Exploring Fallback Solutions in Best-Effort Hardware Transactional Memory

Ricardo Quislant, Eladio Gutierrez, Oscar Plata
Dept. of Computer Architecture
University of Malaga, Spain
Email: {quislant, eladio, oplata}@uma.es

***Abstract*—Transactional Memory (TM) is an optimistic synchronization mechanism for programming shared memory multiprocessors. Companies such as Intel and IBM are offering commercial multicore processors and systems with hardware support for TM (HTM). Due to hardware and system software limitations, all these HTM systems are best-effort (BE-HTM) as they cannot guarantee forward progress for every transactional execution. These designs may show significant performance degradation due to high contention scenarios and different hardware and operating system limitations that abort transactions, e.g. cache overflows, hardware and software exceptions, and so on. To deal with these events and to ensure forward progress, BE-HTM systems usually resort to a software fallback path to execute a lock-based version of the code. The programmer is in charge of adding this piece of code to its transactional application, increasing programming complexity.**

The burden on the programmer of writing a fallback code is relieved in systems such as IBM BlueGene/Q, where a software runtime fallback path is provided by the system, which implements irrevocability. Whenever a transaction is unable to commit after a number of retries it automatically switches to an irrevocable mode, so that it cannot be aborted anymore. In this work we propose a design of a hardware irrevocability mechanism as an alternative to the software fallback path to gain insight into the hardware support that could improve the performance of such a fallback. Our mechanism anticipates the abort that causes the transaction serialization, and stalls other on-going transactions in the system so that transactional work loss is minimized. In addition, we evaluate different software fallback path approaches and propose the use of a ticket lock that holds precise information of the number of transactions waiting to enter the fallback. Thus, the separation of transactional and fallback execution can be achieved in a precise manner with the corresponding performance benefits. The result is an enhanced Lemming effect avoidance. Finally, we outline an in-progress research line based on the concept of lazy lock subscription, where an enhanced lazy hardware-supported irrevocability mechanism is been evaluated with promising results.

The evaluation of the proposals is carried out using the Simics/GEMS simulator and the complete range of STAMP transactional suite benchmarks. We obtain significant performance benefits of around twice the speed-up and an abort reduction of 50% over the software fallback path for a number of benchmarks.

I. Introduction

Transactional Memory (TM) [1] was first presented in 1993 [2] as a non-blocking synchronization mechanism for shared memory chip multiprocessors (CMPs). It is not until recently that some processor manufacturers have included HTM support in their commercial off-the-shelf CMPs [3], [4], [5], [6]. Current industry proposals focus on best-effort solutions (BE-HTM) where hardware limits are imposed on transactions. For instance, transactions cannot survive to capacity overflows, exceptions, interrupts, page faults, migrations,... To deal with these limitations, BE-HTM systems usually provide a software fallback path to execute a non-transactional version of the code, often comprising a global lock.

In this paper we propose an implementation of a hardware irrevocability mechanism as an alternative to the software fallback path to gain insight into the hardware improvements that could enhance the execution of such a fallback. Irrevocability [7], [8] is a transactional execution mode that ensures transaction forward progress since an irrevocable transaction cannot be aborted. Our mechanism anticipates the abort that causes the transaction serialization, and stalls other transactions in the system so that transactional work loss is minimized. In addition, we evaluate the main software fallback path approaches and propose the use of a ticket lock that hold precise information of the number of transactions waiting to enter the fallback. Thus, the separation of transactional and fallback execution can be achieved in a precise manner with the corresponding performance benefits. The result is an enhanced Lemming effect avoidance [9].

The evaluation is carried out using the Simics/GEMS simulator and the complete range of STAMP transactional suite benchmarks. We obtain significant performance benefits of around twice the speedup and an abort reduction of 50% over the software fallback path for a number of benchmarks.

II. Related Work

Irrevocability in the context of HTM was first proposed in TCC [10] to deal with overflowed transactions. Blundell et al. [7] introduces OneTM-Serialized as a system where overflowed transactions gets irrevocable and serializes the system to ensure forward progress. Their implementation comprises a log-based HTM where the irrevocable transaction can be aborted as old data can be recovered from the log. They use a shared transaction status word residing in a fixed virtual location that acts as a mutex lock to implement the irrevocability mechanism. We implement irrevocability with a token-based mechanism distributed through the cache controllers, in the context of a best-effort HTM system, comparing its performance with a software fallback path to gain insight into the hardware that could enhance the fallback.

IBM Blue Gene/Q HTM [5] ensures forward progress on capacity overflows and contention scenarios by means of an irrevocable mode. The irrevocability mechanism is implemented

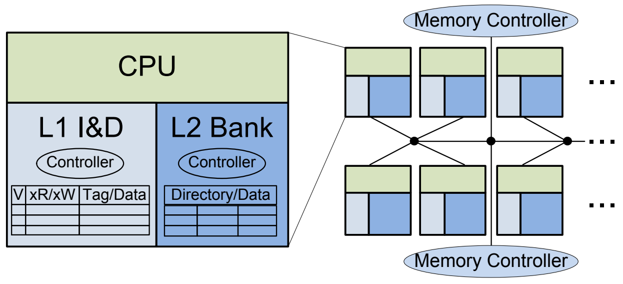

Fig. 1: Baseline architecture of the BE-HTM system.

in a runtime system, thus freeing the programmer from the task of providing a fallback code. The runtime decides if a transaction gets irrevocable in an adaptive way. However, it has to abort a transaction to run it in irrevocable mode, whereas our hardware irrevocable mechanism anticipates the abort and initiates the irrevocable mode without wasting the work done so far by the transaction.

Afek et al. [11] propose a ticket-lock-based technique to improve the performance of Haswell's hardware lock elision (HLE). It is a different approach to our use of the ticket lock. In this case, the ticket lock guards the HLE lock and is acquired by those transactions that abort due to conflicts. Thus, the conflicting transactions are executed speculatively in turn, in parallel with the non-conflicting ones. After a given number of aborts, the transaction holding the ticket lock acquires the HLE lock and aborts all other transactions in the system. In fact, it is a contention management approach.

III. Baseline Architecture

Fig. 1 shows the baseline architecture used in this paper. The system relies on the L1 caches to store new transactional values of memory blocks, while old values are kept into the L2 cache. A pair of read and write transactional bits per L1 cache block marks whether the block was read or written within a transaction. Such bits can be flash-cleared on transaction commit and abort. In case of abort, the blocks whose transactional write bit is set are also invalidated.

The cache coherence protocol maintains strong isolation [12] and implements an eager conflict detection policy. The conflict resolution policy is requester-wins, where the requesting transaction wins the conflict and the requested one is aborted. The baseline cache coherence protocol aborts transactions on evictions. The replacement of a transactional block in an L1 cache implies losing track of transactional loads and stores, which jeopardizes transaction isolation, so transactions must be aborted on these type of evictions. Beside, L2 cache block replacements may abort a transaction because of the inclusion property.

IV. Hardware Irrevocability Mechanism

A common way to deal with hardware capacity overflows and to ensure forward progress in commercial BE-HTM systems is a software fallback path. The code that Intel suggests as fallback path in its optimization manual [13]

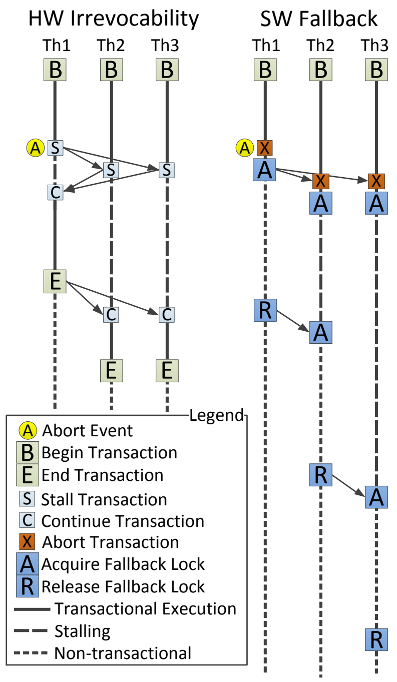

Fig. 2: Execution scenario of hardware irrevocability vs. software fallback.

comprises a global lock to execute a failed transaction as a non-transactional critical section. Once a transaction aborts a given number of times, the fallback path is taken. In addition, when a transaction is successfully started, the fallback global lock is checked. If the lock was acquired, the transaction aborts. If not, the transaction goes ahead with the lock in its read set so that another transaction acquiring the lock can abort it. The clash of transactions and fallback path sections is thereby avoided.

A hardware irrevocability mechanism provides several benefits over a software fallback code of that kind:

- The programmer is not burdened with the task of writing and tuning a fallback code, which reduces the programming effort of transactional applications, one of the main goals of transactional memory.
- There is no need for a lock so it is neither cached nor added to the read set of the transaction, thus freeing limited hardware resources.
- Performance benefits: Fig. 2 shows an execution scenario where a hardware irrevocability mechanism performs better than a software fallback code. The fallback path version aborts transactional execution and retries the transaction as a locked critical section. The other transactions running in the system abort as well, since they read the lock at the beginning[1].

[1]The non-transactional write to the lock causes these aborts by means of strong isolation [12]. Correctness is ensured as locks and transactions are not allowed simultaneously.

Execution is rebooted and serialized. However, the hardware irrevocability mechanism does not discard the transactional work done so far. The other transactions are stalled when a transaction gets irrevocable. Furthermore, the irrevocable one does not have to abort if it gets irrevocable just before the event that causes irrevocability, e.g. before an L1 cache replacement.

A. Implementation

We propose a token-based implementation of the irrevocability mechanism where only the core that owns the token can be irrevocable. Each core has a flag that indicates whether there is an irrevocable transaction running in the system (the I bit). Another flag in the core signals whether the irrevocable transaction belongs to this core or to another core, i.e. if the core owns the token (the T bit). Along with the pair of bits (I,T), each core has a counter (C) that holds the number of transaction retries. The core aks for irrevocability when C=0.

When a transaction reaches the limit of retries, the L1 cache controller of the core checks its (I,T) bits and acts depending on their value:

- *(I,T) = (0,0)*: There are no irrevocable transactions running in the system and the token is not owned. In this case, the controller broadcasts a token request message that will be responded by the core that owns the token. Should the owner just start irrevocability, the token is not sent and the requester keeps stalling until the owner ends its transaction. If the token is received, the T bit is set to 1 and the controller broadcasts an irrevocability request message for the other cores to set the I bit to 1. The requester can safely continue its transaction in irrevocable mode, (I,T) = (1,1), after acknowledgement of the other L1 cache controllers.
- *(I,T) = (0,1)*: The core owns the token, so it can request irrevocability directly.
- *(I,T) = (1,0)*: Someone else is running an irrevocable transaction. Consequently, the transaction stalls. This value for the (I,T) pair can be found on transaction beginning and after receiving an irrevocability message.

We have modified the L1 cache controller to implement the anticipation to a block replacement. Table I shows the modifications made to the protocol highlighted in gray. L1 cache replacements are left untouched whenever either the block to be replaced is not transactional, $\neg(xR \vee xW)$, or the core is in irrevocable mode and owns the token, (I,T) = (1,1). However, if the block is transactional, $xR \vee xW$, the counter (C) is checked. If $C>1$ (1 instead of 0 to anticipate the last abort) the transaction aborts and C is decremented. Conversely, if $C \leq 1$, the core asks for irrevocability and the mandatory queue is recycled [2] so that the event is triggered later on. Should the core manage to get irrevocable, the L1 replacement is performed safely. If irrevocability is not granted, the core stalls by continuously recycling the message that causes the eviction.

In case of L1 transactional block replacements due to L2 cache evictions (L2 Replace events in Table I) we have different scenarios. If the core is running an irrevocable transaction, (I,T) = (1,1), the event is treated as a normal L2 cache replacement. However, if the irrevocable transaction is of another core, (I,T) = (1,0), the transaction in this core must be aborted in favour of the irrevocable one. Thus, the only situation in which a transaction asks for irrevocability on an L2 Replace event is when $C \leq 1$ and there is no other irrevocable transaction in the system, (I,T) = (0,-).

The special case in which several transactions ask for the token at the same time is arbitrated by the controller queue of the core that owns the token. The owner of the first token request message found in such a queue is the one that gets the token. The rest of the token request messages are ignored and the requesters stalled. They will ask for irrevocability again after receiving a message of end of irrevocability.

V. Simulation Environment

The simulation environment comprises the full system simulator Simics [15], and the Wisconsin GEMS [16] toolkit that includes Ruby, a multiprocessor memory system timing simulator, which we have modified to simulate the best-effort HTM system outlined in Section III, and the proposals described in this paper.

The target system is organized as shown in Fig. 1. It comprises 16 in-order single-issue cores, with a private 32KB split 4-way L1 cache where the data cache holds two read and write transactional bits per 64B block. The L2 cache is unified, shared and divided into 16 banks of 512KB each. L2's associativity is 8-way and it does not hold transactional information. The directory keeps a full bit-vector of sharers. Each thread is bound to a core, and so it is the operating system, so that there are not interferences such as migrations and context changes. Consequently, there is a maximum of 15 threads for the use of benchmarks.

The whole Stanford STAMP suite [17] was used for the evaluation. Table II shows the parameters and characteristics of the benchmarks.

VI. Software Fallback Path Evaluation

Fig. 3 shows the fallback path code we have evaluated, which includes a variable to specify the number of transaction retries and the Lemming effect[3] avoidance code [9], [18]. The code defines a thread's local retry variable that is initialized to 0 (line 1). The retry limit is defined globally (`RETRY_LIMIT`). We define two primitives to begin a transaction: (i) `TAKE_XACT_CHECKPOINT` takes a register checkpoint where we want to resume the transaction on abort, but it does not start transactional bookkeeping; (ii) `BEGIN_XACT` begins transactional bookkeeping. Then, we can have non-transactional code between the two primitives

[2]The cache controller comprises queues where coherence messages are buffered until they are served by the controller [14]. In this case, there are a mandatory queue that holds the messages from the CPU to the L1 cache, a request queue that holds request messages from/to the L1 cache and a response queue with response messages from/to the L1 cache.

[3]If one transaction takes the fallback path, the others abort and wait for the fallback path lock to be released, i.e. a complete serialization of the ongoing transactions is carried out.

TABLE I: L1 cache coherence protocol modifications for irrevocability (highlighted in gray).

State	Events					
	L1 Replace ¬(xR∨xW) ∨ (1,1)	L1 Replace (xR∨xW)∧(C>1)	L1 Replace (xR∨xW)∧(C≤1)	L2 Replace ¬(xR∨xW) ∨ (1,1)	L2 Replace (xR∨xW) (1,0)∨(C>1)	L2 Replace (xR∨xW) (0,-)∧(C≤1)
I	–	–	–	ACK	–	–
S	– /I	Abort, C-1 /I	Irre, Z	ACK /I	Abort, C-1 /I	Irre, Zz
E	PUT(no data) /I	Abort, C-1 /I	Irre, Z	ACK /I	Abort, C-1 /I	Irre, Zz
M	PUT+Data /I	Abort, C-1 /I	Irre, Z	ACK+Data /I	Abort, C-1 /I	Irre, Zz

Irre: ask for irrevocability
Z and Zz: recycle mandatory and request queue, respectively
(#,#): pair of bits (I,T)

TABLE II: Workloads: Input parameters and transactional characteristics.

Bench	Input	# Xact	% Time in Xact	*avg*|**RS**|	*avg*|**WS**|
Bayes	-v32 -r1024 -n2 -p20 -i2 -e2 -s1	654	94%	87.64	48.91
Genome	-g512 -s32 -n32768	19496	85%	23.34	3.58
Intruder	-a10 -l16 -n4096 -s1	54933	92%	9.87	3.06
Kmeans	-m15 -n15 -t0.05 -i random-n2048-d16-c16	8235	46%	6.23	1.75
Labyrinth	-i random-x32-y32-z3-n96	222	100%	139.34	95.12
SSCA2	-s14 -i1.0 -u1.0 -l9 -p9	93721	13%	3.00	2.00
Vacation	-n4 -q60 -u90 -r16384 -t4096	4095	95%	63.20	10.16
Yada	-a20 -i633.2	5447	100%	62.45	38.21

to check whether we have to take the fallback path. The code to begin a transaction (lines 2-13) first takes a checkpoint and then increments the thread's local retry variable. Next, if the number of retries is greater than the retry limit (line 5), the fallback path is taken by acquiring a single spin lock (line 6). If the retry limit is not reached, the code executes transactionally and adds the lock to the read set (line 10). The transaction is explicitly aborted if the lock is taken (line 11). It should be noted that the thread waits for the lock to be released just before beginning the transaction to avoid the Lemming effect (line 8). The code to end a transaction (lines 14-19) checks the number of retries to execute either a transaction commit or a lock release.

On the right hand side of Fig. 3 we show an alternative implementation of the fallback path which replaces the single spin lock by a two-variable *ticket lock* [19]. Each thread takes its own ticket before entering the critical section by atomically incrementing and reading the global ticket variable (line 6). Then, the thread waits for his turn by checking it against the global turn variable. The global turn is atomically incremented to release the lock (line 18). The implementation of the Lemming effect avoidance loop (line 8) is more accurate with the ticket lock as the thread waits not only when the lock is taken (`lock != 0`) but also when there is a queue of threads waiting to acquire the lock (`globalTicket >= globalTurn`).

Fig. 4 depicts the speedup results obtained for those STAMP benchmarks that scale to some extent. The fallback code used is that of Fig. 3, with or without Lemming avoidance (±Lemm) and with single or ticket lock. The lazy single lock approach [20] is also shown, which is the same as the single lock without Lemming avoidance but the lock is checked lazily at the end of transactions.[4] The retry limit has been set to 5, which is a frequently used value [4], [6]. We have evaluated 3, 8 and 10 retries as well. An increased number of retries (8 or 10) seems to perform better when the number of threads, and therefore the contention, is high. For a low number of threads, a low number of retries suffices (3 retries up to 4 threads).

The results show that the fallback path versions with Lemming effect avoidance always beat the ones without it, due to the reduction in unnecessary serializations. As far as the type of lock is concerned, the ticket lock reveals itself as a good option since it reduces lock contention and ensures fairness in lock acquisition. But more importantly, the ticket lock provides the information of how many threads are waiting to enter the critical section and therefore, the Lemming loop waits for them to finish. Conversely, the single lock does not provide such information. Thus, the threads waiting at the Lemming loop may begin a transaction while other threads are contending for acquiring the lock. Those transaction will be aborted by the eventual lock acquisition. This fact is more probable in those benchmarks that spend a lot of time in transactions such as Genome, Intruder and Vacation (see Table II), which take advantage of the ticket lock Lemming loop enhancement to avoid unnecessary aborts. SSCA2 and Kmeans are most of the time out of transactions and they are not affected by the type of lock. The lazy single lock yields good results since it encourages parallelism. However, the performance is worse than the ticket lock with Lemming effect avoidance as the number of threads increases, thus increasing the contention (e.g. Intruder, Kmeans and Vacation with 15 threads). The fallback conflicts with the concurrent transactions.

VII. Hardware Irrevocability Results

Fig. 5 shows the speedup of the baseline BE-HTM system with the hardware irrevocability mechanism (Irre) and the software fallback path (Fback) with ticket lock and enhanced Lemming effect avoidance. The hardware irrevocability mechanism counter has been set to 5, as well as the retry counter

[4]In this manner, multiple transactions are allowed to execute in parallel with the one in the fallback path, as long as such transactions commit after the lock release and the fallback code does not conflict with them.

```
localRetries = 0;
void beginTransaction(&localRetries) {
  TAKE_XACT_CHECKPOINT; //Return point on abort
  localRetries++; //Increment xact retries
  if (localRetries > RETRY_LIMIT) { //Fallback?                  ┌myTicket = atomicIncrement(globalTicket);
    while(!lockAcquire(globalLock)) ; //Acquire lock  ←——→       └while(myTicket != globalTurn) ;
  } else { // Execute transactionally
    while(lock != 0) ; //Avoid Lemming effect  ←————————→         while(globalTicket >= globalTurn) ;
    BEGIN_XACT;
    if(lock != 0) //Add lock to the read-set ┐                    ┌if(globalTicket >= globalTurn)
      ABORT_XACT;                            ┘  ←————————→        └  ABORT_XACT;
  }
}
void endTransaction(localRetries) {
  //If not retry limit, assume lock's elided
  if (localRetries <= RETRY_LIMIT)
    COMMIT_XACT; //Commit
  else lockRelease(globalLock);  ←——————————————————→            atomicIncrement(globalTurn);
}
```

Fig. 3: Fallback code with retry limit and Lemming avoidance. Ticket lock alternative on the right.

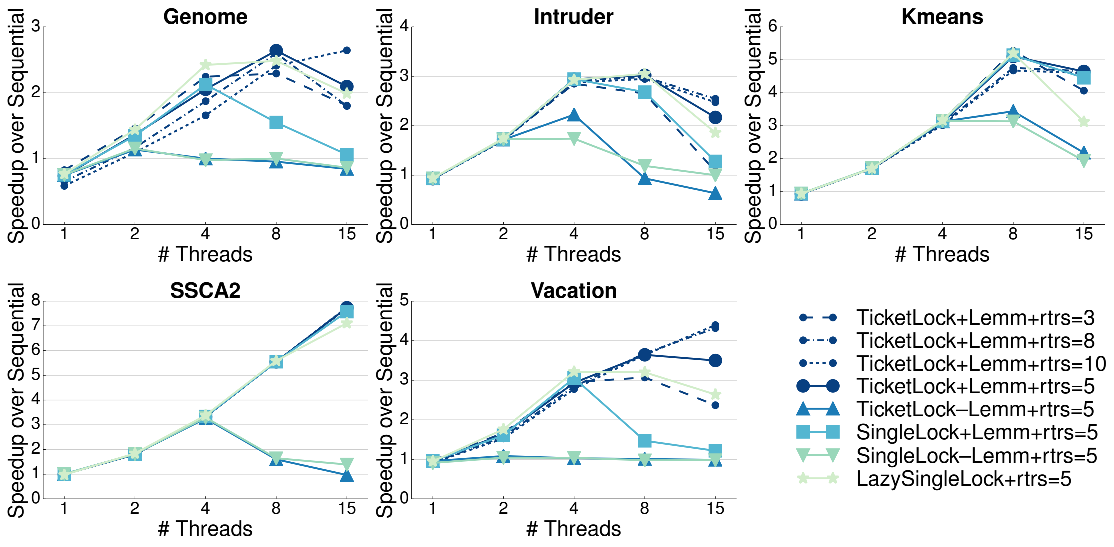

Fig. 4: Speedup over the sequential application for different fallbacks and parameters (Lemm: lemming effect avoidance, rtrs: number of retries).

of the fallback code. From these results we can classify the STAMP benchmarks in the following groups.

a) Bayes, Labyrinth and Yada: The speedup obtained for these benchmarks is barely that of the sequential version. And when there is only one thread the results are even worse than the sequential. The problem with performing worse than the sequential when we have only one thread running in the system is the number of retries before getting irrevocable or taking the fallback (set to 5 in this evaluation). With only one thread there is no abort due to conflicts, so all aborts are because of capacity overflows, that are usually persistent. This can be avoided by maintaining different retry counters as stated in Nakaike et al. [21], where they adapt the number of retries depending on the cause of abort. Three counters are used: one for aborts due to the fallback lock, a second for persistent aborts such as capacity aborts, and a third for transient aborts. In any case, the hardware irrevocability mechanism can implement different counters as well and it performs slightly better than the fallback path due to the last abort anticipation.

Although the irrevocable mechanism is better than the fallback one, these benchmarks do not scale because they exhibit large transactions in average, as shown by Table II. In addition, Table III shows the number of irrevocable transactions and its cause, and the majority of them are due to L1 replacements. We can also see how the number of irrevocable transactions increases with the number of threads because of conflict aborts and capacity overflows due to L2 evictions (the latter primarily in Yada).

b) Kmeans and SSCA2: These two benchmarks scales well and behave similarly either by using hardware irrevocability or the software fallback path. This is due to the short time spent in transactions that amounts to 46% for Kmeans and only 13% for SSCA2, which reduces contention.

The size of transactions in Kmeans and SSCA2 is also a factor to consider. Their small transactions make that the fallback path or hardware irrevocability are barely taken. Actually,

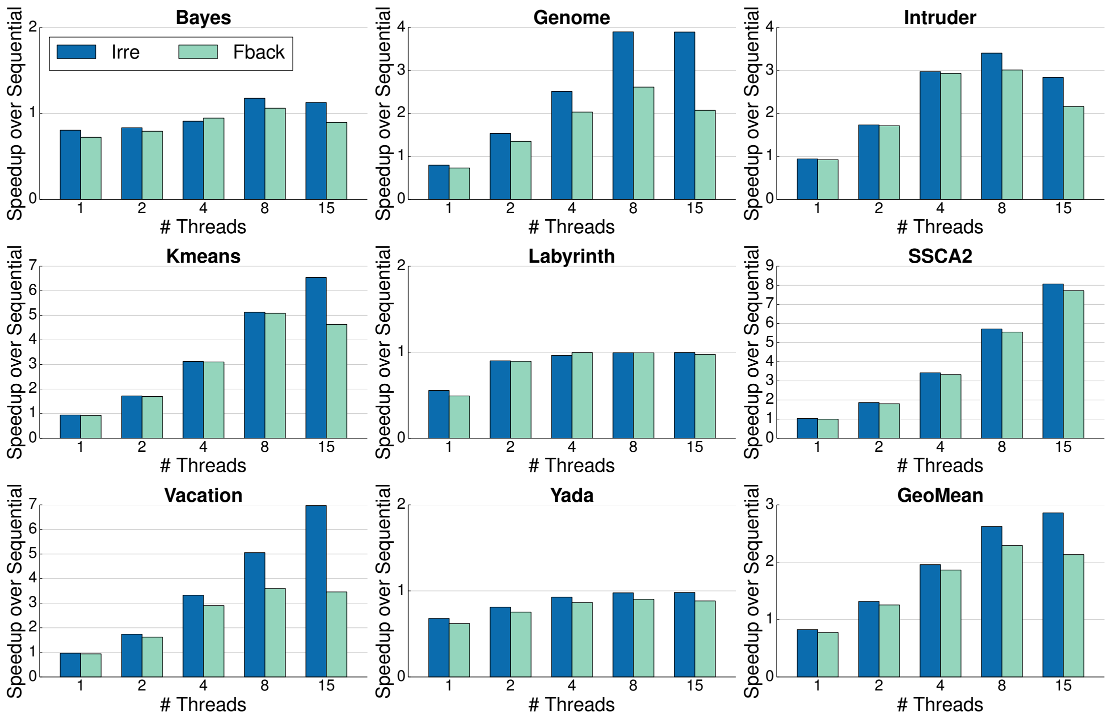

Fig. 5: Speedup of the hardware irrevocability mechanism (Irre) and the software fallback path (Fback) over the sequential application. The geometric mean is also shown (GeoMean).

TABLE III: Average number of irrevocable transactions, broken down into those due to L1 or L2 replacements, and those due to conflicts. Average number of aborts of irrevocability and fallback.

Bench	**# Irrevocable Xacts (L1/L2/Conflicts)**			**Aborts(IRRE/FBACK)**		
	4 th's	**8 th's**	**15 th's**	**4 th's**	**8 th's**	**15 th's**
Bayes	135(86/0/49)	148(68/0/80)	179(37/3/139)	653/728	788/1212	1040/1585
Genome	1203(1120/0/83)	1195(1017/0/178)	1679(1358/21/301)	5022/7942	5582/10821	8217/16321
Intruder	1055(22/0/1033)	3794(36/0/3759)	10562(110/1/10450)	11455/10428	35861/39628	77299/96499
Kmeans	400(0/0/400)	970(0/0/970)	1815(0/0/1815)	2193/2074	5425/6537	10296/19185
Labyrint	97(76/0/21)	122(57/0/64)	160(44/0/116)	435/631	617/783	797/931
SSCA2	127(0/0/127)	283(0/0/283)	515(0/0/515)	657/575	1583/2140	3208/5486
Vacation	249(217/0/33)	347(280/0/68)	433(301/0/132)	1272/2924	1773/6221	2357/9874
Yada	1021(702/0/319)	1245(710/0/535)	1557(628/57/872)	4651/9128	5895/12561	7600/13643

Table III shows 0 irrevocable transactions due to L1 and L2 replacements. However, contention makes some transactions to abort and take the fallback or the irrevocability mechanism when we have more threads. For this configurations we can see a slight benefit of irrevocability over the fallback version, or not so slight for Kmeans and 15 threads, because the irrevocability mechanism stalls the transactions instead of aborting them. Table III shows such an abort reduction that is up to 9000 transactions for Kmeans and 15 threads, which supposes an abort rate of 1.2 with irrevocability in contrast to the 2.32 of the fallback path.

c) Genome, Intruder and Vacation: For this group of benchmarks we obtain considerable benefits by using the BE-HTM system with hardware irrevocability over the fallback configuration. They are benchmarks with medium and small-sized transactions (Genome and Vacation) or that are more contended (Intruder). These characteristics can be noted in the number of irrevocable transactions that are due to replacements or conflicts in Table III.

The hardware irrevocability mechanism not only performs better due to the anticipation of the last abort but also reduces the number of aborts by stalling non-irrevocable transactions instead of aborting them. Table III shows that the number of transaction aborts for the system with irrevocability is usually lower than that of its software fallback counterpart. The amount of wasted work is larger for the fallback path, specially for Genome and Vacation with 15 threads, where the abort reduction is more than 50%.

VIII. Future Work

The lazy single global lock fallback proposed in [20] enhances parallelism between the transaction in the fallback path and the rest. The lock subscription is performed at the end of the transaction, just before committing, therefore transactions can run in parallel with the fallback as long as they commit after the fallback ends. Otherwise, those transactions will be aborted, thus wasting more energy and time than if they were aborted earlier.

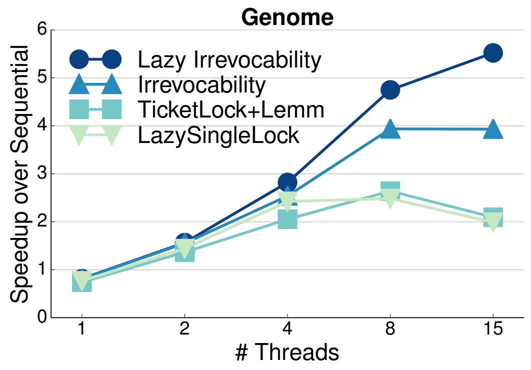

Fig. 6: Speedup of the lazy irrevocability mechanism compared with the proposals in this paper for the Genome benchmark.

Focusing on this problem, we propose an integration of lazy subscription and the irrevocability mechanism where transactions are allowed to run in parallel with the irrevocable transaction. The commit phase of the non-irrevocable transactions is delayed (the transaction is stalled instead of aborted) until the irrevocable one finishes. Such a delay ensures a correct execution and enhances parallelism.

Fig. 6 shows the speedup of the lazy irrevocability mechanism, compared with that of the normal irrevocability presented in this paper, and the fallbacks with ticket lock and Lemming avoidance (TicketLock+Lemm) and with lazy subscription (LazySingleLock). These preliminary results show that the aforementioned integration is worth exploring.

IX. Conclusions

In this paper we propose a hardware implementation of an irrevocability mechanism to gain insight into the hardware enhancements that may speedup the execution of a fallback path in BE-HTM systems. We find that anticipating the abort that causes the execution of the fallback path and stalling the other transactions running in the system yields a significant improvement over the *abort-all* fallback solution.

On the other hand, we propose an enhanced Lemming effect avoidance loop by means of a ticket lock. A ticket lock provides precise information of how many threads are waiting to acquire the lock, so the separation of transactional and non-transactional execution can be performed more precisely.

We suggest having a hardware accelerated fallback path to retain both hardware benefits and software versatility. However, the possibility of having a hardware alternative to the software fallback path can be interesting for the user due to its simplicity.

Acknowledgement: This work has been supported by the Government of Spain under project TIN2013-42253-P and Junta de Andalucía under project P12-TIC-1470.

References

[1] T. Harris, J. Larus, and R. Rajwar, *Transactional Memory, 2nd edition*. Morgan & Claypool Publishers, 2010.

[2] M. Herlihy and J. Moss, "Transactional memory: Architectural support for lock-free data structures," in *20th Ann. Int'l. Symp. on Computer Architecture (ISCA'93)*, 1993, pp. 289–300.

[3] H. W. Cain, M. M. Michael, B. Frey, C. May, D. Williams, and H. Le, "Robust Architectural Support for Transactional Memory in the Power Architecture," in *40th Ann. Int'l. Symp. on Computer Architecture (ISCA'13)*, 2013, pp. 225–236.

[4] C. Jacobi, T. Slegel, and D. Greiner, "Transactional Memory Architecture and Implementation for IBM System z," in *45th Ann. Int'l. Symp. on Microarchitecture (MICRO'12)*, Dec. 2012, pp. 25–36.

[5] 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 *21st Int'l Conf. on Parallel Architectures and Compilation Techniques (PACT'12)*, 2012, pp. 127–136.

[6] R. M. Yoo, C. J. Hughes, K. Lai, and R. Rajwar, "Performance Evaluation of Intel Transactional Synchronization Extensions for High-performance Computing," in *Int'l Conf. on High Performance Computing, Networking, Storage and Analysis (SC'13)*, 2013, pp. 19:1–19:11.

[7] C. Blundell, J. Devietti, E. C. Lewis, and M. M. K. Martin, "Making the Fast Case Common and the Uncommon Case Simple in Unbounded Transactional Memory," in *34th Ann. Int'l. Symp. on Computer Architecture (ISCA'07)*, ser. ISCA '07, 2007, pp. 24–34.

[8] A. Welc, S. Bratin, and A.-R. Adl-Tabatabai, "Irrevocable transactions and their applications," in *20th ACM Symp. on Parallelism in Algorithms and Architectures (SPAA'08)*, june 2008, pp. 285–296.

[9] D. Dice, M. Herlihy, D. Lea, Y. Lev, V. Luchangco, W. Mesard, M. Moir, K. Moore, and D. Nussbaum, "Applications of the Adaptive Transactional Memory Test Platform," in *3rd Workshop on Transactional Computing (TRANSACT'08)*, 2008.

[10] L. Hammond, V. Wong, M. Chen, B. Carlstrom, J. Davis, B. Hertzberg, M. Prabhu, H. Wijaya, C. Kozyrakis, and K. Olukotun, "Transactional memory coherence and consistency," in *31th Ann. Int'l. Symp. on Computer Architecture (ISCA'04)*, 2004, pp. 102–113.

[11] Y. Afek, A. Levy, and A. Morrison, "Programming with hardware lock elision," *ACM SIGPLAN Notices*, vol. 48, no. 8, pp. 295–296, aug 2013.

[12] M. M. K. Martin, C. Blundell, and E. Lewis, "Subtleties of Transactional Memory Atomicity Semantics," *IEEE Computer Architecture Letters*, vol. 5, no. 2, p. 17, 2006.

[13] *Intel 64 and IA-32 Architectures Optimization Reference Manual. Chapter 12.3: Developing an Intel TSX Enabled Synchronization Library*, Sep 2014.

[14] D. J. Sorin, M. D. Hill, and D. A. Wood, *A Primer on Memory Consistency and Cache Coherence*, 1st ed. Morgan & Claypool Publishers, 2011.

[15] P. Magnusson, M. Christensson, J. Eskilson, D. Forsgren, G. Hallberg, J. Hogberg, F. Larsson, A. Moestedt, B. Werner, and B. Werner, "Simics: A full system simulation platform," *IEEE Computer*, vol. 35, no. 2, pp. 50–58, 2002.

[16] 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," *ACM SIGARCH Computer Architecture News*, vol. 33, no. 4, pp. 92–99, 2005.

[17] C. Minh, J. Chung, C. Kozyrakis, and K. Olukotun, "STAMP: Stanford Transactional Applications for Multi-Processing," in *IEEE Int'l Symp. on Workload Characterization (IISWC'08)*, 2008, pp. 35–46.

[18] Y. Liu and M. Spear, "Toxic transactions," in *6th Workshop on Transactional Computing (TRANSACT'11)*. ACM, 2011.

[19] J. M. Mellor-Crummey and M. L. Scott, "Algorithms for scalable synchronization on shared-memory multiprocessors," *ACM Trans. on Computer Systems*, vol. 9, no. 1, pp. 21–65, Feb. 1991.

[20] I. Calciu, T. Shpeisman, G. Pokam, and M. Herlihy, "Improved Single Global Lock Fallback for Best-effort Hardware Transactional Memory," in *9th Workshop on Transactional Computing (TRANSACT'14)*, 2014.

[21] T. Nakaike, R. Odaira, M. Gaudet, M. M. Michael, and H. Tomari, "Quantitative comparison of hardware transactional memory for Blue Gene/Q, zEnterprise EC12, Intel Core, and POWER8," in *42nd Ann. Int'l. Symp. on Computer Architecture (ISCA'15)*, 2015, pp. 144–157.