Analysis of 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 Architecture and Parallel Systems Group University of Murcia
Murcia, Spain
{victor.nicolasc, rtitos, ricardof, aros, meacacio} @um.es

***Abstract*—Hardware Transactional Memory (HTM) allows the use of transactions by programmers, making parallel programming easier and theoretically obtaining the performance of fine-grained locks. However, transactions can abort for a variety of reasons, resulting in the squash of speculatively executed instructions and the consequent loss in both performance and energy efficiency. Among the different sources of abort, conflicting concurrent accesses to the same shared memory locations from different transactions are often the prevalent cause.**

In this work, we characterize, for the first time to the best of our knowledge, how the aggressiveness of the cores in terms of exploiting instruction-level parallelism can interact with thread-level speculation support brought by HTM systems. We observe that altering the size of the structures that support out-of-order and speculative execution changes the number of aborts produced in the execution of transactional workloads on a best-effort HTM implementation. Our results show that a small number of powerful cores is more suitable for high-contention scenarios, whereas under low contention it is preferable to use a larger number of less aggressive cores. In addition, an aggressive core can lead to performance loss in medium-contention scenarios due to an increase in the number of aborts. We conclude that depending on contention, a careful choice over processor aggressiveness can reduce abort ratios.

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

I. Introduction

Architectural support for transactional memory (TM), available in commercial multicore processors [1]–[3], enables easier synchronization of parallel programs, by shifting to the hardware the responsibility of guaranteeing that certain regions of code appear to be executed atomically and in isolation with respect to any other thread. The idea behind TM is that the programmer only needs to declare which code needs to be synchronized (i.e., *transactions*), leaving the burden of how to achieve it to the underlying TM system [4]. Although hardware TM (HTM) systems can provide the aforementioned transactional properties at low overhead by leveraging private caches and coherence protocols, their performance and efficiency get hindered by the occurrence of *conflicts*, i.e., concurrent memory accesses to shared data originating from different threads that pose a risk to the atomicity and isolation of a given transaction. Such conflicts must be detected and dealt with at runtime, typically resulting in the abort and re-execution of competing transactions. Aborts lead to a waste of energy and time due to the speculative work discarded.

Existing HTM implementations adopt an eager approach to detect conflicts as each memory access instruction performs in cache. Once detected, a conflict can be resolved in a number of ways, but the simplest approach, found in commercial processors, opts for favoring the requesting cache. Under this *requester-wins* policy, the transaction running on a core is aborted when its private cache controller observes a conflicting coherence request: an exclusive-ownership request for a block that has been read by the transaction (remote write, local read), or a request for a block that is currently speculatively modified in cache (remote read/write, local write).

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 [5], 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 [6]–[9], which are far from representative of today's execution cores.

In contrast, in this work, we analyze the interactions between the mechanisms used by modern commercial multicore processors to bring high performance at core level (aimed at exploiting Instruction-Level Parallelism or ILP) with the HTM support intended to provide high performance at system level (aimed at exploiting Thread-Level Parallelism or TLP). We find that there are relevant interactions between the HTM system and processor aggressiveness, including instruction-level speculation brought by out-of-order (OoO) cores.

This study shows that the characteristics of the processing core have implications on HTM performance and contention between transactions. Under high contention scenarios, our results show that it is preferable to use fewer powerful cores,

This work was supported by the Spanish MCIU and AEI, as well as European Commission FEDER funds, under grant RTI2018-098156-B-C53, and the European Research Council (ERC) under the Horizon 2020 research and innovation program (grant agreement No 819134).

being the opposite of low contention. Interestingly, in certain scenarios with low to medium contention we observed that overall performance may not benefit from employing more complex cores, as the loss caused by an increase in the number of aborts offsets the gains achieved by exploiting ILP more aggressively. Our work suggests that proper management of the aggressiveness of the cores during HTM execution in modern multicore designs is a promising approach as it can bring performance and efficiency gains.

In the remaining sections we will give some background about instruction-level parallelism and hardware transactional memory, relating it to the state of the art (Section II). Afterward, we will show the system and methodology used to obtain the results from this study (Section III. Then we will display the results obtained from the experiments (Section IV. And finally, we will discuss the conclusions derived from this work (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 [10]–[13], 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) [13], [14], 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 [15]. 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 [16]. 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 [15], [17], [18].

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 [19]. 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 which all transactions must subscribe to (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, e.g. [20] or [21]. Contention managers act whenever a transaction aborts due to conflicts and try to avoid a new conflict or even the same to happen, usually by making transaction wait (back-off) before re-executing [22]. Some techniques try to avoid energy waste even while using a contention manager, e.g., turning the CPU on a low-power mode while it is waiting to re-execute a transaction, as explained in [8].

There have been previous works that tried to resize the LSQ and ROB based on their occupation to reduce energy consumption without negatively impacting performance. More precisely, [23] observes that these structures are not totally used most part of the time, and so reducing them causes passive and active energy savings [24] and does not entail a really great performance loss. The same work also shows that occupation is not the same on every application and that within the same application there are moments of high and low occupancy and structures size can vary depending on this. 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 [23], some other works have also analyzed the occupation of these structures while executing non-transactional code. Dimova *et al.* [25] study the effects over performance of modifying ROB size too and conclude that increasing ROB beyond certain limits can even hurt performance. In Mathis *et al.* [26], 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 [27] 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 [28], with the CPU models of gem5 appropriately adapted to support HTM. 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.

Our HTM model resembles the Intel RTM (Restricted Transactional Memory) ISA extensions introduced in the Haswell microarchitecture [2]. 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 evicted from the L1 cache [18], [29]. Loads are added to the read set at the moment they are retired from the pipeline. 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).

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.

TABLE I
System parameters.

Out-of-order baseline core Settings	
Cores	out-of-order (execute/commit width: 8)
Load queue	72
Store queue + store buffer	56
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

LSQ and ROB occupancy: 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: To observe how contention affects the obtained results, we use two different systems sizes (4 and 16 cores) and run the benchmarks at the corresponding thread counts matching the number of cores.

The benchmark suite that has been used for this study is STAMP [30], which includes diverse workloads with different characteristics in terms of contention, transaction size, read-write set size, etc. The recommended medium inputs [30] 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).

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 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 ×5.7 and ×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

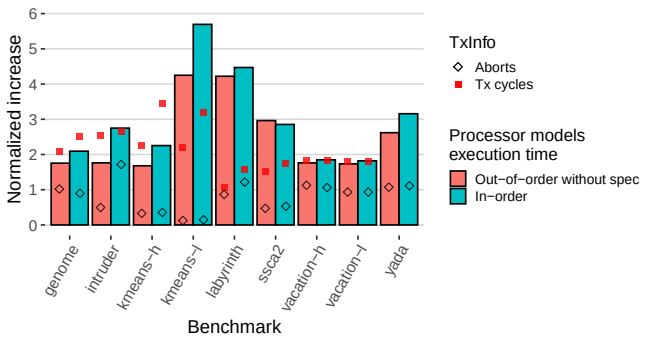

Fig. 1. Differences between out-of-order, out-of-order without instruction level speculation and in-order core models. Red squares represent cycles spent in transactions, black diamonds are the number of aborts and the bars represent execution time. All data is normalized respect to OoO model with speculation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

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 Figure 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 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

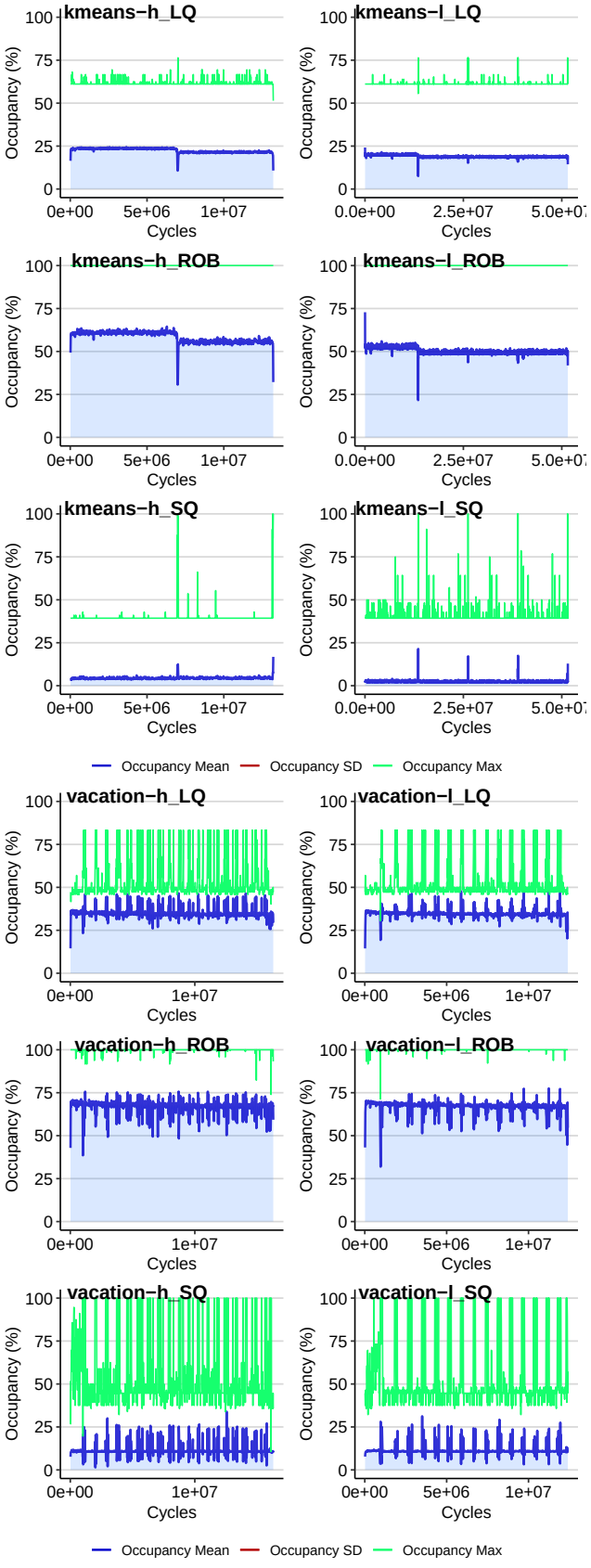

Fig. 2. Occupancy levels in microarchitectural OoO structures while running *kmeans* and *vacation*. The occupation is shown as a percentage of the total structure's size.

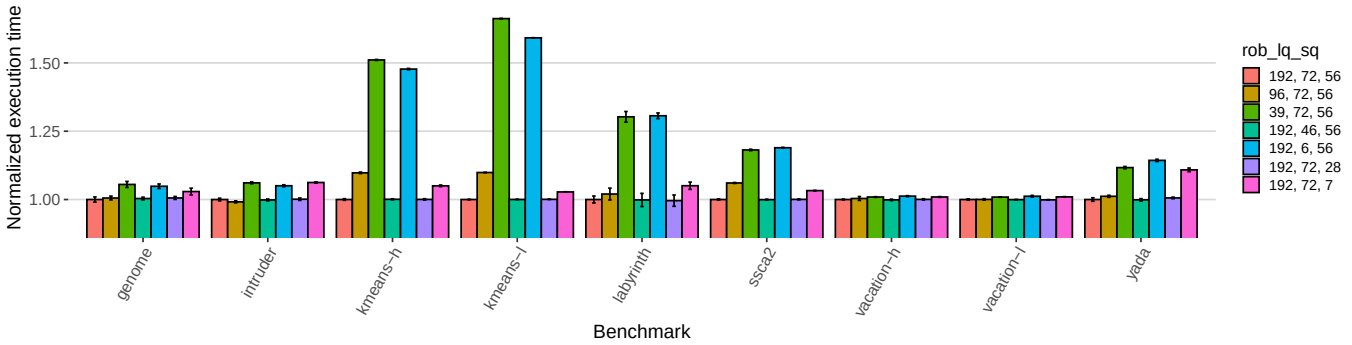

Fig. 3. Results of changing the OoO structures size executing STAMP applications. This figure shows performance expressed in cycles to execute the application. Results are normalized to the base configuration and all experiments are run with 4 cores.

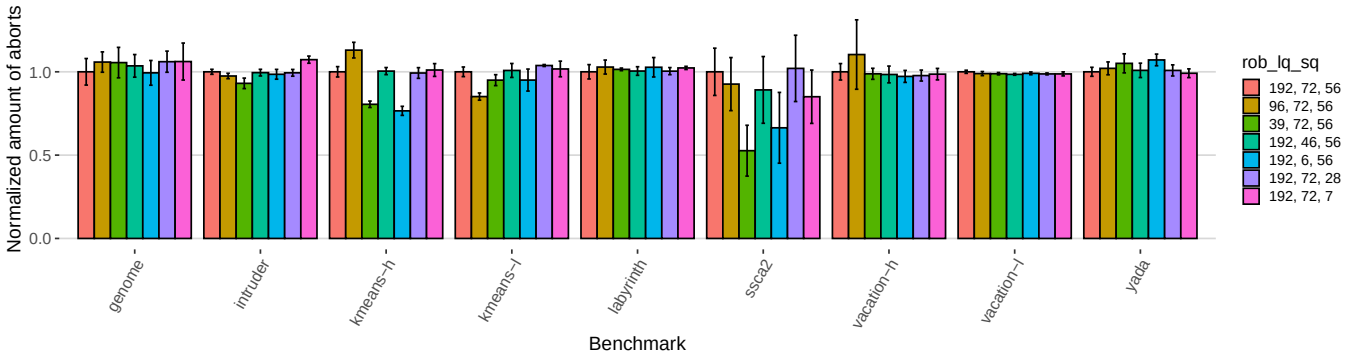

Fig. 4. Results of changing the OoO structures size executing STAMP applications. This figure shows contention expressed in aborts produced during execution. Results are normalized to the base configuration and all experiments are run with 4 cores.

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

Figures 3 and 4 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 SQ size seems 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. This confirms the results from section IV-A where we saw that the maximum occupation level of the LQ was about 60%. 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 in all 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

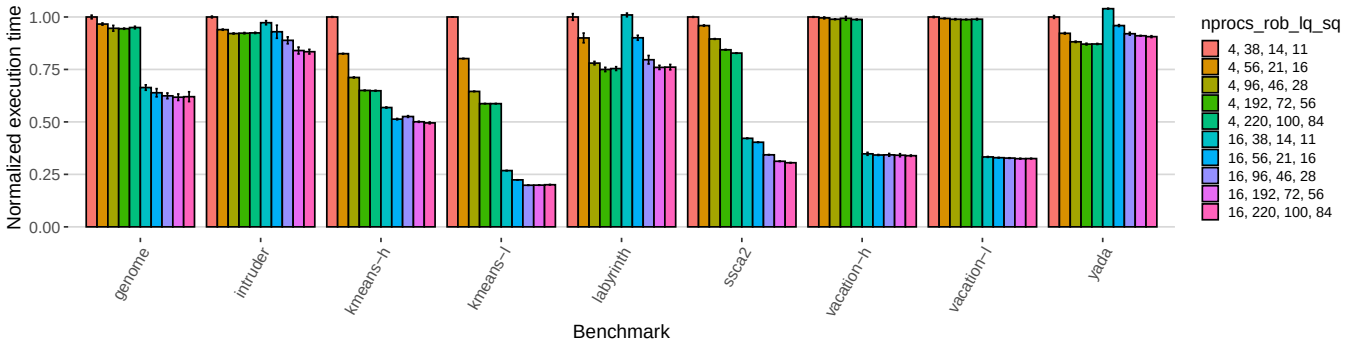

Fig. 5. Relative execution time for five core configuration in 4- and 16-core systems, matching the thread and core counts. Results are normalized to the 4-core system with the simplest core configuration.

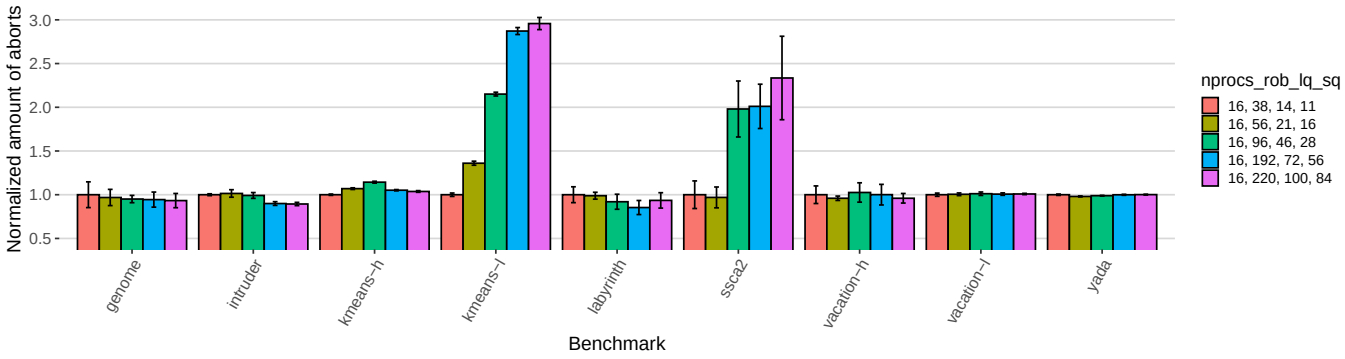

Fig. 6. Relative number of aborts for five core configurations in the 16-core system, normalized to the simplest core configuration.

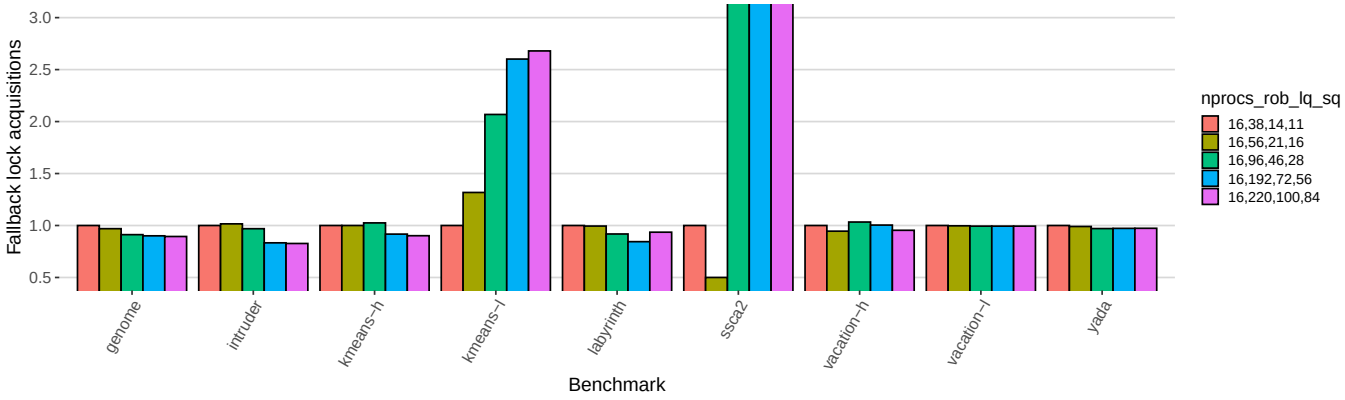

Fig. 7. Relative number of acquisitions of the fallback lock for five core configurations in the 16-core system, normalized to simplest core configuration.

exploit less ILP is lower in *kmeans-h* as a result of its much smaller fraction in non-transactional code than *kmeans-l*.

Interestingly, we see that a benchmark like *vacation* does 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. Together with the low occupation shown earlier, this points to the out-of-order pipeline being poorly utilized 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. 5 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. 6 and 7 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. 5 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* [31] 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. 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 3X increase in the number of aborts and 2.5X 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 2-3X 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. 6 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. 5 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. 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. 6 and Fig. 7, 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.

V. Conclusion

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, 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 by extracting 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 transaction 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. Finally, our analysis also reveals that the load and store queues (LQ and SQ) are often underutilized when executing 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.

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] A.-R. Adl-Tabatabai, C. Kozyrakis, and B. Saha, "Unlocking concurrency: Multicore programming with transactional memory," *Queue*, vol. 4, no. 10, p. 24–33, Dec. 2006. [Online]. Available: https://doi.org/10.1145/1189276.1189288

[5] 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.

[6] 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.

[7] 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.

[8] 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.

[9] 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.

[10] 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

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

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

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

[14] 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.

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

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

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

[18] 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

[19] 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.

[20] 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.

[21] 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.

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

[23] 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.

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

[25] 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.

[26] 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.

[27] 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.

[28] 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.

[29] 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.

[30] 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.

[31] 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.