On the Interactions between ILP and TLP with Hardware Transactional Memory

Víctor Nicolás-Conesa, Rubén Titos-Gil, Ricardo Fernández-Pascual, Alberto Ros, Manuel E. Acacio
Computer Engineering Department
University of Murcia
30100 Murcia (SPAIN)
{victor.nicolasc, rtitos, ricardof, aros, meacacio}@um.es

***Abstract*—Hardware implementations of Transactional Memory (HTM) are designed to facilitate efficient thread synchronization in parallel programs, encouraging the use of larger critical sections. By employing optimistic concurrency control to execute transactions speculatively, HTM systems promise to deliver the performance benefits typically associated with fine-grained locks. In doing so, HTM systems must deal with transaction aborts. While under certain conditions aborts may be caused by the inherent limitations of hardware structures employed to implement TM (e.g., caches), conflicting concurrent accesses to shared memory locations are generally the prevailing cause for squashing the work done by a transaction.**

In this study, we present what we believe to be, to the best of our knowledge, the first characterization of how the aggressiveness of processor cores, particularly their ability to exploit instruction-level parallelism (ILP), interacts with the support for optimistic thread-level speculation offered by HTM systems. We have observed that by adjusting the size of structures that facilitate out-of-order and speculative execution, the number of aborts in the execution of transactional workloads can be altered in best-effort HTM implementations. Our findings indicate that in scenarios with high contention, a smaller number of powerful cores is more suitable, whereas in low contention scenarios, using a larger number of less aggressive cores is preferable. In addition, HTM systems that employ lazy detection and those employing eager detection with requester-stalls resolution, benefit from using simpler cores. In conclusion, abort ratios can be reduced with a careful choice of both processor aggressiveness and design aspects for each application depending on its contention.

***Index Terms*—Hardware Transactional Memory, Out-of-Order and Speculative Execution, Multicore, Characterization.**

I. Introduction

Hardware transactional memory (HTM) is now implemented in several commercial multicore processors with low overhead by leveraging private caches and coherence protocols [1]–[3]. This technology simplifies the synchronization of parallel programs by shifting the responsibility of ensuring atomic and isolated execution of specific code sections from the programmer to the hardware.

However, the occurrence of *conflicts* can jeopardize performance and efficiency. Conflicts occur when concurrent memory accesses to the same cache line from different threads perform, being one of them a write. Since conflicts pose risks to the atomicity and isolation of transactions, when they are detected, typical implementations abort and re-execute the competing transactions, leading to a waste of energy and time. Existing HTM implementations perform an *eager* detection of conflicts by leveraging the cache coherence protocol. Most resolve them in the simplest way, consisting in always satisfying requests (known as *requester-wins* resolution): the transaction running on a core is aborted when its private cache controller observes a conflicting coherence request.

In such eager HTM implementations, both the order in which memory accesses within a transaction are performed in cache, as well as the lifetime of each block in the read-write set (i.e., cycles that each cache block remains as part of it, since the first access until commit), can have an influence on the number of aborts, as they affect the length of the window of vulnerability for the transaction, this is, the cycles that it is exposed to aborts due to remote conflicting accesses.

Since first introduced by Herlihy and Moss [4], a myriad of research works have proposed alternative HTM designs and analyzed their performance. However, nearly all prior work has invariably considered the HTM implementation in isolation from the processing core, treating it as a *black box* that generates a trace of memory accesses, and very often assuming oversimplified CPU models based on single-issue in-order execution [5]–[8], which are far from representative of today's execution cores.

In contrast, this work focuses on the interactions between the strategies employed by contemporary commercial multicore processors to harness of Instruction-Level Parallelism (ILP), and their hardware support to aid the exploitation of Thread-Level Parallelism (TLP) in multithreaded programs that employ transaction-based synchronization.

In particular, our investigation reveals notable synergies and interactions between the HTM support and the processor's aggressiveness. Our results confirm that the characteristics of the processing core have relevant implications on HTM performance as they affect the frequency of conflicting accesses among transactions. In situations characterized by intense contention, we find that opting for a smaller number of powerful cores is the preferred approach, which stands in stark contrast to the low contention scenario. Interestingly,

This work was supported by Grant PID2022-136315OB-I00 funded by MCIN/AEI/10.13039/501100011033/ and by "ERDF A way of making Europe", EU; grant TED2021-130233B-C33 funded by MCIN/AEI/10.13039/501100011033 and by the European Union NextGenerationEU/PRTR; and grant agreement No 819134 funded by the European Research Council (ERC) under the Horizon 2020 research and innovation program.

this work suggests that in certain scenarios of low to moderate contention, overall performance may not benefit from employing the most complex and energy hungry cores, as the loss caused by an increase in the number of aborts offsets the gains achieved by exploiting ILP more aggressively. Therefore, we advocate for proper management of the aggressiveness of the cores during HTM execution in modern heterogeneous multicore designs [9]–[11], as a promising approach that can bring performance and efficiency gains.

This article extends our previously published conference paper [12] by also taking into consideration in our analysis varying HTM design alternatives when it comes to handling conflicts, ranging from eager to lazy detection approaches. We found an interesting interplay between the characteristics of the HTM system and those of the processing core. More precisely, we observe that under low contention scenarios, all HTM implementations show similar performance when commit arbitration is not bottleneck. On the contrary, in high contention situations, we observed that the preferred combination is simpler cores and a more advanced HTM system, as they show same contention and performance levels than with aggressive cores, thus favoring TLP exploiting over ILP. Observing closely, these behaviors on different HTM designs can lead to different insights. On one side, we can see, comparing best-effort simple system (req-wins) against req-stalls, that serialization over fallback lock is the root cause of inefficiencies in those systems that have a more constrained implementation and require forward progress guarantees. On the other hand, comparing eager systems with lazy ones we can observe that the time at which a memory operation is made visible to the rest of the cores is substantially important in terms of aborts and performance.

The rest of the manuscript is organized as follows. In Section II, we will give some background about instruction-level parallelism and hardware transactional memory, relating it to the state of the art. Afterward, we will describe the system and methodology used to obtain the results from this study in Section III. Then, in Section IV, we will show and discuss the results obtained from the experiments. And, finally, we will draw the main conclusions derived from this work in Section V.

II. Background and Related Work

A. Superscalar ILP processors

Current high performance processors are equipped with a variety of techniques aimed at exploiting ILP to improve single-thread performance through superscalar, speculative, out-of-order (OoO) execution [13]–[16], among others. A typical OoO core relies on certain structures to ensure that sequential semantics of the program are preserved, and the memory consistency rules are met. Among the most relevant of such structures, we find the reorder buffer (ROB) and the load-store queue (LSQ) [16], [17], as their size partially determines the aggressiveness of the pipeline at exploiting ILP: the larger these structures are, the more instructions can be considered by the dynamic scheduling logic for execution when their operands are ready. In the analysis presented in subsequent sections, we will focus on the size of the ROB and LSQ as key parameters of an OoO core.

B. Hardware Transactional Memory

HTM systems provide atomicity and isolation to transactions at low overhead through the hardware implementation of two basic mechanisms: conflict detection and data versioning. Both mechanisms have been incorporated at relatively low cost into commercial multicore processors by leveraging existing structures, namely private caches and the cache-coherence protocol [18]. Private caches are typically extended with two bits per block used for tracking the read and write sets of the transaction. Additionally, the cache is extended with logic to perform conditional gang-invalidation of all blocks that have the *speculatively modified* bit asserted, so that tentative updates isolated in the private cache can be discarded in a few cycles when a transaction aborts. In turn, committing a transaction only requires clearing both bits for all cached blocks at once, publishing the updates to the rest of the system in an atomic manner. On the other hand, using the L1 cache for tracking read-write sets and data versioning has the drawback of limiting them to the cache size, which can cause *capacity* aborts whenever a block in the read or write set gets evicted from cache [19]. Some other ways to implement book-keeping include the use of Bloom filters or signatures, which conservatively track any number of addresses at the cost of introducing additional conflicts due to false positives [18], [20], [21].

Conflicts in an HTM implementation are detected by leveraging the cache coherence protocol: to write a block in cache, exclusive ownership must be acquired on the block, which in turn will notify any concurrent readers through invalidation messages, allowing them to detect a conflict if the block is in the read-set of an active transaction. Similarly, any conflicting read to a block that is currently in the write set of a transaction will be detected as the block will be exclusively owned by the writer. In order to resolve conflicts, the most commonly used policy is requester-wins because of its simplicity of integration into existing protocols. With this policy, the transaction that generates the coherence request will cause the abort of any other conflicting transaction. The key drawback of requester-wins is that livelocks may appear if transactions *fire* each other repeatedly. As a result of employing requester-wins resolution as well as the limits in buffering capacity imposed by private caches, commercially available HTM implementations are best-effort: the system tries its best but gives no guarantees about the commit of any transaction. Thus, to ensure forward progress despite contention-induced livelocks or capacity limitations, at the beginning of a transaction an alternative software fallback path must be provided. If the transaction aborts, execution will continue through this specified code section [22]. Depending on the abort status code returned, the abort handler may attempt to re-execute the transaction speculatively several times before resorting to the non-speculative execution of the transaction, using a global

lock to which all transactions must subscribe (have in their read set).

C. Related Work

Several previous works have proposed the use of contention managers to reduce the negative impact of transactional conflicts [23]–[25]. Contention managers act whenever a transaction aborts due to conflicts and try to avoid the same or a new conflict, usually by making the transaction wait (back-off) before re-executing [26] or enforcing some commit ordering by means of priorities [25], [27], [28]. Other conflict managers try to schedule transactional re-execution avoiding the concurrent execution of conflicting transactions using heuristics based on information gathered in the abort manager [29], [30].Some techniques try to take advantage of the contention manager to avoid energy waste, e.g., switching the CPU to a low-power mode while it is waiting to re-execute a transaction, as explained in [7].

There are other authors that have developed different HTM systems that try to solve different problems. Systems that use requester-wins as the conflict resolution policy often suffer from the friendly-fire pathology, where two transactions abort each other continuously, thus preventing forward progress [1], [31]. As an alternative, the requester-stalls conflict resolution policy was designed [19], [31], [32].This policy uses nack messages to communicate to the requester cache that there was a conflict and that its transactional execution must stall. This implies not only adding a nack message to the coherence protocol, but also a mechanism to avoid deadlocks produced by cyclic dependencies between transactions. Some proposals try to improve the performance of requester-stalls by either serializing transactions upon cyclic dependencies [33] or allowing transactions to keep running even if they are conflicting [34]. Other studies prove that delaying the release of stores to other cores as long as possible results in a reduction of the time that the transaction is sensitive to conflicts [35]–[37]. But no study has looked into the interplay between the design choices for supporting HTM mechanisms and those aimed at exploiting instruction-level parallelism. This work tries to provide quantitative answers to questions such as: would a less aggressive processor increase the amount of time a transaction is vulnerable to conflicts? And if so, would a lazy conflict detection mechanism cope better with this situation?

There have been previous works that tried to resize the LSQ and ROB based on their occupancy to reduce energy consumption without negatively impacting performance. More precisely, [38] observes that these structures are not totally used most part of the time, and so reducing them causes passive and active energy savings [39] and does not entail noticeable performance loss. The same work also shows that occupation is not the same in every application and that within the same application there are moments of high and low occupancy and structure size can vary depending on this. Some proposals even show that complex structures like the load queue can be removed without adding much complexity or overhead [40].All of the former has been done with traditional, non-transactional applications, thus neglecting the interplay between ILP mechanisms and HTM, which is the focus of this work.

Apart from [38], some other works have also analyzed the occupation of these structures while executing non-transactional code. Dimova *et al.* [41] study the effects on the performance of modifying ROB size too and conclude that increasing ROB beyond certain limits can even hurt performance. In Mathis *et al.* [42], LSQ occupancy is observed using SMT (Simultaneous MultiThreading) on POWER5 processors and the authors find that these structures are not fully utilized most of the time. Unlike prior works, we characterize for the first time to our knowledge the occupation of OoO structures when running transactional workloads, and analyze the interactions between ILP and HTM mechanisms.

III. Experimental Methodology

We use a full-system simulator based on gem5 [43] to perform our evaluation. While the gem5 simulator has official support for HTM since release 20.1 (currently limited to the Arm ISA), in this work we use an in-house HTM model for the x86 ISA developed earlier atop gem5-17 in which we merged the HTM support for the memory system found in the Ruby module of Wisconsin GEMS [44], with the CPU models of gem5 appropriately adapted and extended to support HTM with several improvements over the original model. Using this modified version of gem5, we model a tiled chip multiprocessor whose cache hierarchy has two levels: the L1 is private to each core, and L2 is shared and distributed among the tiles (one L2 bank per tile). It uses a MESI cache coherence protocol properly extended to support cache-based HTM. The key configuration parameters are shown in Table I, including those for the baseline out-of-order CPU model.

Unlike our previous conference version [12], in this work we use an optimization in the HTM runtime library and an augmented version of the STAMP thread-local memory allocator, to enable the use of a technique known as *heap prefault* [45], [46]. In this way, right before a transaction is started, the wrapper function in the library *touches* the memory pages that will be used to satisfy dynamic memory allocation in transactions. By doing so, most page faults that are the result of the naive thread-local memory allocator implemented in the STAMP library, which would otherwise cause transactions to abort every now and then, are triggered and handled by the kernel before entering the speculative transaction. In this way, thread serialization on the fallback path due to such page-fault induced aborts is avoided. Since their occurrence is independent from the characteristics of the core, we opt for their removal since it helps isolating our analysis from other spurious causes of abort, allowing us to focus on true contention exhibited by the workloads (conflicts due to true data dependencies).

Our base HTM model resembles the Intel RTM (Restricted Transactional Memory) ISA extensions introduced in the Haswell microarchitecture [2] just as in [12]. The conflict resolution policy used is requester-wins and the book-keeping

technique used to track read sets is a perfect signature that allows to track read-set blocks even after being evicted from the L1 cache [21], [32]. For keeping track of the write set, a *speculatively modified* (SM) bit in each of the L1 cache lines is used. We employ eager lock subscription and our abort handler ensures forward progress by acquiring the fallback lock in case of capacity-induced and fault-induced aborts, as well as when the number of retries after a conflict-induced abort exceeds a given threshold (set to 8 in our experiments).

TABLE I: System parameters.

Out-of-order baseline core Settings	
Cores	out-of-order (execute/commit width: 8)
Load queue (LQ)	72
Store queue + store buffer (SQ)	56
Reorder buffer (ROB)	192
Memory Settings	
L1 I&D caches	Private, 32KiB, 8-way, 1-cycle hit latency
L2 cache	Shared, 512KiB per Tile, 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 (2×2), X-Y
Flit size / Message size	16 bytes / 5 flits (data), 1 flit (control)
Link latency / bandwidth	1 cycle / 1 flit per cycle

In the analysis presented in the following section, we vary the following system configuration parameters:

Core models: To analyze the interactions between the execution model and the HTM support, we consider both the TimingSimpleCPU and O3CPU models of gem5. The *timing* CPU is in-order, single-issue, and non-memory instructions take 1 cycle, while memory access instructions stall the CPU until completed in cache. On the other hand, the *O3CPU* resembles a modern superscalar, out-of-order execution pipeline with dynamic instruction scheduling and a large instruction window. Apart from these two well-known CPU models, in the following evaluation, we also consider a variation of the out-of-order model in which we have removed speculation at the instruction level by stalling instruction fetch whenever an unresolved branch is in-flight. To study the actual occupation of the LSQ and ROB structures, we use probe-like tracing measuring the number of entries occupied on these structures for 2048 cycles every 16000 cycles.

Number of cores: We use two different systems sizes (4 and 16 cores) and run the benchmarks at the corresponding thread counts matching the number of cores, to analyze overall performance a different contention levels of the same workload.

Hardware Transactional Memory systems: To check how the choice of conflict detection and resolution policies implemented by the HTM system interacts with the OoO core configuration, we consider two alternative designs apart from our base HTM model: a *requester-stalls* design similar to our baseline except that it resolves conflicts by *nacking* the conflicting request until the offended transaction completes (commits or aborts) and releases isolation over the cache block. A timestamp-based conservative deadlock avoidance mechanism to break cyclic dependencies among transactions similar to that of LogTM [32] is used, so that coherence messages are augmented with timestamps, which indicate the priority (age) of each transaction. On its part, we also compare against an idealized lazy conflict detection design similar in concept to that presented in [31]. Our lazy model uses an unlimited transactional write buffer separated from the private cache to keep speculative values until the transaction has fully executed and is granted permission to commit, at which point coherence actions required to perform transactional writes in cache (acquisition of exclusive ownership for write-set blocks) are issued to the coherent fabric. In this idealized lazy system, we also model a zero-latency, *magic* lazy arbitration scheme that enables fully parallel commits of non-conflicting transactions without any communication overhead.

The benchmark suite that has been used for this study is STAMP [47], which includes diverse workloads with different characteristics in terms of contention, transaction size, read-write set size, etc. The recommended medium inputs [47] were used to run all the benchmarks. We opted for excluding *Bayes* from our evaluation, as it implements a search algorithm whose non-deterministic behavior exhibits a very high variability in execution time depending on the specific thread interleaving (arriving at different solutions for the same input).

In those figures that show the results in terms of accumulated execution cycles (sum of cycles executed by all threads), we categorize cycles into the following disjoint *regions* according to the code executed by each thread: *Parallel* (non-transactional, parallel code); *Stalled* (stalls due to conflicts, in requester-stalls policy); *AbortHandler.Waiting* (cycles spent in the abort handler, including the time needed to acquire the fallback lock); *Committed* and *Aborted* (executing transactions that are eventually committed and aborted, respectively); and finally, *Other* (cycles spent in regions that are not of particular interest for this study such as barriers, kernel code, etc.).

Similarly, in those figures that report aborts, bars are broken into the different causes of abort. In particular, we show the aborts due to conflicting memory accesses amongst several transactions (*Conflicts*); those caused by evictions of transactional blocks from cache (*Size*); aborts due to conflicts caused by the acquisition of the fallback lock by another transaction (*FallbackLock*); those explicit aborts of transactions that find the fallback lock acquired during subscription (*ExplFL*); and last, the remaining sources that are not of particular interest for the study (e.g., exceptions, interrupts, etc.) are grouped under *Other*.

IV. Experimental results

In Fig. 1 we compare execution time and key HTM performance metrics under varying CPU models (out-of-order, out-of-order without speculation, and in-order). The purpose of this experiment is to determine how the execution model used by the cores interacts with the HTM support. In this figure, all the results are normalized to the OoO CPU model with speculation.

Fig. 1 shows, as expected, that in every benchmark the average number of cycles for committed transactions increases

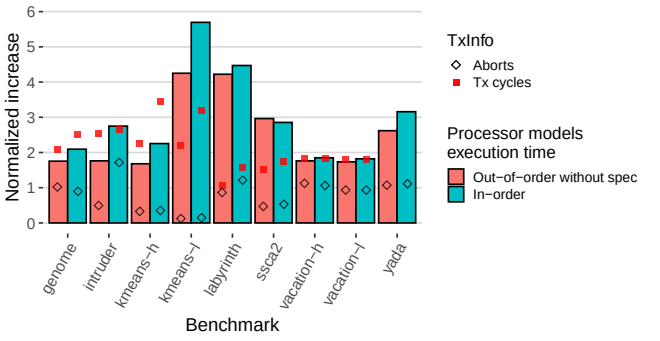


Fig. 1: Differences between out-of-order, out-of-order without instruction level speculation and in-order core models. Red squares represent cycles spent in transactions, black diamonds are the number of aborts and the bars represent execution time. All data is normalized respect to OoO model with speculation and all architectures use 16 cores.

when using the in-order and OoO without speculation when compared to the OoO. By exploiting ILP and branch prediction, the OoO aggressive core is able to execute all the applications faster (execution time grows up to $\times 5.7$ and $\times 4.2$ for the in-order and OoO simpler models, respectively), in *kmeans-l*. Interestingly, results from the transactional information reveal that the number of aborts and the duration of the transactions are affected differently by the CPU model, also depending on the benchmark and its specific characteristics. In the case of *genome, vacation* and *yada*, we can see that the number of aborts does not change much with the CPU model. However, there are cases like *kmeans* and *ssca2* (short-running transactions), where the less aggressive in-order core experiences less contention. This happens because there are fewer memory accesses concurrently in flight (both from transactional and non-transactional regions). Note that *ssca2* has a very small number of aborts and this reduction has negligible impact on performance. On the other side, there are cases like *intruder* and *labyrinth* (long-running transactions) where the in-order core experiences more conflicts. This is because the transactions take more time to execute, increasing the window of time during which they are exposed to conflicts from other transactions.

A. ROB and LSQ occupancy analysis

In Fig. 2 we can see the occupancy level of the LQ, SQ and ROB in our baseline core model (OoO with speculation), which is equipped with a 72-entry LQ, a 56-entry SQ and a 192-entry ROB. For the sake of brevity, we only show the results for two representative benchmarks and two different contention levels in each case. As we can see, there is only a small increase of approximately 5% in the LQ occupancy in *kmeans* benchmark when the contention level increases. This increase is smaller in *vacation*.

This increase in occupancy can also be observed in the SQ and ROB. In the case of the ROB, the occupancy grows around 10% of the whole queue capacity in the high contention

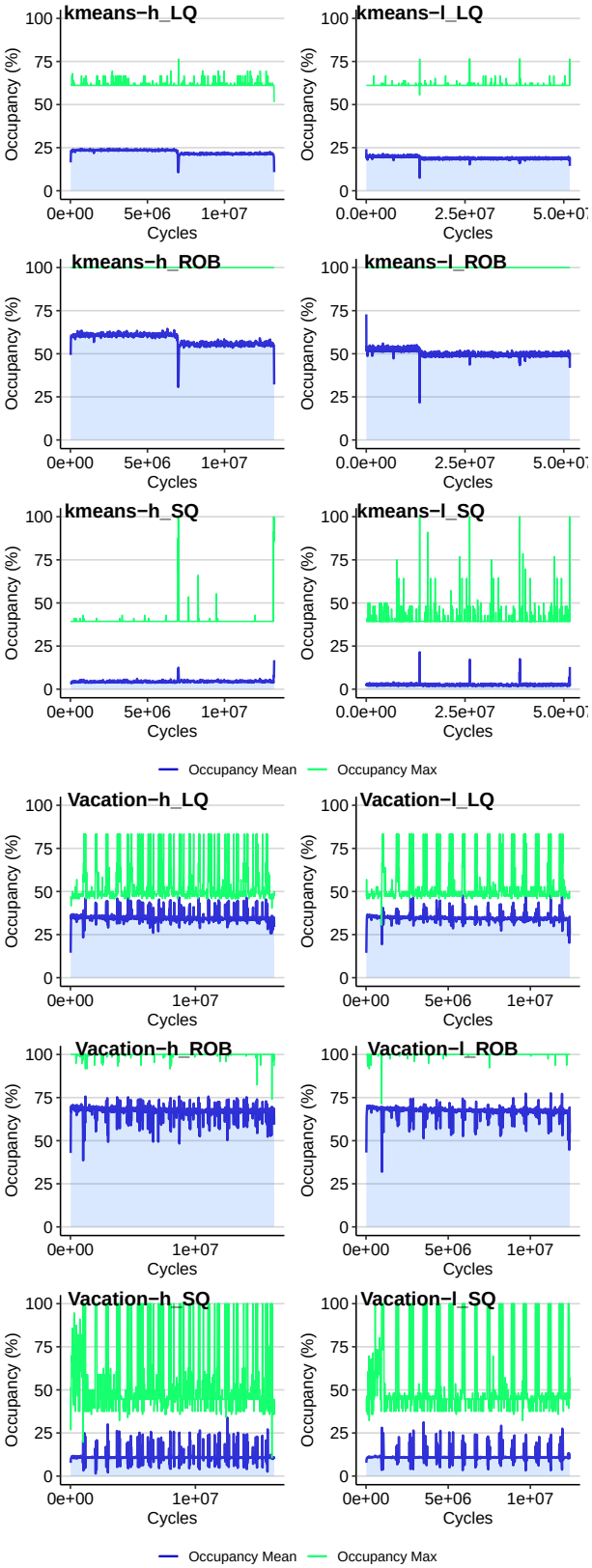
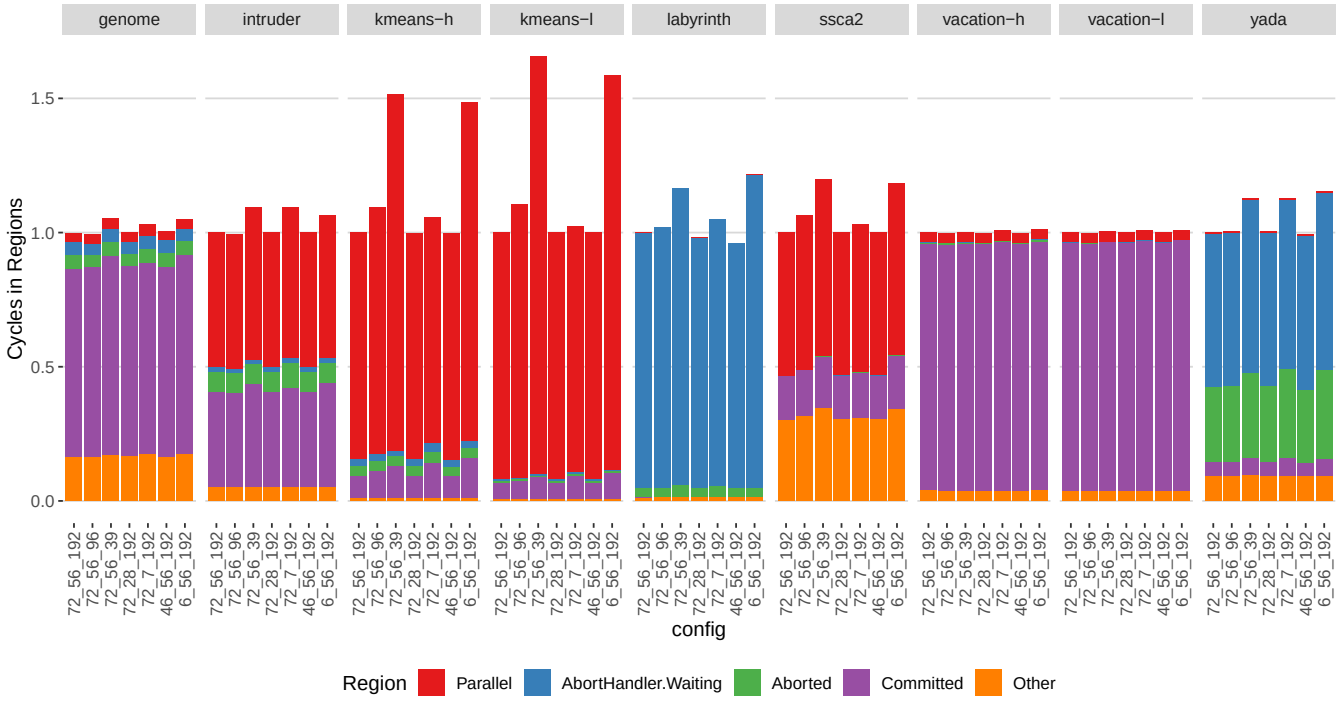
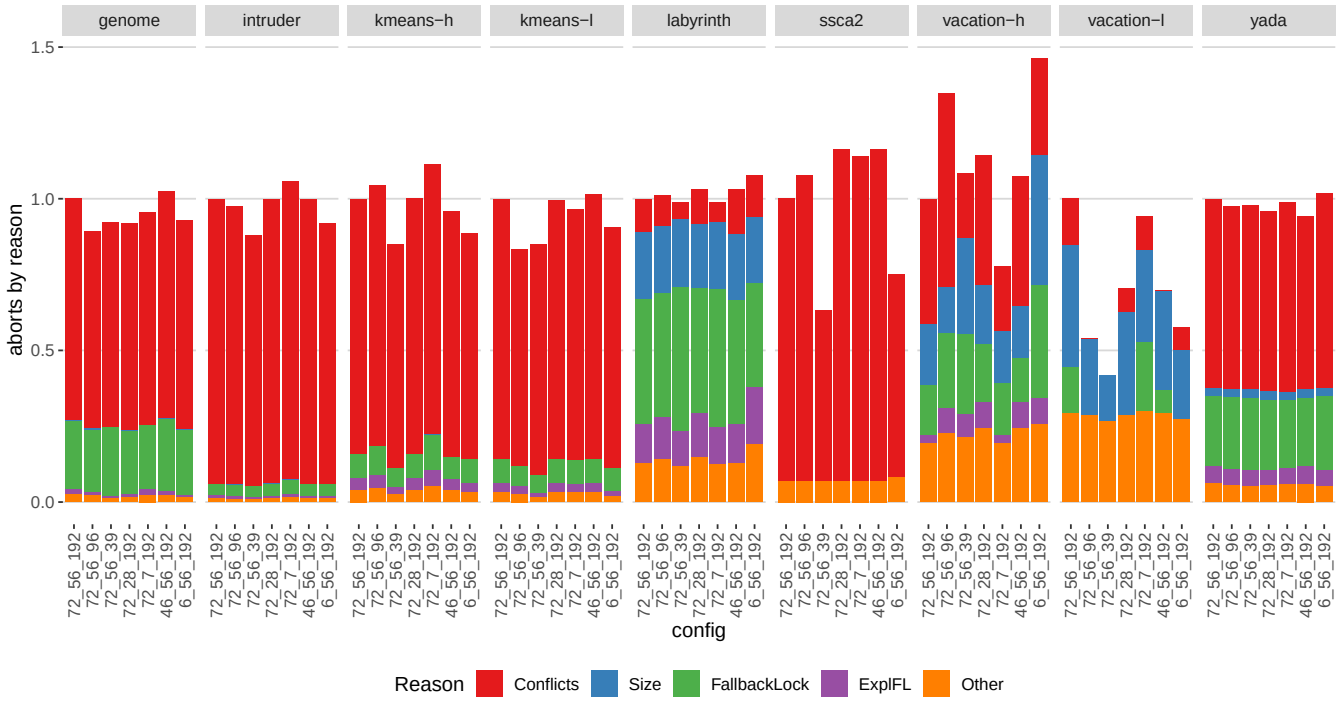


Fig. 2: Occupancy levels in microarchitectural OoO structures while running *kmeans* and *vacation*. The occupation is shown as a percentage of the total structure's size.



(a) Relative cycles spent in different execution regions reducing every structure one by one (Normalized to base configuration of 72 LQ, 56 SQ and 192 ROB entries).



(b) Aborts by reason reducing every structure one by one (Normalized to base configuration of 72 LQ, 56 SQ and 192 ROB entries)

Fig. 3: Results in terms of execution cycles and aborts reducing every OoO structure size one by one.

scenario. For the SQ, occupancy levels in the high contention scenario are higher too. The same observations made for *kmeans* benchmark can apply to *vacation*, although in this case, the increase in occupancy is less noticeable.

As we can see, generally the store queue and load queue occupancy during execution are low in most cases. Maximum occupancy in LQ (green line) does not rise over 80% in either *kmeans* nor *vacation*, while average occupation is around 25% in *kmeans* and 40% on *vacation*. This means that both structures are usually not fully utilized most of the time in these benchmarks. Additionally, different phases during the execution of *kmeans* are visible, with the LQ being used slightly more during the first half of its execution. Furthermore, we can see that the ROB follows the same occupation pattern as the LQ, while the SQ shows the opposite pattern as the LQ.

B. Analysis with varying LQ, SQ and ROB sizes

Fig. 3a and 3b show, respectively, normalized execution time and number of aborts, when varying the number of ROB, LQ and SQ entries, relative to our baseline OoO core. Note that, for easier comparison across all configurations, ROB size is deliberately not adjusted to match the changes in SQ and LQ sizes. The figures show that halving the size of SQseems to have little to no effect on the execution time. Performance losses begin to appear when reducing SQ size to only 7 entries. In most cases, like *genome, kmeans, ssca2* and *vacation*, the performance loss is less than 5%, while it is higher in others like *yada, labyrinth* and *intruder*, which are the benchmarks that have both larger write sets and highest contention. This happens because we reduce the size of the structure further than the average occupation seen in section IV-A. The lack of room in the store buffer causes stalls when trying to dispatch new stores, increasing the time to finish transactions and hence enlarging the vulnerability window, having the same effect as observed in Fig. 1.

Reducing LQ size from 72 to 46 entries barely harms performance in any of the considered benchmarks. Reducing the LQ size to 6 entries decreases the number of aborts produced in *intruder*, *kmeans*, and *ssca2*, but despite discarding less speculative work, execution time increases forall benchmarks as a result of exploiting less available ILP. The slowdown is more pronounced in *kmeans* and *ssca2* than in *intruder*, as the former spends most of its execution in non-transactional code. The degradation in *labyrinth* and *yada* comes from the longer duration of each non-speculative transaction executed, as these benchmarks often resort to the fallback lock due to both contention and capacity limits. In general, we can see that reducing ROB size to 39 entries brings a similar pattern in terms of aborts and execution time as reducing the LQ size to 6 entries.

Because *kmeans* has small transactions that touch a handful of cache blocks in a read-modify-write fashion, the more aggressive the out-of-order pipeline is, the more contention we have as more conflicting memory accesses to such blocks can be in flight at a given moment, while we saw in Fig. 1 that the in-order CPU model suffered significantly less aborts as a result of having at most one memory access per core in flight. However, the penalty in execution time of being able to exploit less ILP is lower in *kmeans-h* as a result of its much smaller fraction in non-transactional code than *kmeans-l*.

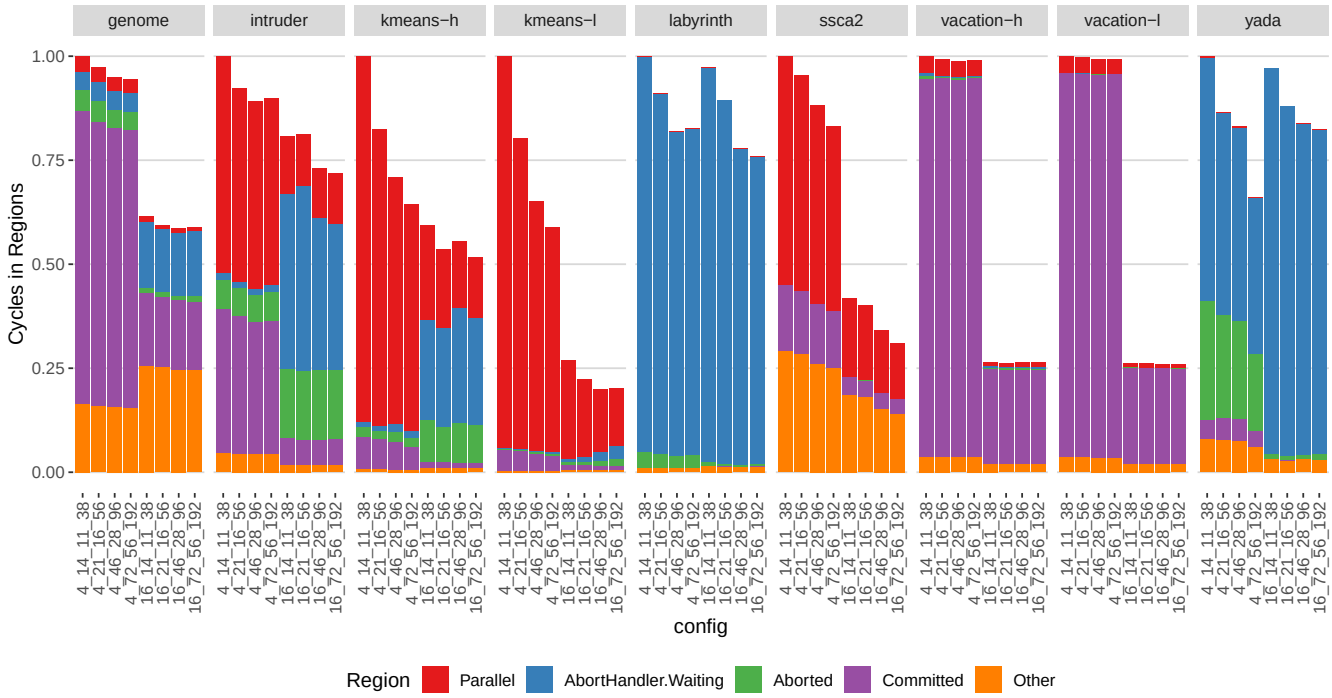
In [12], we saw *vacation* did not see its execution time affected nor a change in the number of aborts seen, in spite of important reductions in ROB, LQ or SQ sizes. In this study, we observe a different behavior as we see strong variations in the number of aborts with different structure sizes. This happened because we removed page faults inside transactions as we are using prefault. Page faults were the main reason that made transactions to abort in vacation and removing them, this benchmark is left with a really reduced amount of aborts, and as it is that small and our data is normalized, a little variation over this number cause a really big change over the figure shown. Still, saving the page faults this benchmark keeps poorly utilizing the out-of-order pipeline, which in turn could be an indication of scarcely available ILP or frequent misses whose latency cannot be hidden by the OoO engine.

C. Analysis with varying thread count

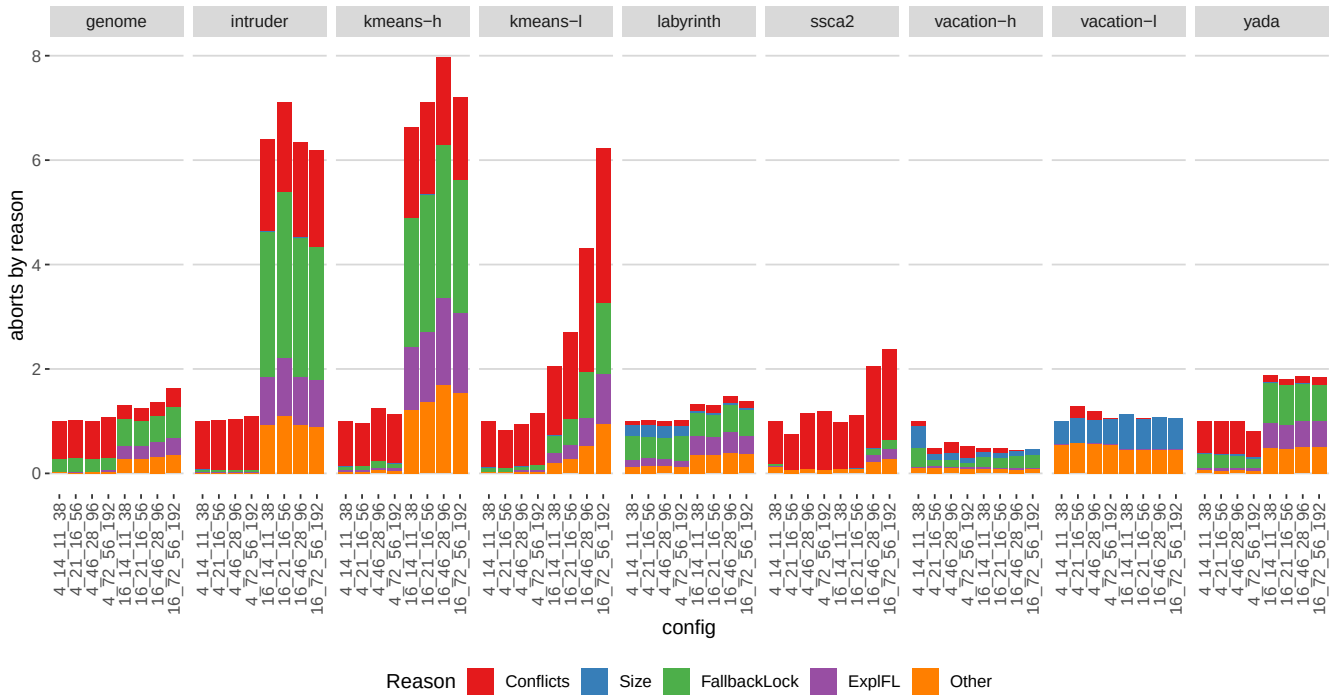
In the previous subsection, we varied the size of key processor structures while keeping the system size fixed to 4 cores. In this subsection, we will compare results as contention levels increase as a result of running a higher thread count in a larger system (16 cores). In Fig. 4a we can see relative execution time running 4 and 16 threads, for five different core configurations of increasing complexity (larger structures), normalized to the 4-thread configuration of lowest core complexity. On its part, Figs. 4b and 5 show the relative number of aborts and fallback lock acquisitions, respectively, for the same five core configurations when running 16 threads, again normalized to the configuration with the simplest core.

We can see in Fig. 4a that benchmarks such as *labyrinth* and *yada* do not reduce their execution time when moving to 16 threads, in both cases largely due to high contention affecting long-running transactions, but also partly because of capacity aborts suffered due to the large write sets of their main transaction. In particular, *labyrinth* is one of the benchmarks from the STAMP suite that shows the worst scalability on best-effort HTM systems, a direct consequence of its huge write set (a local copy of a shared matrix of tens of kilobytes is performed inside the transaction) that makes transactions fail due to insufficient cache capacity. Furthermore, without hardware support for *early release* [48] it becomes impossible for any transaction in *labyrinth* to make progress without aborting all other concurrent transactions (as its main transaction fully reads the shared data structure which gets eventually modified in the same transaction).

This behavior leads to useful work being made almost exclusively while all threads but one are waiting on the fallback lock, while one executes non-speculatively to escape capacity limits and avoid livelock situations. We see that employing a more aggressive core improves execution time by accelerating the *serialized* execution of a transaction holding the fallback lock, but the efficiency of parallel execution is very poor.



(a) Relative execution time in different execution regions (Normalized to configuration with 14 LQ, 11 SQ, 38 ROB and 4 cores/threads).



(b) Relative number of aborts and the reason that caused it (Normalized to configuration with 14 LQ, 11 SQ, 38 ROB and 4 cores/threads).

Fig. 4: Results in terms of execution cycles and aborts using different core/thread count and processor configurations.

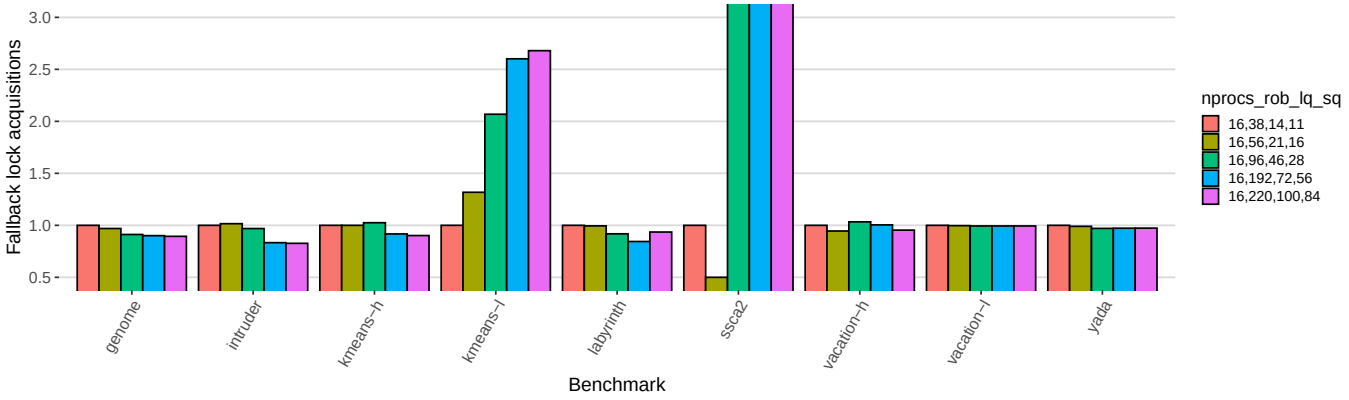


Fig. 5: Relative number of acquisitions of the fallback lock for five core configurations in the 16-core system, normalized to simplest core configuration.

The same considerations apply to *yada*, though in this case, write capacity is not the dominant cause of aborts, but rather the high contention coupled with the *friendly-fire* pathology brought by the requester-wins policy of the underlying HTM system. Although neither benchmark improves with higher thread counts, we can see that labyrinth benefits more than yada from running on aggressive cores: execution time in labyrinth is reduced by 25% when using the most complex core, compared to the results seen for the least complex core, while in *yada* the reduction is about 12%. This is a clear indication that labyrinth exhibits more ILP available than yada, likely a consequence of the regular computations performed on a matrix during the path expansion phase of labyrinth's main transaction, in which an aggressive core can shine due to highly accurate branch predictions and data dependencies not being carried across to the next loop iteration.

In the opposite end in terms of contention we find both versions of *vacation*, *ssca2* and *kmeans-l*: these are the applications from STAMP that scale best in a best-effort HTM system. Their low to medium contention level translates into good scaling up to 16 cores, very close to the ideal speedup. In the case of *vacation*, increasing the size of OoO structures does not improve performance, unlike in *kmeans-l* or *ssca2*, which indicates that *vacation* does not exhibit enough ILP to benefit from complex cores, partly owing to the irregular data structures it employs (red-black trees of linked lists). Though both *kmeans-l*, *ssca2* as well as *vacation* show a similar trend in scalability, their characteristics are very different from each other: short-running transactions and a large non-transactional fraction of its execution in *kmeans* and *ssca2*, versus *vacation*'s long-running transactions that span most of the execution time. Because most of the execution is non-transactional, the $\times 3$ increase in the number of aborts and $\times 2.5$ in the number of fallback acquisitions seen in *kmeans-l* for the most complex core does not translate into any slowdown in execution: the penalty is made up by the ability to exploit more ILP in other phases of its execution such as the calculation of cluster centers, which does not require synchronization. In *ssca2*, the number of aborts is negligible in the baseline configuration, and thus a $\times 2$–$\times 3$ or even higher increase does not have any impact on execution time; on the contrary, more complex cores bring additional gains in execution time by exploiting more ILP. *Ssca2* operates on a large, directed, weighted multi-graph, and the adjacency list of subsequent vertices in the graph can be inspected in parallel, without carrying data dependencies across loop iterations. It is worth noticing how in *kmeans-l* the improvement in execution time when moving from the simplest core to the most aggressive one is more pronounced in the 4-core system (nearly 40% improvement) than with 16 cores (around 20%). In contrast, in *ssca2* aggressive cores seem to improve performance similarly regardless of the thread/core count.

Fig. 4b also shows that a more aggressive CPU can bring in certain cases the opposite effect seen for *kmeans-l* and *ssca2*: a reduction in the number of aborts. Such is the case of *intruder*, and to a lesser degree *genome* and *labyrinth*. In *intruder*, we can see the direct effect of micro-architectural modifications with varying levels of contention. While Fig. 4a shows that when running *intruder* with 4 threads the aggressiveness of the core barely impacts its execution time (flat beyond a 56-entry ROB, in a similar trend to that of *vacation*), when executed with 16 threads we see a notable reduction in execution time as the core becomes more and more aggressive: the 192-entry ROB configuration obtains much better results compared to the simpler core that has only 38 entries in the ROB. This can be explained by the fact that *intruder* resorts much more frequently to the fallback path with 16 threads than with 4. We observe that for the configuration that uses a 56-entry ROB, there is an increment over aborts caused by other reasons different to contention that make the HTM system to acquire the fallback lock more often too. Thus, having a more powerful core to run non-speculative transactions as fast as possible is beneficial, as all other threads are blocked until the fallback lock is released, as was the case for labyrinth. Interestingly, as it can be seen in Fig. 4b and Fig. 5, in the case of intruder, a more aggressive core can slightly reduce the number of

aborts and consequently the number of fallback acquisitions in comparison to thinner cores.

D. Analysis varying HTM conflict detection policies

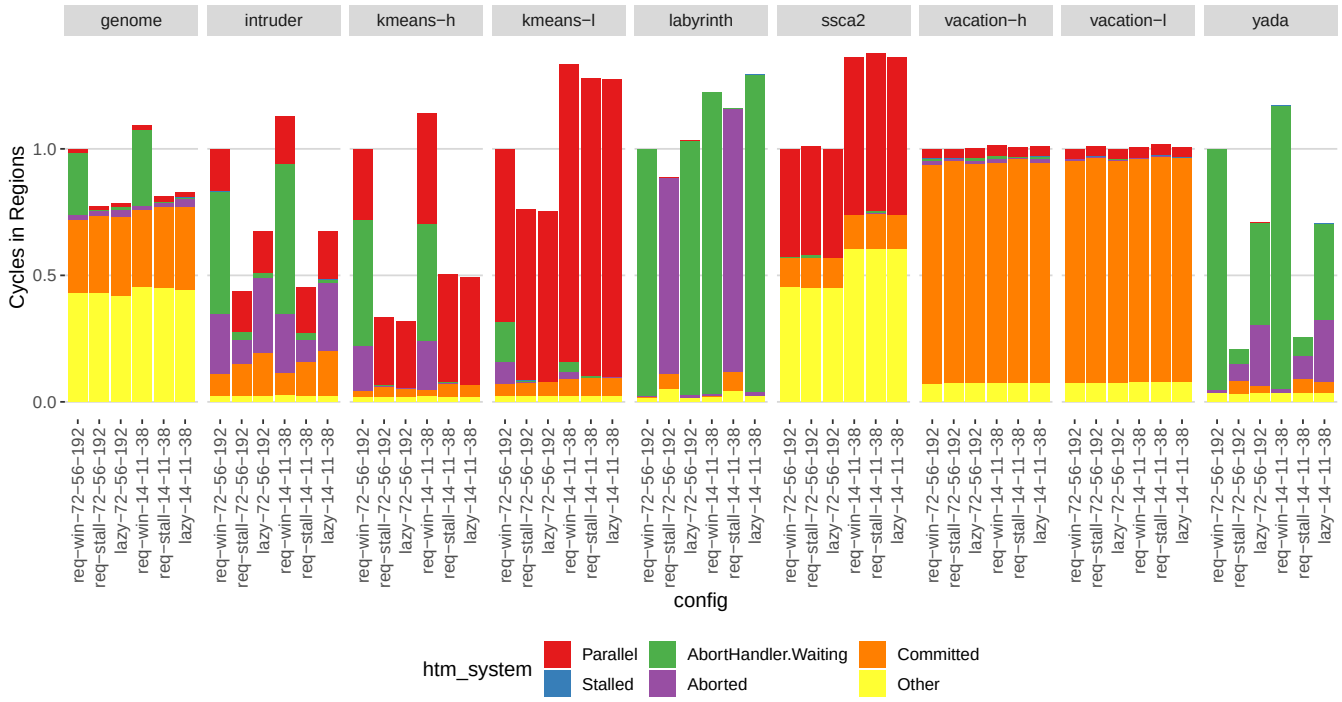
In the previous subsection, we used four core configurations with varying thread and core counts, and observed how contention (via higher core count) is affected by changes in the microarchitecture. In this section, we show the relative performance of three varying HTM design points —our *requester-wins* baseline, a requester-stalls eager alternative, and an ideal lazy HTM design—, to analyze the interplay between the characteristics of the HTM system and the aggressiveness of the OoO cores. We show the results of our experiments in Fig. 6, where we can see relative execution time (Fig. 6a) and amount of aborts (Fig. 6b), for two opposite core configurations: our baseline *fat* core configuration with larger OoO structures (three left-most bars for each benchmark), and a *thin* core with 14 LQ, 11 SQ, and 38 ROB entries (three right-most bars). All results are normalized to the baseline (fat core and requester-wins). Fig. 6a shows the sum of cycles executed by all threads broken into code regions, while Fig. 6b provides further detail on the causes of abort. All results in this figure are for 16 cores and equal number of threads.

In Fig. 6a we observe that, when contention arises, both HTM systems that do not employ requester-wins are able to drastically reduce execution time in comparison to the baseline. In *kmeans-l, ssca2* and *vacation*, execution times seen for the slim core configuration is similar across the three HTM design points considered. In *vacation* and *ssca2* (low contention workloads), all HTM designs reach almost the same performance regardless of the microarchitecture of the cores. The case of *kmeans-l* deserves a closer look: we see that with bigger cores and req-wins, the performance loss with respect to req-stalls and lazy systems is wider than that seen for slim cores. The reason behind is that this benchmark spends most of its time executing parallel non-transactional code. This, added to the fact that the use of simpler cores decreases the amount of aborts and time spent waiting for fallback lock acquisitions with req-wins systems, makes the performance of the base system match that of the more advanced ones. We can see the reason for this reduction in overhead in Fig. 6b as the amount of aborts seen gets smaller and smaller for req-wins as a result of reducing the aggressiveness of the cores, as discussed in the previous section. The same applies to *ssca2* and *vacation*, as contention in all these benchmarks is low, making the particular characteristics of the HTM system employed less relevant, and putting the emphasis on the importance of ILP exploitation.

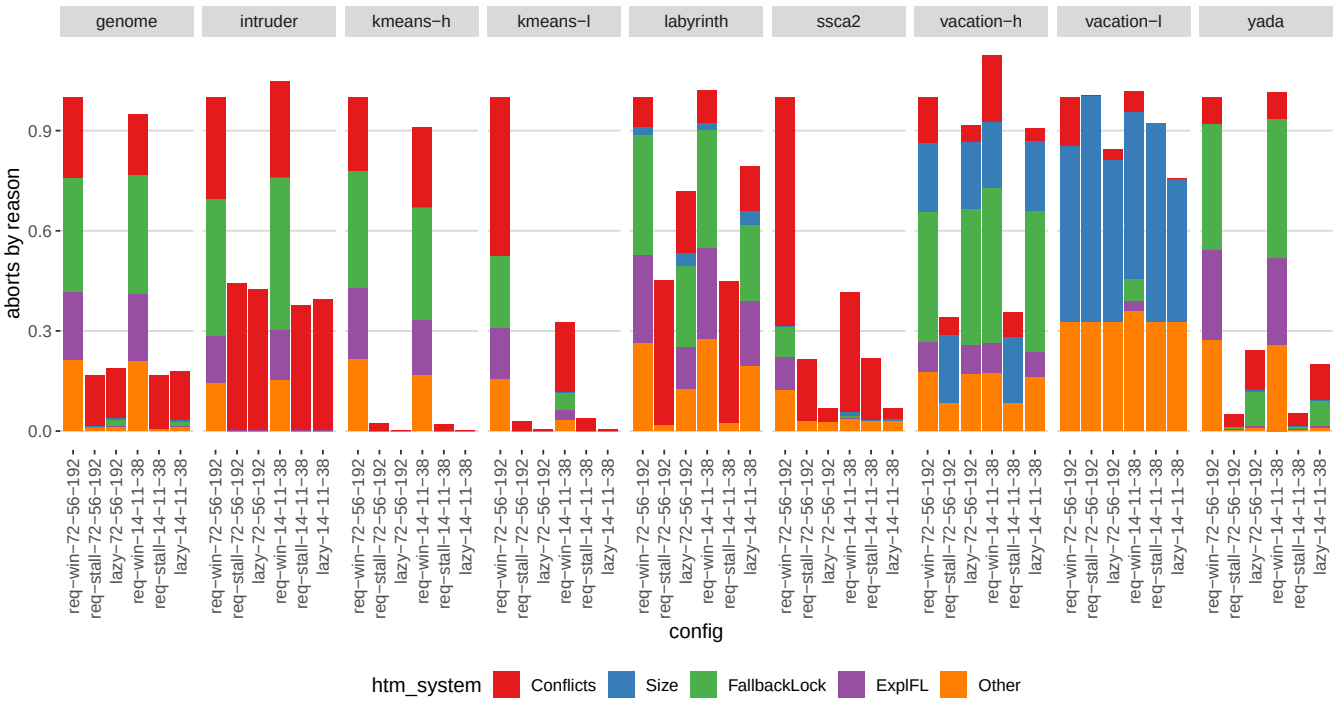
In high-contention workloads like *genome, intruder* and *yada*, we observe that lazy and req-stalls designs see little degradation when moving from big to little cores, unlike in req-wins systems –which invariably see a performance drop as cores get thinner. In *yada*, we observe that the requester-stalls system sees its execution time increased around 5% while the baseline degradation is around 20%, and the lazy CD system has the same performance. In *genome*, we see how both systems increase their execution time, but checking the figure we observe that in the case of the base system this is due to longer serialized sections (due to global lock acquisitions) and in the case of the other systems it is due to more time spent executing transactions that committed and mainly in the "other" region. This means that the main bottleneck in this benchmark for req-wins systems is the friendly-fire pathology [31] that other systems do not suffer. On the other hand, the performance degradation on lazy and req-stalls systems is mainly due to a reduction over the exploited ILP inside transactions with simpler microarchitectures. Nevertheless, in *intruder* and *yada*, the performance seen for the *lazy* system is the same regardless of the aggressiveness of the cores. We observe that the time spent executing the parallel region that is non-transactional is increased as less ILP can be exploited. If we dive deeper in Fig. 6a and 6b we can see that for both benchmarks, req-stall and lazy systems were able to reduce the contention with a simpler architecture. This translates to less transactional work being wasted and therefore more cycles are executed in parallel and less work is discarded, exploiting more TLP. So, as in these benchmarks most of the execution time is spent on transactional code execution, as explained in [47], the ability to leverage as much parallelism between critical sections as possible is of great importance. Because of that, the slowdown produced by less ILP being exploited is compensated in execution time by the capability to avoid wasting transactional work.

In *labyrinth* we see that the lazy system performs worse than req-wins in terms of total amount of cycles. The explanation for this situation is due to the increase over the cycles that aborted. In *labyrinth*, using eager or lazy conflict detection mechanisms would not avoid the conflicts between transactions as the main transaction reads the whole shared data structure at the very beginning of the transaction and it gets modified, so delaying the publication of memory instructions does not have any positive effect. In fact, Fig. 6b shows how aborts due to conflicts in lazy CD system though fewer aborts are produced in the req-wins system. Then, even if there are fewer aborts, the threads still usually have to wait on the fallback lock to execute, as shown in Fig. 6a but with the difference that conflict detection is delayed until commit time. This implies that transactions usually execute during more time until the conflict (which is going to happen anyway) is detected increasing the amount of aborted cycles executed.

In general terms, the number of aborts in requester-stalls and lazy CD systems are more stable than in the requester-wins-based system. In fact, we saw that in some cases, while in the req-wins system the number of aborts increases, in lazy or req-stalls, they decrease. Performance in some other cases does not decrease whenever a less aggressive core microarchitecture is used. This mostly happens in high-contended workloads as TLP exploitation is improved with these systems. As the fallback lock is acquired fewer times, the amount of time spent executing serialized code is reduced, making ILP exploitation less worthwhile and performance loss due to OoO execution aggressiveness reduction less noticeable.



(a) Relative execution time in different execution regions (Normalized to base configuration req-win 72LQ, 56 SQ, 192 ROB).



(b) Relative number of aborts and the reason that caused it (Normalized to base configuration req-win 72LQ, 56 SQ, 192 ROB).

Fig. 6: Results in terms of execution cycles and aborts using varying HTM systems and processor configurations.

The fact that the abort rate is the same (or even less) using lazy conflict detection with simpler cores and more complex ones allows us to think that some of the conflicts are produced due to the specific moment that a memory operation is published to other cores. While in eager conflict detection the memory operations are published in a specific ordering, in lazy conflict detection systems, these are published all in parallel at the very end of the transaction. This can reduce the number of aborts produced in two ways: reducing the number of aborts produced due to the ordering between transactions; and reducing the window of vulnerability, and therefore, friendly-fire performance pathology, as explained in previous sections.

V. Conclusions and future work

In this work, we analyzed the interactions between the mechanisms implemented by contemporary OoO cores to exploit ILP, and the HTM support aimed at exploiting TLP at a lower programming complexity.

Our analysis confirms that when the execution of transactional workloads exhibits low contention, the performance improvements achieved by increasing the number of threads/cores to exploit more TLP exceed the gains that can be attained by more complex OoO cores through the exploitation of more ILP. This work quantitatively shows that, in the case of lightly contended workloads, integrating a higher number of simpler OoO cores on a chip under a given transistor budget can provide better throughput and result in higher efficiency than opting for fewer cores with a more aggressive microarchitecture. On the other hand, in workloads with high contention, performance improvements brought by more aggressive OoO pipelines are of importance, as more complex ILP cores can accelerate execution in two ways: i) by reducing friendly-fire aborts in a requester-wins HTM system (as the window of vulnerability is shrunk as transactions take fewer cycles to complete), and ii) by reducing the synchronization overhead due to threads waiting on the fallback lock to be released to resume parallel execution. An interesting observation in this study is that, under certain conditions and workload characteristics, reducing the aggressiveness of the processing cores may lead to higher contention.

Additionally, we analyzed the interplay between the choice of conflict detection of the HTM implementation and the aggressiveness of the processing cores. We observed that in low-contended workloads, all HTM designs, from the more complex lazy or requester-stalls approaches to the simplest requester-wins solution, perform at par. On the other hand, this work not only confirms the superior performance in high contention workloads of the requester-stalls and lazy detection policies in comparison to requester-wins, but also that there is little difference in terms of contention and performance despite significant reductions in the ability to exploit ILP, making it less relevant in favour of TLP.

Finally, our analysis also reveals that the load and store queues (LQ and SQ, respectively) are often underutilized when executing the majority of the STAMP transactional workloads. This result suggests that proper management of the sizes of the LQ and SQ during HTM execution could bring considerable energy savings to HTM implementations

Although this work has yielded valuable insights regarding the interaction between OoO cores and HTM support, numerous opportunities for additional research exist. A promising prospective path is to investigate how dynamic adaptation mechanisms can improve energy efficiency by throttling the aggressiveness of selected cores, via reconfiguration of key OoO structures, in response to the changing contention level of transactional workloads. A further avenue of investigation entails the examination of the potential benefits that multi-threaded transactional workloads might accrue when executed on modern heterogeneous systems characterized by the coexistence of diverse core types.

References

[1] A. Wang, M. Gaudet, P. Wu, J. N. Amaral, M. Ohmacht, C. Barton, R. Silvera, and M. Michael, "Evaluation of blue gene/q hardware support for transactional memories," in *Proceedings of the 21st International Conference on Parallel Architectures and Compilation Techniques*, ser. PACT '12. Association for Computing Machinery, 2012, p. 127–136.

[2] "Intel xeon e7-4809 specification sheet," https://ark.intel.com/content/www/es/es/ark/products/84676/intel-xeon-processor-e7-4809-v3-20m-cache-2-00-ghz.html, 2015, accessed: 21/05/2021.

[3] "Arm transactional memory extensions documentation." https://developer.arm.com/documentation/101028/0012/16–Transactional-Memory-Extension–TME–intrinsics, 2020, accessed: 27/09/2021.

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

[5] M. M. Waliullah and P. Stenstrom, "Intermediate checkpointing with conflicting access prediction in transactional memory systems," in *Proceedings of the 22nd IEEE International Parallel and Distributed Processing Symposium*, April 2008.

[6] R. Titos-Gil, A. Negi, M. E. Acacio, J. M. García, and P. Stenstrom, "Zebra: Data-centric contention management in hardware transactional memory," *IEEE Transactions on Parallel and Distributed Systems*, vol. 25, no. 5, pp. 1359–1369, 2014.

[7] S. Sanyal, S. Roy, C. A., O. S. Unsal, and M. Valero, "Clock gate on abort: Towards energy-efficient hardware transactional memory," in *2009 IEEE International Symposium on Parallel Distributed Processing*, 2009, pp. 1–8.

[8] M. Lupon, G. Magklis, and A. González, "A dynamically adaptable hardware transactional memory," in *2010 43rd Annual IEEE/ACM International Symposium on Microarchitecture*, 2010, pp. 27–38.

[9] "Arm big.little technology. processing architecture for power efficiency and performance," https://www.arm.com/technologies/big-little, 2023, accessed: 01/11/2023.

[10] "Arm dynamiq technology." https://www.arm.com/technologies/dynamiq, 2023, accessed: 01/11/2023.

[11] "Alder lake architecture overview." https://www.intel.com/content/www/us/en/products/platforms/details/alder-lake-ps.html, 2023, accessed: 01/11/2023.

[12] V. Nicolás-Conesa, R. Titos-Gil, R. Fernández-Pascual, A. Ros, and M. E. Acacio, "Analysis of the interactions between ilp and tlp with hardware transactional memory," in *2022 30th Euromicro International Conference on Parallel, Distributed and Network-based Processing (PDP)*, 2022, pp. 157–164.

[13] J. Shen and M. H. Lipasti, *Modern Processor Design: Fundamentals of Superscalar Processors*, ser. Electrical and Computer Engineering. McGraw-Hill Companies,Incorporated, 2005. [Online]. Available: https://books.google.es/books?id=Nibfj2aXwLYC

[14] S. Lee, *Design of Computers and Other Complex Digital Devices*. Prentice Hall, 2000. [Online]. Available: https://books.google.es/books?id=xgtTAAAAMAAJ

[15] J. Ortega-Lopera, M. Anguita-López, and A. Prieto-Espinosa, *Arquitectura de computadores*. Thomson, 2005. [Online]. Available: https://books.google.es/books?id=6WlSsQFBfNcC

[16] J. L. Hennessy and D. A. Patterson, *Computer Architecture: A Quantitative Approach*. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 2019.

[17] I. Park, C. L. Ooi, and T. Vijaykumar, "Reducing design complexity of the load/store queue," in *Proceedings. 36th Annual IEEE/ACM International Symposium on Microarchitecture, 2003. MICRO-36.* IEEE, 2003, pp. 411–422.

[18] D. Kanter, "Analysis of haswell's transactional memory." https://www.realworldtech.com/haswell-tm/3/ , 2012, accessed: 25/05/2021.

[19] R. Rajwar, "Speculation-based techniques for transactional lock-free execution of lock-based programs," Ph.D. dissertation, Citeseer, 2002.

[20] T. Harris, J. Larus, and R. Rajwar, *Transactional Memory, 2nd Edition*, 2nd ed. Morgan and Claypool Publishers, 2010.

[21] B. H. Bloom, "Space/time trade-offs in hash coding with allowable errors," *Commun. ACM*, vol. 13, no. 7, p. 422–426, Jul. 1970. [Online]. Available: https://doi.org/10.1145/362686.362692

[22] R. Titos-Gil, R. Fernández-Pascual, A. Ros, and M. E. Acacio, "Concurrent irrevocability in best-effort hardware transactional memory," *IEEE Transactions on Parallel and Distributed Systems*, vol. 31, no. 6, pp. 1301–1315, 2019.

[23] R. Guerraoui, M. Herlihy, M. Kapalka, and B. Pochon, "Robust contention management in software transactional memory," in *Proceedings of the OOPSLA 2005 Workshop on Synchronization and Concurrency in Object-Oriented Languages (SCOOL'05)*, ser. OOPSLA '05, no. CONF, 2005.

[24] G. Blake, R. G. Dreslinski, and T. Mudge, "Proactive transaction scheduling for contention management," in *Proceedings of the 42nd Annual IEEE/ACM International Symposium on Microarchitecture*, ser. MICRO 42. Association for Computing Machinery, 2009, p. 156–167.

[25] C. Piatka, R. Amslinger, F. Haas, S. Weis, S. Altmeyer, and T. Ungerer, "Investigating transactional memory for high performance embedded systems," in *Architecture of Computing Systems–ARCS 2020: 33rd International Conference, Aachen, Germany, May 25–28, 2020, Proceedings 33*. Springer, 2020, pp. 97–108.

[26] R. Guerraoui, M. Herlihy, and B. Pochon, "Polymorphic contention management," in *International Symposium on DIStributed Computing*. Springer, 2005, pp. 303–323.

[27] 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. 1–24, 2018.

[28] M. Ansari, M. Luján, C. Kotselidis, K. Jarvis, C. Kirkham, and I. Watson, "Steal-on-abort: Improving transactional memory performance through dynamic transaction reordering," in *High Performance Embedded Architectures and Compilers: Fourth International Conference, HiPEAC 2009, Paphos, Cyprus, January 25-28, 2009. Proceedings 4*. Springer, 2009, pp. 4–18.

[29] W. Maldonado, P. Marlier, P. Felber, A. Suissa, D. Hendler, A. Fedorova, J. L. Lawall, and G. Muller, "Scheduling support for transactional memory contention management," *ACM Sigplan Notices*, vol. 45, no. 5, pp. 79–90, 2010.

[30] N. Diegues, P. Romano, and S. Garbatov, "Seer: Probabilistic scheduling for hardware transactional memory," *ACM Transactions on Computer Systems (TOCS)*, vol. 35, no. 3, pp. 1–41, 2017.

[31] J. Bobba, K. E. Moore, H. Volos, L. Yen, M. D. Hill, M. M. Swift, and D. A. Wood, "Performance pathologies in hardware transactional memory," *SIGARCH Comput. Archit. News*, vol. 35, no. 2, p. 81–91, Jun. 2007. [Online]. Available: https://doi.org/10.1145/1273440.1250674

[32] L. Yen, J. Bobba, M. Marty, K. Moore, H. Volos, M. Hill, M. Swift, and D. Wood, "Logtm-se: Decoupling hardware transactional memory from caches," in *2007 IEEE 13th International Symposium on High Performance Computer Architecture*, 01 2007, pp. 261–272.

[33] S. A. R. Jafri, G. Voskuilen, and T. Vijaykumar, "Wait-n-gotm: improving htm performance by serializing cyclic dependencies," *ACM SIGPLAN Notices*, vol. 48, no. 4, pp. 521–534, 2013.

[34] H. E. Ramadan, C. J. Rossbach, and E. Witchel, "Dependence-aware transactional memory for increased concurrency," in *2008 41st IEEE/ACM International Symposium on Microarchitecture*. IEEE, 2008, pp. 246–257.

[35] A. Armejach, R. Titos-Gil, A. Negi, O. S. Unsal, and A. Cristal, "Techniques to improve performance in requester-wins hardware transactional memory," *ACM Trans. Archit. Code Optim.*, vol. 10, no. 4, dec 2013. [Online]. Available: https://doi.org/10.1145/2541228.2555299

[36] R. Titos-Gil, R. Fernández-Pascual, A. Ros, and M. E. Acacio, "Detras: Delaying stores for friendly-fire mitigation in hardware transactional memory," *IEEE Transactions on Parallel and Distributed Systems*, vol. 33, no. 1, pp. 1–13, 2022.

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

[38] D. Ponomarev, G. Kucuk, and K. Ghose, "Reducing power requirements of instruction scheduling through dynamic allocation of multiple datapath resources," in *Proceedings. 34th ACM/IEEE International Symposium on Microarchitecture. MICRO-34*, 2001, pp. 90–101.

[39] D. Sylvester and H. Kaul, "Power-driven challenges in nanometer design," *IEEE Design Test of Computers*, vol. 18, no. 6, pp. 12–21, 2001.

[40] A. Ros and S. Kaxiras, "The superfluous load queue," in *2018 51st Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*. IEEE, 2018, pp. 95–107.

[41] G. Dimova, M. Marinova, and V. Lazarov, "Performance evaluation of heterogeneous microprocessor architectures," *Journal of Information Technologies and Control*, vol. YEAR X No. 3, pp. 31–36, 01 2012.

[42] H. M. Mathis, A. E. Mericas, J. D. McCalpin, R. J. Eickemeyer, and S. R. Kunkel, "Characterization of simultaneous multithreading (smt) efficiency in power5," *IBM Journal of Research and Development*, vol. 49, no. 4.5, pp. 555–564, 2005.

[43] 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.

[44] M. Martin, D. Sorin, B. Beckmann, M. Marty, M. Xu, A. Alameldeen, K. Moore, M. Hill, and D. Wood, "Multifacet's general execution-driven multiprocessor simulator (gems) toolset," *SIGARCH Computer Architecture News*, vol. 33, pp. 92–99, 11 2005.

[45] R. Titos-Gil, R. Fernández-Pascual, A. Ros, and M. E. Acacio, "Pftouch: Concurrent page-fault handling for intel restricted transactional memory," *Journal of Parallel and Distributed Computing*, vol. 145, pp. 111–123, 2020.

[46] 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 *2014 IEEE 28th International Parallel and Distributed Processing Symposium*, 2014, pp. 615–624.

[47] Chi Cao Minh, JaeWoong Chung, C. Kozyrakis, and K. Olukotun, "Stamp: Stanford transactional applications for multi-processing," in *2008 IEEE International Symposium on Workload Characterization*, 2008, pp. 35–46.

[48] M. Herlihy, V. Luchangco, M. Moir, and W. N. Scherer III, "Software transactional memory for dynamic-sized data structures," in *Proceedings of the twenty-second annual symposium on Principles of distributed computing*, 2003, pp. 92–101.