NV-PhTM: An Efficient Phase-Based Transactional System for Non-volatile Memory

Alexandro Baldassin[1(✉)], Rafael Murari[1], João P. L. de Carvalho[2], Guido Araujo[2], Daniel Castro[3], João Barreto[3], and Paolo Romano[3]

[1] UNESP – Univ Estadual Paulista, São Paulo, Brazil
{alexandro.baldassin,rafael.murari}@unesp.br
[2] UNICAMP – Institute of Computing, São Paulo, Brazil
{joao.carvalho,guido}@ic.unicamp.br
[3] INESC-ID & Instituto Superior Técnico, University of Lisbon, Lisbon, Portugal
daniel.castro@ist.utl.pt, joao.barreto@tecnico.ulisboa.pt, romano@inesc-id.pt

Abstract. Non-Volatile Memory (NVM) is an emerging memory technology aimed to eliminate the gap between main memory and stable storage. Nevertheless, today's programs will not readily benefit from NVM because crash failures may render the program in an unrecoverable and inconsistent state. In this context, *durable transactions* have been proposed as a mechanism to ease the adoption of NVM by simplifying the task of programming NVM systems. Existing systems employ either hardware (HW) or software (SW) transactions with different performance tradeoffs. Although SW transactions are flexible and unbounded, they may significantly hurt the performance of short-lived transactions. On the other hand, HW transactional memories provide low-overhead but are resource-constrained. In this paper we present NV-PhTM, a transactional system for NVM that delivers the best out of both HW and SW transactions by dynamically selecting the best execution mode according to the application's characteristics. NV-PhTM is comprised of a set of heuristics to guide online phase transition while retaining persistency in case of crashes during migration. To the best of our knowledge, NV-PhTM is the first phase-based system to provide durable transactions. Experimental results with the STAMP benchmark show that the proposed heuristics are efficient in guiding phase transitions with low overhead.

Keywords: Transactions · Transactional memory · Persistent memory

1 Introduction

Recent Non-Volatile Memory (NVM) technologies can provide persistency, fast access time and a byte-addressable interface. As NVM's access latency is approaching those of current DRAM technology, its content can be directly read

© Springer Nature Switzerland AG 2020
M. Malawski and K. Rzadca (Eds.): Euro-Par 2020, LNCS 12247, pp. 477–492, 2020.
https://doi.org/10.1007/978-3-030-57675-2_30

or written by the CPU, thus avoiding the overhead involved in block-oriented systems. However, it is challenging to write code for NVM because a system crash may render the program in an unrecoverable state. Durable transactions have been suggested as an appropriate way of programming these systems given their consolidated strong semantics and ease-of-use idiom [4,15].

Most of early works focused on providing durable transactions by carefully extending software transactional memory (STM) libraries with logging and recovery mechanisms [9,21,24,25]. Nevertheless, although flexible and unbounded, software approaches may display a considerable overhead for applications with short-lived transactions. With the availability of microprocessors with hardware transaction extensions (HTM) [16,18], researchers have proposed using this mechanism as a way of speeding up the performance of applications running on durable transactional systems. The key idea of recent HTM-enabled solutions [8,14,20] is to separate the execution of a durable transaction into two parts. In the first one, transactions are executed using the hardware support and construct a volatile redo log. The second stage consists of a transaction persisting its log and ensuring that the order is consistent with some serial execution. One important drawback of hardware-based solutions, however, is that most microprocessors only provide best-effort transactions, meaning that transactions are not guaranteed to always commit in hardware.

Although each approach (HW or SW) has distinct virtues, the decision about which one to use is usually left to programmers. However, making the right choice requires an intricate understanding of workload and system-specific characteristics, and is often dynamic (i.e., the optimal approach changes throughout an application's execution). This work is motivated by the observation that this decision is a fundamental gap that affects the effectiveness of the current state of the art on (SW and HW) durable transactions. In order to fill that gap, we propose NV-PhTM: a Non-Volatile Phased Transactional Memory system that delivers the best out of both HW and SW transactions by dynamically selecting the best execution mode according to the application's characteristics. A key decision in designing NV-PhTM concerns how to handle the concurrent execution of HW/SW transactions.

Before the emergence of NVM, the goal of combining SW and HW transactions had already received plenty of attention in the context of non-durable transactional memory [15]. Historically, the first approaches allowed both HW and SW transactions to concurrently execute in the same application, which is commonly designated as Hybrid Transactional Memory (HyTM). More recently, different studies have shown that HyTM has inherent scalability issues [1,7]. In parallel, the alternative approach of Phase-based Transactional Memory (PTM) was proposed as a pragmatic way of avoiding the fundamental pitfalls of HyTM [6,19] through a simplistic design where SW and HW transactions no longer run concurrently; instead, the execution is split into all-SW and all-HW phases. PTM systems have to deal with challenges such as when to transition the execution to different phases, accomplished through heuristics, and how to efficiently perform the transition.

To the best of our knowledge, NV-PhTM is the first system to apply the principles of PTM in the context of durable transactions. As we discuss later on, directly applying existing non-durable PTM systems to this new context is sub-optimal as it neglects new phenomena and trade-offs that durability brings about. NV-PhTM provides insights regarding the construction of new heuristics and phase transition in a NVM context. In particular, this paper makes the following contributions:

- It proposes NV-PhTM and new heuristics with the aim of allowing phase-based execution of durable transactions (see Sects. 3.1 and 3.2);
- It devises a new strategy to allow the migration between HW and SW transactions while maintaining consistency and persistency (see Sect. 3.3);
- It provides experimental results, based on the STAMP benchmark [22], showing that NV-PhTM is efficient and can provide the best of both HW and SW transactions (see Sect. 4).

The rest of the paper is organized as follows. Section 2 presents the main concepts used in this work. Section 3 gives a detailed description of the NV-PhTM design, whereas Sect. 4 presents its evaluation, comparing it against other state-of-the-art approaches. Section 5 provides an overview of related works and, finally, Sect. 6 concludes the work.

2 Background

This section briefly describes two representative HW and SW systems which provide durable transactions, namely NV-HTM and PSTM. These systems serve as the base in which NV-PhTM is built upon.

NV-HTM [8] is one of the first systems to provide durable transactions over commodity transaction-enabled hardware. Its commit stage is split into *non-durable* and *durable* stages. When a hardware transaction executes, it also stores its updates into a redo log (a per-thread structure). Upon a commit, the hardware makes the updates visible to other concurrent threads but does not necessarily persist them. This is the so-called *non-durable commit*. After that, the transaction's redo log is persisted via software (it might have to wait for the logs of transactions it depends on to be persisted as well), completing the *durable commit* stage. NV-HTM requires instrumenting the procedures to start/commit a transaction and the write operation (to construct the redo log), but read operations can proceed without any instrumentation overhead. A timestamp mechanism is used to enforce consistency: when a transaction is durably committed, all transactions serialized before it by the HTM system are already durably committed. A concurrent checkpointing process is used to persist the snapshot in NVM of all durably committed transactions, as well as pruning the redo logs. In case transactions cannot proceed in hardware, NV-HTM acquires a single global lock and serializes the execution (software transactions are not provided).

The acronym PSTM (Persistent Software Transactional Memory) usually refers to a class of implementations based on the mechanism that Mnemosyne [25]

originally introduced to support durable memory transactions. It is composed of a transaction system and a transaction log. The original proposal described by Mnemosyne is based on TinySTM [13], providing lazy versioning with redo logging and eager conflict detection with encounter-time locking. With lazy versioning, data written by a transaction is stored locally in a buffer (volatile memory) and also added to a log (along with the corresponding addresses). During commit, the log is flushed to NVM and the data is persisted. Notice that lazy versioning requires, for each read operation, checking whether the required data is already present in the local buffer, in which case it contains the most recent value. In order to avoid that, some PSTM systems prefer to adopt undo logs and in-place updates instead [3,8]. Upon each write, the corresponding log entry is flushed to NVM before the data is modified in-place. During commit, the changes are flushed to NVM and a commit marker is added to the log. The cost of durable transactions is two writes to NVM with every update: one for the log entry and another for the data itself.

3 NV-PhTM Design

NV-PhTM allows the execution of HW/SW transactions in phases. It provides the following features in the context of NVM: i) new heuristics to guide transitions among hardware (HW), software (SW) and serialized (GLOCK) phases; ii) a consolidation strategy to enforce system consistency and persistency when transitioning between different phases. This section discusses NV-PhTM general system architecture, transition heuristics and state consolidation strategies.

3.1 System Architecture

The two main building blocks of NV-PhTM are NV-HTM (for HW transactions) and PSTM (for SW transactions), described previously. A general overview of the architecture is presented in Fig. 1. The first step performed by the system is to map the memory region (e.g., by using *mmap*) to the application address space, creating a Working Snapshot (WS)①. As soon as the transaction performs the first access to a page mapped on the PS, the operating system automatically uses copy-on-write (CoW) to create a volatile copy in DRAM. Hence, during execution, the load and store instructions emitted by transactions operate on DRAM-mapped pages of the WS②. When a hardware transaction completes, two actions take place. First, the HTM system non-durably commits the transaction data (volatile memory). Second, the system flushes the redo log to NVM③, in which case the transaction is durably committed. A Checkpoint Process (CP) is responsible for applying the updates stored in the logs④ into a consistent Persistent Snapshot (PS)⑤, as well as pruning the redo logs so that they do not grow beyond a given threshold. The application can also invoke the CP⑥ to perform *memory consolidation*, an operation that drains all the durable logs to the PS and discards every page that has been cloned in DRAM. It is used to consolidate the updates to the PS before migrating to SW mode.

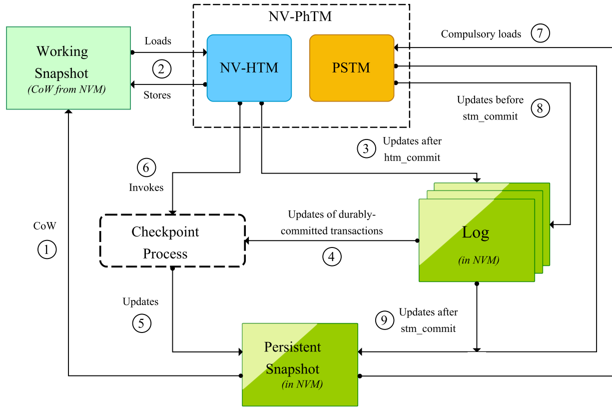
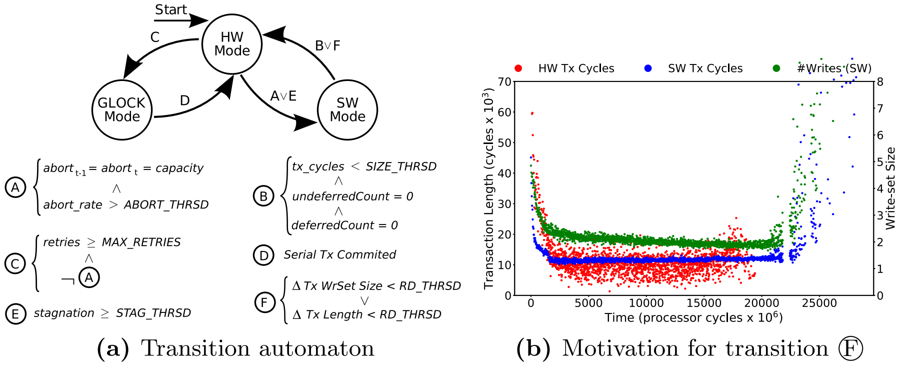


Fig. 1. NV-PhTM system architecture.

For the software part (PSTM), NV-PhTM extends the NOrec transactional system [10] with non-volatile semantics. Differently from the HW component, each transaction here is responsible for persisting its log as well as flushing the updates to PS. During transaction processing, values are initially accessed from the PS ⑦ and stored locally (DRAM) in the transaction's write set. For each memory update performed during the execution of a transaction, its corresponding log entry is flushed to NVM⑧; when a transaction finally commits, it appends a commit marker to the log and persist all updates to PS ⑨. The sequence lock of NOrec is used to order the commit events in the log. Notice that the CP component is not used while in SW mode, since the transactions themselves are responsible for consolidating the updates to PS. Another design option would be to use the idea of splitting the commit operation into stages and let the CP module perform consolidation, similarly to what is done in DudeTM [20]. However, in this initial investigation on phase-based durable transactions we opted for a more traditional design, leaving commit splitting for future work.

3.2 Transition Heuristics

Transition heuristics specify *in what conditions* and *to which phase* the system should migrate. NV-PhTM behavior is better understood by its transition automaton, showed in Fig. 2a. The system initially starts in HW mode. A HW→SW transition is triggered when two consecutive aborts occur due to capacity issues and the abort rate is above a given threshold Ⓐ. These conditions represent situations where transactions are very unlikely to make the most out of current HW transactions, thus execution falls back to SW mode. However, when the length of transactions (measured in cycles) is relatively small, the system may return to HW mode since short-running transactions tend to cause high overhead in SW mode. The SW→HW transition only completes once all deferred transactions (those that caused the HW→SW migration) are com-



(a) Transition automaton **(b)** Motivation for transition Ⓕ

Fig. 2. Design of the NV-PhTM heuristics. (Color figure online)

mitted and no other transaction is active. Internally, this is identified by the two variables, `deferredCount` and `undeferredCount`, being equal to zero Ⓑ. The variable `deferredCount` keeps track of how many transactions invoked the HW→SW transition. These are the transactions that should complete in SW before the system considers returning to HW. As for `undeferredCount`, it counts the transactions that are actively running in SW, but are only doing so because other transaction(s) invoked the switch.

The execution is serialized Ⓒ if: i) most of the aborts are not caused by capacity issues (therefore they are very likely caused by conflicts among transactions), ii) the abort rate is not high enough, and iii) the number of retries reached a given threshold, In this situation it is pointless to migrate to SW mode and thus serializing the execution may be more beneficial – as entering and leaving the GLOCK mode is much faster compared to the SW mode. When the serialized transaction is completed, the system returns to HW mode Ⓓ.

So far, the described heuristics take care of avoiding capacity and contention issues, but they do not address problems caused by NVM. For instance, a particular source of efficiency loss when running in HW mode is the persistent log structure that is used to store the updates of the transactions. Recall that the CP is responsible for pruning this log and consolidating the changes into the PS (steps ④ and ⑤ in Fig. 1). If the number of writes to the log is high, the log will probably fill up before the CP is able to free some space, stalling the execution. We named this scenario as *log-induced stagnation*, since transactions are unable to proceed until there is enough space in the log. Therefore, a new heuristic was added to NV-PhTM in order to force a HW→SW transition when stagnation is problematic (above an empirically determined threshold) Ⓔ. Recall that PSTM does not use the CP and, as a consequence, the log-induced scenario cannot happen while in SW mode.

We observed that, for some applications running solely in HW mode, the stagnation issue tends to dissipate over time. If that happens, then running in HW mode might yield better performance. But if the system migrated to SW due

to heuristic Ⓔ, it has no direct way of knowing if the stagnation level would be low and whether returning to HW is a good idea. In order to have an insight into the new SW→HW heuristic that addresses this point, please refer to Fig. 2b. It shows the average transaction length (left Y axis), for both HW (red) and SW (blue) transactions, as well as the average write-set size (right Y axis, green dots), as time goes on (X axis). The plot is for the Intruder application from the STAMP benchmark running with 12 threads (see Sect. 4 for details on the experimental settings). The key point here is to notice that there is a relationship between the reduction in the write-set size (green dots) and SW transaction length (blue dots) with the stagnation level, as it is possible to observe that HW transactions become faster (red dots) when that happens (around 2 million cycles in the figure). The new SW→HW heuristic developed for NV-PhTM uses this reduction in the write-set size and length of SW transactions to force a transition when the threshold *RD_THRSD* is met Ⓕ.

3.3 Consolidation Strategies

If a crash occurs while executing in either HW or SW mode it is possible to recover the state by replaying the logs. However, inconsistencies might occur due to the transitions between modes. Phase transitions are handled by a shared `modeIndicator` variable, which is always read by HW transactions when they start. When the condition for HW→SW is met, the transaction that triggers the transition atomically changes `modeIndicator` to SW, which aborts all running hardware transitions. Upon restart, these transactions will notice the mode change and will run in SW mode. Notice that this behavior would allow SW transactions to start executing (and change the logs) while the CP might still be executing. Therefore, NV-PhTM requires a *barrier* when switching modes. An extra bit of `modeIndicator` is used to act as a permission flag. When the mode is changed, the flag is atomically set (using a CAS operation); the transactions that detect the HW→SW transition wait for the permission flag before entering SW mode. Meanwhile, the transaction that triggered the migration invokes and waits for the CP to perform a system consolidation procedure before resetting the permission flag. At this point all transactions will start in SW mode and the PS will be correctly updated. Handling SW→HW transition is similar, but does not require waiting for the CP since it is not used in SW mode.

4 Experimental Evaluation

This section presents a thorough quantitative evaluation of NV-PhTM by showing the effectiveness of the new heuristics and speedup numbers against state-of-the-art systems.

4.1 Setup

The experimental evaluation considers the following systems:

NV-HTM: an implementation of the work in Castro et al. [8] using a threshold of 9 consecutive retries for serialization, 10000 log entries per thread, and a log occupancy threshold of 50% (used to activate log pruning);

PSTM: based on NOrec STM [10] with redo log and lazy versioning similar to Mnemosyne [25]. As in NV-HTM, a log of 10000 entries is also used for each thread;

PhTM*: an implementation of NV-PhTM without heuristics Ⓔ and Ⓕ. It is considered here so that the effectiveness of the new NVM-aware heuristics can be assessed;

NV-PhTM: the newly proposed phase-based transactional system with durable transactions described in this paper. It uses the same core parameters of PSTM and NV-HTM, an abort threshold (*ABORT_THRSD*) of 75%, transaction length threshold (*SIZE_THRSD*) of 30000 cycles, and stagnation threshold (*STAG_THRSD*) of 45%. The implementation is very lightweight and based on the `rdtscp` instruction for collecting timing information. The time spent by transactions waiting for the log (stagnation time) and total time are measured and the ratio is computed. The SW→HW transition is triggered when a 15% reduction (*RD_THRSD*) over time of the write-set size and transaction length is detected. Measurements are collected every 1000 committed transactions. See Sect. 4.4 for a brief discussion on how these parameters were selected.

The systems are evaluated using the STAMP (Stanford Transactional Applications for Multi-Processing) benchmark suite [22]. Speedup is calculated by using a NVM-aware sequential version of the applications without any concurrency control as the baseline. The experiments are performed on an 18-core Intel Xeon Gold 5220 machine (with TSX support) clocked at 2.20 GHz, 192 GB physical DRAM, and x86-64 Linux kernel 3.10. The applications were compiled using GCC 7.3.1. The reported results represent the average of 30 runs; a 95% confidence interval bar is also shown. In order to avoid some performance issues induced by the memory allocator [5,11], the TCMalloc allocator with the changes suggested by Nakaike et al. [23] is used. Finally, like previous works [8,9,20], NVM is emulated using DRAM. In particular, slow writes to NVM are modeled by adding a delay of 500ns.

4.2 `SSCA2` and `Intruder`

This section provides a detailed discussion for two of the most representative applications of STAMP: `SSCA2` and `Intruder`. The speedup (Y axis) as the number of threads increases (X axis) is shown in Fig. 3. In order to better understand the behavior of the systems, Table 1 presents the percentage of time spent in the different modes for NV-HTM, PhTM*, and NV-PhTM: `HW` for hardware, `SW` for software, and `GL` for GLOCK (their sum should add to 100% of the total execution time). Table 1 also shows the average percentage of the total time consumed by log-induced stagnation (LIS), that is, the fraction of the total time that threads need to wait for enough log space.

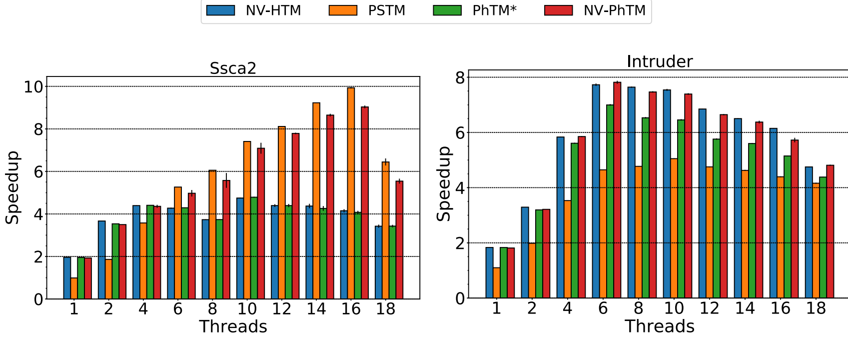


Fig. 3. Speedup numbers evaluating the effectiveness of the proposed heuristics.

Table 1. Fraction of time (%) spent in each mode (HW, SW, GL) and Log-Induced Stagnation (LIS).

App	#t	NV-HTM			PhTM*				NV-PhTM			
		HW	GL	LIS	HW	SW	GL	LIS	HW	SW	GL	LIS
SSCA2	1	100.00	0.00	0.00	100.00	0.00	0.00	0.00	100.00	0.00	0.00	0.00
	2	99.99	0.01	0.26	99.99	0.00	0.01	0.21	99.99	0.00	0.01	0.20
	4	99.99	0.01	19.80	99.99	0.00	0.01	19.78	96.71	3.28	0.01	18.76
	6	99.99	0.01	31.30	99.99	0.00	0.01	30.98	22.83	77.17	0.00	7.83
	8	99.99	0.01	36.68	99.99	0.00	0.01	36.99	14.76	85.23	0.00	6.79
	10	99.98	0.02	37.09	99.99	0.00	0.01	36.97	8.62	91.37	0.00	3.38
	12	99.98	0.02	39.66	99.98	0.00	0.01	39.64	7.65	92.35	0.00	2.55
	14	99.98	0.02	41.60	99.99	0.00	0.01	41.66	9.39	90.61	0.01	3.37
	16	99.99	0.01	43.25	99.98	0.00	0.02	43.45	11.34	88.66	0.01	4.36
	18	99.99	0.01	46.03	99.97	0.01	0.01	46.02	23.67	76.33	0.00	10.59
Intruder	1	99.81	0.19	0.02	99.77	0.02	0.21	0.02	99.80	0.00	0.20	0.02
	2	99.02	0.98	0.44	96.62	2.56	0.81	0.30	99.04	0.02	0.94	0.35
	4	96.17	3.83	2.19	89.15	8.61	2.24	2.20	96.37	0.16	3.47	2.11
	6	92.74	7.26	4.77	81.06	16.07	2.87	4.88	93.46	0.77	5.77	4.67
	8	97.39	2.61	14.79	77.10	19.43	3.47	10.85	90.91	1.12	7.96	11.82
	10	97.31	2.69	21.88	75.37	21.11	3.51	16.10	88.33	2.62	9.05	17.89
	12	95.22	4.78	26.98	71.94	23.05	5.02	19.23	83.10	4.98	11.93	20.89
	14	87.26	12.74	26.99	71.24	23.09	5.68	21.87	77.76	9.48	12.76	22.60
	16	96.70	3.30	35.26	75.22	23.91	0.86	27.75	75.83	18.51	5.66	27.64
	18	83.47	16.53	35.07	61.25	35.14	3.61	25.64	67.62	20.29	12.09	27.47

For `SSCA2`, it is possible to see that NV-HTM performs well up to 4 threads, but PSTM then starts to display a better performance. This is mostly due to the stagnation problem occurring in NV-HTM as showed by the column LIS in Table 1. Since PhTM* does not have the new heuristic that takes into account the stagnation time Ⓔ, it does not transition to SW and therefore performs similarly to NV-HTM, spending most of its time in HW mode. On the other hand,

NV-PhTM is able to switch to the SW mode and follows PSTM's performance closely after 4 threads. Notice also that the LIS column (highlighted) confirms the reduction of stagnation time due to the migration to SW mode. This result highlights the main feature of phase-based systems: its ability to automatically identify the best performing system (NV-HTM up until 4 threads, and PSTM after that). Because PhTM* is not aware of the stagnation issue, it continues following NV-HTM and therefore does not perform well with more than 4 threads.

`Intruder` is a case in which HW performs better throughout all thread configurations and therefore NV-HTM exhibits the best performance numbers. For this application, stagnation is not as severe as with `SSCA2` (LIS column). Contrary to other applications, the stagnation levels in `Intruder` varies during its execution, with peaks at the beginning and end, when the number of writes is more accentuated (see Fig. 2b). Although PhTM* performs better than PSTM, its performance numbers are not as good as NV-PhTM from 6 threads onwards. PhTM* is still able to transition to SW because of the high overall abort rate. However, as Table 1 reveals (highlighted), it spends more time in SW than necessary because it does not have the new SW→HW heuristic based on the reduction of number of writes and transaction length Ⓕ. Overall, the results obtained with NV-PhTM show the effectiveness of the new heuristics for HW→SW (`SSCA2`) and SW→HW (`Intruder`) transitions.

4.3 Remaining STAMP Applications

The performance results for the remaining STAMP applications are shown in Fig. 4. Due to space reasons we only consider a subset of the threads. For `Genome`, `Labyrinth` and `Vacation` there are small performance differences between PhTM* and NV-PhTM, implying that the new heuristics do not play a major role with these applications. Stagnation is not a major issue in `Genome` and, for `Vacation`, there is a large number of capacity aborts that force HW→SW transitions. Indeed, 99% of the total execution time is spent in SW mode in `Vacation` starting from 8 threads. On the other hand, `Genome` spends about 90% of the total execution time in HW mode. `Labyrinth` has very long transactions, forcing HW transactions to abort almost all time. In fact, 99% of the time is serialized with NV-HTM because it employs a global lock as the fallback mechanism in case of high contention. Both PhTM* and NV-PhTM can detect the serialization issue very early and switch to SW mode.

`Kmeans` is an application with a high variability in execution time. Even then, it is possible to see that NV-PhTM follows the best system, NV-HTM, more closely than PhTM* up to 4 cores (the maximum speedup achieved with this application). After that, NV-HTM tends to get worse because stagnation time starts to become an issue and, eventually, at 10 threads, PSTM takes over. At this point NV-PhTM starts following PSTM whereas PhTM* does not, as seen in the configuration with 16 threads. There is a small inaccuracy with NV-PhTM

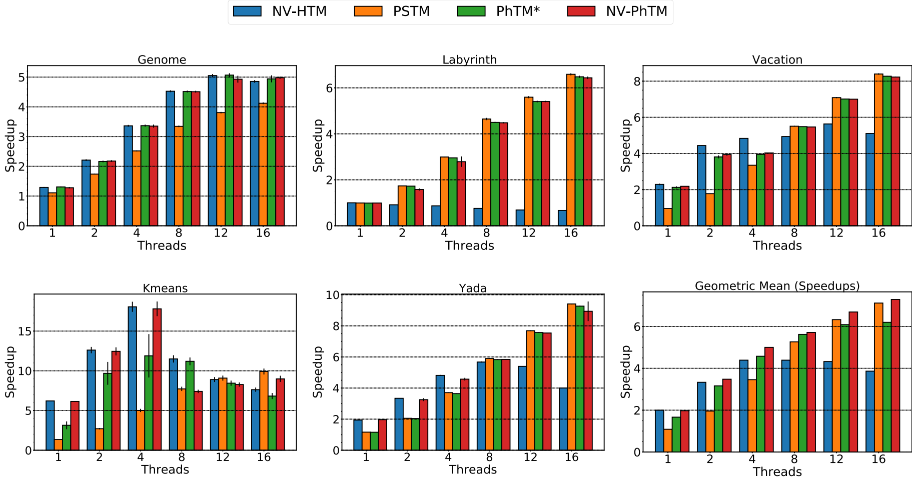


Fig. 4. Speedups for the STAMP applications.

at 8 threads as the stagnation threshold is reached and the system migrates to SW mode. `Yada` is an application in which stagnation is not a major concern. Capacity and conflict aborts are the major cause of its inefficiency, particularly after 4 threads, when the majority of the execution time of NV-HTM is serialized. Here, NV-PhTM correctly starts following NV-HTM but switches to PSTM as serialization starts to dominate the execution time of the HW mode. PhTM* is always using the SW mode because its heuristics force a HW→SW very early given the capacity aborts. It also cannot return to HW because the length of the transactions is way above the minimum threshold. NV-PhTM, on the other hand, can perform a SW→HW transition since `Yada` has a behavior similar to `Intruder`, in which the transaction length decreases rapidly.

The last plot in Fig. 4 shows the geometric mean of the speedups for all the applications considered. It is clear from this plot that NV-HTM tends to be faster than PSTM up until 4 threads when stagnation is still not a serious issue, but after that PSTM starts to dominate. Since the phased systems are very likely to follow the best performing system, they also display good overall results. In particular, the improved heuristics provided by NV-PhTM in the context of NVM makes it a superior option when compared to PhTM*. The reason is clear: when stagnation is not a problem, it performs similarly to PhTM*; but PhTM* heuristics cannot deal with log-induced stagnation and therefore NV-PhTM performs better overall.

Table 2. Percentage of total time spent during phase migrations and the average number of transitions (ANoT).

Application	Transition	Threads									
		1	2	4	6	8	10	12	14	16	18
Genome	HW→SW	0.00	0.01	0.03	0.03	0.03	0.04	0.05	0.06	0.11	0.10
	SW→HW	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.01	0.01
	ANoT	1.80	2.07	1.70	2.03	1.27	1.47	3.73	5.70	17.20	13.97
Intruder	HW→SW	0.00	0.00	0.00	0.01	0.07	0.22	0.56	0.56	0.06	1.02
	SW→HW	0.00	0.00	0.00	0.01	0.01	0.03	0.06	0.06	0.02	0.07
	ANoT	0.67	6.07	7.37	63.33	106.40	198.00	465.57	475.93	125.40	658.70
Kmeans	HW→SW	0.00	0.00	0.01	0.01	0.01	0.01	0.01	0.02	0.02	0.02
	SW→HW	0.00	0.00	0.01	0.00	0.00	0.00	0.00	0.01	0.01	0.01
	ANoT	1.13	1.83	43.17	39.67	48.00	47.27	48.77	47.60	49.17	65.77
Labyrinth	HW→SW	0.00	0.01	0.17	0.24	0.23	0.09	0.21	0.01	0.40	0.13
	SW→HW	0.00	0.21	0.02	0.01	0.01	0.02	0.25	0.26	0.34	0.70
	ANoT	1.00	1.50	1.03	1.00	1.00	1.00	1.00	1.00	1.07	1.00
SSCA2	HW→SW	0.00	0.00	0.00	0.01	0.01	0.01	0.02	0.02	0.02	0.06
	SW→HW	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.01	0.01
	ANoT	0.47	0.23	1.47	3.80	8.77	8.60	16.40	21.27	25.33	119.07
Vacation	HW→SW	0.00	0.01	0.04	0.16	0.03	0.03	0.03	0.03	0.03	0.03
	SW→HW	0.00	0.00	0.00	0.02	0.00	0.00	0.00	0.00	0.00	0.00
	ANoT	2.57	3.27	31.33	167.10	5.17	5.13	5.30	5.03	5.57	6.07
Yada	HW→SW	0.00	0.01	0.01	0.02	0.03	0.03	0.03	0.03	0.03	0.03
	SW→HW	0.00	0.00	0.01	0.01	0.02	0.03	0.03	0.04	0.03	0.04
	ANoT	2.13	3.40	5.10	12.40	15.87	15.20	16.47	13.13	12.40	12.07

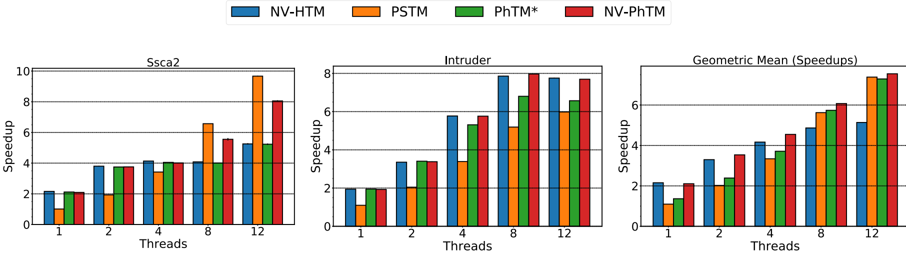


Fig. 5. Results with a Xeon E5-2648L @1.80 GHz using the same thresholds.

4.4 Discussion

The results show that NV-PhTM is able to follow the best performing mode, either HW or SW, but usually does not outperform the best of the two modes for a given configuration. The reason for this, as discussed in details by Carvalho et al. [7], is due to the lack of phases in the STAMP applications. However, the ability of dynamically adapting its behavior to exploit the best execution mode makes it a valuable option. A criticism of phased systems is that transition between modes can take a large fraction of the time because of the possible barriers. In case of NV-PhTM, this has to be analyzed since it does require

a barrier when transitioning from HW to SW because of the NVM consistency requirements. In order to show that this is not a problem, Table 2 shows the percentage of the total time (averaged over 30 runs) consumed by transitions (both HW→SW and SW→HW) and The Average Number of Transitions (ANoT) for each application. As can be noticed, the largest overhead happens with `Intruder` (18 threads) and it only consumes about 1% of the total execution time. The reason for such low overhead is that the heuristics cause very few transitions, as can also be observed in the table.

Finally, the heuristics require some thresholds to be tuned in order to get a good performance. These values were set after a performance analysis with the STAMP benchmark, similarly to previous work [7]. To show that the results presented here are consistent, we repeated the experiments in another machine, a 14-core Intel Xeon E5-2648L running at 1.80 GHz and 32GB of RAM, without changing any of the thresholds. Notice that this machine is slightly slower and has smaller L2 and L3 caches. The main results are shown in Fig. 5 for the `SSCA2`, `Intruder` and the overall geometric mean. As can be seen, the main conclusions carried over.

5 Related Work

The works of Avni et al. with PHTM [3] (Persistent Hardware Transactional Memory) and Wang et al. with PTM [26] (Persistent Transactional Memory) were the first to explore HTM in the context of durable transactions. In both cases, the proposed solutions require changes to existing HTM designs which, in practice, limit their usage. DHTM [17] (Durable Hardware Transactional Memory) is a more recent hardware approach that also requires minor changes to the coherence protocol. HTM-enabled systems that use current microprocessors were developed recently [8,14,20]. These systems use the same idea of splitting the execution of a transaction into two parts: one that operates on volatile memory (using HTM support) and a decoupled phase responsible for persisting the changes into NVM. They differ on how this is achieved. In particular, the approach taken by Castro et al. with NV-HTM [8] avoids the use of a shared logical clock to serialize hardware transactions as in DudeTM [20]. Also, NV-HTM does not require instrumenting load operations as proposed in the work developed by Gilles et al. [14].

The closest work to ours is PHyTM [2] (Persistent Hybrid Transactional Memory), which allows the concurrent execution of both hardware and software transactions. However, PHyTM is based on PHTM [3] and, as such, cannot use current hardware support. NV-PhTM, on the other hand, does not have that limitation since it uses the decoupled HW mechanism of recent works. Moreover, recent studies have showed that hybrid systems have an inherent scalability limitation [1,7]. There are works on phase-based systems [6,7,19] but they target architectures with volatile memory. ProteusTM [12] automatically identifies the best TM implementation based on a multi-dimensional online optimization mechanism but, as with previous phase-based systems, it was design for volatile

memory. Using phase-based systems with NVM, though, raises new problems concerning both the switching policies (which need to take into account specific performance issues, e.g., log stagnation, affecting NVM-related solutions) and the logic used to regulate the switching between the various execution phases (e.g., ensuring that logs are consolidated prior to switching to SW). In Section 4 we have compared the effectiveness of the switching policies used in prior phase-based systems [7] that considered volatile memory and demonstrated experimentally their inadequacy in the context of NVM.

6 Conclusion

In this paper we presented NV-PhTM: an efficient phase-based transactional system for persistent memory. NV-PhTM solves the performance issues of both hardware-only and software-only approaches by dynamically selecting the best operation mode. The key novel contributions of NV-PhTM consist of: i) new lightweight policies that allow for automatically identifying the best performing execution mode (STM or HTM) for arbitrary workloads in a NVM context, and ii) defining an architecture and phase transitioning mechanisms that allow for the safe alternation between the phased execution modes. Experimental results show that NV-PhTM can efficiently select the best execution mode and has low transition overhead.

Acknowledgments. The authors would like to thank the anonymous reviewers for their insightful comments. This work was supported by FAPESP (grants 2013/08293-7, 2016/15337-9, 2018/15519-5, and 2019/10471-7), Center for Computational Engineering and Sciences (CCES), and Fundação para a Ciência e a Tecnologia (under project UIDB/50021/2020).

References

1. Alistarh, D., Kopinsky, J., Kuznetsov, P., Ravi, S., Shavit, N.: Inherent limitations of hybrid transactional memory. Distrib. Comput. **31**(3), 167–185 (2017). https://doi.org/10.1007/s00446-017-0305-3
2. Avni, H., Brown, T.: Persistent hybrid transactional memory for databases. Proc. VLDB Endow. **10**(4), 409–420 (2016)
3. Avni, H., Levy, E., Mendelson, A.: Hardware transactions in nonvolatile memory. In: Moses, Y. (ed.) DISC 2015. LNCS, vol. 9363, pp. 617–630. Springer, Heidelberg (2015). https://doi.org/10.1007/978-3-662-48653-5_41
4. Badam, A.: How persistent memory will change software systems. Computer **46**(8), 45–51 (2013)
5. Baldassin, A., Borin, E., Araujo, G.: Performance implications of dynamic memory allocators on transactional memory systems. In: Proceedings of the 20th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, pp. 87–96 (2015)
6. Carvalho, J.P.D., Araujo, G., Baldassin, A.: Revisiting phased transactional memory. In: Proceedings of the International Conference on Supercomputing, pp. 25:1–25:10 (2017)

7. Carvalho, J.P.D., Araujo, G., Baldassin, A.: The case for phase-based transactional memory. IEEE Trans. Parallel Distrib. Syst. **30**(2), 459–472 (2019)
8. Castro, D., Romano, P., Barreto, J.: Hardware transactional memory meets memory persistency. In: 2018 IEEE International Parallel and Distributed Processing Symposium (IPDPS), pp. 368–377, May 2018
9. Coburn, J., et al.: NV-heaps: making persistent objects fast and safe with next-generation, non-volatile memories. In: Proceedings of the Sixteenth International Conference on Architectural Support for Programming Languages and Operating Systems, pp. 105–118 (2011)
10. Dalessandro, L., Spear, M.F., Scott, M.L.: NOrec: streamlining stm by abolishing ownership records. In: Proceedings of the 15th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, pp. 67–78 (2010)
11. Dice, D., Harris, T., Kogan, A., Lev, Y.: The influence of malloc placement on TSX hardware transactional memory. CoRR abs/1504.04640 (2015)
12. Didona, D., Diegues, N., Kermarrec, A.M., Guerraoui, R., Neves, R., Romano, P.: ProteusTM: abstraction meets performance in transactional memory. In: Proceedings of the Twenty-First International Conference on Architectural Support for Programming Languages and Operating Systems, pp. 757–771 (2016)
13. Felber, P., Fetzer, C., Riegel, T.: Dynamic performance tuning of word-based software transactional memory. In: Proceedings of the 13th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, PPoPP 2008, pp. 237–246 (2008)
14. Giles, E., Doshi, K., Varman, P.: Continuous checkpointing of HTM transactions in NVM. In: Proceedings of the 2017 ACM SIGPLAN International Symposium on Memory Management, pp. 70–81 (2017)
15. Harris, T., Larus, J., Rajwar, R.: Transactional Memory, 2nd edn. Morgan & Claypool Publishers, San Rafael (2010)
16. Intel Corporation: Intel® Architecture Instruction Set Extensions Programming Reference (February 2012)
17. Joshi, A., Nagarajan, V., Cintra, M., Viglas, S.: DHTM: durable hardware transactional memory. In: 2018 ACM/IEEE 45th Annual International Symposium on Computer Architecture (ISCA), pp. 452–465, June 2018
18. Le, H., et al.: Transactional memory support in the IBM POWER8 processor. IBM J. Res. Dev. **59**(1), 8-1 (2015)
19. Lev, Y., Moir, M., Nussbaum, D.: PhTM: phased transactional memory. In: Workshop on Transactional Computing (Transact) (2007)
20. Liu, M., et al.: DudeTM: Building durable transactions with decoupling for persistent memory. In: Proceedings of the Twenty-Second International Conference on Architectural Support for Programming Languages and Operating Systems, pp. 329–343 (2017)
21. Memaripour, A., et al.: Atomic in-place updates for non-volatile main memories with Kamino-Tx. In: Proceedings of the Twelfth European Conference on Computer Systems, pp. 499–512 (2017)
22. Minh, C.C., Chung, J., Kozyrakis, C., Olukotun, K.: Stamp: stanford transactional applications for multi-processing. In: 2008 IEEE International Symposium on Workload Characterization, pp. 35–46, September 2008
23. Nakaike, T., Odaira, R., Gaudet, M., Michael, M.M., Tomari, H.: Quantitative comparison of hardware transactional memory for blue Gene/Q, zEnterprise EC12, intel core, and POWER8. In: Proceedings of the 42nd Annual International Symposium on Computer Architecture, pp. 144–157 (2015)

24. Team, T.N.L.: PMEM.IO: persistent memory programming. http://pmem.io/. Accessed 19 Sept 2019
25. Volos, H., Tack, A.J., Swift, M.M.: Mnemosyne: lightweight persistent memory. In: Proceedings of the Sixteenth International Conference on Architectural Support for Programming Languages and Operating Systems, pp. 91–104 (2011)
26. Wang, Z., Yi, H., Liu, R., Dong, M., Chen, H.: Persistent transactional memory. IEEE Comput. Architect. Lett. **14**(1), 58–61 (2015)