2011 IEEE International Parallel & Distributed Processing Symposium

The Impact of Non-coherent Buffers on Lazy Hardware Transactional Memory Systems

Anurag Negi* Rubén Titos-Gil† Manuel E. Acacio† José M. García† Per Stenstrom*
Universidad de Murcia†
*Chalmers University of Technology**
{*rtitos,meacacio,jmgarcia*}*@ditec.um.es*
{*negi,per.stenstrom*}*@chalmers.se*

Abstract

When supported in silicon, transactional memory (TM) promises to become a fast, simple and scalable parallel programming paradigm for future shared memory multiprocessor systems. Among the multitude of hardware TM design points and policies that have been studied so far, lazy conflict resolution designs often extract the most concurrency, but their inherent need for lazy versioning requires careful management of speculative updates. In this paper we study how coherent buffering, in private caches for example, as has been proposed in several hardware TM proposals, can lead to inefficiencies. We then show how such inefficiencies can be substantially mitigated by using complete or partial non-coherent buffering of speculative writes in dedicated structures or suitably adapted standard per-core write-buffers. These benefits are particularly noticeable in scenarios involving large coarse grained transactions that may write a lot of non-contended data in addition to actively shared data. We believe our analysis provides important insights into some overlooked aspects of TM behaviour and would prove useful to designers wishing to implement lazy TM schemes in hardware.

1. Introduction

Transactional memory (TM) [7] enables programming constructs that provide optimistic concurrency control in multithreaded applications and largely eliminate the need for programmers to worry about semantics and safety of locks. Regions of code that rely upon a consistent view of shared data are enclosed in atomic blocks (or transactions) that guarantee properties of atomicity and isolation. By leveraging traditional cache coherence mechanisms, hardware transactional memory (HTM) systems can support such constructs efficiently.

Lazy HTM protocols [7] commit updates made to shared data in a transaction at the end of its execution. Any concurrent transaction that may have a conflict (data race) with the committing one is aborted. To do so while preserving TM semantics requires establishment of a logical global order over all transactions that commit. Non-concurrent transactions represent the trivial case. The problem becomes more interesting and complex when we consider concurrent transactions attempting to commit simultaneously. A solution in such a case would be to have a global arbiter allow one transaction to commit at a time [6], [1]. Another solution, as presented in [3] allows a greater degree of parallelism at commit time in multi-banked directory based distributed shared memory (DSM) architectures. Both approaches gain exclusive ownership to the entire write-set of a transaction before its commit can be considered complete. This results in a burst of coherence activity at commit time, which, as we highlight in this study, can be reduced significantly through the use of non-coherent write buffers, i.e. buffers that contain writes from the processor before these enter the cache hierarchy. Additionally, such buffers also improve the cache hit rates during transactional execution in high-contention scenarios, since aborts would only result in a buffer flush and not in cache line invalidations. Although write buffering is commonplace in microprocessors, its effects on the behaviour of transactions have not been investigated in depth in HTM literature so far and, by our estimates, its impact on performance is large enough to merit a detailed study.

A large fraction of data accessed by a typical coarse grained transaction is either thread-private or not actively contended during the lifetime of the transaction. Allowing early interaction of writes to such data with coherence mechanisms results in unnecessary

1530-2075/11 $26.00 © 2011 IEEE
DOI 10.1109/IPDPS.2011.205

pessimistic actions to be taken to ensure no violations of TM guarantees occur. Write-backs and coherence state transitions must occur to allow detection of races. Such transitions are then reverted on commit. In workloads with high concurrency and large transactions such actions are particularly wasteful. On the other hand, in workloads that exhibit high contention, a large number of aborts would occur causing updated lines to be discarded through invalidation. When the transaction re-executes, these lines are bound to be accessed again, causing expensive misses, lengthened execution times and exacerbated contention. In summary, coherent buffering in private caches excites what can be regarded as pathological case for such workloads and, moreover, amplifies its effect.

In this paper we attempt to quantify the cost of having coherent buffering for speculative data in lazy HTMs. We examine, in detail, the behaviour of such mechanisms in a directory based DSM HTM implementation. We then show that incorporation of non-coherent buffers can mitigate this effect, thereby increasing performance and improving other key design metrics. We believe this study would prove insightful to designers thinking about implementing lazy HTM designs in silicon.

2. Related Work

This work is a study of the impact of non-coherent write buffering in lazy HTM systems. The insights it encompasses apply to a large body of work done on the topic. The TCC proposal [5] implements coherent buffering in a private cache with global commit arbitration using a bus. A committing transaction writes back all its speculative updates to the shared memory hierarchy. A later proposal [3] provides a commit algorithm which allows for considerable parallelism in directory based DSM systems. It works by dividing the directory into several banks. Transactions can commit in parallel if they do not observe directory bank conflicts. Commit sequence numbers are assigned to prioritize transactions when such conflicts occur. Tomic et al. [11] describe an eager conflict detection design that commits transactions lazily, utilizing directory coherence in MESI based systems with two levels of private caching. Negi et al. developed a broadcast-based lazy commit protocol in [10] that eliminates the need for write-backs or cache-line invalidation messaging at commit. Sanyal et al. [13] present mechanisms to filter thread-local variables in lazy HTMs but need support mechanims in both the core and the operating system. Moreover, in several workloads separation between contended and non-contended data is not as clean as that between thread-local and shared data.

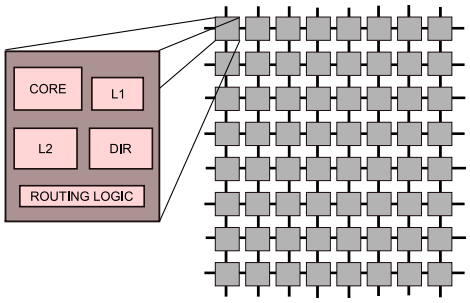

Figure 1: Tiled-CMP architecture used for this study.

One common characteristic in the proposals mentioned in this section is that they do not investigate different speculative buffering mechanisms which, as we show in this paper, can cause significant variation in key performance metrics.

3. Buffering speculative updates in coherent caches

We choose a tiled CMP design as reference because its modular nature has made it popular in several commercial many-core designs and the availability of reliable simulation models [9] makes comparison of various policies and architectural features less daunting. The basic architecture comprises several tiles overlaid over point-to-point interconnects forming a mesh-based network-on-chip. This arrangement is depicted in Figure 1. Each tile has a processing core, one level of private cache, a slice each of the shared inclusive level 2 cache and the directory and some routing logic. A MESI protocol utilizes the banked directory to keep private caches coherent.

To buffer speculative data in private caches, per-cache line meta-data is augmented with two bits, SR and SM, which indicate whether a line has been speculatively read or speculatively modified, respectively. During the course of execution of a transaction writes appear as non-invalidating reads to the coherence protocol. In order to preserve its last globally consistent value, a dirty (non-speculative) line is written back to the shared memory hierarchy prior to the first speculative update to it in a transaction, resulting in a downgrade of its coherence state from M to S. Commits imply acquisition of ownership over all lines with SM set, while aborts imply invalidation of all such lines.

Let us now consider the case presented in Figure 2. A line that is only written by one transaction (it is either thread-private or not actively shared in the

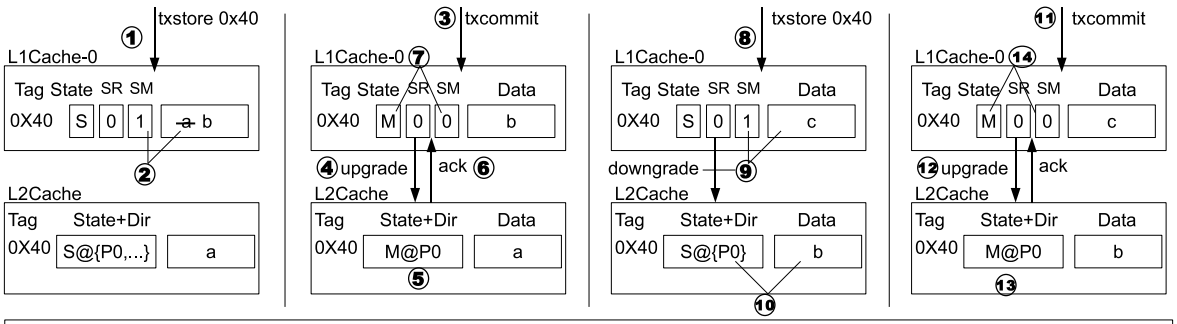

Figure 2: *Downgrade miss*: Redundant cache-state changes when a transaction eventually commits.

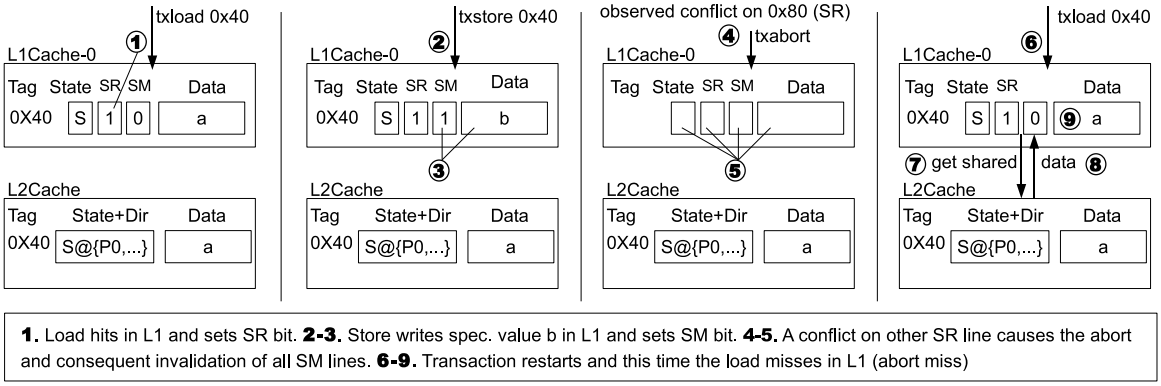

Figure 3: *Abort miss*: Invalidations that could be avoided with non-coherent buffering.

current phase of the workload) might, in the steady state, be found with high probability in M state in the private cache. To preserve the old content of the line in the absence of non-coherent write buffering, it must be written back prior to the write, resulting in a transition to S, as shown in Figure 2 (step 9). On commit we need to reacquire exclusive ownership to the line. Since such a line will not have any sharers, this work (M⟶S and S⟶M) to ensure no races exist is largely redundant. We refer to such events where unnecessary work prolongs the commit phase of a transaction as a *downgrade miss*. Coarse grained transactions that have a relatively large fraction of the write set as private data (stack, thread-local storage) are expected to show the most degradation in performance. In Section 5 we examine in detail the impact of downgrade misses and see that for applications like genome and vacation (refer to [2]) their elimination results in a significant contraction of commit delays.

Figure 3 depicts the alternate case. A transaction speculatively updates a non-contended line present in its cache and then aborts. As depicted in Figure 3 (step 6), the line would not be found in the private cache on re-execution as all lines in the write-set have now been invalidated. We refer to such an event as an *abort miss*. Such misses have also been referred to as *contamination* misses [12]. Workloads with large write sets and high contention over small amounts of shared data would experience the greatest drop in private cache hit rates. As Section 5 will show, elimination of such misses using non-coherent buffering results in a marked overall improvement in private cache hit rates.

Some transactional workloads also suffer from write-write conflicts. These are not true conflicts (not data races per se) but need to be resolved in in-validation based coherence protocols. Consider two transactions that write (but do not read) to a certain cache line. When one commits the other needs to be aborted since the invalidation message only tells us that the cache line might need merging and that cannot be handled without making the coherence protocol a lot more complex.

4. Use of non-coherent buffers

Private caches are present primarily to keep frequently used data close to the processor core. Their use in buffering uncommitted data should be made conservatively. Non-coherent write-buffers can be used to prevent transactional updates from polluting the coherent cache hierarchy. The idea can also be extended to inclusive two-level private caching schemes, wherein the first level private cache can be made non-coherent when handling transactional updates.

Non-coherent write buffering can be incorporated very simply into the design by capturing writes issued by the processor and then releasing those to the private cache in a controlled manner. In a lazy HTM, writes would be captured throughout the execution of a transaction or as long as buffer capacity is not exceeded. On commit, the buffer would be flushed causing all writes to enter the memory hierarchy as quickly as possible. On abort, the buffer contents would simply be discarded. Such a buffer would obviously also participate in any store forwarding mechanism. In fact, existing write buffering schemes could be suitably modified to enable such functionality.

It can be observed quite easily that unnecessary switches in coherence state and invalidations of aborts can be completely eliminated if write-sets are fully contained in write-buffers. Since speculative data can be recorded in the non-coherent buffer, we can eliminate write-backs and downgrades of M lines to shared (S) state. On commit, since the line would likely be present in the cache in the M state, it can simply be written into the private cache without any coherence action. An abort results in the speculative contents inside the write buffer to be discarded. No cache lines need invalidation and, thus, the transaction on re-execution would still find such lines in the private cache.

Another benefit of having non-coherent write buffers is reduction in the number of write-write conflicts, which are purely an artifact of using coherence messages to detect possible modifications to different parts of the same cache line. The cache, when it does not buffer any speculative updates, only records the read set of a transaction. Hence any invalidations resulting from transaction commits result in aborts only when there is a possible true data race, i.e. the write set of the committer conflicts with the read set of the other. We would like to point out that this does not eliminate conflicts due to reader-writer false sharing.

5. Methodology and Evaluation

In this section, we evaluate the performance implications of both coherent and non-coherent approaches to the buffering of speculative writes in a lazy HTM systems.

5.1. Experimental Setup

We use a full-system execution-driven simulator based on the Wisconsin GEMS tool-set [9], in conjunction with Virtutech Simics [8]. We use the detailed timing model for the memory subsystem provided by GEMS, with the Simics in-order processor model. Simics provides functional simulation of the SPARC ISA and boots an unmodified Solaris 10. The lazy HTM system modelled in this evaluation is an extension to the LL system considered by Bobba et al. [1], available in the GEMS v2.1 release. While Bobba's LL system models a private, per processor infinite write buffer, for this study we extended the simulator to precisely model finite buffering for transactional writes. We limited the capacity of the non-coherent write buffer, so that once it fills up, transactional stores happen in the private data cache. Unlike writes to the L1 cache, which need the line present in cache to be able to complete, writes to the write buffer proceed even if they encounter a miss in the L1 cache, when a non-blocking *prefetch-read* for the line is sent to the L2 cache in such case. We modified the replacement policy of the L1 data cache by giving the highest priority to speculatively written lines, in order to minimize the number of transactional overflows when executing large transactions. Nonetheless, we avoid serialization penalty due to limited buffering capacity when such overflows happen – as has been done in [1] by Bobba et al.– by incorporating an unlimited speculative victim buffer. In our simulations only yada experiences a few such evictions and a tiny victim buffer with 8 entries proves sufficient. We use an ideal book-keeping scheme to track read sets (*perfect signatures*) even when some speculatively read lines have been evicted, in an attempt to isolate our study from the effects of false conflicts arising from non-ideal signature schemes like bloom filters. A simple commit token algorithm is used to serialize transaction commits: Transactions arbitrate for the token using a zero-latency broadcast bus. Once the token is acquired, a transaction enters the commit phase and issues coherence requests to gain exclusive ownership over all lines in its write set.

We perform our experiments on a 16-core tiled CMP system, as described in Figure 1. We use a 16-core configuration with private L1I and L1D caches and a

Table 1: System parameters.

MESI Directory-based CMP	
Core Settings	
Cores	16, single issue in-order, non-memory IPC=1
Memory and Directory Settings	
L1 I&D caches	Private, 32KB, split 4-way, 1-cycle latency
Write Buffer	Non-coherent, private, 128 bytes
L2 cache	Shared, 512KB per tile, unified 8-way, 12 cycle-latency
L2 Directory	Bit vector, 6-cycle latency
Memory	4GB, 300-cycle latency
Network Settings	
Topology	2D Mesh
Link latency	1 cycle
Link bandwidth	40 bytes/cycle

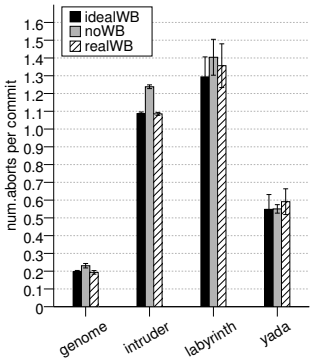

Figure 4: Contention in selected workloads.

shared, multi-banked L2 cache consisting of 16 banks of 512KB each (one L2 slice per tile). The L1 caches maintain inclusion with the L2. The cores and L2 cache banks are connected through a 2D mesh network. The private L1 caches are kept coherent through an on-chip directory (at L2 cache banks), which maintains a presence-bit vector of sharers and implements the MESI protocol.

Workloads. For this evaluation, we have selected seven transactional applications from the STAMP suite [2]: genome, intruder, kmeans, labyrinth, ssca2, vacation and yada. The application, bayes, was excluded since it exhibits unpredictable behaviour and high variability in its execution time [4], [10]. For kmeans and vacation, both high and low contention configurations were used, resulting in a total of 9 benchmarks. Small input parameters, detailed in [2], were used.

Buffering configurations. We consider three different buffering schemes for the lazy HTM system. In order to establish an upper bound on the performance achievable by the introduction of non-coherent buffering, we simulate an ideal write buffer of unlimited size (*idealWB*). At the other end of this design space, we

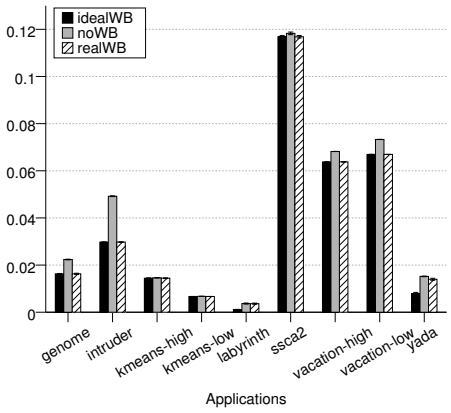

Figure 5: L1 data cache miss rates.

simulate a lazy HTM system that relies solely on its coherent buffering (private L1 caches) for storing speculative updates (*noWB*). The third scheme models a more realistic design (*realWB*) which combines a 128-byte, non-coherent write buffer and uses the L1 data cache in case the write buffer runs out of space.

5.2. Results and Discussion

In this section, we analyze the impact of the three buffering schemes described in the previous section and quantify the effectiveness of non-coherent buffers in improving cache performance and reducing the number of coherence actions required on commit.

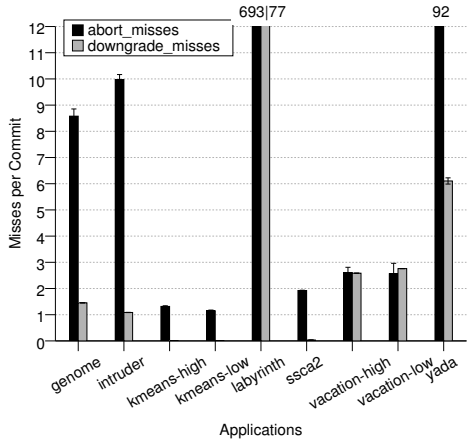

Figure 6: Downgrade and abort misses.

Figure 5 shows the average miss rate of L1 data caches for each STAMP benchmark. In Figure 6, we present the number of abort misses suffered on average by a transaction that restarted at least once. The same plot shows the average number of downgrade misses

per committed transaction. Figure 7 shows the execution time breakdown of all applications, normalized to the execution time of the configuration with no write buffer, running 16 threads. The execution time is broken down into eight components that indicate how threads spend their time. The first component (*barrier*) is a measure of the time spent waiting at barriers. The second component (*non-txnal*) corresponds to the number of cycles spent executing non-transactional code. The third and fourth components (*tx-useful, tx-aborted*) represent cycles spent in transactional execution, split into useful and aborted cycles, respectively, depending on the final outcome of the transaction. The fifth component (*stall*) is the time a transaction spent stalled in a data access, because such data was in the write set of a committing transaction. The six component corresponds to the back-off cycles during which an aborted transaction delayed its restart, determined using a linear-backoff algorithm. Finally, components seven and eight represent the overheads experienced at commit time, due to arbitration for the commit token (*arbitration*), and acquisition of exclusive ownership over modified lines (*acquisition*). The sum of these two components corresponds to the commit phase overhead imposed by the lazy nature of the HTM system, and it is depicted separately in Figure 8.

A dedicated write buffer reduces miss rate for almost every benchmark. The improvement in L1 cache performance is most significant in intruder, an application with moderate contention, a very large number of transactions and a medium-sized write set (about 50 bytes spread across 6 cache lines on average for its main transaction). Here, the impact of pollution of private cache is considerable. As described in Section 3, repeated aborts cause a number of invalidations of speculatively dirty data, which then result in misses when the transaction re-executes. As shown in Figure 6, the number of abort misses suffered by restarted transactions is significant in intruder, with an average of 10 such misses until a (perhaps repeatedly) restarted transaction eventually commits. This causes a severe degradation in the L1 cache miss rate. The use of a write-buffer completely eliminates abort misses for this application, and effectively reduces its cache miss rate by 40%, as shown in Figure 5, for both configurations with speculative write-buffering enabled. Figure 5 shows how, in general, 32 words (128 bytes) are enough to buffer all transactionally written data in the common case, except for those benchmarks with exceptionally large write sets. For intruder, the improvement in L1 cache performance shortens the duration of the transaction and thus reduces its probability of conflicting with another concurrent transaction. The nett effect is a substantial decrease of the number of aborted transactions (from almost 14000 in *noWB* to around 12200 in *realWB/idealWB*) as well as in the contention of the application, as shown in Figure 4. This explains the reductions in both *tx-aborted* and *backoff* components of the total execution time.

Yada also exhibits a high degree of cache pollution in the configuration with coherent buffering, with 92 abort misses per each commit of a restarted transaction. However, its very large write set (60 cache lines on average for its main transaction, with 2124 bytes written) makes it impossible for the 128-byte write buffer configuration to keep up with its ideal counterpart.

The effect of non-coherent buffering in benchmarks with low to medium contention –like genome, ssca2 and vacation– is substantially different. Most of the improvement in cache performance when write buffers are introduced is due to substantial reduction in the number of redundant coherence requests for local or non-actively shared data (downgrade misses).

In vacation, for example, each transaction suffers an average of 2.6 downgrade misses (see Figure 6) that can be avoided with non-coherent buffering. Due to the large transaction size in vacation, the elimination of such misses does not have a significant effect on the arbitration component of the commit phase, as depicted in Figure 8. However, the acquisition phase is substantially shortened, although it has minimal influence in the overall execution time of the application because of its relatively small number of transactions (see Figure 7). The small improvement in execution time is primarily due to slightly better L1 cache performance, caused by the removal of abort misses.

For low contention benchmarks with small to medium transaction sizes, such as genome or ssca2 the key bottleneck is the arbitration for commits. Thanks to the introduction of write buffers, the arbitration delays in genome are reduced from almost 90% of the total commit overhead to around 30%, which in turn results in a reduction of around 15% in the overall execution time. The reason for this dramatic improvement in the performance of genome is the elimination of downgrade misses that occur during the acquisition phase. These coherence upgrades lengthen the duration of each commit, which combined with the simple arbitration scheme, creates a snowball effect: A slight increase in the commit phase may cause severe contention for the commit token, as exhibited by genome (see Figure 8).

The very large number of downgrade misses that Figure 6 shows for labyrinth (77) is worth an explanation. In this benchmark, each thread replicates the global grid into its thread-local memory, and then

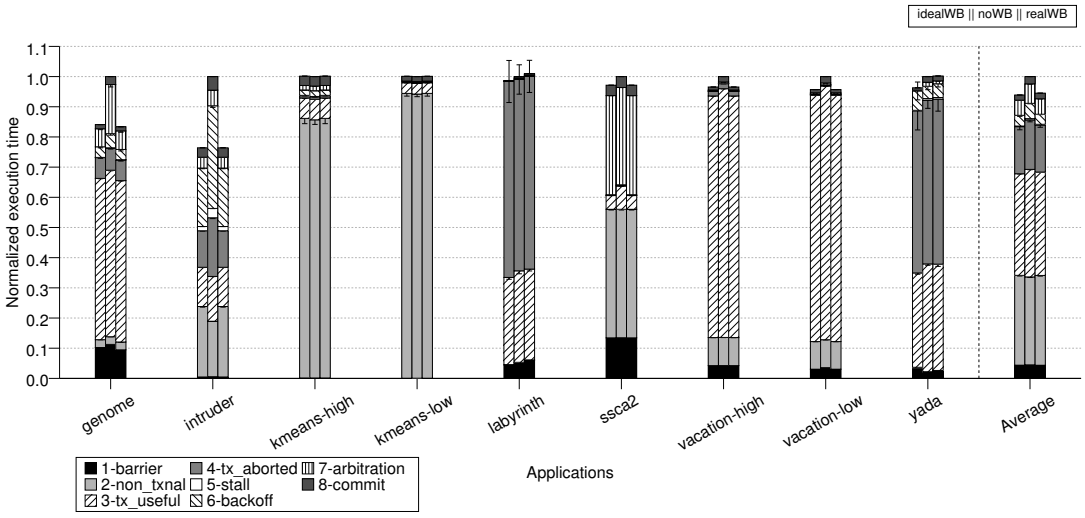

Figure 7: Normalized execution time breakdown.

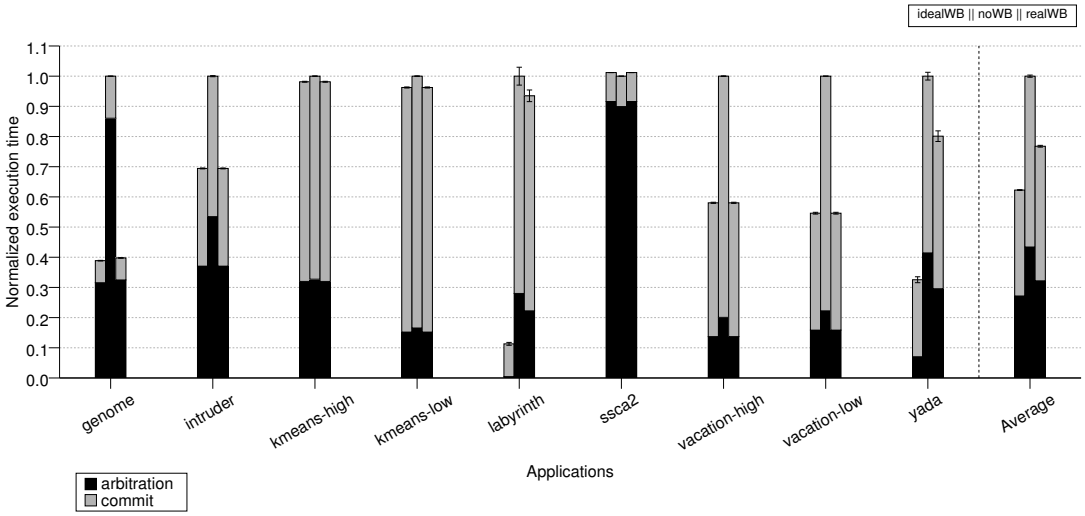

Figure 8: Commit and arbitration delays.

applies Lee's routing algorithm on a local grid. Every time a thread creates a copy of the global grid, the cache lines that contain the local grid are likely to be still in modified state since the last commit, and thus must be written back to the L2 as well as downgraded to shared state before being speculatively modified again. At commit time, the writes to the local grid are indistinguishable from those to the global structure, and hence result in a large number of redundant coherence requests. The inclusion of an infinite write buffer improves L1 hit rates slightly and this is reflected in the minor improvement seen in useful transactional time (see Figure 7). The finite write buffer is too small to buffer any significant portion of the write-set (13 KB across 217 lines) and, hence, its performance is almost the same as that of the case without it.

For intruder, the elimination of downgrade misses during acquisition shortens the total commit delay by 30% with respect to the *noWB* configuration, as shown in Figure 8. The removal of such misses is responsible for a 3% decrease in the overall execution time, while the remaining 20% improvement comes from the lowered contention levels achieved through the elimination of abort misses, as discussed earlier.

SSCA2, a workload with a high commit rate, shows improvement in execution time 7 when write-buffering is enabled. A shorter commit phase, as a result of no downgrade misses, is the primary contributing factor. The contraction in execution time results in commits being concentrated in a shorter span of time than in the *noWB* case, resulting in a slight increase in the proportion of commit arbitration time (see Figure 8).

6. Conclusion and Future Work

In this work we have described and analyzed the inefficiencies that can be caused by buffering of speculative writes in coherent structures like private caches. While we do not recommend exclusive use of non-coherent write buffers since area and power restrictions may severely limit flexibility, the importance of having such buffering to support the common case has been underlined. The performance impact of non-coherent buffering has been quantified and shown to yield significant improvements in the set of benchmarks analyzed here. The expectation is that TM programming constructs would eventually enable workloads with coarse grained transactions, where significant amounts of non-contended data could be written along with actively contended data. In scenarios of high contention abort misses would result in significant degradation of cache performance. In scenarios with low contention but high commit throughput downgrade misses might result in substantial slowdown due to prolonged arbitration.

In the future we would like to extend this study to include the mitigation of write-write conflicts between transactions when using non-coherent buffering. Its implications in eager HTM systems also appear to be of some importance.

Acknowledgment

This collaborative work was supported by the Spanish MEC and MICINN, as well as European Comission FEDER funds, under grants Consolider Ingenio-2010 CSD2006-00046 and TIN2009-14475-C04. It was also partly supported by the HiPEAC-2 NoE under contract FP7/IST-217068. Anurag's work at Chalmers has been supported by the European Commission FP7 project VELOX (ICT-216852). Rubén Titos has a research grant from the Spanish MEC under the FPU National Plan (AP2006-04152) and was awarded a collaboration grant from HiPEAC to visit Chalmers.

References

[1] Jayaram Bobba, Kevin E. Moore, Luke Yen, Haris Volos, Mark D. Hill, Michael M. Swift, and David A. Wood. Performance pathologies in hardware transactional memory. In *ISCA-34*, pages 81–91, Jun 2007.

[2] Chi Cao Minh, JaeWoong Chung, Christos Kozyrakis, and Kunle Olukotun. STAMP: Stanford transactional applications for multi-processing. In *Proc. of The IEEE Intl. Symposium on Workload Characterization*, pages 35–46. Sept 2008.

[3] Hassan Chafi, Jared Casper, Brian D. Carlstrom, Austen McDonald, Chi Cao Minh, Woongki Baek, Christos Kozyrakis, and Kunle Olukotun. A scalable, non-blocking approach to transactional memory. In *HPCA-13*, pages 97–108, 2007.

[4] Aleksandar Dragojevic and Rachid Guerraoui. Predicting the scalability of an STM. In *TRANSACT '10: 5th Workshop on Transactional Computing*, Feb 2010.

[5] Lance Hammond, Brian D. Carlstrom, Vicky Wong, Mike Chen, Christos Kozyrakis, and Kunle Olukotun. Transactional coherence and consistency: Simplifying parallel hardware and software. *IEEE Micro*, 24(6), Nov-Dec 2004.

[6] Lance Hammond, Vicky Wong, Mike Chen, Brian D. Carlstrom, John D. Davis, Ben Hertzberg, Manohar K. Prabhu, Honggo Wijaya, Christos Kozyrakis, and Kunle Olukotun. Transactional memory coherence and consistency. In *ISCA-31*, pages 102–113, Jun 2004.

[7] James R. Larus and Ravi Rajwar. *Transactional Memory*. Morgan & Claypool, 2006.

[8] Peter S. Magnusson, Magnus Christensson, Jesper Eskilson, Daniel Forsgren, Gustav Hallberg, Johan Hogberg, Fredrik Larsson, Andreas Moestedt, and Bengt Werner. Simics: A full system simulation platform. *IEEE Computer*, 35(2):50–58, Feb 2002.

[9] Milo M.K. Martin, Daniel J. Sorin, Bradford M. Beckmann, Michael R. Marty, Min Xu, Alaa R. Alameldeen, Kevin E. Moore, Mark D. Hill, and David A. Wood. Multifacet's general execution-driven multiprocessor simulator (GEMS) toolset. *Computer Architecture News*, pages 92–99, September 2005.

[10] Anurag Negi, M.M. Waliullah, and Per Stenstrom. LV*: A low complexity lazy versioning HTM infrastructure. In *IC-SAMOS X)*, pages 231–240, July 2010.

[11] Sasa Tomic, Cristian Perfumo, Chinmay Kulkarni, Adria Armejach, Adrián Cristal, Osman Unsal, Tim Harris, and Mateo Valero. EazyHTM: Eager-lazy hardware transactional memory. In *MICRO-42*, 2009.

[12] M.M. Waliullah and P. Stenstrom, Classification and Elimination of Conflicts in Transactional Memory Systems. Tech. Report 2010:09, Department of Computer Science and Engineering, Chalmers University of Technology, Sweden, December 2010.

[13] Sutirtha Sanyal, Adrián Cristal, Osman S. Unsal, Mateo Valero and Sourav Roy. Dynamically Filtering Thread-Local Variables in Lazy-Lazy Hardware Transactional Memory. In HPCC '09: Proc. 11th Conference on High Performance Computing and Communications, June 2009.