, transactions avoid most of the pitfalls attributed to locks (e.g., deadlocks and priority inversion), while still providing similar or better performance [2].

To date, many TM systems were proposed and implemented either in hardware (HTM), software (STM) or some combination of both (HyTM). Recently, IBM and Intel have added basic HTM support to their processors [18], opening up new opportunities for TM research and improved performance.

2.1 Hardware Transactional Support

Most of the processors with transactional memory support available today only implement best-effort transactions, meaning that the hardware does not guarantee whether or not transactions will eventually complete; this task is left to the programmer or runtime.

Transactional support in Intel processors (*Haswell*) is available through the *Transactional Synchronization Extension* (TSX) [5], [19] and comes in two flavors: RTM (*Reduced Hardware Transactions*), a set of instructions that enable the control of transactions, and HLE (*Hardware Lock Elision*), two prefixes used to elide writes in lock variables and run critical sections speculatively. RTM is Intel best-effort implementation of hardware transactions and adds three main instructions: XBEGIN and XEND to mark the transactional code region, and XABORT to explicitly abort a transaction. When a transaction aborts, the code is redirected to a fallback path specified by the XBEGIN instruction. In this case, the hardware provides a register (EAX) which stores the possible causes of the abort.

The Power8 processor is the first IBM processor that implements hardware transactional support directly by the Power ISA [20]. Power8 is also the first HTM-ready processor that supports non-transactional operations inside of transactions through SUSPEND and RESUME instructions. All operations executed within a SUSPEND/RESUME block are not rolled-back on a transactional abort. Power8's hardware only provides best-effort progress guarantees and, similar to RTM, relies on the programmer to provide an alternative execution path (fallback) to be executed when transactions are unable to complete in hardware. The instructions TBEGIN, TEND and TABORT were added to the Power ISA to explicitly start, commit and abort transactions. Detailed information about the causes of transactional failures are provided by a special-purpose register (TEXASR).

2.2 Hybrid Transactional Memory

The main goal of hybrid transactional memory is to allow concurrent execution of both hardware and software transactions. The research on hybrid mechanisms gained momentum with the introduction of hardware support, but the approach has been studied at least since 2006 as shown by the works of Kumar et al. [6] and Damron et al. [7].

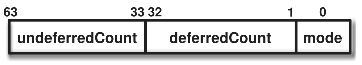
The advent of more efficient STM designs and real HTM implementations gave rise to new hybrid proposals. The NOrec STM [21], presented in 2010 by Dalessandro et al., simplified the implementation of STM by using a single global sequence lock and employing value-based validation. Hybrid algorithms based on NOrec (HyTM-NOrec) were proposed by Dalessandro et al. in 2011 [8]. The authors investigated different implementations with contrasting tradeoffs. For instance, while a straightforward implementation can provide opacity, it comes with drawbacks such as a single software transaction aborting all the hardware ones (regardless of having real data conflicts). More advanced algorithms using lazy subscription were also discussed, but they required sand-boxing or the execution of non-transactional instructions inside hardware transactions (not available on most commercial processors at the time).

Around the same time (circa 2011), Riegel et al. [9] proposed a family of hybrid implementations for AMD ASF (Advanced Synchronization Facility) [22]. Two specific implementations are discussed: one based on the Lazy Snapshot Algorithm (LSA) [23] and the other on NOrec [21]. Similar to some approaches described by Dalessandro et al. [8], their algorithms heavily rely on non-speculative operations inside transactions.

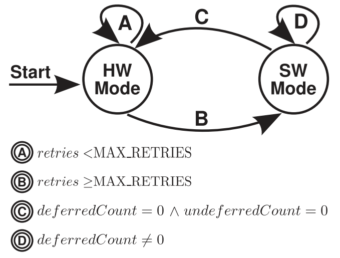
Matveev and Shavit proposed two hybrid approaches employing a multi-level fallback mechanism: RH-TL2 [10] in 2013, and RH-NOrec [11] in 2015. In general, the idea is to have transactions starting in hardware mode (fast path) but switching to a "mixed" slow path in case it fails. If it fails again, it then switches to a pure software mode (slow path). For RH-TL2, the "mixed" slow path mode executes the transaction body in software mode, but the commit operation is performed using a hardware transaction. As for RH-NOrec, the "mixed" slow path works by adding two small hardware transactions: (1) an HTM prefix, executing the largest possible number of reads in hardware (adjusted dynamically based on abort rate); (2) an HTM posfix, encapsulating all the writes. The modifications allowed the fast path to be executed without instrumentation.

Another hybrid approach, named Invyswell, was proposed by Calciu et al. in 2014 [12]. Invyswell uses a modified version of InvalSTM [24] as the software fallback. While HTM performs better with small transactions, InvalSTM is designed to handle large transactions and high contention. The idea of Invyswell is to combine both systems so as to provide good overall performance. Invyswell actually provides five execution modes: two in hardware and three in software. Having many different modes helps in adapting the system to different workloads, but complicates the design and requires efficient heuristics for transitioning between modes. Similarly to RH-NOrec, Invyswell performed well for some STAMP applications, but for others (such as Kmeans and Yada) a noticeable slowdown was noticed when compared to other approaches (either pure hardware or software schemes).

Although not a hybrid system per se, Xiao et al. [25] propose a decision tree-based approach to decide between hardware and software TM implementations. However, the proposed solution chooses between HW and SW statically,



(a) `modeIndicator`



(b) PhTM transition automaton

Fig. 2. PhTM – (a) Bit field with state configuration and (b) transition automaton.

differently from PhTM* [15], and therefore would not perform as good as the phase-based approach with long-running applications that change phases periodically.

Common to all hybrid approaches is the need to coordinate the concurrent execution of hardware and software transactions. In turn, this usually introduces some kind of instrumentation (slowing the fast path) and complicates the design of the algorithm. On the other hand, as described in the next section, the phased approach (PhTM) only allows one execution mode (e.g., hardware or software) at a time, making the algorithm design much simpler and more flexible.

2.3 Phased Transactional Memory

Phased Transactional Memory is a phase-based transactional system that executes transactions in modes selected according to properties like transactional length, level of contention and speculative capacity requirements. Unlike hybrid systems, in PhTM simultaneously executing transactions use the same running mode (e.g., HTM or STM). All the discussion in this section is based solely on Lev et al.'s [14] description of the PhTM implementation; no other algorithm nor pseudo-code has been published on this subject since then.

The prototype PhTM implementation described in [14] has two modes of execution: a pure hardware mode (HW), and a pure software mode (SW). While in HW, transactions execute using the HTM support available in the machine, whereas in SW mode transactional execution is provided by means of an STM library. Given that most HTM implementations available today have smaller overheads than STM libraries to manage a transaction (start, commit and abort), and that all accesses inside a hardware transaction are treated as transactional (no read/write barriers), the paths executed in HW or SW mode are respectively called the *fast-path* and *slow-path*. The phase transitions as well as the choice of which path to take are driven by three global variables: `mode`, `deferredCount` and `undeferredCount`. The system must be able to access these variables atomically, but most current processors only have support to atomically access at most 64bits. A way to circumvent this problem is to store them as bit fields of a 64 bit variable (`modeIndicator`): one bit for the mode state, and the remaining 63 bits split into two counters (`deferredCount` and `undeferredCount`). A possible configuration for the `modeIndicator` variable is shown in Fig. 2a.

As shown in Fig. 2b, the PhTM system starts in HW mode (`mode` bit not set). When a hardware transaction aborts, it increments a local counter with the number of retries. As described in [14], while this number is less than nine, the system remains in HW mode Ⓐ. Otherwise, the transactions that repeatedly failed initiate a transition to SW phase Ⓑ: each failed transaction increments `deferredCount` if, and only if, the system is still in HW, and sets the `mode` bit if not yet set. Therefore, a phase transition automatically causes all currently running hardware transactions to abort because variable `modeIndicator` belongs to the read-set of all hardware transactions. All aborted transactions restart and check which path to choose by reading `modeIndicator`.

The system remains in SW mode while `deferredCount` is nonzero Ⓓ. In this mode, transactions that incremented `deferredCount` (while still in HW mode) decrement it when they complete running in SW mode. Each *undeferred transaction* (those that did not increment `deferredCount`) first increment `undeferredCount` (making sure that the mode is SW and no HW mode transition is in place) and decrements it again once they finish. A transaction that decrements one of the counters to zero, while the other counter is also zero, forces a phase transition to HW Ⓒ.

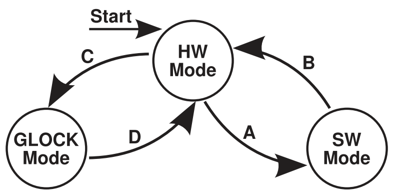
It is easy to see that PhTM has two major shortcomings concerning its phase transition policies. First, only the number of aborts is considered when changing the system to SW mode, so transactions that cannot complete in HW mode due to hardware limitations, e.g., capacity constraints, will fail 9 times before they have a chance to complete in SW mode. In addition, hardware transactions can fail arbitrarily (spuriously), so even transactions that could finish in hardware may be forced to switch to SW mode. Second, the system switches back to HW mode as soon as all deferred transactions, possibly only one, finishes and no undeferred transactions are running. As a consequence, the system may start going back and forth between modes: a failed transaction switches the system mode to SW, it then starts and finishes in this mode before any other transaction is able to run, forcing the system to switch back to HW almost instantly.

In the next section we propose PhTM*, an efficient Phased TM algorithm that addresses the above described drawbacks of PhTM. PhTM* avoids unnecessary switches to SW mode by considering not only the number, but also the categories of hardware transactions' aborts.

3 PhTM*– An Efficient Phased TM Algorithm

In this section we describe the first implementation of a phase-based transactional system designed with real HTM support on the fast-path. Our proposed implementation (PhTM*) is capable of using feedback information from HTM execution to guide the phase transition process. Contrary to PhTM, in PhTM* there are three modes of execution for all running transactions: (a) hardware mode (HW); (b) software mode (SW); and (c) serial mode (GLOCK). While in HW or SW, transactions execute using the TM support available in hardware or the support provided in any STM library, respectively. Transactions in GLOCK mode execute holding a global lock, irrevocably and in total isolation.

Similarly as in PhTM, both mode transitions are controlled using a single global variable, `modeIndicator`. The



$$Ⓐ \begin{cases} \exists t, t \in [0..9] | abort_{t-1} = abort_t = capacity \\ \wedge \\ abort\_rate > ABORT\_THRSD \end{cases}$$

$$Ⓑ \begin{cases} tx\_cycles < SIZE\_THRSD \\ \wedge \\ undeferredCount = 0 \\ \wedge \\ deferredCount = 0 \end{cases}$$

$$Ⓒ \begin{cases} retries \geq \text{MAX\_RETRIES} \\ \wedge \\ \neg Ⓐ \end{cases}$$

$$Ⓓ \; Serial\ Tx\ Commited$$

Fig. 3. PhTM*'s transition automaton

field configuration is similar to the one in Fig. 2a, only the lengths of the fields change: 2 bits for mode and 31 bits for each counter. A transaction in PhTM* starts in HW mode and uses the hardware transaction's abort information to decide which mode to choose, SW or GLOCK. Our proposed algorithm, differently of other usual hybrid systems (e.g., HyTM-NOrec), does not require any special HTM or STM implementations. In fact, any HTM providing explicit instructions to start, commit and abort transactions are suitable to be used with PhTM*.

To simplify the description of the algorithm we use HW→SW, HW→GLOCK, GLOCK→HW, and SW→HW to describe the transitions among hardware, software and serial modes. Fig. 3 shows the mode transition automaton of the PhTM* algorithm. As shown, PhTM* only switches to SW Ⓐ as a last resort when a HW transaction experiences an abort rate over a fixed threshold (`ABORT_THRSD`) and has persistent capacity aborts. This policy aims to detect long transactions and direct them to run in SW. A capacity abort is considered persistent if, and only if, it happens at least twice in a row. The abort rate is calculated considering all hardware abort causes, except capacity, and is given by the following equation:

$$abort\_rate_t = \alpha \cdot abort\_rate_{t-1} + (1 - \alpha) \cdot abort,$$

where $abort$ is 1 when a HW transaction aborts, and 0 otherwise. α is used to either give more weight to the past history ($abort\_rate_{t-1}$) or the current abort rate ($abort$). The abort rate computation was inspired by the contention intensity formulation introduced by Yoo et al. [26].

Mainly two reasons make long transactions better suited to run in SW rather than HW. First, transactions that persistently fail because of capacity issues will never be able to complete in HW as HTMs typically have limited transactional capacity [27]. Second, because the conflict detection granularity in STMs is usually finer than in HTMs, and all accesses inside a transaction are tracked to detect conflicts in most HTMs, hardware transactions are more susceptible to false conflicts than software transactions. Moreover, the longer a transaction the more likely it is to conflict with others, so two long transactions have a higher conflict probability in HW than in SW.

Once the system is in SW, deferred transactions start immediately. Deferred transactions are those which could not complete in HW and therefore trigger a HW→SW transition. Undeferred transactions are those which switched to SW because at least one deferred transaction initiated a HW→SW transition. Undeferred transactions must first increment `undeferredCount` while checking if both the system mode is still SW and there is no SW→HW transitions in place (`deferredCount` equals 0). Therefore, newly started undeferred transactions only increment their counter if deferredCount is not zero and the current state is SW. If an undeferred transaction sees that the mode changed to HW or that `deferredCount` is zero, it decrements `undeferredCount` if it has restarted and wait for the SW→HW transition to happen. After completion, undeferred transactions only have to decrement `undeferredCount`. On the other hand, deferred transactions must decrement `deferredCount`. When a deferred transaction commits for the N-th time it computes the average size, in cycles, of the transactions it has executed. If such average is above a fixed threshold (`SIZE_THRSD`), then the system remains in SW and the value of N is doubled. A SW→HW transition is started once transactions are small enough and both counters are zero Ⓑ. The system is always capable to return to HW simply because once a deferred transaction commits, it decrements deferredCount, and when such counter reaches the value zero, no other transactions starts. It is possible that deferredCount reaches zero while undeferredCount does not. In this scenario, the thread which decreased deferredCount to zero performs a SW→HW when condition Ⓑ is met.

After a hardware transaction starts, it checks if `modeIndicator` is zero or, in other words, if the system is in HW and no HW→SW is in place. If so, the transaction can safely execute in HW, given that the `modeIndicator` variable belongs to the read-set of all transactions in HW and any mode transition will conflict with and abort all hardware transactions. In such case, either the mode changed or a transition to SW is in place. If the mode changed to GLOCK the transactions wait until the serial transaction completes and retries in HW. On the other hand, if the system is in SW, or is attempting a HW→SW transition, the transactions will simply restart in SW.

PhTM* serializes transactions only when they fail repeatedly due to non-persistent capacity aborts Ⓒ. This policy prevents transactions from livelocking and allows PhTM* to quickly return to HW. A transition to GLOCK may fail because another thread may already have switched the system to SW. Notice that only a single thread can run in GLOCK at a time; the other threads must wait the GLOCK transaction to complete Ⓓ. As PhTM* tries to serialize only short transactions, the amount of time waiting for the GLOCK transaction is usually very small (see Section 4). Given that GLOCK transactions run irrevocably and in isolation, the only post completion action necessary is to switch the system to HW, atomically writing 0 to `modeIndicator`.

4 Experimental Results

The purpose of this section is to present a quantitative analysis of the performance of our phase-based transactional system, PhTM*, comparing it against the original phased transactional prototype, hardware, software, and hybrid systems.

4.1 Experimental Setup

The following transactional systems are evaluated[1]:

- *RTM:* Intel hardware implementation of TM. Each transaction may retry up to 9 times.[2]In case this threshold is reached or an abort is persistent, indicated by a hardware flag, the transaction is serialized with a global lock. The *lemming effect* [28] is treated by delaying the restart of new HW transactions till the lock is free;
- *PowerTM:* Power8 hardware implementation of TM. This system shares the implementation code of RTM, but uses Power8's HTM instructions instead of TSX's;
- *NOrec:*This is the code developed by the Rochester Synchronization Group and released as part of the RSTM package [29]. NOrec employs a global sequence lock, performs deferred versioning and eager conflict detection by means of value-based validation;
- *HyTM-NOrec:* We implemented and used the 2-location version of HyTM-NOrec, with eager subscription of the software sequence lock. The code for the software path of HyTM-NOrec is the same used for NOrec. Besides adding the hardware path, the only changes necessary to implement HyTM-NOrec are: (i) eagerly subscribe the sequence lock after starting a HW transaction; (ii) increment the global hardware commit counter in the pre-commit phase; and (iii) re-validate the software transactions whenever such counter changes;
- *HyCo:* We also implemented and used the Hybrid Cohorts [13] algorithm. Transactions on the software path first try to writeback their updates inside a HW transactions, allowing concurrent execution of both SW and HW transactions. In the experiments, we used the same threshold values as Spear et al. [13]: HW transactions are retried up to 20 times and transactions are serialized with a global lock after 5 SW failed commit validations. The hardware-assisted commit in SW is only retried twice;
- *PhTM:* The prototype implementation described by Lev et al. [14] and presented in Section 2.3. The maximum number of HW retries (`MAX_RETRIES`) was set to 9 and the software component of PhTM is based on NOrec;
- *PhTM*:* Our phase-based transactional system using the algorithm described in Section 3. The implementation shares much of the same code used by PhTM, including the software path, which is based on NOrec. The number of HW retries (`MAX_RETRIES`) was again set to 9, the abort rate threshold (`ABORT_THRSD`) and value of α were set to 60 and 75 percent, respectively. The value of $30k$ cycles was used as the threshold for transaction length (`SIZE_THRSD`). When running in software mode, a transaction length is initially averaged every 100 executions (this value is doubled, to a maximum of 1000, everytime a new average is calculated). These values were chosen based on a performance analysis conducted with the STAMP benchmark. We discuss the implications of changing these thresholds further in Section 4.6.

The results reported in this section represent the mean of 20 executions. All applications and algorithm implementations were compiled with GCC (GNU C/C++ Compiler) 7.2 and optimization level three (`-O3`). In this section we present experiments conducted on two different machines, one powered with a *Broadwell* processor and another with a IBM *POWER8*. Each machine has the following configuration:

- *Broadwell:* This machine has a Intel Xeon E5-2660 v4 2.0 GHz processor and 64 GB of RAM, running CentOS 7 Linux kernel version 3.10. The E5-2660 v4 processor has 14 cores, each with 2 hardware threads (total of 28 SMT threads). The processor's transactional capacity (for writes) is limited by the size of the L1 data cache (32 KB) and the granularity of conflict detection is at the level of a cache line (64 bytes);
- *Power8:* This machine has a IBM Power8 Turismo SCM 3.0 GHz processor and 16 GB RAM, running Ubuntu 14.04 with linux kernel version 3.16. The Power8 processor has 4 cores, each with 8 hardware threads (total of 32 SMT threads). The processor's transactional capacity (both reads and writes) is limited by a 64-entry directory structure associated with the L2 cache (CAM) [20], and conflicts are detected at the level of a cache line (128 bytes).

In neither of these machines we made use of hyperthreading as it tends to decrease performance due to capacity aborts. Moreover, performance issues induced by the memory allocator [30], [31] were avoided by using the TCMalloc allocator with the changes suggested by Nakaike et al. [18]. Our analysis makes use of the STAMP [16] benchmark suite. STAMP consists of scientific applications from a diversity of fields such as bioinformatics (`Genome`), security (`Intruder`) and computational geometry (`Yada`). STAMP is parallelized using the Single Program Multiple Data (SPMD) style, where every thread executes the same block of code with different input. STAMP thus favors regularity and code homogeneity.

4.2 Results Using TSX

Fig. 4 shows speedup (y-axis), over sequential time, for 7 STAMP applications up to 14 threads (x-axis). The last set of bar graphs (`geomean`) is the geometric mean speedup of all applications. The `Bayes` application was not used because of its known high execution time variation [32].

In Table 1 we present the number of transitions between different modes and the percentage of time spent at each mode for RTM, PhTM, and PhTM* when executing with 14 threads. Table 1 also shows the percentage of commits at each mode when running RTM, PhTM and PhTM* also on

1. Source code of PhTM* and all evaluated TM system can be found at https://github.com/jaopaulolc/PhTM-Star

2. This threshold is actually used by all the evaluated systems and follows the recommendation of Lev et al. [14] for their PhTM system. During our tests, values ranging from 5 to 20 were used leading to quite the same results, on average.

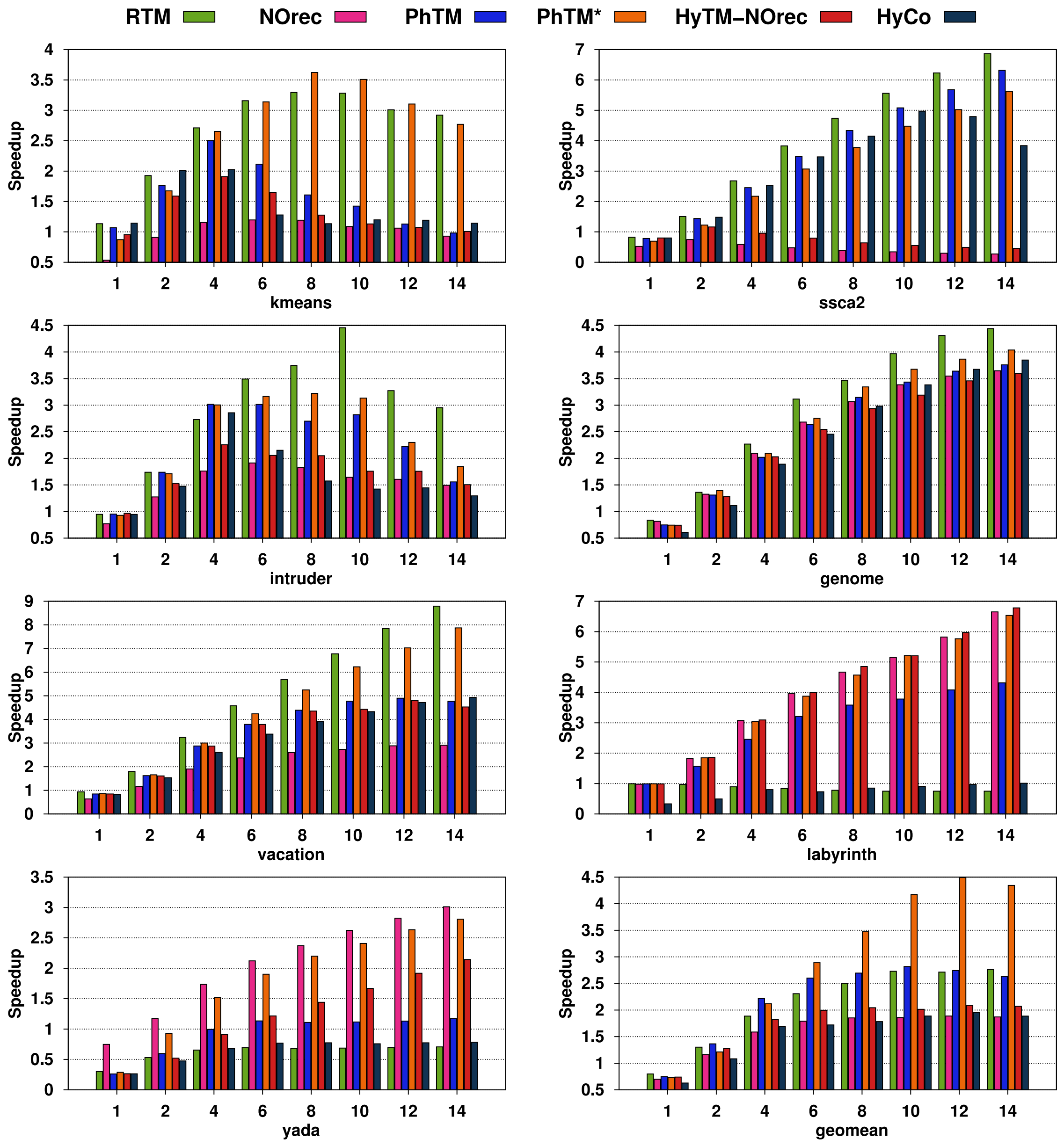


Fig. 4. Speedup results for the STAMP benchmark.

14 threads. The percentages are computed as the number of commits of a given mode over the total number of commits of the system.

The first observation one can draw from the graphs is that RTM outperforms NOrec in almost all (5 out of 7) applications. NOrec only outperforms RTM for Labyrinth and Yada. Therefore, in order to better analyze the results, we will discuss first the HTM-friendly applications (Intruder, Kmeans, SSCA2, Vacation and Genome). After that we present results explaining the remaining STM-friendly applications (Labyrinth and Yada). All the discussion in this section considers the results with 14 threads, unless explicitly stated otherwise.

Of all HTM-friendly applications, Kmeans and SSCA2 are those with the shortest transactions. Short transactions reduce the probability of data conflicts and rarely cause persistent capacity aborts, thus RTM completes most Kmeans's and SSCA2's transactions. However, when running with the software implementation, the overhead of read/write barriers is noticeable. The abort rate for SSCA2 is very low, around 0.48 percent with 14 threads, while for Kmeans it is significantly higher, around 69.93 percent. Even with such a high abort rate, RTM is able to outperform NOrec due to very low capacity aborts (less than 0.2 percent) and because conflicts of short transactions have a low probability to persist. As Table 1 shows, almost all transactions are executed in HW with RTM. Since SSCA2 has low abort rate, both PhTM and PhTM* are able to keep the transactions in HW mode and perform almost as good as RTM. Both phased approaches fall slightly short of RTM because both implementations add overhead due to duplication of code for both execution paths. PhTM* overhead is larger than PhTM due to its more sophisticated heuristic. For Kmeans, the increased abort rate causes PhTM to switch to SW much more than PhTM* (141676.5 and 34.1, respectively with 14 threads), explaining the great performance advantage of the latter. PhTM* avoids switching to SW repeatedly because the number of capacity aborts in Kmeans is very low, around

TABLE 1
Transitions, Percentage of Time and Commits in each System Mode for STAMP with 14 Threads

System	RTM					PhTM					PhTM*							
	Trans.	% of Time		% of Commits		Trans.	% of Time		% of Commits		Trans.		% of Time			% of Commits		
Application	HW→GL	HW	GL	HW	GL	HW→SW	HW	SW	HW	SW	HW→SW	HW→GL	HW	SW	GL	HW	SW	GL
`Kmeans`	605896.4	99.9	00.0	88.5	11.4	141676.5	23.4	76.5	56.0	43.9	34.1	404178.1	51.3	27.8	20.7	79.1	09.7	11.6
`SSCA2`	0	100.0	00.0	100.0	00.0	0	100.0	00.0	100.0	00.0	0	0	100.0	00.0	00.0	100.0	00.0	00.0
`Intruder`	1819652.5	46.7	53.2	94.3	05.6	476510.5	24.9	75.0	63.2	36.8	429.2	1010588.3	19.1	61.7	19.1	74.9	19.0	06.2
`Genome`	19060.3	72.0	27.9	99.2	00.7	2354	51.3	48.6	69.5	30.4	10	15023.6	59.2	12.5	28.2	82.9	16.5	00.5
`Vacation`	38496.6	86.4	13.5	99.0	00.9	12829.7	33.2	66.7	62.3	37.6	9.1	37930.2	82.6	06.9	10.4	95.9	03.1	00.9
`Labyrinth`	518.7	52.6	47.3	50.7	49.2	52.6	02.6	97.3	25.2	75.1	1	02.8	00.0	96.8	03.1	01.5	98.2	00.2
`Yada`	442611.4	66.8	33.1	82.8	17.1	124810	09.7	90.3	54.2	45.8	161.4	6129.2	01.3	94.6	04.0	01.1	98.6	00.2

0.15 percent with 14 threads. When it does switch to SW, it quickly switches back to HW because the transactions are very small. This feature shows the effectiveness of PhTM*'s more elaborated heuristic over PhTM. PhTM* performs noticeably better than RTM for `Kmeans` with 8, 10 and 12 threads. The increase on abort rate leads RTM to transition much more frequently to GLOCK than PhTM*. In addition, for those specific thread configurations PhTM* spends most of its execution time in HW, above 51.3 percent, while RTM spends the same fraction of execution time in GLOCK. In fact, RTM remains 32.9, 31.4 and 35.5 percent in HW while PhTM* spends 90.2, 77.6 and 64.5 percent, respectively, with 8, 10 and 12 threads. However, with 14 threads PhTM* spends a considerable amount of execution time in SW, explaining the observed performance degradation.

The difference between RTM and NOrec increases with the number of threads for `Intruder`, `Vacation` and `Genome`. Amongst these three applications, `Genome` has the lowest abort rate, around 8.94 percent with 14 threads, but the majority of the aborts are capacity ones. The occurrence of capacity aborts explain the performance gap between RTM and PhTM*. These aborts, although spurious, leads PhTM* to stay over 12.5 percent in SW instead of retrying either on HW or GLOCK. Nonetheless, the few times PhTM* switches to SW (around 10 on average with 14 threads) it is able to go back to HW where it spends most of its time, about 59.2 percent. The lack of a serialization mode causes much frequent mode changes in PhTM (around 2354) and the total amount of time it stays in SW mode is 48.6 percent, explaining why PhTM* performs better. The same behavior explains why PhTM* is slightly better than PhTM in `Intruder` with 14 threads. Although the abort rate is higher in `Intruder` (about 57.72 percent), PhTM* spends less time in SW, about 61.71, against PhTM's 75.07 percent. PhTM* starts spending more time in SW than in HW mode beyond the 8-thread, explaining the performance degradation. The increase on conflict aborts triggers more HW→SW transitions. Nonetheless, PhTM* still performs better than PhTM.

`Vacation` has a moderate abort rate compared with the other STAMP applications, revolving around 20 percent. Two thirds of these aborts are due to capacity reasons, explaining the migrations to SW in PhTM and the total of 66.7 percent of the execution time spent in this mode. This is in contrast with PhTM*, which spends little time in SW (less than 6.9 percent). In addition, the number of migrations back and forth to SW in PhTM is very high, causing extra overhead (due to the SW→HW barrier) and thus allows PhTM* to perform increasingly better beyond the 8-thread boundary. Similar to `Intruder`, capacity aborts become more frequent beyond 8 threads in `Vacation`. However, PhTM* rarely switches to SW due to `Vacation`'s lower abort rate. RTM outperforms NOrec for `Intruder`, `Genome` and `Vacation` due to the contention on the software sequence lock. Table 1 reveals that PhTM* not only spends most of its time in HW, but also that HW is in fact very effective. In particular for `SSCA2` and `Vacation`, the system stays in HW 100 and 82.6 percent while commiting the majority of transactions, respectively, 100 and 95.9 percent.

For the remaining two applications, `Labyrinth` and `Yada`, NOrec performs noticeably better than RTM. These applications, particularly `Labyrinth`, have very long running transactions. As all accesses inside a hardware transaction are versioned, RTM's transactions are more vulnerable to capacity and conflict aborts than NOrec, which uses the manually instrumented accesses provided as STAMP annotations. As a consequence, hardware transactions in RTM exhibit an abort rate over 90 percent for `Labyrinth`, most being capacity aborts, and over 67 percent for `Yada`, from which 63 percent are due do conflicts and the remaining 36 percent due to capacity. As a consequence, a significant percentage of transactions in both applications are serialized, as Table 1 shows. With NOrec, the abort rate is close to 25 and 11 percent for `Yada` and `Labyrinth`, respectively. Again, Table 1 makes it clear that that SW is the most effective system for both `Labyrinth` and `Yada`.

PhTM*'s heuristic quickly (after about 8 serializations) switches to SW and never switches back, given the long transactions in `Labyrinth`, allowing it to perform very close to NOrec for this application. Although Table 1 shows a percentage of 97.3 percent in SW for PhTM, against PhTM*'s 96.8 percent, it should be noticed that PhTM keeps migrating between modes much more frequently and the overhead to switch from SW to HW (the cost of waiting all software transactions to finish) is showed as part of SW. After instrumenting the code, we found out that this cost is up to 30 percent of the total time spent in SW, explaining why PhTM* was better than PhTM. For `Yada` something similar happens, except that `Yada` also presents a lot of conflict aborts. This causes PhTM* to switch between modes much more frequently, but still a lot less than PhTM (161.4 versus 124810). As a result, the penalty for switching from

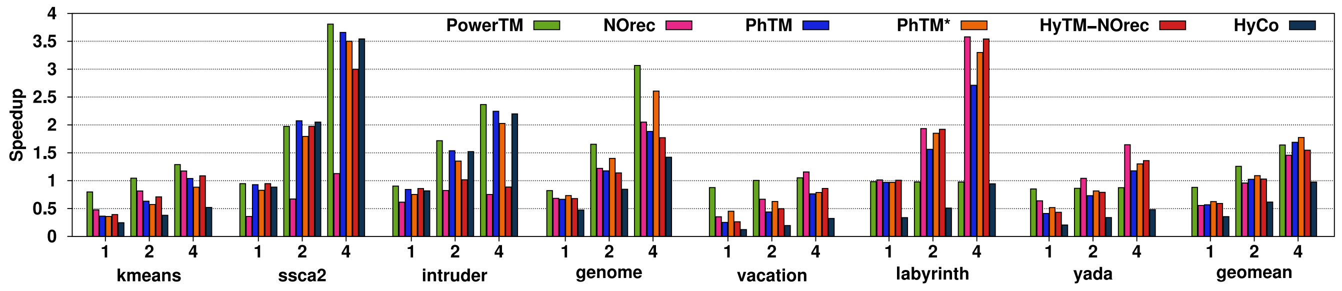


Fig. 5. Speedup results for the STAMP applications on the Power8 machine.

TABLE 2
Transitions, Percentage of Time and Commits in each System Mode for STAMP with 4 Threads

System	PowerTM					PhTM					PhTM*							
	Trans.	% of Time		% of Commits		Trans.	% of Time		% of Commits		Trans.		% of Time			% of Commits		
Application	HW→GL	HW	GL	HW	GL	HW→SW	HW	SW	HW	SW	HW→SW	HW→GL	HW	SW	GL	HW	SW	GL
Kmeans	880.3	99.9	00.0	99.9	00.0	5.8	99.9	00.0	100.0	00.0	0	5.8	100.0	00.0	00.0	99.9	00.0	00.0
SSCA2	1	100.0	00.0	100.0	00.0	0	100.0	00.00	100.0	00.0	0	0	100.0	00.0	00.0	100.0	00.0	00.0
Intruder	5757525.4	50.1	49.8	74.4	24.5	1050867.1	32.3	67.6	63.0	36.9	24.2	4632403.9	63.2	00.5	36.1	79.5	00.4	20.1
Genome	54733	59.0	40.9	99.2	00.7	14088.3	56.9	43.0	91.2	08.7	0	52463.2	89.4	00.0	10.5	99.4	00.0	00.5
Vacation	4100667.8	50.3	49.6	02.3	97.6	754836.7	15.6	84.3	01.9	98.0	36.8	4052586.6	38.0	00.5	61.4	02.6	00.5	96.7
Labyrinth	514.4	49.0	50.9	50.1	49.9	90.4	01.4	98.5	34.3	65.7	1	1	00.0	98.6	01.3	00.6	99.2	00.1
Yada	468711.4	51.0	48.9	82.0	17.9	102626.1	15.0	84.9	54.1	45.8	370.4	94189.2	07.9	63.8	28.1	14.2	82.7	03.0

SW to HW due to the barrier is only 2.33 percent of the time spent in SW, against a cost of 37.6 percent for PhTM. This is a strong evidence that the serialization mode used by PhTM* is efficient in minimizing the barrier cost in situations that would involve a lot of migrations.

As Fig. 4 also shows, both PhTM* and HyTM-NOrec achieved close speedups in 2 out of the 7 applications: Labyrinth and Yada. In fact, Labyrinth was the only application to effectively benefit from HyTM-NOrec's ability to run transactions in different modes simultaneously. However PhTM* outperforms HyTM-NOrec for Yada, showing the effectiveness of PhTM*'s heuristic to quickly detect the best running mode. HyCO only achieves speedups close to PhTM*'s for SSCA2 and Genome. Nonetheless, RTM is followed much closely by PhTM*. Since PhTM* is able to follow the best execution mode for each application, it presented the best overall results for STAMP among all studied TM systems, with a 1.64x gain over PhTM, 2.08x over HyTM-NOrec and 2.28x over HyCO considering the geometric mean for the 14 threads execution. Both hybrid systems evaluated (HyTM-NOrec and HyCO) performed worse overall than both phased approaches (PhTM and PhTM*). The main causes are discussed in great detail later on this work (4.4 and 4.5).

4.3 Results Using Power8

Our analysis of the results for the Power8 machine follows the same structure of TSX's, in which Fig. 5 shows the speedup numbers and Table 2 the transitions and percentage of time spent in each execution mode for PowerTM, PhTM and PhTM*. Table 2 also shows the percentage of commits at each mode when running PowerTM, PhTM and PhTM* with 4 threads. Here again the percentages are computed as the number of commits of a given mode over the total number of system commits.

Similarly to the results on *Broadwell*, there is a group of applications which performs better with RTM (Kmeans, SSCA2, Intruder and Genome) and another which performs better with NOrec (Labyrinth and Yada). Differently of what was observed on Broadwell, but as also noted by Nakaike et al. [18], both Intruder and Vacation did not scale with PowerTM. Indeed, Intruder and Vacation exhibit a capacity abort rate over 88 and 99 percent, respectively, with 4 threads. We discovered that such applications also do not scale with NOrec due to the fact that both Intruder's and Vacation's transactions are not large enough to mitigate the overhead of starting, committing and aborting transactions in SW. We also noticed that this overhead is higher on Power8 than it is on Broadwell and we believe that this is due to the memory model of each processor. PowerPC's memory model is more relaxed than x86's, therefore explicit CPU fences are necessary to guarantee that reads and writes are not reordered with respect to NOrec's sequence lock operations. Such fences are not present in the x86's version of NOrec because the processor's memory model (TSO) guarantees the expected ordering.

In the first group, Kmeans and SSCA2 have very short transactions, explaining why both performed best with PowerTM. NOrec's higher overhead to start, commit and abort transactions is noticeable and severely limited the speedup of these applications. As Table 2 shows, both PhTM and PhTM* are able to stay most of their time in HW, thus performing as good as PowerTM (4 threads). This happens because the number of transitions in both PhTM and PhTM* are much smaller than in PowerTM. Despite of the less frequent mode transitions of PhTM*, PhTM still performs slightly better. Genome, differently of Kmeans and SSCA2, has small to medium-sized transactions and scales with NOrec. However, the SW overhead is still quite

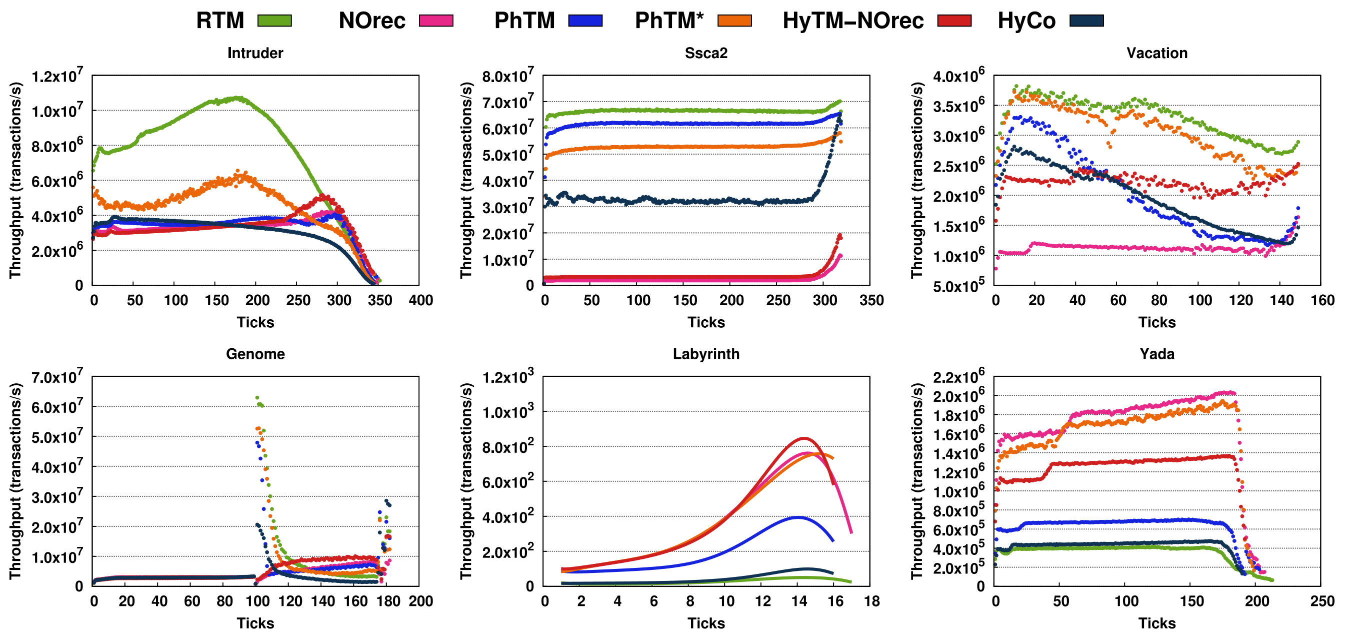


Fig. 6. Commit-rate over time of all STAMP applications running on the *Broadwell* machine.

noticeable. PowerTM's transactional capacity is large enough to hold most `Genome`'s transactions (up to 4 threads) and, as it adds negligible transactional overhead, it performs better than NOrec. PhTM*, by employing a more elaborated heuristic, does not transition to SW as PhTM does on `Genome` and thus is able to run most of the time in HW. The high frequency of mode transitions in PhTM makes the threads waste about 13 percent of `Genome`'s execution time waiting on the SW→HW barrier and causes more HW aborts due to writes on `modeIndicator`.

NOrec performs noticeably better than PowerTM in the two remaining applications, `Labyrinth` and `Yada`. Both applications have very long transactions, `Labyrinth` particularly. PowerTM, similarly to RTM, implements implicit transactions, meaning that all accesses inside a transaction are versioned which increases the likelihood of both capacity and conflict aborts. Indeed, over 54 percent of `Labyrinth`'s transactions are aborted in PowerTM, most of them for exceeding transactional capacity. As a consequence, PowerTM serializes, respectively, 50.9 and 48.9 percent of `Labyrinth`'s and `Yada`'s transactions, as Table 2 shows.

As also observed on TSX, PhTM*'s heuristic quickly switches to SW in `Labyrinth` and never switches back, explaining why its performance is very close to NOrec. Despite Table 2 showing a close percentage of time in SW for PhTM and PhTM*, threads in PhTM spent over 26 percent of `Labyrinth`'s total execution time waiting on the barrier against 0 percent in PhTM*. Notice that the amount of time waiting on the SW→HW barrier also counts as SW mode time. Neither PhTM nor PhTM* were able to scale well with `Yada`. The former due to the high frequency of HW→SW transitions and the wastage of time waiting on the SW→HW barrier (over 23 percent). The latter because about half of `Yada`'s transactions did not had an abort rate high enough to trigger a HW→SW transitions, explaining why PhTM* spent around 28.1 percent of the time in GLOCK mode.

Fig. 5 also shows that both PhTM* and HyTM-NOrec achieved close speedups in 2 out of 7 applications (`Labyrinth` and `Yada`). Moreover, PhTM* outperformed HyTM-NOrec in 3 out of 7 (`SSCA2`, `Intruder` and `Genome`). PhTM* also performed as good as HyCO in 2 out of 7 applications (`SSCA2` and `Intruder`). However, on the 5 remaining applications HyCO fell significantly behind due to its inherent limitations discussed further on Section 4.5. `Labyrinth` was the only application in which HyTM-NOrec was significantly faster than PhTM* (less than 8 percent). This result shows that a simpler design choice to execute transactions in phases, such as PhTM*, can lead to the same, or even better performance than a more complex hybrid system such as HyTM-NOrec. Confirming the results obtained with TSX, PhTM* presented the best results overall for STAMP among all studied TM systems, with a 1.18x gain over PhTM, 1.36x over HyTM-NOrec and 1.81x over HyCO considering the geometric mean with 4 threads.

4.4 Behavior of STAMP Applications

In order to understand the dynamic behavior of STAMP applications we sampled their commit-rate over time. Figs. 6 and 7 shows the throughput (y-axis), number of committed transactions per second, over the execution time of each application (x-axis) when executed on *Broadwell* and *POWER8* with 14 and 4 threads, respectively. Notice that each point in the plot is actually the *instantaneous* commit-rate calculated as the average number of commits of the last `n` transactions. The value of `n` was chosen based on the size and number of transactions. Applications with medium sized transactions were executed with $n = 1K$ (`Genome` and `Yada`), many transactions with $n = 5K$ (`Intruder` and `SSCA2`), large but few transactions with $n = 5$ (`Labyrinth`) and short but many transactions with $n = 2K$ (`Vacation`). We omit the results for `Kmeans` due to the high variation on the completion time (of individual transactions) induced by its very small transactions composed of only a couple of memory updates. As each point is ordered, the plots show how the commit-rate changes over time. As in the previous section, first we present the HTM-friendly applications

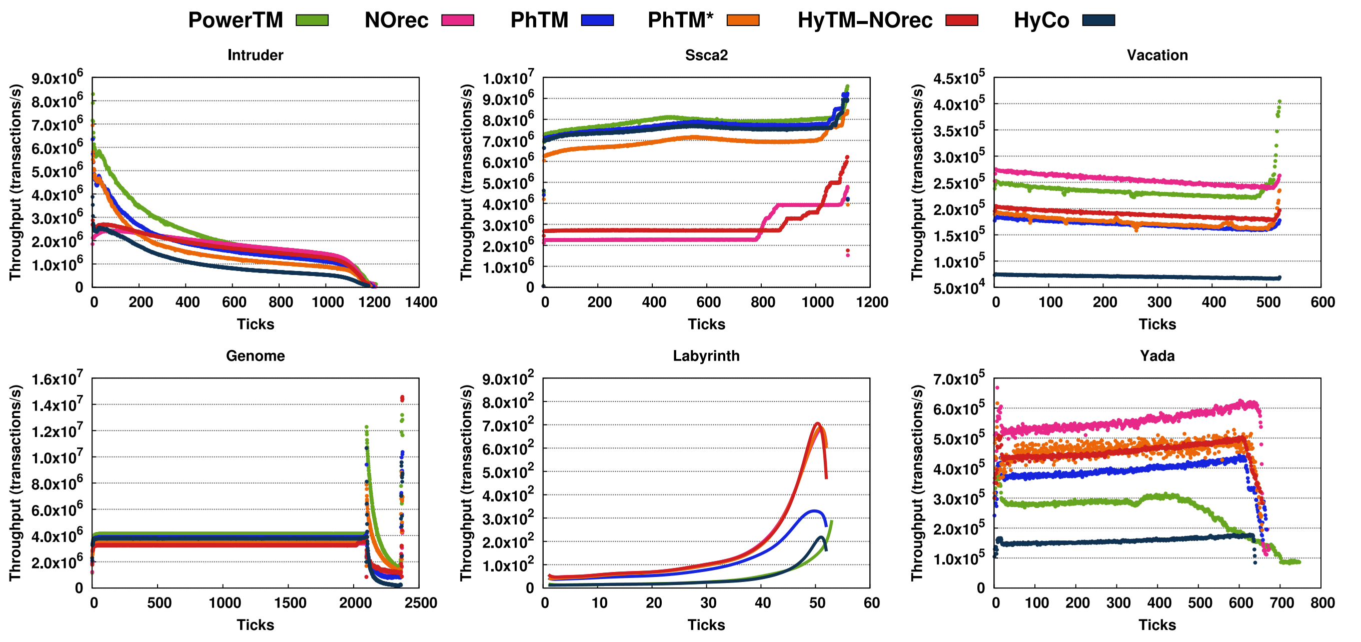


Fig. 7. Commit-rate over time of all STAMP applications running on the *POWER8* machine.

(`Kmeans`, `SSCA2`, `Intruder`, `Genome` and `Vacation`) and later those that performed better with STM (`Labyrinth` and `Yada`) as Section 4.2 shows.

Fig. 6 makes it clear that, although the commit-rate changes over time, the best performing system remains the same throughout the whole execution. This result shows, for the first time, that STAMP applications do not exhibit hybrid behavior. As a result, a phase-based TM system is better suited than a hybrid alternative because there is no point in time in which any two transactions will diverge on the best performing execution mode. The SPMD nature of STAMP applications combined with the fact that a single transaction dominates execution time [33] also supports our claim. Fig. 6 also shows how effective PhTM* is to both detect and remain on the best mode. This mono-phase characteristic lends STAMP a poor candidate to evaluate hybrid systems as it will not stress the scenario where two or more transactions are required to execute on different modes.

`Genome` is the only exceptional case with a more noticeable commit-rate change. After the instant 100, HW is able to complete transactions at a higher rate than SW. Even with this abrupt change PhTM* was capable of quickly switching and remaing in HW. Around instant 140, SW becomes the best running mode and again PhTM* followed SW closely. The results of `Intruder` reveal that PhTM* and HyTM-NOrec were not able to closely follow the best mode (HW), showing that there is still space for improving the state transition heuristics. The plot of `Labyrinth` shows significantly fewer points than the other applications because `Labyrinth` has to execute only about 500 transactions in order to terminate while the others must execute hundreds or even thousands of thousands (e.g., `Intruder` and `Yada`).

Confirming the results obtained with *Broadwell*, Fig. 7 shows that STAMP applications do not exhibit hybrid behavior on *POWER8* either. Such consistent results, obtained with two different architectures, are strong evidences that the behavior depicted on both Figs. 6 and 7 are indeed intrinsic of STAMP and not induced by the systems characteristics. Therefore, STAMP is a poor candidate to evaluate hybrid transactional systems on both *Haswell* and *POWER8* platforms.

4.5 Performance in the Presence of Phases and Hybrid Behavior

In the previous Section 4.4 we have showed that STAMP, the *de facto* standard in TM systems evaluation, do no exhibit sufficient hybrid behavior to stress dynamic mode transition heuristics. Therefore, in order to properly evaluate PhTM* and all previous transactional systems (See Section 4.1) we adapted the `Intset` microbenchmark, from the TinySTM code base [34], to enable it to execute phases composed of operations over two of its data structures, a linked-list (LL) and a red-back tree (RB). Intset aims to measure how effective a given synchronization mechanism implementation is under different scenarios [35]. The microbenchmark performs a stress test in a sorted integer set by randomly alternating between its search, insert and delete operations. Therefore, a configuration with a 20 percent update rate means that 10 percent of all operations are insertions, 10 percent are deletions and the remaining 80 percent are lookups. Notice that each of these data structures has its own intrinsic characteristics, such as memory footprint for each operation, which limits HW and SW performance in different ways. In general, RB operations are usually HW-Friendly, while LL operations tend to be more SW-Friendly. As a result, the microbenchmark can emulate phase changes by simply switching between these two data structures.

Fig. 8 shows the throughput (y-axis), committed transactions per second, over time (x-axis) for a 3-phase configuration of our new microbenchmark: two LL phases alternated by a RB phase, both with 20 percent of all operations being updates. Both data structures were initialized with 4K elements. Each data point is the average number of commits for the last 1K transactions. In all of them PhTM* is capable to effectively detect and remain in the best runing mode. Clearly neither HyTM-NOrec or HyCO were able to keep up

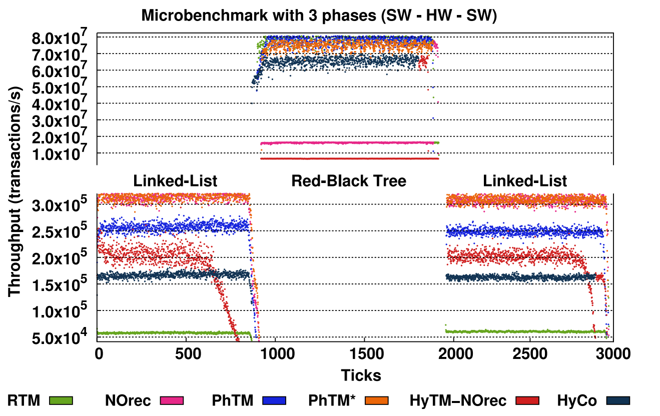


Fig. 8. Commit-rate over time of microbenchmark running on *Broadwell* with 14 threads.

with PhTM or PhTM*. In fact, HyTM-NOrec performed worse than pure SW for the RB 20 percent updates phase. HyCO performed better than HyTM-NOrec, however much worse than both PhTM and PhTM* for the LL 20 percent phase. Both hybrid systems must fail 10 times in HW before switching to SW. Once in SW, as soon as the transactions commit, they will start again in HW, even though SW is the best suitable execution mode. This result shows, for the first time, the inability of conventional hybrid systems to dynamically adapt to program phase changes.

The previous results showed that conventional hybrid systems are unable to cope with phases. However they were designed for hybrid-behaved applications, those which execute both HW-Friendly and SW-Friendly transactions concurrently. Since we failed to find applications with such characteristics, we implemented a RB-tree forest to evaluate the ability of conventional hybrid systems to execute both SW and HW concurrently. In this microbenchmark, each forest operation consists of insertions, deletions and queries on a number of trees in the forest. The number of trees is randomly chosen individually by each thread before performing an operation. As a result, transactions with different memory footprints, and therefore, different SW and HW friendliness, will execute concurrently.

Fig. 9 shows the throughput of each transactional system (y-axis) for each number of threads (x-axis). We show the results for a forest with 1, 20, 45 and 90 trees, respectively from top to the bottom graph. Each graph shows groups of bars for the results with 20 and 60 percent updates, respectively from left to right. As can be seen, HyCO was only able to perform close to both PhTM and PhTM* with a forest of a single tree. Once the number of trees in the forest increase and threads start to modify the forest, PhTM* was the only system that scaled with the number of threads. The main reason for that is, although both HyTM-NOrec and HyCO are able to execute SW and HW-Friendly concurrently, neither of them can effectively commit transactions in SW and HW simultaneously.

Even though HyCO enables HW commits, transactions that failed due to capacity aborts will most likely abort during

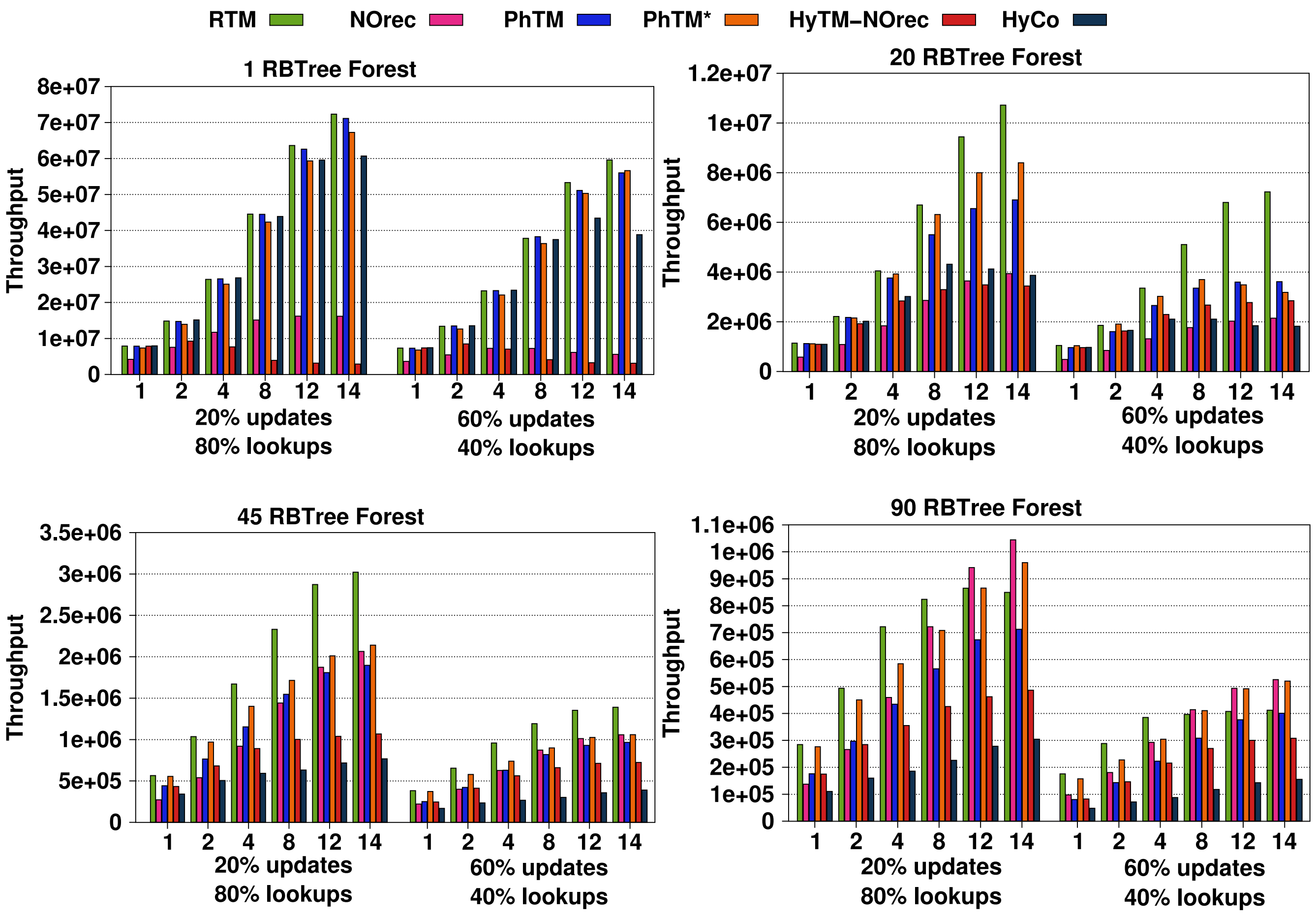


Fig. 9. Throughput of each transactional system for the hybrid microbenchmark running on *Broadwell*.

hardware-assisted writeback (HC state). Once HyCO's transactions start to commit in SW, all HW transactions are aborted via `stx_kill` flag. In addition, HW transactions can only start after SW commit ends. HyTM-NOrec exhibits a similar limitation. Every HW commit triggers a SW read-set revalidation via the increment of a hardware commit counter. Besides that, every SW commit aborts all HW transactions due to the eager subscription to the sequence lock. The aforementioned characteristics are inherent limitations of conventional hybrid systems, as also discussed by Shavit et al. [17].

4.6 Challenges and Limitations of Phased TM Systems

The previous results demonstrated that PhTM* is a viable and competitive alternative to standard hybrid transactional systems. As Lev et al. pointed out [14], there are three main challenges in implementing phase-based systems: (i) identifying efficient modes for various scenarios; (ii) managing correct (and fast) transitions between modes; and (iii) deciding when to switch modes. PhTM* illustrates a possible way to effectively deal with the first two. We identified three efficient modes: full concurrent HW, serial (GLOCK) and SW modes. The transitions are usually very fast, but returning to HW from SW may become an issue because a barrier is needed to wait for the software transactions to finish. The problem of deciding when to switch modes turned out to be the most challenging one.

The core idea of PhTM* is to avoid switching to software mode too early and use it as a last resort. Full hardware mode is the default and the preferred one. If transactions cannot complete concurrently in HW, serial mode is used first. However, if the abort rate is still too high and hardware resources are exhausted (indicated by two consecutive capacity aborts), PhTM* migrates to SW. When to return to HW once in SW is the most difficult task. On the one hand, it should not return too quickly because it might have to pay the cost of the barrier. Moreover, it may also cause a "ping-pong" effect, where the system keeps transitioning between HW and SW continuously. On the other hand, if too much time is spent in SW it might not benefit from the fast execution provided by running transactions without any instrumentation in HW. The decision made by PhTM* is to only return when: (1) the transactions aborted in HW have been committed in SW; and (2) the transaction length is below a given threshold. Therefore, if transactions are still very long, PhTM* assumes that they still will not be able to complete in HW, and therefore stays in SW.

The drawback of the heuristic proposed by PhTM* is that several thresholds need to be tuned: the abort rate threshold (`ABORT_THRSD`) and its corresponding α, the minimum transaction length (`SIZE_THRSD`), and the minimum number of times a transaction needs to run in software before its length is averaged. We set these values after a performance analysis conducted with the STAMP benchmarks. The results showed in the last two sections indicate that they provided very good results. For instance, it is very clear that PhTM*'s heuristic avoided a lot of unnecessary transitions to software for all applications when compared to PhTM (see Tables 1 and 2). Avoiding the "ping-pong" effect was crucial for the observed performance gains. Even though we have tuned the parameters using *Broadwell* as the baseline, the good results carried over to *POWER8*, showing the robustness of the heuristic proposed by PhTM*.

The strongest argument against phase-based transactional systems is the alleged performance impact caused by a single transaction requiring execution in a slower mode (e.g., software) [9], [10]. Our results show that the STAMP applications do not exhibit frequent scenarios wherein a single long transaction requires PhTM* switching to a software mode. This is mostly due to the way these applications were constructed (i.e., SPMD style), which does not favor transaction heterogeneity. Therefore, most applications have transactions with well-defined capacity utilization behavior and a simpler phase-based approach such as PhTM* offers a competitive solution when compared to a more complicated HyTM approach.

5 Conclusion

In this article we made a case for phased transactional memory by providing a new implementation, PhTM*, and showing that it is competitive with current hybrid systems using both Broadwell and Power8 processors. Our results show that for the STAMP benchmarks, a simpler transactional system based on phases can achieve better overall performance than state-of-the-art hybrid systems while providing a simpler and more flexible implementation. It also shows that STAMP applications do not exhibit hybrid behavior to justify the use of conventional hybrid systems (HyTM-NOrec and HyCO), therefore making PhTM* a better suited solution. And for the first time we show that conventional hybrid systems do not perform better than phased-based system in a scenario with hybrid-behaved transactions.

Acknowledgments

The authors would like to thank the anonymous reviewers for their insightful comments. This work was supported by FAPESP (grant 2016/15337-9), Center for Computational Engineering and Sciences (CCES), and CNPq (grant 446160/2014-8).

References

[1] T. Harris, J. Larus, and R. Rajwar, *Transactional Memory*, 2nd ed. San Rafael, CA, USA: Morgan & Claypool Publishers, Jun. 2010.
[2] M. Herlihy and J. E. B. Moss, "Transactional memory: Architectural support for lock-free data structures," in *Proc. 20th Annu. Int. Symp. Comput. Archit.*, Jun. 1993, pp. 289–300.
[3] A. Wang, M. Gaudet, P. Wu, J. N. Amaral, M. Ohmacht, C. Barton, R. Silvera, and M. Michael, "Evaluation of Blue Gene/Q hardware support for transactional memories," in *Proc. 21st Int. Conf. Parallel Architectures Compilation Tech.*, Sep. 2012, pp. 127–136.
[4] C. Jacobi, T. Slegel, and D. Greiner, "Transactional memory architecture and implementation for IBM system Z," in *Proc. 2012 45th Annu. IEEE/ACM Int. Symp. Microarchitecture*, Vancouver, B.C., CANADA, 2012, pp. 25–36, doi: 10.1109/MICRO.2012.12.
[5] *Intel® Architecture Instruction Set Extensions Programming Reference*, Intel Corporation, Feb. 2012.
[6] S. Kumar, M. Chu, C. J. Hughes, P. Kundu, and A. Nguyen, "Hybrid transactional memory," in *Proc. 11th Symp. Principles Practice Parallel Program.*, Mar. 2006, pp. 209–220.
[7] P. Damron, A. Fedorova, and Y. Lev, "Hybrid transactional memory," in *Proc. 12th Int. Conf. Architectural Support Program. Lang. Operating Syst.*, Oct. 2006, pp. 336–346.
[8] L. Dalessandro, F. Carouge, S. White, Y. Lev, M. Moir, M. L. Scott, and M. F. Spear, "Hybrid NOrec: A case study in the effectiveness of best effort hardware transactional memory," in *Proc. 16th Int. Conf. Architectural Support Program. Lang. Operating Syst.*, Mar. 2011, pp. 39–52.

[9] T. Riegel, P. Marlier, M. Nowack, P. Felber, and C. Fetzer, "Optimizing hybrid transactional memory: The importance of nonspeculative operations," in *Proc. 23rd Annu. ACM Symp. Parallel Algorithms Architectures*, Jun. 2011, pp. 53–64.

[10] A. Matveev and N. Shavit, "Reduced hardware transactions: A new approach to hybrid transactional memory," in *Proc. 25th Annu. ACM Symp. Parallel Algorithms Architectures*, Jul. 2013, pp. 11–22.

[11] A. Matveev and N. Shavit, "Reduced hardware NOrec: A safe and scalable hybrid transactional memory," in *Proc. 20th Int. Conf. Architectural Support Program. Lang. Operating Syst.*, Mar. 2015, pp. 59–71.

[12] I. Calciu, J. E. Gottschlich, T. Shpeisman, G. A. Pokam, and M. Herlihy, "Invyswell: A hybrid transactional memory for haswell's restricted transactional memory," in *Proc. 23rd Int. Conf. Parallel Architectures Compilation Tech.*, Aug. 2014, pp. 187–200.

[13] W. Ruan and M. Spear, "Hybrid transactional memory revisited," in *Proc. Int. Symp. Distrib. Comput.*, 2015, pp. 215–231.

[14] Y. Lev, M. Moir, and D. Nussbaum, "PhTM: Phased transactional memory," in *Proc. 2nd ACM SIGPLAN Workshop Transactional Comput.*, Aug. 2007.

[15] J. P. L. de Carvalho, G. Araujo, and A. Baldassin, "Revisiting phased transactional memory," in *Proc. Int. Conf. Supercomputing*, 2017, pp. 25:1–25:10. [Online]. Available: http://doi.acm.org/10.1145/3079079.3079094

[16] C. C. Minh, J. Chung, C. Kozyrakis, and K. Olukotun, "STAMP: Stanford transactional applications for multi-processing," in *Proc. IEEE Int. Symp. Workload Characterization*, Sep. 2008, pp. 35–46.

[17] D. Alistarh, J. Kopinsky, P. Kuznetsov, S. Ravi, and N. Shavit, "Inherent limitations of hybrid transactional memory," in *Proc. Int. Symp. Distrib. Comput.*, 2015, pp. 185–199.

[18] T. Nakaike, R. Odaira, M. Gaudet, M. M. Michael, and H. Tomari, "Quantitative comparison of hardware transactional memory for blue gene/q, zenterprise ec12, intel core, and POWER8," in *Proc. 42nd Annu. Int. Symp. Comput. Archit.*, 2015, pp. 144–157.

[19] R. M. Yoo, C. J. Hughes, K. Lai, and R. Rajwar, "Performance evaluation of Intel transactional synchronization extensions for high-performance computing," in *Proc. Int. Conf. High Perform. Comput. Netw. Storage Anal.*, 2013, pp. 19:1–19:11.

[20] 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 J. Res. Develop.*, vol. 59, no. 1, pp. 8–1, 2015.

[21] L. Dalessandro, M. F. Spear, and M. L. Scott, "NOrec: Streamlining STM by abolishing ownership records," in *Proc. 15th Symp. Principles Practice Parallel Program.*, Jan. 2010, pp. 67–78.

[22] J. Chung, L. Yen, S. Diestelhorst, M. Pohlack, M. Hohmuth, D. Christie, and D. Grossman, "ASF: AMD64 extension for lock-free data structures and transactional memory," in *Proc. 43rd ACM/IEEE Int. Symp. Microarchitecture*, Dec. 2010, pp. 39–50.

[23] T. Riegel, P. Felber, and C. Fetzer, "A lazy snapshot algorithm with eager validation," in *Proc. 20th Int. Symp. Distrib. Comput.*, Sep. 2006, pp. 284–298.

[24] J. E. Gottschlich, M. Vachharajani, and J. G. Siek, "An efficient software transactional memory using commit-time invalidation," in *Proc. Int. Symp. Code Generation Optimization*, Apr. 2010, pp. 101–110.

[25] Y. Xiao, T. Jeyakumaran, E. Atoofian, and A. Jannesari, "Improving Performance of Transactional Memory Through Machine Learning," *Concurrency Computat.: Practice Experience*, vol. 30, 2017, Art. no. e4397.

[26] R. M. Yoo and H.-H. S. Lee, "Adaptive transaction scheduling for transactional memory systems," in *Proc. 20th Annu. ACM Symp. Parallel Algorithms Archit.*, Jun. 2008, pp. 169–178.

[27] B. Goel, R. Titos-Gil, A. Negi, S. A. McKee, and P. Stenstrom, "Performance and energy analysis of the restricted transactional memory implementation on haswell," in *Proc. IEEE 28th Int. Parallel Distrib. Process. Symp.*, 2014, pp. 615–624.

[28] D. Dice, Y. Lev, M. Moir, and D. Nussbaum, "Early experience with a commercial hardware transactional memory implementation," in *Proc. 14th Int. Conf. Architectural Support Program. Lang. Operating Syst.*, Mar. 2009, pp. 157–168.

[29] V. J. Marathe, M. F. Spear, C. Heriot, A. Acharya, D. Eisenstat, W. N. Scherer, and M. L. Scott, "Lowering the overhead of nonblocking software transactional memory," in *Proc. 1st ACM SIGPLAN Workshop Lang. Compilers, Hardware Support Transactional Comput.*, Jun. 2006.

[30] A. Baldassin, E. Borin, and G. Araujo, "Performance implications of dynamic memory allocators on transactional memory systems," in *Proc. 20th Symp. Principles Practice Parallel Program.*, Feb. 2015, pp. 87–96.

[31] D. Dice, T. Harris, A. Kogan, and Y. Lev, "The influence of malloc placement on TSX hardware transactional memory," *CoRR*, vol. abs/1504.04640, 2015. [Online]. Available: http://arxiv.org/abs/1504.04640

[32] D. Christie, J.-W. Chung, S. Diestelhorst, M. Hohmuth, M. Pohlack, C. Fetzer, M. Nowack, T. Riegel, P. Felber, P. Marlier, and E. Riviere, "Evaluation of AMD's advanced synchronization facility within a complete transactional memory stack," in *Proc. 5th Eur. Conf. Comput. Syst.*, Apr. 2010, pp. 27–40.

[33] J. P. L. de Carvalho, R. P. Murari, and A. Baldassin, "Reacessing sTAMP applications on a new transactional hardware," in *Proc. Workshop High-Perform. Comput. Syst.*, Oct. 2015.

[34] P. Felber, C. Fetzer, P. Marlier, and T. Riegel, "Time-based software transactional memory," *IEEE Trans. Parallel Distrib. Syst.*, vol. 21, no. 12, pp. 1793–1807, Dec. 2010.

[35] M. Herlihy, V. Luchangco, M. Moir, and W. N. Scherer, "Software transactional memory for dynamic-sized data structures," in *Proc. 22nd Annu. Symp. Principles Distrib. Comput.*, Jul. 2003, pp. 92–101.

João P. L. de Carvalho received the master's degree in computer science from Universidade Estadual Paulista (UNESP), Brazil, in 2016. He is currently working toward the PhD degree at the University of Campinas (UNICAMP). His research interests include concurrent programming, high performance computing, and compiler-aided profiling and parallelization.

Guido Araujo received the PhD degree in electrical engineering from Princeton University, in 1997. He is a full professor of computer science and engineering with UNICAMP. His current research interests include code optimization, parallelizing compilers, transactional memory and cloud computing, which are explored in close cooperation with industry partners.

Alexandro Baldassin received the PhD degree in computer science from the University of Campinas (UNICAMP), Brazil, in 2009. Since 2010, he has served as an assistant professor with the Department of Statistics, Applied Mathematics and Computation (DEMAC), Sao Paulo State University (UNESP), Brazil. His main research interests include computer architecture, multicore processors, and parallel programming.

▷ **For more information on this or any other computing topic, please visit our Digital Library at** www.computer.org/publications/dlib.