# WoperTM: Got nacks? Use them!

Víctor Nicolás-Conesa, Rubén Titos-Gil, Ricardo Fernández-Pascual, Manuel E. Acacio, Alberto Ros
University of Murcia
Email: *victor.nicolasc@um.es, rtitos@um.es, rfernandez@ditec.um.es, meacacio@um.es, aros@ditec.um.es*

***Abstract*—The simplicity of requester-wins has made it the preferred choice for conflict resolution in commercial implementations of Hardware Transactional Memory (HTM), which typically have relied on conventional locking to escape from conflict-induced livelocks. Prior work advocates for combining requester-wins and requester-loses to ensure progress for higher-priority transactions, yet it fails to take full advantage of the available features, namely, protocol support for *nacks*. This paper introduces WoperTM, a dual-policy, best-effort HTM design that resolves conflicts using *requester-loses* policy in the common case. Our key insight is that, since *nacks* are required to support priorities in HTM, performance can be improved at nearly no extra cost by allowing regular transactions to benefit from requester-loses, instead of only those involving a high-priority transaction. Experimental results using gem5 and STAMP show that WoperTM can significantly reduce squashed work and improve execution times by 12% with respect to *power transactions*, with negligible hardware overhead.**

***Index Terms*—Transactional memory, conflict resolution.**

## I. Introduction

HArdware Transactional Memory (HTM) can simplify parallel programming by shifting the complexity of synchronizing threads from the programmer to the hardware. The hardware then makes use of speculative execution to run critical sections from multiple threads concurrently. Speculative transactions can be executed and committed in parallel as long as no *conflicts* arise among them —a conflict occurs when two or more threads access the same shared memory location and at least one those accesses is a write—. Thus, HTM systems implement *conflict detection* (CD) and *resolution* (CR) mechanisms to handle these conflicts, aborting the transactions involved to guarantee correctness. When a transaction aborts, its speculative updates must be discarded as if it had not been executed at all. To this end, HTM systems implement a *version management* (VM) scheme. CD, CR and VM are the three main dimensions in the HTM design space [1], and there are alternatives for each of these dimensions [2], [3].

Existing *best-effort* implementations of HTM have opted for eager CD and lazy VM policies, given their easier integration into existing designs. Eager CD leverages the messages generated by the cache coherence protocol to detect conflict upon each memory access, while lazy VM takes advantage of private L1 caches to keep a transaction's speculative updates invisible to other threads. The additional complexity of lazy CD or eager VM disqualifies such policies from cost-effective HTM implementations in the context of best-effort HTM systems [3]. This leaves CR as the only major design dimension that computer architects may consider exploring.

Despite being prone to livelocks during contention [2], commercial HTM implementations appear to have opted for a *requester-wins* (RW) approach [3]. This choice of CR is part of the reasons why HTM support has only demonstrated performance benefits in lightly contended workloads [2], [3]. As contention raises and transactions begin *squashing* each other, performance quickly deteriorates, not only because of the work discarded, but also the need to switch to the *slow path* to ensure progress: the TM runtime resorts to non-speculative execution in mutual exclusion using a single global lock.

Among the alternatives to RW, the most relevant policies are *requester-loses* (RL) [4] and requester-stalls (RS) [1]. Both can offer better overall performance than RW during contention. This is particularly true for RS, as it can resolve conflicts through stalls rather than aborts, though its need for a deadlock avoidance scheme makes RS less appealing than RL for best-effort HTM. Unlike RW, the main downside of RL, shared also by RS, is that it requires extending typical cache coherence protocols with messages to allow for unsuccessful completion of requests. In directory-based coherence, supporting RL means introducing negative acknowledgments (*nacks*) sent in response to conflicting memory requests.

Prior work on best-effort HTM systems has demonstrated the performance benefits of supporting RL [5], [6]. Dice *et al.* propose a CR scheme in which a *power transaction* can survive conflicts by responding to offending requests with *nacks*, while regular transactions invariably favor the requester, effectively creating a dual-priority, RW/RL HTM design. The resulting design improves concurrency by reducing the frequency in which the *slow path* is used while experiencing contention, since an intermediate *power path* prioritizes a chosen transaction to guarantee progress without resorting to mutual exclusion. Considering that the protocol must be extended to support both RW and RL, a naturally occurring question is which of the two policies to use by default. Prior work has opted for RW, so that at most one transaction at a time benefits from *nacks*, while the rest will remain subject to the *friendly fire* pathology of RW.

Our key insight is that, if designers are willing to incorporate *nacks*, they can be leveraged more effectively than what has been proposed. In this work, we propose WoperTM[1], a best-effort HTM design built on the ideas of *power transactions*. WoperTM employs RL as the default policy, and only reverts to RW if the requester has higher priority. Our evaluation with gem5 and STAMP shows that WoperTM, requiring only one extra bit in coherence requests, consistently outperforms power transactions in contended workloads by minimizing discarded work from aborted transactions.

[1]A wordplay flipping the first two consonants of "Power", highlighting how our approach inverts the conflict resolution strategy of power transactions.

## II. Background and Related Work

Given the ability to deny invalidation or ownership transfer requests for a cache block under certain conditions, a simple approach to CR could handle all conflicts by sending a *nack* response, which would abort the requester. Different flavors of the *nack* response have been described in academia [5] and industry (e.g., AMBA CHI's Retry Ack response). Under this always-RL policy, transactions become more and more immune to aborts as their read and write sets grow larger, heuristically tending to favor transactions that have done more work at the point of the conflict, minimizing wasted work upon aborts. However, such a naive RL approach is subject to a different performance pathology which indefinitely hinders the progress of writer transactions attempting to modify a cache block that is accessed by many concurrent readers [2].

Preventing *starving writer* pathology in RL is possible as long as RW is also supported by the coherence substrate, simply by adopting a hybrid RL/RW policy that allows writers to simultaneously abort concurrent readers [2]. In today's multicores, RW can be supported without extending the coherence protocol, by leveraging the existence of two levels of private cache: Only minor behavioral changes in the private cache controllers are required to ensure that remote conflicting requests always obtain the non-speculative version of the cache block from L2. Therefore, support for *nacks* does not hard-wire RL as the only CR policy, but rather enables the system to dynamically choose between RL and RW at each conflict.

An alternative strategy when combining RW and RL is to give higher priority to *all* accesses made by a certain transaction. This is the approach followed by Dice *et al.* with *power transactions*, an example of hardware-software co-design that manages priorities with minimal ISA changes, in order to have two modes for running speculative transactions, regular or *power*. Using software-controlled entry into *power mode*, the complexity of determining whether it is safe to execute a transaction with high priority is shifted to the TM runtime —a regular lock ensures that at most one *power* transaction exists at a time—. The choice of CR policy made by private cache controllers then depends on the local *power mode* bit for that core. When a conflicting remote request is observed by the cache controller and the power mode bit is unset, the local transaction aborts (RW policy). On the other hand, if the local power mode bit is set, then the *speculative* status of the remote request must be considered to select the policy, since conflicts with non-speculative code cannot be resolved using RL, for safety. Thus, the proposal by Dice *et al.* had to augment coherence requests with an extra *speculative* bit to distinguish requests originating from transactions.

## III. WoperTM

Our observation is that employing RL as default CR policy is an alternative design to *power transactions* that makes better use of *nacks*. In this way, conflicts caused either by power transactions or by non-transactional code are resolved similarly, using RW as the CR policy, whereas conflicts among regular transactions are resolved using RL, in order to make the best out of the effort made to augment the coherence protocol to support *nacks*.

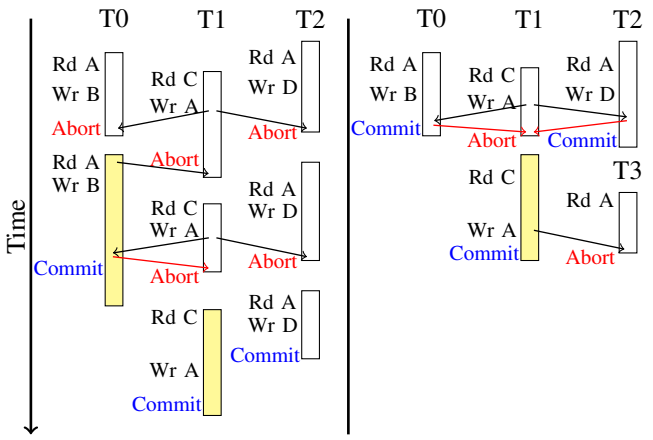


Fig. 1: Power versus WoperTM. Red arrows are *nacks*.

Fig. 1 illustrates the key behavioral differences between prior work (*PowerTM*) [6] (left) and our proposal, *WoperTM* (right). For this example, we assume that after a conflict-induced abort, threads will attempt to enter the *power mode*, if the token is available. The figure depicts a scenario where two threads run non-conflicting transactions T0 and T2 —readers of location A—, while another thread runs T1 and writes location A. In the case of PowerTM, the use of RW as default policy results in *friendly fire* among the three transactions: T1 aborts T0 and T2 when writing A, and then T1 is aborted after T0 (or T2) restarts and re-reads A. A subsequent restart of T1 will once again abort T2 when reading C, before T1 itself is squashed upon receiving a *nack* from T0 (now in power mode), which denied T1's write to A. T2 might be able to read A and commit as long as it completes before T1 writes A, which now commits in power mode. In the case of WoperTM, resolving conflicts using RL allows T0 and T2 to commit in parallel, since none of them are vulnerable to the conflicting write to A from T1. After being denied its write to A, T1 restarts in power mode and thus will win any potential conflicts that may arise with subsequent reader transactions (T3), effectively escaping the *starving writer* pathology.

Fig. 2 depicts the architectural changes required to support WoperTM, PowerTM and RL policies, compared to a baseline RW design. The *speculative* bit (S) that tracks whether the core is running a transaction is present in all four designs. To prevent *nacking* of non-transactional requests, the S bit must propagated to coherence requests generated in all cases except RW. PowerTM and WoperTM extend the HTM state with a *power mode* bit (P), and a new ISA instruction to begin a transaction in said mode. The key difference between PowerTM and WoperTM is that coherence requests generated by transactions must also carry the P bit in WoperTM, while in PowerTM this bit is never propagated outside the core. Thus, in WoperTM, conflicting remote requests observed by private cache controllers are resolved based on their P and S bits.

Any policy other than RW needs minor changes in both private cache controllers and the directory controller in order to account for the case where a request is denied due to a conflict, so that a coherence transaction can complete without

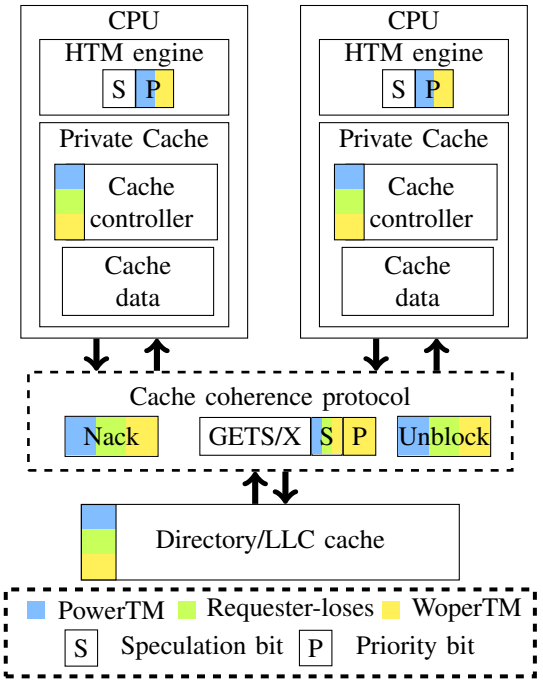


Fig. 2: Requirements of PowerTM, RL, and WoperTM

TABLE I: System parameters.

| | |
|---|---|
| Cores | 16, out-of-order, x86-64 |
| Struct. size | ROB: 512, LQ: 192, SQ: 114, IQ: 140 |
| L1I cache | Private, 32KiB, 8-way, 1-cycle hit latency |
| L1D cache | Private, 48KiB, 12-way, 1-cycle hit latency |
| L2 cache | Private, 1.25MiB, unified, 10-way, 4-cycle min. roundtrip |
| L3 cache | Shared, 32MiB, unified, 16-way 30-cycle min. roundtrip |

obtaining the required data/permissions. On the private cache controller side, besides the *nack* response itself, a new type of *nacked-unblock* message must be sent to the directory, so as to inform that the request did not succeed. When the directory controller receives such a *nacked-unblock*, it does not update the coherence state for the block nor its sharers/owner; instead, the coherence state for the block transits back to its initial state, as if the request had not been served.

## IV. Methodology and Evaluation

We use Gem5 in full-system mode to model an Intel RTM-like HTM system with lazy VM and eager CD, whose key parameters are shown in Table I. The L1 cache is used for speculative versioning, writing back non-speculative version to L2 before speculation when needed. Write sets are tracked using one bit per L1 cache line, while read-set blocks can reside in either private level. The replacement policy takes into account such bits to minimize capacity aborts.

We conducted our experiments using the STAMP benchmark suite, excluding *bayes* due to its high variability. We run 16-thread configurations with standard medium inputs.

Four CR flavours are evaluated: our proposed **WoperTM**, **PowerTM** [6] and their respective fixed-policy underlying design (**RL** and **RW**, respectively). PleaseTM [5] is a well-known CR policy that avoids the use of nacks and is outperformed by PowerTM [6] and nack-based RL implementations [5]. Furthermore, its lack of atomicity during validation re-fetch complicates the implementation of a priority-based mechanism like WoperTM due to parallel memory accesses from different cores. Consequently, we excluded it from our experiments.

A *fallback* global lock ensures progress despite capacity limitations or exceptions. Page-fault-induced aborts arising from STAMP's simplistic memory allocator are mitigated through prefaulting.For RL and RW, the abort handler takes the fallback lock after 6 and 10 retries, respectively. WoperTM and PowerTM employ identical abort handlers, where an extra lock serves as a *power token* that threads attempt to acquire after a conflict-induced abort.

Fig. 3 presents the execution time across the benchmarks split in four regions. No_Transactional represents the time the CPU spent executing code outside transactional regions. Committed and Aborted correspond to the time spent in transactional regions that resulted in a commit or an abort, respectively. Finally, Wait_Fallback_Lock measures the time that CPUs waited for another core to release the global lock. All values are normalized to RW. Fig. 4 illustrates the proportion of aborted cycles relative to total transactional cycles, categorized by atomic region in the source code (TID) to help us attribute performance differences among CR policies to specific data access patterns. As we can see in Fig. 4, ssca2 and vacation have very light contention levels and thus are unaffected by the choice of CR policy. In labyrinth, the lack of *early release* support in HTM limits available parallelism and leads lots of wasted transactional cycles.

The choice of CR policy impacts performance in workloads with moderate to high contention, such as genome, intruder, kmeans-h or yada. As we can see, WoperTM emerges as a robust alternative that balances efficiency and performance, consistently outperforming the state-of-the-art in all benchmarks. Overall, WoperTM stands out as the most efficient CR policy among the four configurations, with remarkable reductions in aborted cycles that lead to an average reduction in execution times of 12% when compared to PowerTM.

The best results for WoperTM are seen for yada, obtaining speedups of around 2x and 3x, respectively, when compared to PowerTM and RW. Despite its simplicity, WoperTM has a unique ability to exploit irregular parallelism available in a very large transaction with moderate contention, as is the case of yada's main transaction: TID-2 performs triangle refinement and often reaches read set sizes well above one hundred cache lines. WoperTM's performance stems from its two salient features, namely, its trend towards favoring transactions that have done more work (i.e., accessed more cache lines) and its ability to escape starving writer without aborting unrelated transactions. On their part, RW and PowerTM suffer a wide window of vulnerability to conflicts resulting from the very long transaction duration. In RL, the key bottleneck is wasted work stemming from the huge cost of entering the fallback path: when a writer repeatedly fails to make progress and acquires the fallback lock, it causes the squash of thousands of speculative cycles in most other threads. In contrast, in WoperTM, the power-mode writer only causes the abort of truly conflicting readers, while unrelated transactions working on independent parts of the mesh continue unaffected.

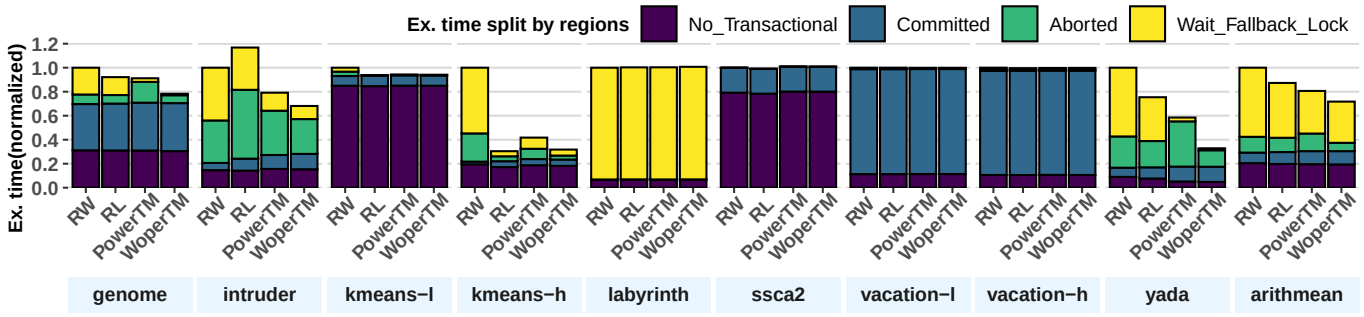


Fig. 3: Normalized execution time split in several regions.

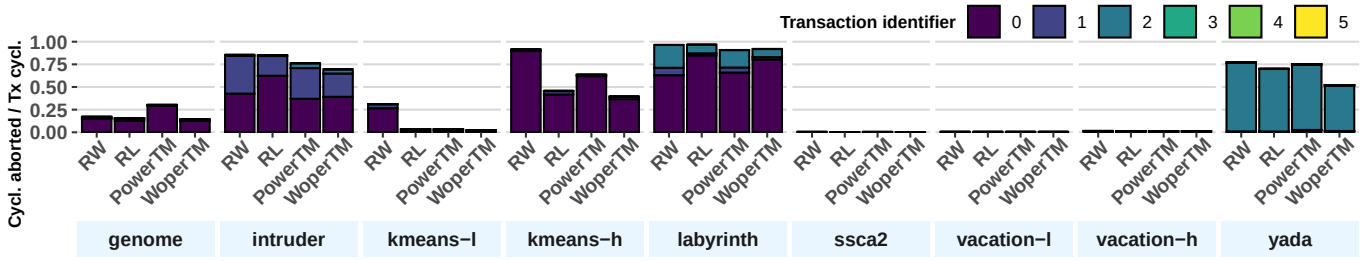


Fig. 4: Proportion of aborted transactional cycles (Cycles aborted / Total transactional cycles).

The case of genome showcases a key advantage of WoperTM: efficiency. Because priorities guarantee progress during contention, regular transactions never resort to the fallback lock and thus may be subject abort each other until each one of them eventually becomes a power transaction. This is perhaps the key shortcoming found in the state-of-the-art: the additional concurrency enabled often buys marginal performance gains at the cost of lots of wasted work. Precisely this happens in genome TID-0: Although PowerTM achieves slightly better performance than RL, by avoiding the switch to the fallback path, it nearly doubles the aborted cycles compared to RL and WoperTM.

Kmeans' main transaction (TID-0, update of the new cluster centers) performs a read-modify-write (RMW) access to increment the centers length, followed by a loop where cluster centers are updated. In RW, this pattern makes the transaction vulnerable to any concurrent access since its very beginning, since the write to the contended variable occurs shortly after transaction start. On the other hand, in RL, this pattern acts as a *shield* that quickly makes the transaction invulnerable to concurrent accesses to the same cluster center. Furthermore, because the compiler generates an exclusive load for the RMW access, the contended cache line jumps from core to core in exclusive ownership. This prevents the *starving writer* pathology and explains why RL and Woper perform alike.

As for intruder, the extremely high contention in TID-0 (queue pop) harms overall performance, despite the fact that most of the parallel work is done by TID-1 (traversal of a tree of lists). RL manages to reduce aborted work in the main transaction (TID-1), yet it fares worse than RW because the RMW access pattern to the queue pop pointer seen for TID-0: several concurrent transactions read it and then all of them fail in their attempt to modify it. While RW can make progress in this scenario (since the first upgrade request to reach the directory will abort all other readers), recurring aborts in TID-0 constantly switch to the fallback path for both RL and RW. WoperTM and PowerTM side-step this switch, but the limitation of having only two priority levels limits further improvements: TID-0 and TID-1, though independent, compete for the power token to deal with contention. As in yada, WoperTM can handle contention in TID-1 better than Power, by allowing readers that have already navigated the tree to survive eventual write attempts to remove specific nodes.

## V. Conclusion

We presented WoperTM, a dual-policy CR that improves performance and efficiency of HTM systems. WoperTM allows regular transactions to benefit from nacks at barely no cost, in order to make them more capable of surviving conflicts, while offering a way to escape pathological scenarios without the penalty of falling back to mutual exclusion.

## References

[1] K. Moore, J. Bobba, M. Moravan, M. Hill, and D. Wood, "Logtm: Log-based transactional memory," in *International Symposium on High-Performance Computer Architecture*, 2006, pp. 254–265.

[2] J. Bobba, K. Moore, H. Volos, L. Yen, M. Hill, M. Swift, and D. Wood, "Performance pathologies in hardware transactional memory," in *International Symposium on Computer Architecture*, 2007, p. 81–91.

[3] T. Harris, J. Larus, and R. Rajwar, *Transactional Memory, 2nd Edition*, ser. SLCA. Morgan & Claypool Publishers, 2010.

[4] M. Herlihy and J. E. B. Moss, "Transactional memory: architectural support for lock-free data structures," in *Proc. of the International Symposium on Computer Architecture*, 1993, p. 289–300.

[5] S. Park, M. Prvulovic, and C. J. Hughes, "Pleasetm: Enabling transaction conflict management in requester-wins hardware transactional memory," in *International Symposium on High Performance Computer Architecture*, 2016, pp. 285–296.

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