This article has been accepted for publication in IEEE Transactions on Parallel and Distributed Systems. This is the author's version which has not been fully edited and content may change prior to final publication. Citation information: DOI 10.1109/TPDS.2021.3085210

DeTraS: Delaying Stores for Friendly-Fire Mitigation in Hardware Transactional Memory

Rubén Titos-Gil[1], Ricardo Fernández-Pascual[1], Alberto Ros[1] and Manuel E. Acacio[1]

Abstract—Commercial Hardware Transactional Memory (HTM) systems are best-effort designs that leverage the coherence substrate to detect conflicts eagerly. Resolving conflicts in favor of the requesting core is the simplest option for ensuring deadlock freedom, yet it is prone to livelocks. In this work, we propose and evaluate DeTraS (Delayed Transactional Stores), an HTM-aware store buffer design aimed at mitigating such livelocks. DeTraS takes advantage of the fact that modern commercial processors implement a large store buffer, and uses it to prevent transactional stores predicted to conflict from performing early in the transaction. By leveraging existing processor structures, we propose a simple design that improves the ability of requester-wins HTM systems to achieve forward progress in spite of high contention while side-stepping the performance penalty of falling back to mutual exclusion. With just over 50 extra bytes, DeTraS captures the advantages of lazy conflict management without the complexity brought into the coherence fabric by commit arbitration schemes nor the relaxation of the single-writer invariant of prior works. Through detailed simulations of a 16-core tiled CMP using gem5, we demonstrate that DeTraS brings reductions in average execution time of 25% when compared to an Intel RTM-like design.

Index Terms—

1 Introduction and Motivation

Hardware Transactional Memory (HTM) systems implement optimistic concurrency control by allowing multiple transactions to run speculatively in parallel. If the atomicity of any of these speculative transactions cannot be guaranteed, the transaction is aborted and its changes discarded. Atomicity is in danger when two or more concurrent transactions access the same data and at least one of the accesses is a write. These *conflicts* are commonly detected by monitoring the coherence traffic generated by loads and stores to the memory subsystem.

Moreover, HTM support in current commercial processors is best-effort, which means that the hardware gives no guarantees that a speculative transaction will ever succeed [1]. Therefore, a non-speculative alternative software path, often called *fallback path*, must be combined with the HTM support to ensure forward progress in circumstances that otherwise would cause livelock because of insufficient speculative buffering capacity, page faults, high contention, etc. The recommended implementation of the fallback path [1] uses a single global lock, commonly referred to as the *fallback lock*, which is read by all speculative transactions as soon as speculation begins. This approach, known as *eager subscription*, ensures that a non-speculative transaction never executes concurrently with other speculative transactions. When the lock is acquired, all subscribed speculative transactions are aborted as a result of the conflict on the fallback lock, and subsequently forced to wait for the lock to be released before they can retry speculatively. Such serialization of threads on the fallback path hurts application performance.

Transactions being aborted as a consequence of a conflict do not resort immediately to the fallback path. Instead, since conflicts may be a transient condition resulting from specific thread interleavings, they are retried several times before the fallback path must be definitely taken after a certain threshold of unsuccessful speculative attempts.

In addition, current implementations of HTM, as the *Restricted Transactional Memory* (RTM) extensions provided by Intel processors, employ an eager approach to resolve conflicts using a *requester-wins* policy: conflicts are always resolved in favor of the requesting transaction. The low integration complexity into existing coherence protocols of a requester-wins conflict resolution policy makes it appealing to chip manufacturers wanting to provide HTM support, but it has a fundamental drawback: in situations of high contention, transactions may repeatedly abort each other over and over in a pathological scenario known as *friendly fire* [2], which spoils forward progress and hurts performance as transactions are forced to proceed in mutual exclusion.

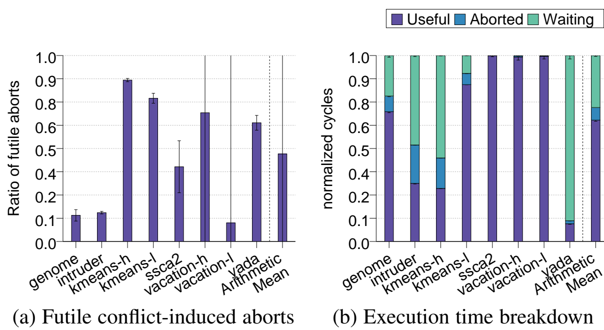
Fig. 1a shows the fraction of futile aborts suffered in STAMP benchmarks with a requester-wins policy, *i.e.*, aborts caused by a transaction that in turn ends up aborting later on because of a conflict (see Section 4 for configuration details). As can be observed, nearly half of all conflict-induced aborts are futile. The consequences of such a high percentage of futile aborts on performance are drawn in Fig. 1b, which plots the fraction of total cycles spent waiting for a non-speculative transaction that acquired the lock: on average, over one fifth of all execution cycles are spent by threads waiting on the fallback lock because of repetitive aborts, even though some of them may not have conflicts with the non-speculative transaction.

The typical alternative conflict management strategy to requester-wins is *requester-stalls* [3], [4], [5], [6], which resolves conflicts by stalling the conflicting memory access until the offended transaction ends. Requester-stalls avoids livelocks and may reduce the number of aborts when compared to requester-wins, provided that cyclic dependencies among conflicting transactions are infrequent. On the downside, it requires hardware changes in the coherence protocol (*e.g.*, negative acknowledgments or *nacks*)

• [1]*Dept. Ingeniería y Tecnología de Computadores, Universidad de Murcia, 30100 Murcia (SPAIN)*
E-mail: {rtitos, rfernandez, aros, meacacio}@ditec.um.es

© 2021 IEEE. Personal use is permitted, but republication/redistribution requires IEEE permission. See https://www.ieee.org/publications/rights/index.html for more information.

This article has been accepted for publication in IEEE Transactions on Parallel and Distributed Systems. This is the author's version which has not been fully edited and content may change prior to final publication. Citation information: DOI 10.1109/TPDS.2021.3085210



(a) Futile conflict-induced aborts (b) Execution time breakdown

Fig. 1: Characterization of requester-wins HTM.

as well as a deadlock avoidance scheme in order to detect and break cycles. As a consequence, no current commercial HTM implementation builds on requester-stalls.

All things considered, an ideal HTM design would try to combine the simplicity of a requester-wins policy with the low abort ratio of a requester-stalls policy while skipping their respective disadvantages. A well-known observation towards this goal in the requester-wins conflict management strategy is that a transaction's exposure to aborts by remote concurrent accesses is minimized if writes within the transaction are delayed as much as possible, ideally being unveiled to the other caches at the very end. Prior works have exploited this observation and showed that significant performance gains can be achieved [7], [8]. In particular, they extend the underlying coherence substrate with an *incoherent* state that relaxes the single-writer invariant for write-set blocks, allowing transactional stores to complete in cache and retire from the processor without requiring exclusive ownership. Apart from the changes required to the highly-optimized L1 data caches and cache controller, current approaches require some sort of directory to track such incoherent blocks, as well as associated logic to manage it. Thus, the complexity added to the memory system by such proposed solutions may jeopardize the simplicity appeal of requester-wins, which is ultimately what has driven chip makers to choose this policy over others. To this end, the influence of the store buffer micro-architecture on HTM performance has been largely overlooked in the literature, with only a few exceptions [9], [10]. Most academic papers on HTM neglect the presence of the store buffers found in any modern out-of-order pipeline, and too often abstract away the CPU model as a black box that feeds a memory sub-system augmented to support HTM.

In this work, we adopt a different strategy, previously unexplored, for ameliorating friendly fire through simple yet effective micro-architectural modifications in the processor cores, and leaving the memory system unaltered. Particularly, our proposal is based on the following observations: (i) store operations that are committed can be exposed some time after commit without affecting correctness, that is, they can delay the write to memory, thus postponing potential conflicts and reducing the probability of aborting unnecessarily; (ii) stores within a transaction can be transparently reordered, given that the target address is different, thus increasing store-level parallelism and reducing commit time; (iii) stores that are suspected not to cause conflicts with other transactions can be written to the memory system as soon as possible, thus freeing their entries in the store buffer; and (iv) conflict-free stores can be identified in a feasible way based on their past executions.

Based on these four observations, we propose DeTraS (which stands for Delayed Transactional Stores), an HTM-aware design of the store buffer that selectively delays stores, that is, postpones the write of transactional stores predicted as conflicting, until the transaction has fully executed or the store buffer is full. DeTraS is the first *hybrid-policy* best-effort HTM design that adapts store management policy to observed contention so as to enable reader-writer concurrency during contention without incurring a performance penalty on low contention workloads due to the extra commit latency. This way, our proposed design addresses the shortcomings of prior memory-side techniques to mitigate friendly fire and leaves the memory subsystem unmodified (except for an extra bit in coherence responses).

Although delaying a small amount of conflicting stores may not have performance implications for systems whose consistency model allows relaxing the order of the stores, in systems that require a total store order (*e.g.*, Intel or AMD –x86–) the execution of large transactions could increase processor stalls as non-conflicting stores cannot be performed until the older conflicting ones have been retired from the store buffer. To avoid inducing extra processor stalls in this scenario we propose two inexpensive techniques: (i) re-ordering transactional stores, motivated by the fact that transactions are atomic and the stores within a transaction are not visible by definition by other processors, and (ii) compacting the store buffer to fill the gaps left by stores that perform out-of-order. Without loss of generality, we implemented and evaluated DeTraS on an x86-TSO system to demonstrate its feasibility even in a stricter memory model, and that transparent relaxation of the TSO consistency model inside transactions can bring performance gains at little cost. DeTraS is fully applicable to weakly ordered designs, where it would be simplified since reordering and compaction would happen naturally.

Our results show that with adequate management of transactional stores in the store buffer and with about 50 additional bytes, the choice of conflict resolution policy becomes less relevant for the performance of HTM systems, allowing requester-wins designs to perform nearly at par with more complex conflict management strategies. In spite of lacking progress guarantees, the proposed HTM system largely prevents most temporary livelock scenarios that arise because of the requester-wins policy, and as a result alleviates the frequency at which the fallback path is taken to achieve progress in such contended circumstances, and consequent performance degradation. Through detailed simulations using gem5, we show that DeTraS reduces execution time of STAMP by 25% on average (up to 68% for contended benchmarks) when compared to a typical requester-wins best-effort HTM design.

To summarize, our main contributions are:

- To propose delaying stores in the store buffer to avoid conflicts in Hardware Transactional Memory, and show its effectiveness in reducing the number of aborts.
- To propose the reordering of transactional stores without affecting the consistency model of the system (as stores in a transaction are all visible at once), while leaving the memory subsystem virtually unmodified.
- To show that a large amount of the stores do not cause conflicts and propose a predictor to detect them and get them out of the store buffer as soon as possible.
- To propose an efficient compaction technique for FIFO store buffers.

© 2021 IEEE. Personal use is permitted, but republication/redistribution requires IEEE permission. See https://www.ieee.org/publications/rights/index.html for more information.

This article has been accepted for publication in IEEE Transactions on Parallel and Distributed Systems. This is the author's version which has not been fully edited and content may change prior to final publication. Citation information: DOI 10.1109/TPDS.2021.3085210

2 BACKGROUND

2.1 Store instructions and the store buffer

Modern processors with support for out-of-order execution need to maintain the order of store instructions for three main reasons. First, to preserve sequential semantics, stores to the same or overlapping addresses need to write to memory in order. Second, sequential semantics maintenance also requires that loads must obtain the latest value written by a store to the same address. Thus, when loads perform, all previous in-flight stores are searched, and in case of an address match, the value of the younger store is taken. This is known as store-to-load forwarding. Third, if a memory consistency model like Total Store Order (TSO) is supported, as is the case of x86-64 processors among others, stores (to same or different addresses) need to write to memory in-order. This is known as store→store order. To this end, when dispatched stores are inserted in program order into a circular FIFO *queue* [1], where they reside until each of them becomes the oldest non-completed store and therefore can already write to memory. To avoid stalls caused by long latency misses, TSO allows stores to commit before writing into cache. The part of the queue that holds non-committed stores is known as the *store queue* and the part that holds committed stores is known as the *store buffer* (SB). DeTraS requires changes in the behavior of committed stores, and along the paper we will refer only to the SB part of the queue. Note that the proposed extensions to the SB impact the whole queue.

Each store can initiate the write to memory as soon as the preceding stores in the SB have already started it. This implies that stores will also enter in order in the cache access pipeline. In case of cache hits, stores perform the write in order; in the event of a miss, the cache pipeline is squashed and the subsequent stores will initiate the write again once the previous (missing in cache) store completes the write.

2.2 Store buffer and cache extensions for HTM

When a transaction aborts, speculative stores in the SB and the speculative writes performed in cache have to be canceled. In order to squash only the transactional stores and not the non-speculative pre-transactional stores, the store buffer needs to distinguish them. One alternative is adding a *transactional* bit to each entry in the SB. Another option is to add a pointer that delimits the transactional and non-transactional parts of the SB, as stores reside in order. Both are equivalent, and without loss of generality we assume the former one for the description of DeTraS.

We assume an HTM implementation that follows the Intel RTM specification [1], in which the `xbegin` and `xend` instructions are used to delimit transaction boundaries. According to the *Transactional Synchronization Extensions* (TSX) specification, *a committed transaction has the same ordering semantics as a locked instruction* (*i.e.*, reads or writes cannot be reordered with it), which means that `xend` acts as a full memory barrier, similar to `mfence`, ensuring that no loads nor stores can be reordered with the transaction. Thus, the `xend` instruction can only commit once the SB has been fully drained and no abort has been signaled.

Transactional stores that are written to cache remain speculatively modified and invisible to the other cores until the `xend` instruction commits. Private caches are therefore extended with an *speculatively modified* (SM) bit, so that in case of an abort, SM cache blocks are discarded. The cache coherence protocol is appropriately augmented to accommodate speculative versioning. In particular, the first transactional write to a dirty cache block must write it back to the next cache level to preserve the non-speculative version in case the transaction aborts.

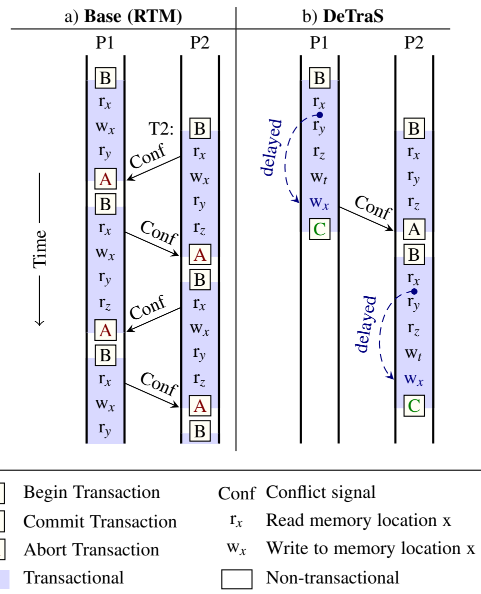


Fig. 2: Motivation for delaying transactional stores. Left: typical friendly-fire scenario in *requester-wins* HTM designs. The transaction comprises the following accesses: r_x, w_x, r_y, r_z, w_t. Right: friendly-fire mitigation by delaying w_x.

3 DETRAS

A key contribution of this work is to show for the first time that the management of transactional stores within the SB plays a prominent role in the performance of an HTM implementation. Fig. 2 shows the following observation: livelocks stemming from the friendly-fire pathology become less and less likely as the contended cache blocks get written later in the transaction. Hence, if selected store instructions delay their write to memory, the friendly-fire scenario represented in Fig. 2.a can be completely eliminated as depicted in Fig. 2.b. This section presents our novel SB design that delays selected transactional stores. Our description builds on top of a standard SB design, as described in the previous section, and it is presented as a set of subsequent refinements. Note that DeTraS does not strictly require a unified queue nor a FIFO structure, and the ideas described here are also applicable to processors with weak memory models.

3.1 Delaying Transactional Stores

The less time a cache block remains speculatively modified in a private cache, the less susceptible a transaction is to conflicts due to remote memory accesses. This well-known observation is what motivates DeTraS. The novelty here is that we achieve it with minimal changes, mainly focused on the SB. Our first proposal to shrink the window of exposure of transactional stores is to simply delay the write to cache of *all* transactional stores for as long as possible, that is, until the `xend` instruction is ready to commit, or until the fill-up of the SB prevents subsequent stores to be dispatched, a condition henceforth called *SB overflow*. Either of these two situations triggers a *SB drain* event: all delayed transactional stores kept in the SB are resumed, so that they write to memory in strict program order as usual.

© 2021 IEEE. Personal use is permitted, but republication/redistribution requires IEEE permission. See https://www.ieee.org/publications/rights/index.html for more information.

This article has been accepted for publication in IEEE Transactions on Parallel and Distributed Systems. This is the author's version which has not been fully edited and content may change prior to final publication. Citation information: DOI 10.1109/TPDS.2021.3085210

Implementation. This scheme requires the addition of a *delay* bit to each entry in the SB. This bit is set to the value of the *transactional* bit when a store commits. Stores with the delay bit set do not initiate the write to cache even if they are at the head of the SB. The *delay* bit of all SB entries gets flash-cleared upon SB drain (`xend` ready to retire, or SB fill-up) as well as upon abort signal. Therefore, no store can be delayed forever and deadlock freedom is easily guaranteed.

3.2 Reordering Transactional Stores

The window of cycles that a transaction is vulnerable to aborts due to conflicts can be further narrowed by reducing SB drain time, provided that all transactional stores can be accommodated in the SB. We therefore propose to perform transactional stores out of order, thus increasing the parallelism on SB drains, not only reducing the exposure of stores to conflicts, but also transaction commit time and pipeline stalls when the SB is prematurely drained upon fill-up.

The key observation to allow reordering stores is that values speculatively modified in cache remain invisible to remote requests until the `xend` instruction retires (*i.e.*, the transaction commits). This observation guarantees that the store→store order is not broken. Each store can start the write to cache as soon as the previous store initiates it and succeeds even if some previous stores miss in cache and get effectively reordered. Note that since stores enter the cache access pipeline in order and stores to the same or overlapped addresses will either both hit or both miss in cache, the sequential semantics are effortlessly respected. In case of a miss, the second store will coalesce with the first store waiting at the miss status holding register (MSHR) until the block is placed in cache. For simplicity, we do not allow reordering between non-transactional and transactional stores (transactional stores cannot enter the cache access pipeline until all preceding non-transactional stores have completed). This is important to make sure that the first transactional store to a dirty block writes back the correct consistent value to the next cache level (it may be incorrect if a transactional store overtakes a pending non-transactional to the same block).

Implementation. Supporting out-of-order writes of transactional stores in TSO architectures requires simple behavioral changes in the SB and cache access pipeline. Transactional stores that miss in cache do not need to squash subsequent stores in the cache pipeline. As no pipeline squashes are possible for a transactional store, it can leave the SB and free their entry, *i.e.*, the SB head is incremented as soon as the cache access is initiated.

3.3 Selective Delay of Transactional Stores

A fundamental drawback of delaying all stores is that it does not fare well for transactions which cannot contain all their stores in the SB. For low-contention workloads, SB overflows are not as harmful since performing some stores too early may not significantly increase friendly fire. Nonetheless, a penalty is incurred upon SB overflow, since pipeline stalls may occur preventing subsequent stores to be dispatched until stores at the SB head complete. On the other hand, highly contended transactions may jeopardize the ability to side-step friendly fire, as each SB overflow exposes the transaction to conflict-induced aborts from the SB-drain point on. In turn, delaying all stores penalizes low-contended transactions that do not fill the SB by pointlessly increasing SB drain time on `xend`, threatening its ability to hide write miss latency.

We observed that very few store instructions are generally responsible for the majority of conflicts and thus the likelihood of SB overflow can be minimized by selectively delaying only certain stores. Following this observation, typical PC-based prediction can determine whether a transactional store is probable to cause conflicts or not based on its past history. Hence, for transactions that are likely to exceed the SB resources, we propose to delay only transactional stores predicted to cause conflicts. To do so, information about whether a completed store caused a remote conflict or not must be exposed so that the predictor can be updated accordingly. This straightforward addition constitutes the only modification in the memory subsystem that DeTraS implies. We must also bear in mind that the penalty of a mispredicted-as-non-conflicting store is generally much higher than the penalty of a mispredicted-as-conflicting store: the former may result in a futile abort, while the latter only increases SB drain latency. In sight of this, our prediction scheme is conservative for stores that appear early in a transaction, since a misprediction in those dramatically widens the window of vulnerability to aborts. Thus, the default prediction under low SB occupancy is selected by monitoring global contention: if the processor has not recently caused nor suffered conflict-induced aborts, stores are not delayed; otherwise, stores are delayed (up to a certain SB occupancy threshold). Once the threshold of SB occupation is surpassed, the local (PC-based) predictor is used.

Since different stores to overlapped memory locations could have contradicting predictions, a mechanism to prevent reordering of overlapped stores is required in order to guarantee sequential semantics. To this end, each transactional store that commits snoops the SB looking for a delayed overlapped store and in case of a hit, the current store will be also delayed independent on the predictor decision.

Selective delay of transactional stores is important for three reasons: i) to avoid penalizing transaction commit latency under low contention (in which all stores would initiate the write as soon as they commit); ii) to reduce the latency of SB drains for large transactions (by reducing the amount of delayed writes), which would decrease the overall duration of the transaction and thus the aforementioned window of exposure to aborts; and iii) to pave the way for better utilization of released SB resources (Section 3.4). This refinement reduces the likelihood of SB overflow in certain scenarios, since entries for non-delayed stores can be released up to the first delayed store. Non-delayed stores following delayed stores can complete but their SB entries remain unavailable (as entries can only be released from the SB head).

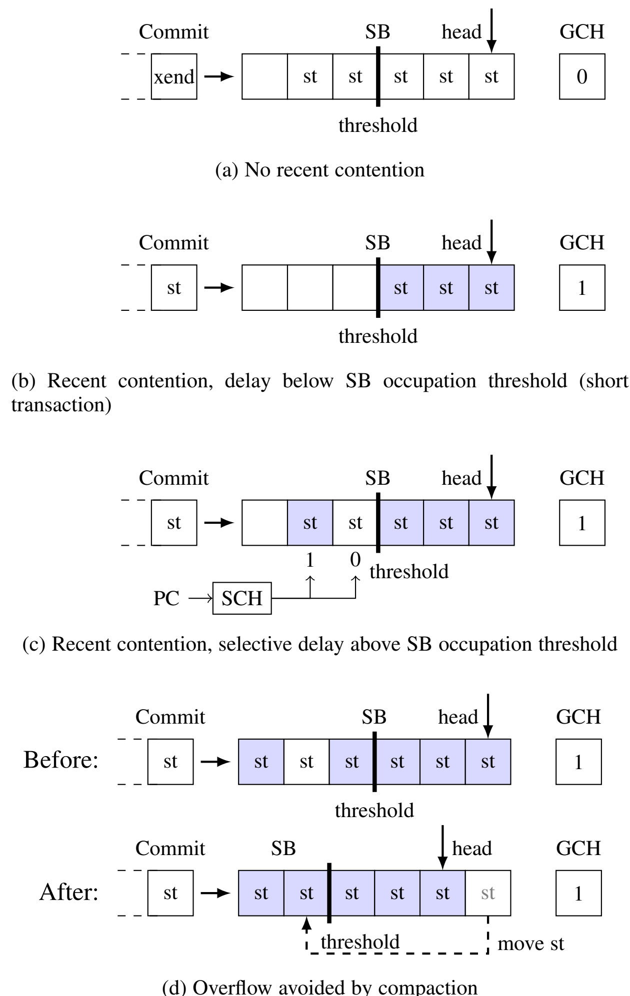
Implementation. DeTraS' prediction scheme employs a *global conflict history* (GCH) counter, a PC-based *store conflict history* (SCH) table, and an *offending transaction* (OT) bit. The SCH is updated after each transactional store completes in cache, with the information given by a *conflict* bit piggybacked in coherence responses, whose value is copied into the corresponding 1-bit SCH entry using the hashed PC as index. When a coherence response with the *conflict* set arrives, the OT bit is asserted (and remains set until the transaction ends) and the GCH is set to its maximum value. The GCH is also saturated upon local conflict-induced aborts, and it is only decremented by transactions that commit while the OT bit remains unset. When the SB is below half capacity, the predictor only checks the GCH: stores are delayed if the GCH is not 0 (an indication of recent contention). Once

© 2021 IEEE. Personal use is permitted, but republication/redistribution requires IEEE permission. See https://www.ieee.org/publications/rights/index.html for more information.

This article has been accepted for publication in IEEE Transactions on Parallel and Distributed Systems. This is the author's version which has not been fully edited and content may change prior to final publication. Citation information: DOI 10.1109/TPDS.2021.3085210

delayed stores fill the SB above half its capacity, the prediction for each committing store is taken from its local history by looking up the SCH: the store is delayed if its SCH bit is set. Fig. 3 illustrates how the GCH, the SCH and the current store buffer occupation determine store handling in DeTraS (blue-shaded SB entries represent a delayed stores). Fig. 3-a shows that stores are never delayed if no recent contention was caused or suffered ($GCH = 0$). This avoids penalizing transaction duration by keeping the SB drain latency observed by `xend` unaffected. Fig. 3-b shows that in the presence of contention ($GCH > 0$), all transactional stores below the 50% SB occupation threshold are invariably delayed, emulating a *committer-wins* design for small-footprint transactions whose speculative writes can be fully contained within the processor structures until commit, in order to hide such writes from the coherence substrate and thus overcome their exposure to aborts that causes the friendly-fire pathology. Once SB occupation threshold grows above the threshold, the PC-based SCH table determines whether a committing store must be delayed, as shown in Fig. 3-c.

The extra SB snoop required by transactional stores reuses exactly the same logic as store-to-load forwarding. Contention between loads snooping the SB and committing stores is always resolved in favor of the former, delaying the retirement of the store until a SB port is available (often the next cycle). When a port is available, SB snoop and GCH/SCH access are done in parallel to determine the value of the *delay* bit. If unset, the store is immediately sent to cache. For efficiency purposes, two global bits are maintained by the SB: *delayedStores* and *needSnoop*. The former is set as the first delayed store enters the SB, while the latter acts as the enable signal to the SB snoop logic for transactional stores, and is set when a store that is not delayed enters the SB and finds *delayedStores* asserted. Both are cleared upon SB drain.



(a) No recent contention

(b) Recent contention, delay below SB occupation threshold (short transaction)

(c) Recent contention, selective delay above SB occupation threshold

(d) Overflow avoided by compaction

Fig. 3: Overflow avoided with compaction.

3.4 Store Buffer Compaction

In the previous sections it is assumed that when the SB is full and a delayed store sits at its head, a SB drain resumes the write of all delayed stores in order to prevent the processor from stalling. The selective delay of transactional stores opens up the possibility of a better utilization of SB resources through *compaction*, again driven by the observation that transactional stores can perform out of order. The only requirement for correctness is that stores to the same address or overlapped addresses cannot be reordered.

Thus, a delayed store sitting at the head of the SB can be moved to any previous completed entry in the SB as long as no overlapped stores exist in between both positions. In order to maximize the number of SB entries that can be released after each compaction, the entry selected as destination when moving a delayed SB entry should be the completed SB entry that lays closest to its tail (i.e., towards the last committed store).

Partially overlapping stores cannot be reordered. Because partially overlapped stores are rather infrequent, we opt for a conservative but simple approach to SB compaction that guarantees their correct handling: partially overlapping delayed stores at the head of the SB will prevent compaction. While other implementations exist, such as extending the SB entry with a *sentinel* pointer to restrict the portion of the SB suitable for movement, we found that our simple policy overcomes most SB overflows. On the other hand, stores that overlap completely can be *silently* coalesced. That is, when a committing transactional store fully overwrites another older committed store, the older store can be directly marked as completed, as it does not need to perform the write since the data will be completely overwritten by the newer store.

Implementation. Fig. 3-d shows how DeTraS avoids SB overflows by performing compaction when the SB is full, its head points to an entry whose *delay* bit is set, and there are completed stores in the SB that do not partially overlap with any older store. Compaction consists simply in copying the head entry to the completed SB entry closest to the tail of the SB and then freeing it (moving the head pointer). Fig. 3-d shows the state of the SB before and after compaction is done.

To do this efficiently, the SB keeps a *completedStores* counter that indicates the number of completed stores that are still occupying an SB entry. It is increased when a store is completed and decreased when its entry is freed (i.e., the SB head moves). Also, each SB entry is extended with two additional bits, *pinned* and *coalesced*, which are mutually exclusive, and get updated for the entries in the SB on each SB snoop. The *pinned* bit is set if the committing store partially overlaps the store in the SB entry and the *coalesced* bit is set if the committing store fully contains it.

Compaction cannot be performed if the *completedStores* counter is 0 or if the head SB entry has the *pinned* bit set. When a *coalesced* entry sits at the head of the SB, then *silent coalescing* is performed and the entry is considered completed, so the head pointer is just moved without further actions. In case of overflow,

© 2021 IEEE. Personal use is permitted, but republication/redistribution requires IEEE permission. See https://www.ieee.org/publications/rights/index.html for more information.

This article has been accepted for publication in IEEE Transactions on Parallel and Distributed Systems. This is the author's version which has not been fully edited and content may change prior to final publication. Citation information: DOI 10.1109/TPDS.2021.3085210

TABLE 1: System parameters.

Core Settings	
Cores	16 out-of-order (execute/commit width: 4)
Load queue	72
Store queue + store buffer	56
Memory Settings	
L1 I&D caches	Private, 32KiB, 8-way, 1-cycle hit latency
L2 cache	Shared, 8 MiB, unified, 16-way 24(tag)+12(data)-cycle latency
Memory	3GB, 200-cycle latency
Protocol	MESI, directory-based
Network Settings	
Topology and Routing	2-D mesh (4×4), X-Y
Flit size / Message size	16 bytes / 5 flits (data), 1 flit (control)
Link latency / bandwidth	1 cycle / 1 flit per cycle

TABLE 2: HTM systems evaluated.

System	Description
Base	Baseline, perfect read signature, SM-bits in L1 cache
DeTraS-D	Delayed Transactional Stores, always delay
DeTraS-R	Delayed Transactional Stores, always delay + reorder
DeTraS-C	Delayed Transactional Stores, selective delay + reorder + compact
MELSI	MESI + incoherent *Lazy* (L) state [7], [8], unlimited L-directory
ReqStalls	Baseline + *LogTM-like* [5] (timestamp-based) conflict resolution

the SB is not completely drained, but only the store at the head clears its delayed bit and gets resumed.

Note that our proposal does not need to block snoops on the SB when moving from one entry to another: if the snoop of a load races with the compaction (same cycle), it will either get the data from the source entry (if performed in the cycle immediately before the destination gets written), or from the destination entry itself (if in the same cycle).

4 Evaluation Methodology

Simulation environment. We have extended the widely used gem5 simulator [11] with transactional memory support, in order to model a variety of HTM implementations. We use the detailed timing model for the memory subsystem provided by Ruby, combined with the out-of-order processor model. Gem5 provides full-system functional simulation of the x86-64 ISA and boots an unmodified Ubuntu Linux 16.04 with kernel version 4.8.13. We perform our experiments on a 16-core tiled CMP system, as described in Table 1. Each tile contains a processing core with private L1 instruction and data caches, and a slice of the shared L2 cache with associated directory entries. A 2-D mesh NoC is employed to interconnect the tiles. The L1 caches maintain inclusion with the L2. The private L1 caches are kept coherent through an on-chip distributed directory (associated with L2 cache banks), which maintains bit vectors of sharers and implements the MESI protocol.

HTM systems evaluated. Table 2 summarizes the HTM systems evaluated in Section 5. Our baseline (*Base*) is an RTM-compliant best-effort design that uses SM bits in L1 data cache to track write sets, and a *perfect signature* to track read sets, so that it can maintain much larger read-sets than write-sets as seen in commercial chips [12], [13]. We implement DeTraS on top of our baseline without introducing any changes to the memory subsystem, except for a *conflict* bit on coherence responses. We evaluate three different flavors of DeTraS following the incremental refinements of Section 3. The version that delays all stores is denoted as *DeTraS-D*. *DeTraS-R* extends the delay approach to allow reordering of stores. *DeTraS-C* is our full-fledged design that selectively delays stores (using a 256-entry SCH table of 1-bit

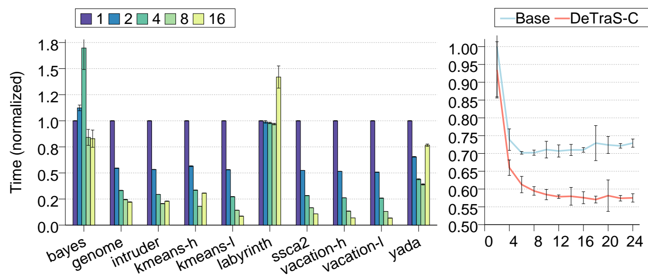


Fig. 4: Left: Execution time in Base up to 16 threads, normalized to sequential. Right: Sensitivity analysis for maximum number of retries before acquiring fallback lock.

predictors), reorders them and performs compaction. The remaining systems in Table 2 model related works and are described in Section 6.1.

Benchmarks and methodology. The STAMP benchmarks [14] with recommended medium inputs are used as workloads. The results presented are for 16-thread runs in all benchmarks. Fig. 4 (left part) shows the scalability up to 16 threads for our baseline, where we can see the poor parallel performance of bayes and labyrinth in all HTM systems as reported by other authors [12], [15], largely due to the large write sets of their transactions. Moreover, bayes implements a search algorithm that leads to highly random work levels executed for the same input [16]. In labyrinth, the inability to release its main data structure from the read set prevents concurrent transactions from committing useful work. Consequently, we exclude both benchmarks from our performance evaluation.

To isolate our evaluation from the effects of page-fault-induced aborts when accessing dynamically allocated data inside transactions, we implement a software scheme of heap pre-faulting [13] as part of our TM library function to begin a transaction, which follows [1] in regards to eager lock subscription. Our sensitivity analysis shown in Fig. 4 (right part) indicates that the optimal value for the maximum number of attempts to execute a transaction speculatively before taking the fallback lock differs between Base (6) and DeTraS (12); DeTraS benefits from a higher threshold because threads can make forward progress despite high contention, and resorting to the fallback lock after a small number of retries results in unnecessary loss of concurrency. Note how in DeTraS, performance is not hurt as the threshold value increases over 12 retries, an early indication that DeTraS infrequently resorts to the non-speculative path. In sight of this analysis, in Section 5 DeTraS-D and DeTraS-R employ the same maximum retry threshold as *Base* (6), while in DeTraS-C we set it to its optimal value (12). We also consider *DeTraS-R**, which uses the same threshold as DeTraS-C (12).

All the results correspond to the parallel part of the applications and we have accounted for the variability of parallel applications. For each workload-configuration pair we gather average statistics over 10 randomized runs, by adding a random jitter of up to 1 extra cycle to the DRAM response time. Threads were pinned to cores in order to avoid migration. The results shown in Section 5 are always normalized to Base and we use the arithmetic mean when summarizing results across benchmarks [17].

© 2021 IEEE. Personal use is permitted, but republication/redistribution requires IEEE permission. See https://www.ieee.org/publications/rights/index.html for more information.

This article has been accepted for publication in IEEE Transactions on Parallel and Distributed Systems. This is the author's version which has not been fully edited and content may change prior to final publication. Citation information: DOI 10.1109/TPDS.2021.3085210

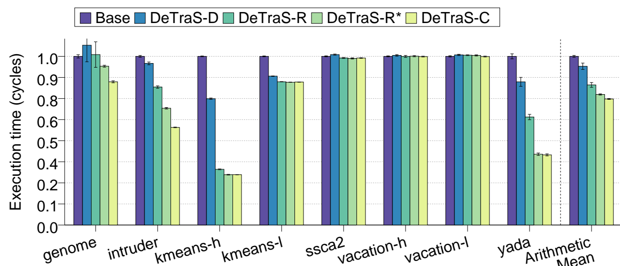


Fig. 5: Normalized execution time.

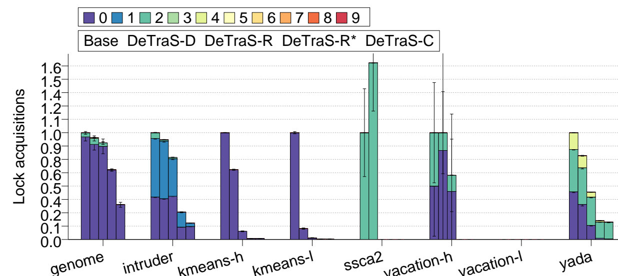


Fig. 6: Acquisitions of the fallback lock, per *xid*.

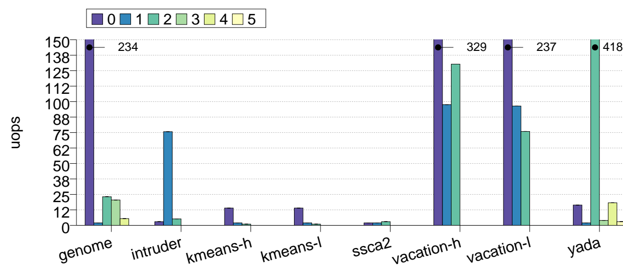


Fig. 7: Store micro-ops in STAMP, per transaction *xid*.

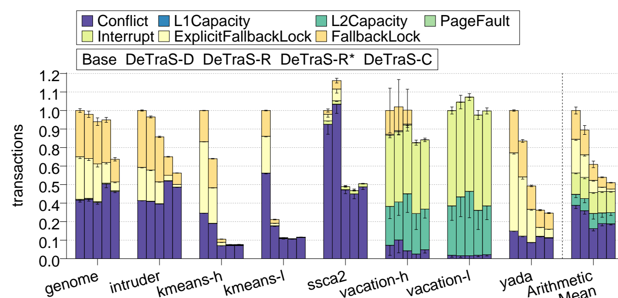


Fig. 8: Aborted transactions by cause.

5 Performance Evaluation

Fig. 5 compares the relative execution time of the three incrementally-refined configurations of DeTraS, relative to Base. Behind the improvement achieved by DeTraS lays its ability to cope with high contention and achieve forward progress without having to resort to non-speculative execution, *i.e.*, without having to revert to mutual exclusion when accessing shared data. Fig. 6 shows the number of acquisitions of the fallback lock in all DeTraS configurations normalized to Base, broken down per transaction in the source code of the benchmark (we assigned them a unique identifier –*xid*–, in program order). Fig. 7 shows the average number of store micro-ops for each *xid* in each benchmark. Note that x86-64 instructions are decoded into RISC-like instructions internally used by gem5's CPU models. Fig. 8 shows the total number of aborts, broken down by cause. Figs. 6, 7 and 8 jointly provide a radiography of each benchmark, by depicting which transactions resort to the fallback lock because of contention (*e.g.*, genome's xid_0 and xid_2, intruder's xid_0 and xid_1, kmeans' xid_0, yada's xid_0 and xid_2) or capacity-induced aborts (vacation's xid_0). Additionally, Fig. 9 shows a breakdown of conflict-induced aborts, with categories named as *killed-killer* (*e.g.*, $W-R$ means *writer killed by reader*) and writers are further split according to the source of the write to cache: xend (W_{xend}), overflow ($W_{overflow}$) or not delayed (W). Results in Fig. 9 correspond to the *Conflict* category of Fig. 8. Note that the high variability exhibited by vacation-l and vacation-h in Fig. 6 and Fig. 9 comes from the fact that conflict-induced aborts are uncommon, accounting only for 3 to 7% of all aborted transactions, respectively. The low level of contention in vacation allows it to achieve close to ideal scalability even in Base, as seen in Fig. 4, leaving little room for improvement. Similarly, techniques aimed at mitigating conflicts have little effect on the overall performance of ssca2, which is also a low-contention benchmark where aborts are very infrequent and transactional execution only account for less than 10% of all executed cycles (see Fig. 11 in Section 6). In the following analysis we will focus in the remaining benchmarks (genome, intruder, kmeans and yada), which have moderate to high contention. We begin with an overview of the results, and then we analyze the effectiveness of the mechanisms that comprise DeTraS.

In Fig. 5 we see that the naive approach of DeTraS-D already improves execution time for contended transactions that contain only a few stores such as intruder's xid_2, kmeans' xid_0 and yada's xid_0, which can be buffered in the SB without running into overflows. Simply delaying SB drain mitigates the friendly-fire pathology that affects Base in kmeans and yada, improving executing time by 10, 12 and 20% in kmeans-l, yada and kmeans-h, respectively, due to roughly 90, 20 and 30% less fallback lock acquisitions. However, this technique by itself hurts performance in genome and barely helps intruder. By introducing reordering, DeTraS-R virtually eliminates lock acquisitions in both kmeans configurations (95-99% of those seen Base) and by completely resolving the friendly fire pathology, DeTraS-R achieves an impressive reduction in execution time of 63% for kmeans-h over Base. DeTraS-R also resorts to the non-speculative path less often than DeTraS-D in intruder's xid_1 and yada's xid_0 with the consequent reduction in execution time. Once the maximum retry threshold is raised, we see DeTraS-R* ability to commit speculative transactions despite high contention, effectively capturing the benefits of the *committer-wins* policy found in systems with lazy conflict resolution. The reduction in lock acquisitions achieved by DeTraS-R* in yada and intruder (75-80%) explains the notable performance gains seen in Fig. 5 for these two benchmarks (24 and 56% improvement over Base, respectively). These results make DeTraS-R, given its simplicity, an appealing design choice that largely mitigates livelocks in a requester-wins HTM system for contended transactions whose stores can be accommodated in the SB. By contrast, DeTraS-C addresses this limitation and further lessens SB overflows in transactions that exceed SB capacity (*e.g.*, genome's xid_0 and intruder's xid_1, as seen in Fig. 7), thus minimizing friendly fire also in coarse grain transactions that experience high contention (see Fig. 6). By narrowing the window of exposure to aborts through efficient utilization of the SB,

© 2021 IEEE. Personal use is permitted, but republication/redistribution requires IEEE permission. See https://www.ieee.org/publications/rights/index.html for more information.

This article has been accepted for publication in IEEE Transactions on Parallel and Distributed Systems. This is the author's version which has not been fully edited and content may change prior to final publication. Citation information: DOI 10.1109/TPDS.2021.3085210

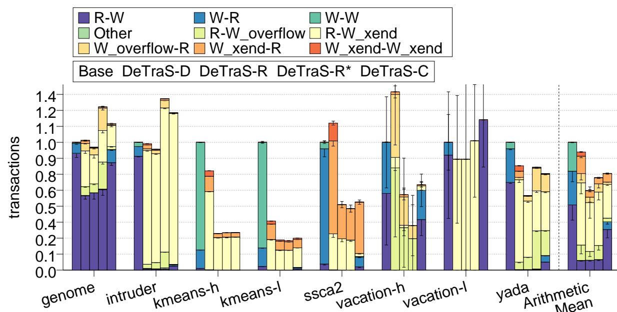


Fig. 9: Distribution of types for conflict-induced aborts.

DeTraS-C reduces execution time by 25% over Base on average.

5.1 Delaying all stores

When SB capacity is enough to accommodate all transactional stores, friendly fire experienced by highly contended transactions that subsequently read, modify and write (RMW) a set of memory locations (*e.g.*, kmeans) can be mitigated by delaying all stores. Fig. 9 shows that DeTraS-D reduces the number of conflict-induced aborts suffered in kmeans-l by 60% through the elimination of nearly all futile $W-W$ aborts suffered in Base, where transactional RMW operations eagerly request exclusive ownership (migratory pattern). We see that by delaying stores, most aborted transactions are now readers killed by "committing" stores ($R-W_{end}$), with a smaller fraction caused by the inverse scenario ($W_{end}-R$). On the downside, DeTraS-D penalizes low-contended transactions due to the extra SB drain latency on `xend`, which increases transaction duration thus its probability of conflict. The additional duration of each transaction hurts performance in several low-contention phases of genome, in spite of leaving the number of aborts nearly unaffected (see Figs. 8 and 9). The extra commit latency also results in more $W_{xend}-W_{xend}$ aborts seen for ssca2 in Fig. 9 when compared to Base).

5.2 Reordering stores

By allowing store misses during SB drain to be served concurrently and out of order, DeTraS-R shrinks the window of contention and consequently reduces $W_{xend}-*$ and $*-W_{xend}$ aborts seen for DeTraS-D, as can be observed in Fig. 9 for nearly all benchmarks, most notably in those that contain transactions with few stores, *e.g.*, intruder (xid_0), kmeans, ssca2 and yada (xid_0 and xid_4). The higher maximum retry threshold before taking the fallback path used in DeTraS-R* increments conflict-induced aborts compared to DeTraS-R in high contention benchmarks where SB overflows still occur (*e.g.*, genome and intruder and yada), but in turn reduces the number of fallback-lock-induced aborts as depicted in Fig. 8. This produces a somehow counter-intuitive result: allowing more retries after conflict-induced aborts may reduce the overall number of aborts (by reducing fallback-lock-related aborts) and ultimately improves performance. This is because all concurrent transactions are aborted upon acquisition of the fallback lock and force to wait until the non-speculative transaction completes. This serialization is particularly harmful for transactions working on unrelated data structures which had no potential conflicts with the lock owner. However, since such *false* lock-induced conflicts are indistinguishable from *true* data conflicts, they contribute to reaching the maximum retry threshold faster. Consequently, resorting to non-speculative execution to deal with contention in one data structure may exacerbate contention in other transactions that access unrelated data structures, as it happens for instance in intruder (xid_0 and xid_1). Note that such false fallback-lock-induced aborts are unavoidable, as employing a single fallback lock is the only way to achieve progress while guaranteeing atomicity in all cases without affecting the programming model nor requiring complex dependence analysis on the compiler side. When the maximum retry threshold is increased from DeTraS-R to DeTraS-R*, Fig. 9 also shows that $W_{overflow}-R$ and $R-W_{overflow}$ become a more pronounced fraction of all conflict-induced aborts in benchmarks whose contended transactions exceed SB size like genome, intruder and yada.

5.3 Selectively delaying stores. SB compaction

In Figs. 8 and 9 we can observe that DeTraS-C reduces both fallback-lock- and conflict-induced aborts seen in DeTraS-R* for genome, intruder and yada, precisely those benchmarks whose high-contention transactions comprise too many stores to be fully contained in the SB (Fig. 7). Fig. 9 shows that DeTraS-C eliminates virtually all SB overflow-induced aborts in genome and intruder but not in yada, because of the huge number of stores of its xid_2. The reduction in the total number of conflict-induced aborts achieved by DeTraS-C in absolute numbers does not reflect its actual impact on performance: On the one hand, part of the $W_{overflow}-R$ and $R-W_{overflow}$ aborts in DeTraS-R* simply do not occur in DeTras-C as the latter enables higher reader-writer concurrency, giving readers a better chance of committing before the writer. On the other hand, the remaining $R-W_{xend}$ aborts in DeTraS-C have a high probability of being useful aborts caused by a transaction that commits successfully. This explains the important reduction in fallback lock acquisitions observed for genome's xid_0. In intruder, we see that the removal of $R-W_{overflow}$ and $W_{overflow}-R$ aborts in its main transaction (xid_1) cuts the number of non-speculative executions of this transaction to one-quarter of those done by DeTraS-R*. Note that a fraction of such SB overflow-induced aborts in DeTraS-R* transits to the categories $R-W$ and $W-R$ (mispredicted-as-non-conflicting stores) in DeTraS-C, which can be attributed to SCH predictor warm up, and also to the reactive design of the GCH (recall that DeTraS-C does not delay stores if the GCH indicates no recent contention). In summary, the prevalence of $R-W_{xend}$ aborts in DeTraS-C is the quantitative proof of DeTraS' ability to consistently achieve forward progress under contention while avoiding serialization of threads on the fallback lock.

5.4 Area and energy considerations

The area overhead of DeTraS-C is very modest: Given the 56-entry SB considered, the total number of additional bytes is around 50: 168 extra bits in the SB (three bits per SB entry, *delayed*, *pinned* and *coalesced*), 256 bits for the PC-based predictor (SCH), plus a handful of small counters (4-bit GHC, 6-bit *completedStores*) and several flags. The SCH is a tagless direct-mapped small structure whose energy consumption is negligible. Compared to the energy incurred by a SB snoop (reading up to 56 tagged entries with a priority decoder), accessing a 1-bit predictor for each retired transactional store is an inexpensive operation. Therefore, we focus on the extra SB snoops that DeTraS-C performs in order to prevent reordering of overlapped stores.

© 2021 IEEE. Personal use is permitted, but republication/redistribution requires IEEE permission. See https://www.ieee.org/publications/rights/index.html for more information.

This article has been accepted for publication in IEEE Transactions on Parallel and Distributed Systems. This is the author's version which has not been fully edited and content may change prior to final publication. Citation information: DOI 10.1109/TPDS.2021.3085210

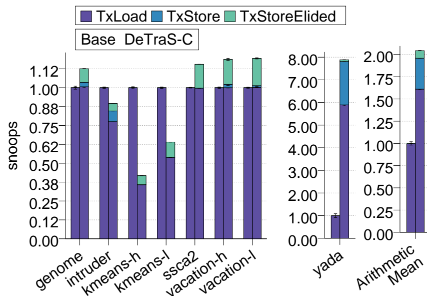


Fig. 10: Left: SB snoops done by transactional loads and stores.

Fig. 10 (left part) shows that the number of extra store-induced SB snoops incurred by DeTraS-C is small. The *TxStoreElided* component indicates stores whose snoop was elided thanks to the *needSnoop* bit. We can see that in kmeans and ssca2, all transactional stores elide the snoop, while the presence of both delayed and not delayed stores in the remaining benchmarks requires a fraction of all transactional stores to snoop the SB. Nonetheless, the total number of snoops grows only slightly (2-3%) in genome and vacation, while in intruder and kmeans is less than in Base as a result of having less aborts.

In the case of yada's large foot-print, long-running xid_2, DeTraS-C speculatively executes much larger fractions of it than Base while avoiding conflicts. The additional reader-writer concurrency achieved by DeTraS-C leads to much more work being speculatively executed compared to Base, explaining the huge difference in the number of transactional loads seen in Fig. 10. By exploiting available parallelism more aggresively, DeTraS-C reduces execution time by 60% but this comes at the cost of more speculative work getting discarded than Base (see Fig. 11). Thus, a fraction of the SB snoops seen for DeTraS-C in yada correspond to loads from transactions that abort much later than in Base, while another fraction is for loads from committed transactions which would anyways snoop the SB in Base when the same loads are executed non-speculatively (not shown in Fig. 10). Lastly, apart from lowering energy consumption by shortening execution time in high contention workloads, DeTraS also reduces the number of aborts by almost 50% on average (see Fig. 8).

6 Related Work: Discussion & Evaluation

The works by Armejach *et al.* [7] and Park *et al.* [8] are most related to this work, as they leverage the observation that delaying transactional writes can mitigate friendly fire in requester-wins HTM designs. *ForgiveTM* [8] shares many similarities with *WriteBurst* [7], as both propose to delay acquisition of exclusive ownership for transactional store misses while using the L1 cache to keep *incoherent* copies of speculatively modified blocks with deferred coherence actions. Their fundamental difference with DeTraS is the level at which delaying and reordering of transactional writes happens in the architecture: while DeTraS leverages the SB itself to alter the order in which stores are presented to the memory system, *WriteBurst* and *ForgiveTM* write to cache in program order (baseline is TSX), and then change the order in which exclusive coherence permissions for speculatively written blocks are requested. Though conceptually this may seem a subtle difference, the overhead brought by prior approaches is far greater than that of DeTraS: as a result of neglecting the presence of the SB, prior works must introduce new structures in the memory system, often duplicating functionalities already available in the processor (*e.g.*, wait for completion of all transactional stores in the SB before `xend` can commit). In particular, delaying transactional writes in the memory system involves: i) adding a new coherence state that relaxes the single-writer invariant for write-set blocks; ii) a directory-like structure for tracking cached blocks with pending coherence actions (henceforth referred to as *L-directory*); and iii) a scheme to select the appropriate victim upon overflow of the aforementioned L-directory. Considering the additional structures required by *ForgiveTM* and their size, and its extra *lazy* bit in cache, its storage overhead adds up to more than 4 kilobits: 640 bits for the 16-entry L-directory, over 3 kilobits bits for the 64-entry scoring table of $\{address, score\}$ pairs, plus 512 bits in the 32 KiB L1 data cache. This is roughly 10 times the area overhead of DeTraS-C, not including ForgiveTM's logic to traverse the L-directory at commit, nor the extensions in the L1 cache controller for the special handling of transactional write misses. Given that the integration cost into existing memory systems is the key motivation behind the choice of requester-wins in existing processors, we believe this extra complexity to tackle the limitations of such policy ultimately jeopardizes its simplicity appeal. In contrast, DeTraS leaves private caches and the existing coherence protocol almost completely unmodified; its only requirement is that a *conflict* signal gets delivered by the coherence protocol upon completion of each write miss in cache, so that it is used to update both local and global conflict history. The implementation complexity of a single bit piggybacked in invalidation acknowledgments and data responses, which merely gets routed from the L1 cache controller to the SB, is trivial.

Apart from their higher area overhead and complexity, existing approaches have two notable limitations. On the one hand, prior works invariably delay all stores as long as L-directory capacity allows, which increments commit latency unnecessarily for transactions in low contention, widening the window of vulnerability. On the other hand, previous proposals never delay transactional writes that hit on dirty blocks (*i.e.*, blocks in MESI M-state), regardless of the history of conflicts for the block; using the coherence state as a proxy of the block's probability of conflict can help minimize pressure on the L-directory (by filtering writes to thread-local or non-actively shared data), but may expose to aborts when consuming data in L1 cache that has been produced by an earlier transaction running in the same processing core.

Improving concurrency in requester-wins best-effort HTMs has also been the subject of other prior works [18], [19], [20], [21], [22], [23]. The goal of Dice *et al.*'s *Power Transactions* [23] is to reduce the frequency at which contended transactions resort to the fallback path, by prioritizing a speculative transaction over the rest. To do so, the coherence protocol must support *nacks*, so that power transactions invert the conflict resolution policy from requester-wins to *requester-loses*. Like DeTraS, others have proposed hybrid (eager/lazy) policies to improve concurrency of eager HTM designs [7], [8], [24], [25], [26]. In this regard, the work by Lupon *et al.* [27] and Titos *et al.* [28] share some similarities with our work, as they apply the same strategy of handling transactional stores eagerly or lazily depending on their contention characteristics. Unlike DeTraS, both *DynTM* and *ZEBRA* are virtualized HTM systems built atop LogTM (hence requester-stalls) that extend it to handle lazily managed writes, and basically differ from each other in the granularity at which policy is selected (transactions vs cache blocks). Other authors have also proposed extending LogTM with side buffers to maintain *nacked*

© 2021 IEEE. Personal use is permitted, but republication/redistribution requires IEEE permission. See https://www.ieee.org/publications/rights/index.html for more information.

This article has been accepted for publication in IEEE Transactions on Parallel and Distributed Systems. This is the author's version which has not been fully edited and content may change prior to final publication. Citation information: DOI 10.1109/TPDS.2021.3085210

transactional stores, so as to allow the in-order processor model to execute past conflicting stores [29], [30]. The Rock processor [9] leveraged the SB to implement minimalistic best-effort HTM support, since L1 caches did not support speculative versioning. In contrast, ASF [10] allowed speculative stores to remain in the SB until commit under certain conditions to maximize speculative buffering capacity beyond the L1 cache limits.

Memory ordering. This is the first work to realize transactional store reordering within the SB of a TSO processor in a completely transparent manner to the memory consistency model. Most prior works on HTM either do not model an out-of-order core [7], [29], [30] or if they do, the SB is disregarded so that reordering of coherence actions for written blocks is entirely done by private caches [8]. The implementation of DeTraS would be simplified in processors with weak models (*e.g.*, store coalescing makes compaction unnecessary).

6.1 Performance comparison to related works

We quantify the relative performance of DeTraS against the closest prior works [7], [8] by means of our *MELSI* model. *MELSI* extends the underlying MESI protocol of our baseline with a new state, *Lazy* (L), which allows transactional store instructions to complete in cache and retire from the processor without requiring an exclusive copy of the block (exclusive ownership is still brought by M and E states). In this way, transactional stores to shared blocks perform immediately, the block transits from S to L state, and its address is added to an unlimited-sized L-directory. Since there are no replacements in this idealized L-directory, *MELSI* does not implement any prediction scheme to select replacement victims (*i.e.*, no *conflict set signature* [7] nor *scoring mechanism* [8]). When the processor attemps to retire `xend`, the L1 cache controller sends coherence upgrade requests for each address in the *L-directory* (all requests sent in parallel). Transactional stores to blocks in M- and E- states perform as usual as they do not need any further coherence actions. Finally, transactional stores that do not find a copy of the line in L1 cache attempt to fetch a shared copy from the L2 cache, though this request may be serviced with an exclusive copy of the block if the L2 finds that there are no privately-cached copies in other caches. Note that this feature found in our underlying MESI protocol allows write misses to thread-local data (*e.g.*, misses resulting from gang-invalidation of write-set blocks in a previous abort, also known as *contamination misses* [31]) to bypass the L-state, and thus do not penalize the commit latency unnecessarily. Our sensitivity analysis of the maximum number of retries before taking the fallback path indicates that MELSI achieves its best performance with higher values than Base, for the same reasons as DeTraS (improved ability to achieve forward progress without resorting to serialization of threads to the fallback lock). Consequently, we use the same threshold in MELSI as in DeTraS (12).

Additionally, we compare DeTraS against prior works which avoid friendly fire by implementing a *requester-stalls* [2] policy. This requires adding *nacks* into the coherence protocol, as well as a mechanism to prevent deadlocks. Note that implementing nacks in certain scenarios (*e.g.*, snoop-based protocols) might be challenging, limiting the applicability of such policies. To maintain a fair comparison across all HTM designs considered, we deliberately choose not to compare against LogTM-SE [6] (virtualized HTM system); instead, *ReqStalls* implements LogTM's policy atop our best-effort baseline by appending 32-bit timestamps to all coherence messages, following its same deadlock avoidance scheme [5], and preventing the *starving writer* pathology by allowing an older writer to simultaneously abort concurrent younger readers [2].

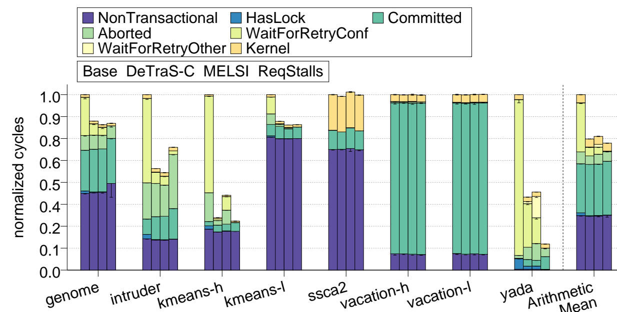


Fig. 11: Normalized execution time breakdown.

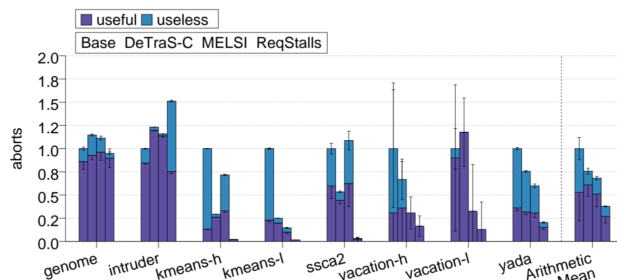


Fig. 12: Distribution of conflict-induced aborts.

Fig. 11 compares the execution time breakdown of Base and DeTraS-C against the aforementioned related works; each execution cycle is attributed to one of the following categories: non-transactional (*NonTransactional*); handling of interrupts/page-faults (*Kernel*); holding the fallback lock (*HasLock*); executing speculative transactions (*Committed* and *Aborted*); waiting on the fallback lock before retrying (where *WaitForRetryConf* corresponds to acquisitions after repetitive conflicts, and *WaitForRetryOther* for the rest). Fig. 12 shows the ratio of futile aborts for the HTM systems shown in Fig. 11, normalized to Base.

MELSI. Fig. 11 shows that with roughly 400 extra bits added to the SB, DeTraS-C outperforms (by 2% on average) an idealized implementation of MELSI in which the number of L-state blocks is only limited by the size/associativity of the L1 data cache itself. Note that for a 32 KiB L1 data cache, this would require a 512-entry L-directory, whose storage overhead would be close to 20 kilobits, two orders of magnitude larger than our proposal. It is interesting to note that how MELSI introduces a 2% performance penalty over Base in ssca2 due to the extra commit latency that MELSI invariably imposes because of its fixed *delay-always* policy in transactions with small write-sets. In this way, low contention workloads see the transaction duration increased by MELSI, which is also reflected in higher probability of conflict and thus many more futile aborts than DeTraS-C, as seen in Fig. 12. The performance advantage of MELSI over DeTraS-C in genome and intruder (2-3%) comes from the unlimited L-directory of our idealized model of MELSI, which results in its lower number of conflict-induced aborts contended transactions that exceed SB size: the unlimited L-directory makes these transactions completely invulnerable to conflicts until commit time, whereas limited SB resources and mispredictions in DeTraS-C occasionally lead to conflicting writes being unveiled to other

© 2021 IEEE. Personal use is permitted, but republication/redistribution requires IEEE permission. See https://www.ieee.org/publications/rights/index.html for more information.

This article has been accepted for publication in IEEE Transactions on Parallel and Distributed Systems. This is the author's version which has not been fully edited and content may change prior to final publication. Citation information: DOI 10.1109/TPDS.2021.3085210

caches prematurely. On the other hand, DeTraS-C exhibits higher performance in kmeans-h, with around 60% less aborts that lead to a 25% reduction in execution time over MELSI. MELSI still suffers many futile aborts in a high-contention workloads such as kmeans-h because of its aggressive fully parallel drain of the L-directory, which at times causes two concurrently committing transactions with mutual conflicts to provoke each other's abort, as each transaction wins the race at the directory for a different cache line. In DeTraS, writes to cache are still done on a store-by-store basis, and the presence of multiple stores to the same cache block in kmeans (array of cluster centers) makes exclusive coherence requests slightly more spaced in time, minimizing mutual aborts upon SB drain. Furthermore, DeTraS-C also outperforms MELSI in yada, in this case because MELSI suffers more capacity-induced aborts (*i.e.*, aborts due to the eviction of speculatively modified blocks from L1 cache). This kind of aborts seen in yada are less frequent in DeTraS-C because it takes advantage of the space in the SB to maintain some of its many transactional writes away from cache, effectively alleviating the pressure on L1 cache speculative buffering capacity.

ReqStalls. The results presented for ReqStalls must be analyzed keeping in mind that it models unrealistic sizes for the timestamps that get appended to coherence messages. More realistic timestamp sizes may harm performance in various ways, as timestamp wraparound can lead to priority inversion or even livelock. Note that extending the coherence messages with timestamps also incurs a fixed overhead that hinders energy efficiency in non-transactional workloads.

As anticipated, Fig. 11 shows that introducing priorities among transactions results in an HTM design that tolerates contention better than requester-wins sytems, for two reasons: First, priorities allow resolving some conflicts by stalling the requester transaction, which can resume execution once the other transaction(s) commits. In benchmarks like kmeans and ssca2, where cyclic dependencies among transactions are infrequent, ReqStalls resolves nearly all conflicts without resorting to aborts, as we can see in Fig. 12; in vacation and yada, cycles do arise, yet aborts are still considerably reduced by ReqStalls in comparison to requester-wins designs. The second key advantage of supporting transaction priority in hardware is that, even when a cyclic dependence is detected and conflict-induced abort occurs, the software abort handler can retry without ever resorting to mutual exclusion, since the transaction will eventually become the eldest (highest priority) transaction and will be guaranteed to survive conflicts with any other younger transactions. As a result, we can see in Fig. 11 that ReqStalls does not have a *HasLock* component in any of the benchmarks, which explains the superior performance demonstrated by this conflict resolution policy over MELSI and DeTraS-C in yada. In this benchmark, all requester-wins designs fall back to non-speculative execution to achieve progress, consequently bringing a performance penalty depicted by the *WaitForRetryConf* component, which ReqStalls simply skips. Note that cycles spent in the software handler after a conflict-induced abort are attributed to this category, and in ReqStalls only become visible in intruder because of many repeated aborts.

Remarkably, DeTraS-C outperforms ReqStalls in a highly contended benchmark such as intruder. This unexpected result has its roots in ReqStalls' ability to resolve conflicts through stalls rather than aborts, which proves advantageous to performance in most benchmarks, but also suffers from its own pathological behaviour in certain contended workloads where cyclic dependencies arise very frequently. Hence, the *futile stall* pathology [2] can become a significant overhead for this policy by increasing aborted cycles as a consequence of more useless aborts, as seen in Fig. 12 for intruder. From a bird's-eye view, we can see that the eager approach to resource acquisition used by ReqStalls limits reader-writer concurrency, whereas DeTraS-C tends to acquire exclusive owernship over contended resources when the transaction is ready to commit. This prevents the *blocking effect* that causes serialization of stalled threads in requester-stalls.

7 Conclusions

This work underlines the impact that appropriate management of transactional stores in the store buffer (SB) of an out-of-order processor has on HTM performance, and proposes simple behavioural changes to the SB found in TSO processors that allow requester-wins HTM systems to cope with friendly fire. DeTraS leverages large store buffers present in modern commercial processors, often underutilized by transactional workloads which exploit irregular parallelism and work on sparse data structures. With minimal extensions to the SB (about 50 extra bytes), DeTraS is a low-complexity, practical extension over currently commercially available best-effort HTM systems. Our work proposes an HTM-aware SB design whose visible change outside the processing core is the fact that speculative writes to the L1 data cache will be seen in a different order. By maximizing concurrency under contention while keeping the memory subsystem unmodified, our proposal may result more compelling to chip manufacturers. DeTraS combines the appeal of simplicity brought by requester-wins, with the ability to provide progress despite contention of an HTM implementation with lazy conflict management. We show that our design improves performance with respect to the baseline architecture up to 68% in contended benchmarks and 25% on average across the STAMP benchmark suite.

Acknowledgments

Work supported by the Spanish MCIU and AEI, as well as European Commission FEDER funds, under grant RTI2018-098156-B-C53; the Spanish MCIU under grant ERC2018-092826; and the European Research Council (ERC) under the Horizon 2020 research and innovation programme (grant agreement No 819134).

References

[1] Intel Corporation, “Intel 64 and IA-32 architectures optimization reference manual, chapter 16: Intel TSX recommendations,” pp. 16:1–16:30, 2020.

[2] J. Bobba, K. E. Moore, L. Yen, H. Volos, M. D. Hill, M. M. Swift, and D. A. Wood, “Performance pathologies in hardware transactional memory,” in *34th Int'l Symp. on Computer Architecture (ISCA)*, 2007, pp. 81–91.

[3] R. Rajwar and J. Goodman, “Transactional execution: Toward reliable, high-performance multithreading,” *IEEE Micro*, vol. 23, no. 6, pp. 117–125, 2003.

[4] R. Rajwar and J. R. Goodman, “Transactional lock-free execution of lock-based programs,” in *10th Int'l Conf. on Architectural Support for Programming Language and Operating Systems (ASPLOS)*, 2002, pp. 5–17.

[5] K. E. Moore, J. Bobba, M. J. Moravan, M. D. Hill, and D. A. Wood, “LogTM: Log-based transactional memory,” in *12th Int'l Symp. on High-Performance Computer Architecture (HPCA)*, 2006, pp. 254–265.

[6] L. Yen, J. Bobba, M. R. Marty, K. E. Moore, H. Volos, M. D. Hill, M. M. Swift, and D. A. Wood, “LogTM-SE: Decoupling hardware transactional memory from caches,” in *13th Int'l Symp. on High-Performance Computer Architecture (HPCA)*, 2007, pp. 261–272.

© 2021 IEEE. Personal use is permitted, but republication/redistribution requires IEEE permission. See https://www.ieee.org/publications/rights/index.html for more information.

This article has been accepted for publication in IEEE Transactions on Parallel and Distributed Systems. This is the author's version which has not been fully edited and content may change prior to final publication. Citation information: DOI 10.1109/TPDS.2021.3085210

[7] A. Armejach, R. Titos-Gil, A. Negi, O. S. Unsal, and A. Cristal, "Techniques to improve performance in requester-wins hardware transactional memory," *ACM Transactions on Architecture and Code Optimization (TACO)*, vol. 10, no. 4, pp. 42:1–42:25, 2013.

[8] S. Park, C. J. Hughes, and M. Prvulovic, "Forgive-tm: Supporting lazy conflict detection in eager hardware transactional memory," in *28th Int'l Conf. on Parallel Architectures and Compilation Techniques (PACT)*, Sep. 2019, pp. 192–204.

[9] S. Chaudhry, R. Cypher, M. Ekman, M. Karlsson, A. Landin, S. Yip, H. Zeffer, and M. Tremblay, "Rock: A high-performance Sparc CMT processor," *IEEE Micro*, vol. 29, no. 2, pp. 6–16, 2009.

[10] 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 *43rd IEEE/ACM Int'l Symp. on Microarchitecture (MICRO)*, 2010, pp. 39–50.

[11] N. Binkert, B. Beckmann, G. Black, S. K. Reinhardt, A. Saidi, A. Basu, J. Hestness, D. R. Hower, T. Krishna, S. Sardashti *et al.*, "The gem5 simulator," *ACM SIGARCH Computer Architecture News*, vol. 39, no. 2, pp. 1–7, 2011.

[12] R. Yoo, C. Hughes, K. Lai, and R. Rajwar, "Performance evaluation of intel transactional synchronization extensions for high performance computing," in *ACM/IEEE Conf. on Supercomputing (SC)*, 2013.

[13] 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 *28th Int'l Parallel and Distributed Processing Symp. (IPDPS)*, 2014, pp. 615–624.

[14] C. Cao Minh, J. Chung, C. Kozyrakis, and K. Olukotun, "STAMP: Stanford transactional applications for multi-processing," in *IEEE Intl. Symposium on Workload Characterization*, 2008, pp. 35–46.

[15] 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 *42nd Int'l Symp. on Computer Architecture (ISCA)*, 2015, pp. 144–157.

[16] A. Dragojevic and R. Guerraoui, "Predicting the scalability of an STM," in *5th ACM SIGPLAN Workshop on Transactional Computing*, 2010.

[17] J. E. Smith, "Characterizing computer performance with a single number," *Communications of the ACM*, vol. 31, no. 10, pp. 1202–1206, 1988.

[18] H. Litz, D. Cheriton, A. Firoozshahian, O. Azizi, and J. P. Stevenson, "SI-TM: Reducing transactional memory abort rates through snapshot isolation," in *19thInt'l Conf. on Architectural Support for Programming Language and Operating Systems (ASPLOS)*, 2014, pp. 383–398.

[19] L. Xiang and M. L. Scott, "Software partitioning of hardware transactions," in *20th Int'l Symp. on Principles & Practice of Parallel Programming (PPoPP)*, 2015, pp. 76–86.

[20] S. Park, M. Prvulovic, and C. J. Hughes, "PleaseTM: Enabling transaction conflict management in requester-wins hardware transactional memory," *22ndInt'l Symp. on High-Performance Computer Architecture (HPCA)*, pp. 285–296, 2016.

[21] R. Quislant, E. Gutierrez, E. L. Zapata, and O. Plata, "Enhancing scalability in best-effort hardware transactional memory systems," *Journal of Parallel Distributed Computing (JPDC)*, vol. 104, no. C, pp. 73–87, 2017.

[22] R. Quislant, E. Gutiérrez, E. L. Zapata, and O. G. Plata, "Lazy irrevocability for best-effort transactional memory systems," *IEEE Transactions on Parallel and Distributed Systems (TPDS)*, vol. 28, no. 7, pp. 1919–1932, 2017.

[23] D. Dice, M. Herlihy, and A. Kogan, "Improving parallelism in hardware transactional memory," *ACM Transactions on Architecture and Code Optimization (TACO)*, vol. 15, no. 1, pp. 9:1–9:24, 2018.

[24] S. Tomic, C. Perfumo, C. Kulkarni, A. Armejach, A. Cristal, O. Unsal, T. Harris, and M. Valero, "EazyHTM: Eager-lazy hardware transactional memory," in *42nd IEEE/ACM Int'l Symp. on Microarchitecture (MICRO)*, 2009, pp. 145–155.

[25] L. Zhao, W. Choi, and J. Draper, "SEL-TM: Selective eager-lazy management for improved concurrency in transactional memory," in *26thInt'l Parallel and Distributed Processing Symp. (IPDPS)*, 2012, pp. 95–106.

[26] A. Shriraman, S. Dwarkadas, and M. L. Scott, "Flexible decoupled transactional memory support," in *35th Int'l Symp. on Computer Architecture (ISCA)*, 2008, pp. 139–150.

[27] M. Lupon, G. Magklis, and A. González, "A dynamically adaptable hardware transactional memory," in *43rd IEEE/ACM Int'l Symp. on Microarchitecture (MICRO)*, 2010, pp. 27–38.

[28] R. Titos-Gil, A. Negi, M. E. Acacio, J. M. Garcia, and P. Stenstrom, "Zebra : A data-centric, hybrid-policy hardware transactional memory design," in *25th Int'l Conf. on Supercomputing (ICS)*, 2011, pp. 53–62.

[29] A. Negi, R. Titos-Gil, M. E. Acacio, J. M. Garcia, and P. Stenstrom, "Eager meets lazy: The impact of write-buffering on hardware transactional memory," in *40th Int'l Conf. on Parallel Processing (ICPP)*, 2011.

[30] R. Titos-Gil, A. Negi, M. E. Acacio, J. M. Garcia, and P. Stenstrom, "Eager beats lazy: Improving store management in eager hardware transactional memory," *IEEE Transactions on Parallel and Distributed Systems (TPDS)*, vol. 24, no. 11, pp. 2192–2201, 2013.

[31] M. Waliullah and P. Stenstrom, "Classification and elimination of conflicts in transactional memory systems," Chalmers University of Technology, TR 2010:09, Dept. of Computer Science, Tech. Rep., 2010.

Rubén Titos-Gil received MS and PhD degrees in Computer Science from the University of Murcia, Spain, in 2006 and 2011, respectively. As a PhD student, he was awarded a FPU scholarship from the Spanish Government. After holding post-doctoral positions at Chalmers University of Technology, Sweden, and at the Barcelona Supercomputing Center, Spain, in 2015 he rejoined the University of Murcia where he has since served as an adjunct professor. His research focuses on hardware support for synchronization in parallel processors, with an emphasis on transactional memory.

Ricardo Fernández-Pascual received the MS and PhD degrees in computer science from the Universidad de Murcia, Spain, in 2004 and 2009, respectively. In 2004, he joined the Computer Engineering Department as a PhD student with a fellowship from the regional government. In 2006, he joined the Computer Engineering Department of the Universidad de Murcia where he is currently an associate professor. His research interests include general computer architecture, memory hierarchies for parallel processors, and performance simulation.

Alberto Ros Alberto Ros is Associate Professor at the University of Murcia, Spain. He received the Ph.D. degree in computer science from the same university, in 2009, after being granted with a fellowship from the Spanish government to conduct the Ph.D. studies. He hold post-doctoral positions at the Universitat Politècnica de València and at Uppsala University. He has co-authored more than 60 research papers in international journals and conferences. His research interests include cache coherence protocols, memory hierarchy designs, and memory consistency for multicore architectures.

© 2021 IEEE. Personal use is permitted, but republication/redistribution requires IEEE permission. See https://www.ieee.org/publications/rights/index.html for more information.

This article has been accepted for publication in IEEE Transactions on Parallel and Distributed Systems. This is the author's version which has not been fully edited and content may change prior to final publication. Citation information: DOI 10.1109/TPDS.2021.3085210

Manuel E. Acacio is a Full Professor of computer architecture and technology at the University of Murcia, Spain. Dr. Acacio obtained his PhD degree in Computer Science in March 2003. Before, in the summer of 2002, he worked as a summer intern at IBM TJ Watson, Yorktown Heights (NY). Currently, Dr. Acacio leads the Computer Architecture & Parallel Systems (CAPS) research group at the University of Murcia. He is author of about 100 papers in refereed international conferences and journals. As well, he has served as a committee member of numerous international conferences. His research interests are focused on the architecture of multiprocessor systems. From April 2011 to April 2015, Dr. Acacio served as an associate editor of IEEE TPDS Journal, since August 2016 he is member of the editorial board of MPDI Computers Int'l Journal, and more recently, since September 2018, he serves as academic editor in the editorial board of Hindawi Scientific Programming journal. He is also member of the board of distinguished reviewers of ACM TACO Journal since May 2014.

© 2021 IEEE. Personal use is permitted, but republication/redistribution requires IEEE permission. See https://www.ieee.org/publications/rights/index.html for more information.