Improving Performance by Reducing Aborts in Hardware Transactional Memory

Mohammad Ansari[1⋆], Behram Khan[2], Mikel Luján[2], Christos Kotselidis[2], Chris Kirkham[2], and Ian Watson[2]

[1] Department of Computer Science, Umm Al-Qura University
[2] School of Computer Science, University of Manchester
mmansari@uqu.edu.sa
{bkhan, mlujan, ckotselidis, ckirkham, iwatson}@cs.manchester.ac.uk

Abstract. The optimistic nature of Transactional Memory (TM) systems can lead to the concurrent execution of transactions that are later found to conflict. Conflicts degrade scalability, and may lead to aborts that increase wasted work, and degrade performance. A promising approach to reducing conflicts at runtime is dynamically, and transparently, reordering the execution of transactions upon discovery of conflicts. This approach has been explored in Software TMs (STMs), but not in Hardware TMs (HTMs). Furthermore, STM implementations of this approach cannot be ported to HTMs easily.
This paper investigates the feasibility of such reordering in HTMs, and presents two designs that are scalable, independent of the on-chip interconnect, require only minor modifications to each core, and add no execution overhead if no conflicts occur. The evaluation takes LogTM-SE as a base line and considers benchmarks with different levels of contention (transactional conflicts). The results show that the preferred design increases HTM performance by up to 17% when contention is low, 57% when contention is high, and never degrades performance. Finally, the designs are orthogonal to LogTM-SE; they require no modification to cache structures, and continue to support transaction virtualization, open and closed unbounded nesting, paging, thread suspension, and thread migration.

1 Introduction

Traditionally, locks have been used to provide synchronization between threads that access shared data concurrently. Locks are known to be challenging to use, with well-documented challenges such as deadlocks, race conditions, convoying, and debugging. Transactional Memory (TM) [1] proposes a programming model to simplify safe access to shared data, which is achieved by providing *implicit synchronization*; the programmer marks, as transactions, those blocks of code that access shared data, and TM ensures correct synchronization when those blocks of code execute concurrently.

TM provides implicit synchronization by checking, at runtime, whether accesses by concurrently executing transactions intersect, i.e., conflict. If a transaction completes

⋆ A large part of this work was conducted while the author was with the School of Computer Science, University of Manchester.

executing and detects no conflict, it commits, but if a conflict is detected, one of the conflicting transactions is usually aborted. TM implementations may detect conflicts eagerly (upon access to a data element), or lazily (when the transactional code block has been completely executed). TM implementations write to shared data, i.e., perform version management, eagerly (write to shared data in place), or lazily (write to a buffer). The former has a fast commit phase, but a slower abort phase as it requires the transaction to undo all its updates to shared data, while the latter has a fast abort phase, but a slower commit phase as it must copy updates from the buffer to the shared data.

TM has been implemented in hardware (HTM) [2–7], software (STM) [8–13], or a hybrid of the two (HyTM) [14–17]. The advantage of HTMs is a low overhead in performing transactional conflict detection, but at the cost of limiting the total accesses of each transaction to the size of the L1 or L2 cache. STMs remove this limitation, but at the cost of increased conflict detection overhead. Research in TM has focused on reducing the overhead of conflict detection, but also on understanding TM behavior [18], and even on adapting to dynamic workload characteristics [19, 20]. This paper focuses on HTMs.

As the number of cores on a chip multiprocessor (CMP) rises, efficiently exploiting the cores to achieve high speedup becomes more challenging, even with TM. TM applications that scale ideally up to, say, 16 cores, may well find that they scale poorly when executed on 128 cores due to more and more transactions conflicting and aborting. To make matters worse, TM implementations have often tried to optimize the execution of a committing transaction at the cost of penalizing aborts, for example, by using eager version management.

Steal-on-Abort (SOA) [21] is our technique to improve the performance of TM when noticeable contention (i.e., transactional conflicts) occurs. SOA targets a pathological interaction between conflicting transactions called *repeat conflicts*. This occurs when a specific transaction A conflicts with, and is aborted by, a specific transaction B. Transaction A is restarted after its abort, but performs an access that causes it to *repeat* its conflict with transaction B, and then transaction A aborts again. This scenario may repeat a number of times. SOA proposes that transaction A not be restarted, and instead be stolen by transaction B, to prevent it from being re-executed until transaction B commits. Once B commits, A is made available for execution. By not executing transaction A again, a potential repeat conflict and abort is avoided, which could have wasted cycles, power, and degraded application performance. Additionally, on SOA-enabled STMs [21], the thread on which transaction A was running acquires a new, third, transaction C, to execute. If transaction C commits, application performance may improve.

However, implementations of SOA exist only in STMs [21]. Furthermore, they have used dynamic data structures such as double-ended queues (deques) that make it difficult to perform a straightforward port of SOA to HTMs. As a result, the feasibility of implementing SOA in HTMs remains unexplored.

This paper is the first to investigate implementing SOA in HTMs, and presents two designs: SOA-HTM-PURE, and SOA-HTM-UTLZN. The former guarantees repeat conflicts are eliminated, but implements a restricted form of SOA compared to STMs. The latter implementation is less restricted, but permits repeat conflicts in cer-

tain scenarios. Notably, both implementations require only simple modifications to each core, are independent of the on-chip interconnect, and highly scalable. For evaluation, the designs are implemented in LogTM-SE [7], and continue to offer all the advantages of LogTM-SE such as unmodified cache structures, and support for transaction virtualization, open and closed unbounded nesting, paging, thread suspension, and thread migration. Results show that the benefit of SOA seen in STMs extends to HTMs; improving speedup up to 57%, reducing processor usage up to 26%, and reducing the number of aborts up to 54%. In addition, the HTM designs of SOA also improve performance in low contention benchmarks, and, promisingly, improve speedup by increasing margins as the number of cores rises.

The remainder of this paper is organized as follows. Section 2 presents the designs of SOA for HTM, and Section 3 discusses how they impact other structures. Section 4 evaluates the designs by implementing them in LogTM-SE, and executing a range of benchmarks. Section 5 discusses related work, and Section 6 concludes the paper.

2 Steal-on-Abort Hardware Implementation

SOA abstractly consists of the three following actions:

1. Upon abort, a transaction is stolen, and hidden, by its opponent.
2. Upon commit, a transaction makes available for execution any transactions it stole.
3. Optionally, another transaction is acquired and executed in place of the stolen one.

The first two actions are enough to support SOA: they prevent repeat conflicts between two transactions by preventing them from executing concurrently. The third action attempts to increase speedup (if the new transaction commits).

However, these actions are non-trivial to support in hardware. For example, transactions are often tightly-coupled to the threads on which they are executing, as threads maintain the execution state of an application. It may be impossible for a core to steal an opponent transaction without stealing the thread on which it is executing. Assuming transactions can be stolen, storing them in hardware is another challenge as there is no limit to the number of steals that may be performed by a transaction. Thus, it may become necessary to overflow stolen transactions to memory, which could significantly slow down an executing transaction, and increase interconnect bandwidth usage. Nevertheless, promising results from SOA on STMs give incentive for exploring if an efficient design for SOA on HTMs can be achieved.

This paper proposes two carefully constructed designs that aim to minimise performance degradation, interconnect traffic, and modifications to cores and cache structures. The latter is particularly important for keeping the designs practically feasible. The first proposal is called SOA-HTM-PURE, which supports only the first two actions mentioned above, and the second is called SOA-HTM-UTLZN, which implements all three actions.

2.1 SOA-HTM-PURE

A single register, called SOA_AMAP (SOA abort map), is added to each core, and has one bit for each core in the CMP. If a core aborts another, it sets the relevant bit in

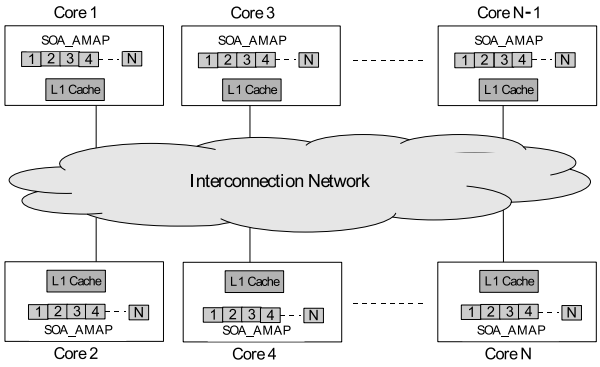

Fig. 1. Architecture for SOA on HTM. Only one additional register per-core needed, called SOA_AMAP.

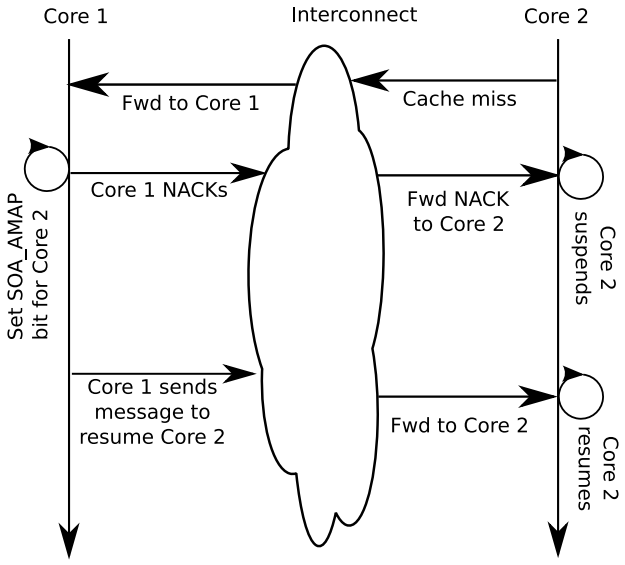

Fig. 2. SOA example. Core 2 has a (data) cache miss, and makes a request to the interconnect that is forwarded to Core 1. Core 1 responds with a NACK, and records the NACK in its SOA_AMAP. Core 2 receives the NACK, aborts, and suspends. Upon commit, Core 1 notifies all cores recorded in its SOA_AMAP to resume.

SOA_AMAP. Since aborting an opponent requires communicating over the interconnect, setting a bit in the register adds negligible overhead. Cores that are aborted stall indefinitely, and are restarted later by their opponents. For now we assume threads do not migrate; we address this issue later.

Once a core commits, it checks if any bits are set in its SOA_AMAP. If all the bits are clear, the core commits as normal. In this way, SOA-HTM-PURE adds no overhead when there is no contention. If one or more bits are set, the corresponding cores are resumed by sending a message across the interconnect. The exact mechanism for sending such a message is architecture specific. In Section 4.3, the messaging mechanism is described for our implementation using LogTM-SE, and results in a single outgoing message from a committing core, and a single multi-cast message from a directory. In this way, commit overhead is kept low by only adding a single, non-broadcast, message.

SOA should reduce communication traffic if repeat conflicts exist as fewer data requests will be received from cores that restart aborted transactions, and fewer abort messages will be sent to them in reply. Furthermore, it may be possible to power down stalled cores to save energy.

The design can feasibly scale to 2048 cores, requiring only a 2048 bit register per core (existing HTM implementations, for example, have suggested implementing 2048 bit signatures per core [7]), easily exceeding the number of cores expected on CMPs in the near future.

2.2 SOA-HTM-UTLZN

SOA-HTM-UTLZN is an acronym for "SOA on HTM for utilization", and extends SOA-HTM-PURE to add the last action of SOA. SOA-HTM-UTLZN piggybacks on hardware thread context support that is common in CMPs [22]. Hardware context support allows a core to store several thread contexts in hardware registers, and swap execution between them quickly, primarily to hide memory latency. SOA-HTM-UTLZN extends SOA-HTM-PURE by swapping threads in hardware contexts if the currently executing thread is stalled (due to executing a transaction that has been aborted). For now we assume thread contexts do not migrate; we address this issue later.

SOA-HTM-UTLZN adds a single bit, called CTXT_SOA, to each hardware thread context, which is set if the transaction being executed by the thread is aborted. A core does not switch to any context that has its CTXT_SOA bit set. When a core sends a resumption message to another core, the other core clears the CTXT_SOA bit in all its thread contexts. This reintroduces the chance of repeat conflicts as the resumption message will have been sent for only one of the contexts on the other core, and waking up all contexts prematurely allows them to repeat their conflict with their respective opponents. However, the benefit of this approach is that it leaves the SOA_AMAP register unchanged; one bit per core, maintaining the scalability of the design. To support resuming specific contexts SOA_AMAP must map one bit per context, which requires either the register to increase in size, or the the potential scalability to be reduced.

It should be noted that using hardware contexts to increase utilization has its limits; if all contexts on a core are stalled due to transactional conflicts, then that core can no longer execute transactions until a resumption message is received. One option may be

to swap contexts with another core, but there are several design trade-offs involved with such a mechanism, and we leave it for future work.

3 Impact of SOA

The previous sections described two proposals for implementing SOA in HTM. This section explores the impact of those designs on processor architecture, transactional execution, and the operating system.

3.1 Processor Architecture

Each core is extended with a single register called SOA_AMAP. A simple messaging protocol is also required to resume cores, requiring the interconnect to simply forward the necessary messages to predefined destinations. A strength of the SOA designs proposed is that no other change is required. No other hardware modifications are needed, and in particular the pipeline, private caches, and shared caches of the core are left unchanged. This significantly reduces the impact on design verification, making the SOA proposals attractive for practical implementation.

3.2 Transactional Execution

SOA is only applicable to eager conflict detection, as lazy conflict detection only detects conflicts with transactions that have committed, which rules out repeat conflicts. The use of eager version management may increase the benefit of SOA, as it should reduce the overhead of roll backs if it reduces the number of aborts. The benefit of SOA may also increase if it is used in conjunction with signature-based conflict detection, as they may lead to false-positives, which may increase the number of repeat conflicts.

A committing core incurs an overhead of sending messages to resume other cores if any bits in its SOA_AMAP are set. In our implementation, only a single message is sent by a committing core. There is no increase in overhead on aborting cores. Nested transactions, both open and closed, are orthogonal to SOA, and work in harmony with it. For example, the Deque benchmark used in the evaluation executes nested transactions.

3.3 OS Context Migration

Earlier, the SOA designs were restricted to prevent threads from leaving their cores, because a stalled thread expects to receive a resumption message from the core of the opponent thread, and the opponent core holds only enough information to send a resumption message to the core that it aborted, not the stalled thread itself. This restriction is simple to remove. First, a stalled thread that is removed from a core needs to be marked as active, and not stalled. No change is needed in SOA_HTM_PURE, and in SOA_HTM_UTLZN the CTXT_SOA bit should be cleared for the thread context in question.

However, removing this restriction reintroduces repeat conflicts, as the migrated thread is no longer stalled waiting for its opponent, and could begin re-executing immediately. Furthermore, cores may send resumption messages that are no longer needed,

possibly resuming threads that are not their opponents, which further increases the chance of repeat conflicts. Nevertheless, earlier work with SOA on STMs suggested SOA is highly effective even when the implementation reintroduced repeat conflicts, and contention was already high [21]. Thus, not only is it possible to override the above restriction, but past results have shown that it may be an acceptable decision.

3.4 OS Virtual Memory Paging

The SOA designs do not peek at memory addresses, and as such are compatible with support for paging. Furthermore, the modifications required to implement the SOA designs do not impact HTM-specific support for paging.

4 Evaluation

SOA-HTM-PURE and SOA-HTM-UTLZN are evaluated using full-system simulations with a range of benchmarks, and results compared with a "Base" implementation that has SOA disabled. The evaluation shows that the designs improve speedup, and reduce aborts, although performance of SOA-HTM-UTLZN is mixed; in some cases it improves performance, while in others it degrades it.

4.1 Methodology

SOA-HTM-PURE and SOA-HTM-UTLZN are implemented in LogTM-SE built on Simics 3.0.31 [23], and GEMS 2.1 [24] Ruby pipeline and memory timing model. The simulated platform uses simple in-order SPARC ISA cores running an unmodified Solaris 9. Experiments are executed with 1, 4, 8, and 16 cores (and corresponding benchmark threads). Each benchmark thread is bound to an individual processor, using Solaris' `pset_bind()`. As a result, OS thread migration and context switching are implicitly disabled. The architecture of the evaluated CMP is described in Section 4.3.

SOA-HTM-UTLZN is executed with four hardware contexts per processor, and consequently each benchmark is launched with four times as many threads. In order to isolate the performance benefit of hardware context switching to SOA-HTM-UTLZN alone, hardware context switching is only permitted when a thread stalls due to SOA, i.e., cannot be used to hide memory latency.

4.2 Workloads

The microbenchmarks Deque and Btree, and the non-trivial benchmarks Kmeans and Vacation (from the the STAMP benchmark suite [17]), are used to evaluate SOA-HTM. In deque, transactions attempt to push or pop a double-ended queue. Transactions in Btree insert, delete, or lookup items in a B-tree. Kmeans is a clustering algorithm, and contention is controlled by the number of clusters to which objects are assigned. We experiment with 1, 5, and 15 clusters, which lead to progressively lower contention. Finally, Vacation is a travel database simulating multiple customers concurrently booking flights, hotels, and cars.

Benchmark	Parameters
Btree	tx:5000, inserts:20%
Deque	tx:1024, bkoff:32
Kmeans C1	m:1, n:1, threshold:0.05, input_file:random-n2048-d16-c16.txt
Kmeans C5	m:5, n:5, threshold:0.05, input_file:random-n2048-d16-c16.txt
Kmeans C15	m:15, n:15, threshold:0.05, input_file:random-n2048-d16-c16.txt
Vacation	tx:1024, n:8, q:10, u:80, r:65536

Table 1. Benchmark parameters.

4.3 Evaluated CMP Configurations

SOA-HTM-PURE and SOA-HTM-UTLZN are implemented in the LogTM-SE HTM that is provided with GEMS 2.1. LogTM-SE and the SOA are a complementary union. LogTM-SE aims to keep cache structures unmodified as this improves the chances of adoption. Similarly, the SOA designs require minimal changes for each core. LogTM-SE attempts to achieve high scalability by using a non-broadcast commit phase, and directory coherency. The SOA designs add no overhead to the commit phase if conflicts do not occur, and is agnostic of the interconnect or coherency protocol. LogTM-SE uses eager validation, which is a requirement for SOA.

LogTM-SE is configured to use eager version management, eager conflict detection, and a conflict resolution policy of self-abort, with exponential backoff (increase backoff on retry). Note that this choice should reduce the benefit of SOA-HTM, as choosing to abort the opponent is likely to generate more repeat conflicts, and exponential backoff also reduces repeat conflicts, but at the risk of backing off for too long and harming performance. A 2048 bit H_3 signature is used for conflict detection [25].

Feature	Description
L1 cache	32KB 4-way split, 64-byte blocks, 1-cycle access.
L2 cache	8MB 8-way unified, 64 byte blocks, 34-cycle access.
Memory	16GB, 500 cycle off-chip access.
L2-Directory	Full-bit vector sharer list; 6-cycle latency.
Interconnect	grid, 64-byte links, 3-cycle link latency.

Table 2. *Simulation parameters for SOA-HTM.*

Figure 3 presents a block diagram of the 16 core CMP architecture. Further configurations include 1, 4, and 8 core CMPs. In each case, the number of L2 banks is equal to the number of cores. Cores are connected by a packet-switched interconnect in a grid topology using 64-byte links and adaptive routing. On-chip memory controllers connect

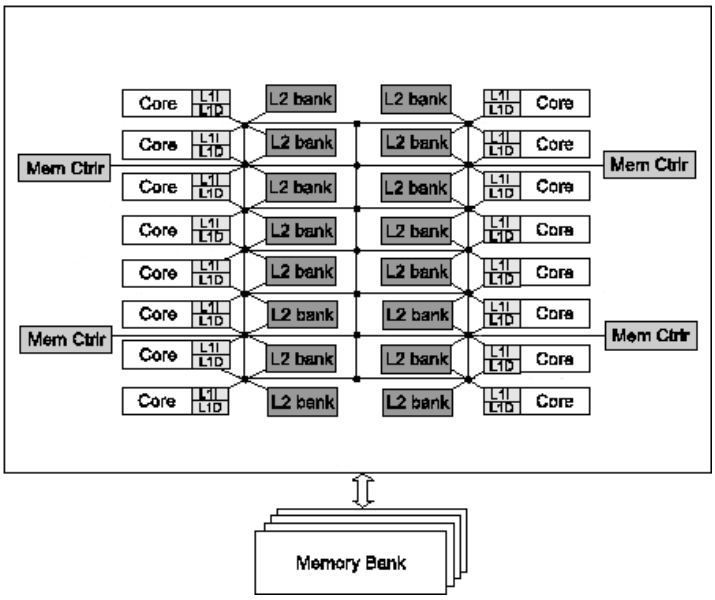

Fig. 3. *Base Log-TM-SE CMP configuration.*

to standard DRAM banks. A MESI directory protocol enforces inclusion at L2. Each L2 tag contains a bit-vector of the L1 sharers and a pointer to the exclusive copy, if it exists. Table 2 summarizes system parameters that remain fixed for each configuration.

The SOA communication protocol is as follows. Upon commit, a core checks its SOA_AMAP register, and if any bit is non-zero, it sends a single CORE_RESUME_REQ message to its local directory, containing the complete value of SOA_AMAP. The directory sends a single multi-cast DIR_RESUME_REQ message out to each core for which the corresponding bit is set in the received SOA_AMAP value. The design creates minimal overhead; a core only needs to send a single message if any bit is set in SOA_AMAP. If transactions are committing in the common case, then there is little or no overhead of SOA as NACKs will be rare, and CORE_RESUME_REQ/DIR_RESUME_REQ messages will also be rare, as they are only sent if there is a waiting core. Similarly, if commits are common then there is little change in traffic for the directory. If aborts occur, then the design will increase commit overhead, and stall cores, but should compensate by leading to a net increase in performance.

On its own, this protocol is susceptible to deadlock; two transactions may signal each other to abort, and consequently never restart. However, LogTM-SE itself is susceptible to such deadlocks, and thus includes a multi-stage abort mechanism that detects and prevents deadlock cycles. Our extensions to LogTM-SE do not interfere with this mechanism, and therefore only abort (and stall) transactions when doing so will not lead to a deadlock. Consult the LogTM [6] and LogTM-SE [7] papers for further details.

4.4 Results

The evaluation explores the impact of SOA on three scenarios: low contention, high contention with low repeat conflicts, high contention with high repeat conflicts. The first scenario should trigger SOA rarely, and is used to illustrate the minimal impact of SOA on performance when aborts are negligible. The second scenario should trigger SOA often, but provide little performance improvement since there are few repeat conflicts, and may even degrade performance due to SOA overhead. The third scenario should trigger SOA often, and could result in larger performance improvements.

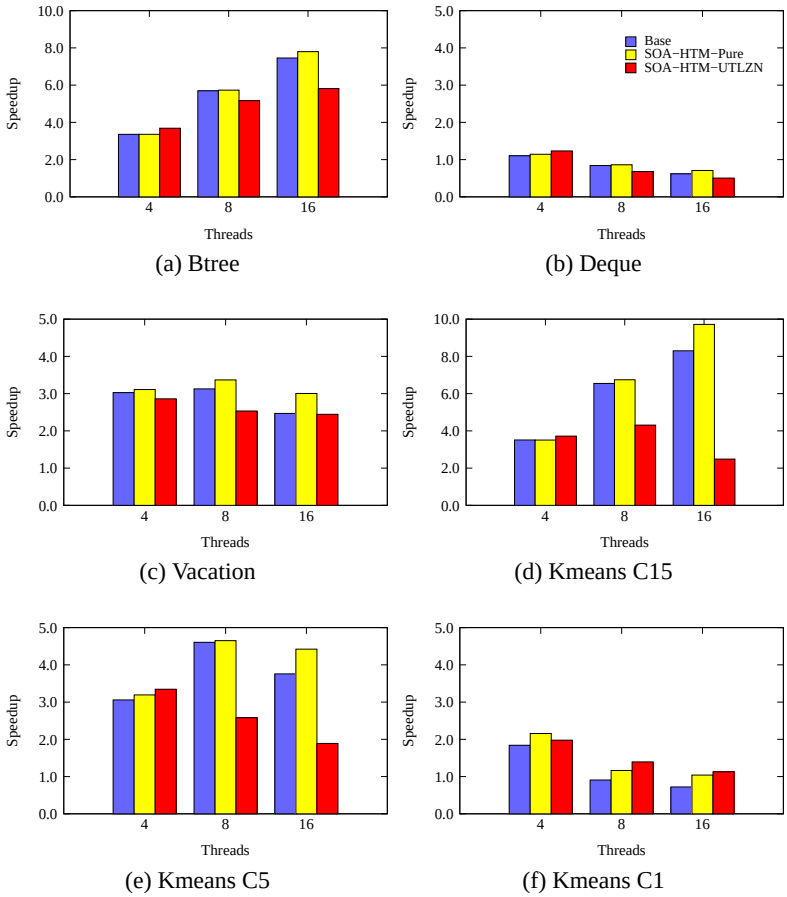

Fig. 4. Speedup over single-threaded execution. Note different y-axis ranges.

Figures 4a-4f illustrate speedup. A cursory look reveals two important findings. First, SOA-HTM-PURE gives similar or better performance than Base in all cases.

Second, SOA-HTM-UTLZN improves upon SOA-HTM-PURE in Kmeans C1, but in most other cases it degrades performance compared to Base.

Btree and Kmeans C15 have low contention, aborting on average 10% and 20% of transactions with 16 cores. In these benchmarks both Base and SOA-HTM-PURE scale similarly, which is indicative of the low overhead of SOA when aborts are rare. SOA-HTM-UTLZN degrades performance in both, and the degradation is more severe in Kmeans C15, which also has a higher percentage of aborts. Profiling data reveals that SOA-HTM-UTLZN is thrashing local caches by context switching. In Kmeans C15 with 16 cores, we find that the number of L1 data misses increases 10 to 15 fold over Base and SOA-HTM-PURE.

Deque and Kmeans C5 are benchmarks with a large amount of contention, but few transactional retries, and thus little scope for repeat conflicts. At 16 cores, Kmeans C5 aborts 65% of its transactions, but retries on average only 1.9 times. However, SOA-HTM-PURE still improves performance by 16%. In deque an almost identical situation arises at 16 cores; 78% contention, and 3.5 retries on average increases the scope for repeat conflicts. The scalability of Deque is limited by transactions accessing either end of the deque structure, making repeat conflicts highly likely, and this is confirmed by the 16% performance improvement with SOA-HTM-PURE. For SOA-HTM-UTLZN the cache misses due to context switching are 1.5 to 2 times higher than Base and SOA-HTM-PURE at 16 cores, degrading performance by 18% over Base in Deque, and 50% in Kmeans C5.

Vacation and Kmeans C1 are benchmarks with a large amount of contention, and a high number of retries. In Vacation, this occurs at 8 and 16 cores, where 77% and 90% of transactions abort, and the average number of retries is 3.2 and 9.0, respectively. Modest performance improvements of 6% and 21% are observed with SOA-HTM-PURE. In Kmeans C1 there is little exploitable parallelism, as all transactions update a single cluster. By 16 cores, Kmeans C1 aborts 96% of transactions, and its average number of retries is 24.4. SOA-HTM-PURE results in a performance improvement of 44%, while SOA-HTM-UTLZN, in one of the few cases where it improves performance, does so by 57%. Kmeans' larger performance improvement than Vacation at 16 cores, despite having a lower number of retries, is indicative of repeat conflicts representing a smaller number of retries in the latter. It is worth noting that contention is rising in these benchmarks as the number of cores increases, and SOA-HTM-PURE provides correspondingly larger performance improvements. Thus we would expect even larger performance improvements if the benchmarks were executed using a larger number of cores.

SOA-HTM-UTLZN improved performance in a limited number of cases, and in all those cases SOA-HTM-PURE improved performance similarly. However, SOA-HTM-PURE results in better performance in many cases that SOA-HTM-UTLZN does not. Thus, for brevity we limit further analysis to Base and SOA-HTM-PURE.

Table 3 illustrates the impact of SOA-HTM-PURE on the average number of transactional retries. For the scalable benchmarks (Btree, Kmeans C15) there are marginal differences in retries. Only Kmeans C15 at 16 cores is significant, and likely to be responsible for the performance improvement seen earlier. The remaining benchmarks all see marked reductions in the number of retries, which increase with the number of

Benchmark	Base			SOA-HTM-PURE		
	4 cores	8 cores	16 cores	4 cores	8 cores	16 cores
Btree	0	0	0.15	0	0	0.10
Kmeans C15	0	0.1	1.6	0	0.1	0.2
Deque	0	1.6	4.2	0	1.3	3.3
Kmeans C5	0.1	1.9	7.5	0.1	0.6	5.5
Vacation	0.4	3.2	9.0	0.2	1.5	4.1
Kmeans C1	2.0	19.5	24.4	0.7	14.0	16.6

Table 3. Average number of retries.

cores, suggesting again that SOA may provide even better results with larger numbers of cores.

Table 4 shows the number of cycles saved by SOA-HTM-PURE, which is the difference between Base and SOA-HTM-PURE in the number of cycles spent stalling. Recall that SOA-HTM-PURE stalls cores upon abort, and those cycles spent stalling are effectively saved. These saved cycles could be used for executing other applications, or SOA-HTM-PURE could be extended to sleep cores on abort, and resume upon notification from the opponent core, thus saving energy. The cycles are not wasted in executing transactions that abort. The table shows that 8-26% of cycles can be saved in the high contention experiments, while maintaining or improving speedup over Base. In some cases SOA-HTM-PURE uses more cycles than Base (shown with negative numbers). Although the increase represents a small fraction of the total execution cycles, small variations can occur since stall cycles also include stalling for cache misses.

Benchmark	4 cores	8 cores	16 cores
Btree	-597 (-0.03)	-922 (-0.09)	-6,793 (-0.84)
Kmeans C15	-662 (-0.03)	-3,863 (-0.3)	-17,701 (-1.72)
Deque	800 (0.12)	49,191 (5.7)	99,876 (8.51)
Kmeans C5	16,265 (1.1)	13 (0)	166,329 (13.81)
Vacation	-32,990 (-0.53)	455,190 (7.61)	1,766,740 (23.34)
Kmeans C1	-20,462 (-1.3)	388,982 (12.2)	1,061,083 (26.44)

Table 4. Average reduction in number of cycles used to execute the benchmarks using SOA-HTM-PURE. In parenthesis: as a percentage of Base total execution cycles.

5 Related Work

SOA was first implemented in an STM [26, 21] by adding two dynamically sized deques to each thread: one that held ready-to-execute work, and one which held stolen transactions. Transaction stealing was performed by abstracting transactions into job objects that held sufficient metadata to enable any thread to execute the transaction. The implementation resulted in a pseudo thread pool framework for executing transactions. This implementation permitted repeat conflicts, and performance results revealed

it to be highly effective at reducing repeat conflicts, unlike SOA-HTM UTLZN, which suffered due to increased cache misses. The difficulties in implementing SOA on HTM using the STM-based solution inspired the work in this paper.

Little other work exists on automatically reducing the impact of contention, or attempting to improve performance when contention occurs. Early work on contention management [27, 28] developed intricate backoff and work-estimation metrics to try and resolve conflicts. Recently, CAR-STM [29] implemented a similar framework to SOA [21] for a different STM. Additionally, CAR-STM allows users can define a routine to serialize transactions they expect to conflict, although such functionality is similar to that presented by Bai *et al.* [30].

Our earlier work [31, 19] on dynamically adapting to available parallelism in an STM application, by changing the number of threads permitted to execute transactions (in a thread pool), reduced the number of aborts, and reduced wasted work. Yoo and Lee [32] implemented a STM transaction scheduling framework that queues threads onto a global queue if they greater than a user-specified threshold of aborts over a history window of transactions, which resulted in similar functionality to our adaptive work, although our solution has the ability to be more responsive in certain cases. In contrast, the STM transaction scheduling framework of Yoo and Lee is more amenable to an HTM implementation.

6 Conclusions

This paper has presented the first proposals for SOA in HTM. The two proposed implementations are scalable, require minimal architectural modifications, and independent of the on-chip interconnect. The two implementations were seamlessly integrated into LogTM-SE [7], and were evaluated using a range of benchmarks and contention scenarios. The results showed SOA-HTM-PURE to be consistently well performing. Although SOA-HTM-UTLZN outperforms SOA-HTM-PURE in two cases, in most other cases it provides the worst performance. In scenarios where the benchmark was highly scalable, SOA-HTM-PURE resulted in no observable performance degradation, and in one case a performance improvement of 17%. When contention rose, performance began to improve more consistently, ranging from 16% to 44% with 16 cores, suggesting that SOA-HTM-PURE may benefit applications with even low contention. Performance improvements increased with the number of cores, strongly suggesting that larger improvements may be observed with larger numbers of cores. Finally, SOA-HTM-PURE reduced the average number of retries (i.e., aborts) by 54%, and saving up to 26% of the execution cycles.

References

1. James R. Larus and Ravi Rajwar. *Transactional Memory*. Morgan and Claypool, 2006.
2. C. Scott Ananian, Krste Asanovic, Bradley C. Kuszmaul, Charles E. Leiserson, and Sean Lie. Unbounded transactional memory. In *HPCA '05: Proceedings of the 11th International Symposium on High-Performance Computer Architecture*, February 2005.

3. Jayaram Bobba, Neelam Goyal, Mark D. Hill, Michael M. Swift, and David A. Wood. Tokentm: Efficient execution of large transactions with hardware transactional memory. In *ISCA '08: Proceedings of the 35th Annual International Symposium on Computer Architecture*. Jun 2008.
4. 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 '04: Proceedings of the 31st Annual International Symposium on Computer Architecture*, 2004.
5. Behram Khan, Matthew Horsnell, Ian Rogers, Mikel Luján, Andrew Dinn, and Ian Watson. An object-aware hardware transactional memory system. In *HPCC '08: Proceedings of the 2008 10th IEEE International Conference on High Performance Computing and Communications*, 2008.
6. Kevin E. Moore, Jayaram Bobba, Michelle M. Moravan, Mark D. Hill, and David A. Wood. Logtm: Log-based transactional memory. In *HPCA '06: Proceedings of the 12th International Symposium on High-Performance Computer Architecture*. 2006.
7. Luke Yen, Jayaram Bobba, Michael M. Marty, Kevin E. Moore, Haris Volos, Mark D. Hill, Michael M. Swift, and David A. Wood. Logtm-se: Decoupling hardware transactional memory from caches. In *HPCA '07: Proceedings of the 13th International Symposium on High-Performance Computer Architecture*. 2007.
8. Dave Dice, Ori Shalev, and Nir Shavit. Transactional locking II. In *DISC '06: Proceedings of the 20th International Symposium on Distributed Computing*, September 2006.
9. Tim Harris and Keir Fraser. Language support for lightweight transactions. In *OOPSLA '03: Proceedings of the 18th Annual ACM SIGPLAN conference on Object-Oriented Programing, Systems, Languages, and Applications*, 2003.
10. Maurice Herlihy, Victor Luchangco, and Mark Moir. A flexible framework for implementing software transactional memory. In *OOPSLA '06: Proceedings of the 21st Annual Conference on Object-Oriented Programming Systems, Languages, and Applications*, October 2006.
11. Yossi Lev, Victor Luchangco, Virendra Marathe, Mark Moir, Dan Nussbaum, and Marek Olszewski. Anatomy of a scalable software transactional memory. In *TRANSACT '09: Fourth ACM SIGPLAN Workshop on Transactional Computing*, February 2009.
12. Virendra Marathe, Michael Spear, Christopher Herio, Athul Acharya, David Eisenstat, William Scherer III, and Michael L. Scott. Lowering the overhead of software transactional memory. In *TRANSACT '06: First ACM SIGPLAN Workshop on Transactional Computing*, June 2006.
13. Bratin Saha, Ali-Reza Adl-Tabatabai, Richard L. Hudson, Chi Cao Minh, and Benjamin Hertzberg. McRT-STM: a high performance software transactional memory system for a multi-core runtime. In *PPoPP '06: Proceedings of the 11th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, March 2006.
14. Lee Baugh, Naveen Neelakantam, and Craig Zilles. Using hardware memory protection to build a high-performance, strongly atomic hybrid transactional memory. In *ISCA '08: Proceedings of the 35th Annual International Symposium on Computer Architecture*. June 2008.
15. Peter Damron, Alexandra Fedorova, Yossi Lev, Victor Luchangco, Mark Moir, and Daniel Nussbaum. Hybrid transactional memory. In *ASPLOS-XII: Proceedings of the 12th International Conference on Architectural Support for Programming Languages and Operating Systems*, 2006.
16. Yossi Lev, Mark Moir, and Dan Nussbaum. PhTM: Phased transactional memory. In *TRANSACT '07: Second ACM SIGPLAN Workshop on Transactional Computing*, 2007.
17. Chi Cao Minh, Martin Trautmann, JaeWoong Chung, Austen McDonald, Nathan Bronson, Jared Casper, Christos Kozyrakis, and Kunle Olukotun. An effective hybrid transactional

memory system with strong isolation guarantees. In *ISCA '07: Proceedings of the 34th Annual International Symposium on Computer Architecture*, June 2007.
18. Jayaram Bobba, Kevin E. Moore, Haris Volos, Luke Yen, Mark D. Hill, Michael M. Swift, and David A. Wood. Performance pathologies in hardware transactional memory. In *ISCA '07: Proceedings of the 34th annual international symposium on Computer architecture*, June 2007.
19. Mohammad Ansari, Christos Kotselidis, Kim Jarvis, Mikel Luján, Chris Kirkham, and Ian Watson. Advanced concurrency control for transactional memory using transaction commit rate. In *EUROPAR '08: Fourteenth European Conference on Parallel Processing*, August 2008.
20. Virendra Marathe, William Scherer III, and Michael L. Scott. Adaptive software transactional memory. In *DISC '05: Proceedings of the 19th International Symposium on Distributed Computing*, September 2005.
21. Mohammad Ansari, Christos Kotselidis, Kim Jarvis, Mikel Luján, Chris Kirkham, and Ian Watson. Steal-on-abort: Dynamic transaction reordering to reduce conflicts in transactional memory. In *HIPEAC '09: Fourth International Conference on High Performance and Embedded Architectures and Compilers*, January 2009.
22. Poonacha Kongetira, Kathirgamar Aingaran, and Kunle Olukotun. Niagara: A 32-way multithreaded sparc processor. *IEEE Micro*, 25(2), April 2005.
23. Peter S. Magnusson, Magnus Christensson, Jesper Eskilson, Daniel Forsgren, Gustav Hållberg, Johan Högberg, Fredrik Larsson, Andreas Moestedt, and Bengt Werner. Simics: A full system simulation platform. *IEEE Computer*, 35(2), 2002.
24. 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. *SIGARCH Computer Architecture News*, 33(4), 2005.
25. Luke Yen, S.C. Draper, and M.D. Hill. Notary: Hardware techniques to enhance signatures. In *MICRO '08: Proceedings of the 41st IEEE/ACM International Symposium on Microarchitecture*, Nov. 2008.
26. Mohammad Ansari, Mikel Luján, Christos Kotselidis, Kim Jarvis, Chris Kirkham, and Ian Watson. Steal-on-abort: Dynamic transaction reordering to reduce conflicts in transactional memory. In *SHCMP '08: First Workshop on Software and Hardware Challenges of Manycore Platforms*, June 2008.
27. William Scherer III and Michael L. Scott. Contention management in dynamic software transactional memory. In *CSJP '04: Workshop on Concurrency and Synchronization in Java Programs*, July 2004.
28. William Scherer III and Michael L. Scott. Advanced contention management for dynamic software transactional memory. In *PODC '05: Proceedings of the 24th Annual Symposium on Principles of Distributed Computing*, July 2005.
29. Shlomi Dolev, Danny Hendler, and Adi Suissa. Car-stm: Scheduling-based collision avoidance and resolution for software transactional memory. In *PODC '08: Proceedings of the 27th annual ACM symposium on Principles of distributed computing*, August 2008.
30. T. Bai, X. Shen, C. Zhang, W.N. Scherer, C. Ding, and M.L. Scott. A key-based adaptive transactional memory executor. In *IPDPS '07: Proceedings of the 21st International Parallel and Distributed Processing Symposium*, March 2007.
31. Mohammad Ansari, Christos Kotselidis, Kim Jarvis, Mikel Luján, Chris Kirkham, and Ian Watson. Adaptive concurrency control for transactional memory. In *MULTIPROG '08: First Workshop on Programmability Issues for Multi-Core Computers*, January 2008.
32. Richard M. Yoo and Hsien-Hsin S. Lee. Adaptive transaction scheduling for transactional memory systems. In *SPAA '08: Proceedings of the twentieth annual symposium on Parallelism in algorithms and architectures*, New York, NY, USA, 2008.