Lazy Irrevocability for Best-Effort Transactional Memory Systems

Ricardo Quislant, Eladio Gutierrez, Emilio L. Zapata, and Oscar Plata

Abstract—IBM and Intel now offer commercial systems with Transactional Memory (TM), a programming paradigm whose aim is to facilitate concurrent programming while maximizing parallelism. These TM systems are implemented in hardware and provide a software fallback path to overcome the hardware implementation limitations. They are known as best-effort hardware TM (BE-HTM) systems. The software fallback path must be provided by the user to ensure forward progress, which adds programming complexity to the TM paradigm.

We propose a new type of irrevocability (a transactional mode that marks transactions as non-abortable) to deal with BE-HTM limitations in a more efficient manner, and to liberate the user from having to program a fallback path. It is based on the concept of lazy subscription used in the context of software fallback paths, where the fallback lock is checked at the end of the transaction instead of at the beginning. We propose a hardware lazy irrevocability mechanism that does not involve changes in the coherence protocol. It solves the unsafe execution problem of premature commits associated with lazy subscription fallbacks, and can be triggered by the user via an ISA extension, for the sake of versatility. It is compared with its software counterpart, which we propose as an enhanced lazy single global lock with escaped spinning at the end of the transaction. We also propose the lazy irrevocability with anticipation, a mechanism that cannot be implemented in software, which significantly improves the performance of codes with multiple cache evictions of transactional data.

The evaluation of the proposals is carried out with the Simics/GEMS simulator along with the STAMP benchmark suite, and we obtain speedups from 14% to 28% over the fallback path approaches.

Index Terms—Best-effort Hardware Transactional Memory; Irrevocability; Lazy Subscription; Transaction Fallback Path

1 Introduction

Transactional memory (TM) is a programming paradigm whose aim is to facilitate concurrent programming while maximizing parallelism in shared memory chip multiprocessors (CMPs). It is not until twenty years after its publication by Herlihy and Moss [1] that hardware transactional memory (HTM) becomes a reality with the release of Intel's Haswell [2] and IBM's BlueGene/Q [3] in 2013. IBM also releases two different systems with HTM included, System z [4] and Power 8 [5]. All these systems implement a best-effort form of HTM (BE-HTM), where transactions not encountering any problems are executed entirely in hardware. However, those that find limitations, such as hardware capacity overflows, interrupts, persistent conflicts, etc., have to be managed by a software fallback code to ensure forward progress.

The software fallback path of a transaction must be programmed by the user in the majority of the aforementioned systems, which defeats the purpose of simplicity that the TM paradigm claims. However, the burden on the programmer is relieved in the IBM BlueGene/Q system, with the provision of a software runtime fallback that implements irrevocability [6]. Whenever a transaction is unable to commit after a number of retries it automatically switches to irrevocable mode, so that it cannot be aborted anymore. In such a mode, the system may not be able to keep track of the data set of the transaction due to capacity overflow, therefore the rest of the transactions must be aborted or stalled to ensure a correct execution.

In this paper, we propose *lazy irrevocability*, an enhanced irrevocability mechanism based on the concept of lazy subscription [7] in the context of transactional fallback paths for BE-HTM systems [8]. In a simple global lock fallback implementation, transactions must subscribe to the fallback lock by reading it at the beginning of the transaction, so that they are aborted if the lock is acquired by the fallback path. Lazy subscription, though, subscribes to the lock at the end of the transaction, right before the commit, allowing more parallelism. Correctness is ensured because transactions are aborted if they commit before the fallback ends. Transactional isolation and strong isolation [9] make sure that a transaction is aborted whenever it accesses an inconsistent location. Considering this, our contributions are the following:

- A hardware lazy irrevocability mechanism: Transactions are allowed to run in parallel with the irrevocable one, but they are stalled when they reach the commit instruction as long as the irrevocable transaction is ongoing. Thus, the computation is preserved unless the irrevocable transaction or other normal transactions conflict with the stalling ones. Irrevocability is arbitrated by a token protocol without the need of a software lock, and the coherence protocol is not modified.
- Lazy irrevocability solves the problem of unsafe execution associated with lazy subscription fallbacks, where an inconsistent read from a transaction in parallel with the fallback may lead the transaction to jump to a commit instruction without performing the lock subscription.
- A lazy irrevocability mechanism with overflow anticipation: Both software fallbacks and irrevocability mechanisms abort transactions when there is a capacity overflow, and then the execution is serialized. However, we make

• *All the authors are with the Department of Computer Architecture, University of Málaga, Spain, 29071.*
E-mail: {quislant, eladio, zapata, oplata}@uma.es

Manuscript received Month XX, 20XX; revised Month XX, 20XX.

slight modifications to the coherence protocol in order to anticipate a transactional data cache eviction to start irrevocability from there on. Thus, less computation and energy are wasted. The overflow anticipation cannot be implemented in software.

- An enhanced software lazy single global lock fallback: We propose a software fallback path counterpart for the lazy irrevocability mechanism, which is the lazy single global lock proposed in [8] with an escaped spinning subscription. Such a subscription artificially delays the commit of transactions until the end of the fallback. The HTM system must support escape actions [10] to implement this fallback.

With such proposals we aim for a solution that does not require the user to program a fallback path for the transactions, and encourages parallelism when a transaction takes the fallback path. However, we also propose an ISA extension for the user to trigger irrevocability, for the sake of versatility.

The proposals in this paper are evaluated in a simulated BE-HTM system based on Simics [11]/GEMS [12] with the benchmarks of the STAMP [13] transactional suite. We carry out a performance comparison of our lazy irrevocability mechanisms along with the proposed enhanced software lazy fallback and other fallback codes, showing significant performance benefits for certain benchmarks.

2 Background and Related Work

Intel Haswell [2], IBM System z [4] and IBM Power 8 [5] include BE-HTM synchronizations facilities that require a software fallback path to guarantee forward progress. All of them use the private caches of the cores to keep transactional metadata, and the coherence protocol is used to detect conflicts. Regarding escape actions [10], only the Power 8 HTM system supports a form of non-transactional instructions within transactions via the so-called suspended transactional mode [14].

IBM Blue Gene/Q [3] uses a different approach to a BE-HTM system. BG/Q stores transactional values into the L2 cache, implementing a multi-versioned set-associative policy where a block can stay both transactionally and non-transactionally in the same set. To ensure forward progress on capacity overflows and contention scenarios, BG/Q implements an irrevocable mode by means of a software 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. If a transaction exceeds a given abort ratio, its ID is inserted into a hash table after being executed in irrevocable mode. Subsequent instances of the transaction switches into the irrevocable mode immediately after just one abort. Nevertheless, BG/Q has to abort a transaction to run it in irrevocable mode, whereas we propose a mechanism that anticipates an abort and initiates the irrevocable mode without discarding the transactional work. We also delay the commit of other transactions running in parallel with the irrevocable one to preserve their computation. Moreover, the effect of lock contention is not present in our lazy hardware irrevocability mechanism.

As far as the fallback path is concerned, a single global lock is the simplest form of ensuring forward progress. Those transactions that have to enter the fallback contend for acquiring the lock, while all transactions subscribe to the lock (i.e. they read the lock to include it into the read set) to be aborted whenever a fallback path is executed. Several fallback improvements have been proposed in the literature [15]. Dice et al. [16] propose a way to prevent the so-called *lemming effect*, by which one transaction that takes the fallback path aborts all other transactions once and again until they end up executing the fallback as well. Their proposal consists of busy waiting for the lock to be released before starting a transaction. Liu and Spear [17] propose a similar solution in the context of software TM (STM) called Hourglass. Aborts due to capacity overflow are non-existent in STM, but some transactions can be so conflicting that they are labelled as toxic. When a toxic transaction is detected no other transaction can begin until the toxic transaction commits. Hourglass is viewed as an alternative to existing contention management policies.

In the context of contention management policies, Afek et al. [18] propose a ticket-lock-based technique to improve the performance of Haswell's hardware lock elision (HLE). 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 to ensure forward progress.

The lazy single global lock fallback [8] is proposed to enhance parallelism in BE-HTM systems such as Haswell. Concurrency is allowed between the fallback path and the other transactions running in the system, as long as the transactions commit after the fallback ends. Otherwise the transactions are aborted. The lock subscription is done lazily at the end of transactions just before committing. Strong isolation and hardware sandboxing enforce correctness, although inconsistent reads may be a hazard.

Irrevocability in the context of HTM was first proposed in TCC [19] to deal with overflowed transactions. TCC relies on a commit arbitration protocol to grant commit permissions to one transaction at a time. During the commit window the network interconnect is owned by the transaction that holds commit permissions, which broadcasts its read/write sets to all cores for them to check for conflicts. TCC uses this mechanism to ensure forward progress of overflowed transactions. The transaction holds commit permissions from the time of the overflow to the commit time.

In the literature, irrevocability has been used to support operations that cannot be rolled back (like I/O and system calls), and to ensure forward progress due to contention and capacity overflow [6]. Blundell et al. [20] introduces OneTM-Serialized as a system where an overflowed transaction gets irrevocable and all other threads, both transactional and non-transactional, are stalled. They implement the irrevocability mechanism in a virtualized log-based HTM context where the irrevocable transaction can be explicitly aborted after serialization since transactional data can be recovered from a log. OneTM uses a private per-thread register with overflowed information and a shared transaction status word residing in a fixed virtual location that acts like a mutex lock to implement the access to irrevocable mode. Although such an approach is simple it hinders concurrency, therefore they propose OneTM-Concurrent, an implementation that allows non-overflowed transaction as well as non-transactional code to execute in parallel with the irrevocable transaction. For that purpose, they introduce persistent read/write bits in each block of physical memory to track the read/write sets of the overflowed transaction. Thus, conflicts with the irrevocable transaction can be detected by checking these bits. Our lazy irrevocability mechanism achieves the same concurrency purpose in a BE-HTM system without the need for such a drastic augmentation.

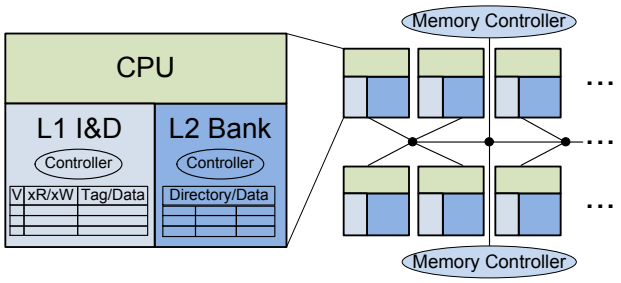

Fig. 1. Baseline architecture of the BE-HTM system.

3 Baseline Architecture

The baseline architecture used in this paper is shown in Figure 1. The system uses the private L1 caches to store new transactional values of memory blocks, while old values are kept in the shared L2 cache. A pair of read and write transactional bits (`xR/xW`) 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 [9] and implements an eager conflict detection policy. As far as the conflict resolution policy is concerned, the system implements requester-wins, where the requesting transaction wins the conflict and the requested one is aborted. This is the usually implemented policy in BE-HTM systems because of its simplicity. Other policies such as requester-stalls [21] may complicate the coherence protocol.

The cache coherence protocol is a MESI directory protocol that holds a full bit vector of sharers and has the following modifications to support the execution of transactions:

- *Backup on first transactional store*: If an L1 cache block is in M state and its write transactional bit is not set, the L1 cache has to send the data to the L2 cache before a transactional store is performed. This way the L2 cache holds the last old value for the block.
- *Abort on evictions*: The replacement of a transactional block in an L1 cache implies losing track of transactional loads and stores, which compromises transaction isolation; so transactions must be aborted on these type of evictions. Furthermore, L2 cache block replacements may abort a transaction because of the inclusion property, which is held in the baseline system.
- *L2 cache serves data of aborted transactions*: If a transaction requests data of an aborted transaction, it must be served by the L2 cache. There are two different situations: (i) The directory detects that the requester was the owner of the block, and assumes that the transaction was aborted and it is requesting the same data that was invalidated in the abort (invalidations are silent). In this case, the L2 cache serves the data without further ado; (ii) The directory forwards the request to the owner of the block, which is different from the requester. Next, the L1 cache that owned the block receives a forwarded message for a block that is not in the cache due to a recent abort. In this case, the L1 cache informs the L2 cache and the L2 cache serves the data. This implies an additional hop to the cache access.

Other remarks on the baseline architecture are the following. It does not include a randomized back-off policy to address contention scenarios that may lead to livelock [22]. Nested transactions [10] are flattened and treated as if they were part of the outer one. Escape actions [10] are allowed. The BE-HTM system is implicitly transactional, meaning that all memory operations within a transaction (except for escape actions) are implicitly taken as transactional.

4 Lazy Irrevocability

The lazy irrevocability mechanism involves changes in the procedures of beginning and committing a transaction, and it comprises a communication protocol to signal the switch to irrevocable mode of a transaction to the rest of the cores in the CMP. However, neither the cache coherence protocol nor the ISA need modifications, although we also propose an ISA augmentation to offer more versatility.

The execution flow of a transaction using lazy irrevocability is depicted in Figure 2. The `Begin Xact` procedure first checks whether the transaction has to be executed in irrevocable mode. This usually requires verifying that the transaction has reached the limit of retries. If the transaction has to be irrevocable we request irrevocability (Section 4.3 describes the protocol used) and the transaction is stalled until irrevocability is granted. Another transaction in irrevocable mode could make us stall until it ends. Once we have been granted irrevocability the transaction is executed with the transactional bookkeeping disabled, which means that the code is executed as non-transactional code, so it cannot be aborted. When the `Commit` instruction executes we know that the transaction was irrevocable, and we only have to notify the rest of the cores we are not in irrevocable mode anymore.

If the transaction must not get irrevocable yet, `Begin Xact` proceeds normally by enabling transactional bookkeeping so that subsequent loads and stores marks the `xR` and `xW` cache block bits respectively. Next, the body of the transaction is executed as usual. Conflicts are detected not only between the transaction and normal transactions, but also between the transaction and an irrevocable one, due to strong isolation. When the transaction reaches the commit phase the existence of an irrevocable transaction is checked. If the check is positive the transaction is stalled until the irrevocable transaction finishes. Then, it is allowed to commit. It should be noted that the transaction can be aborted while waiting for the irrevocable transaction to end.

The mechanism described above can be used transparently for the user as a sort of HLE feature, or it can be presented to the user via an ISA extension called `go_irrevocable` which returns whether irrevocability was granted or not. Thus, the user is free to tailor the code that decides to switch to irrevocability (number of retries, back-off, number of counters,...):

```
     localRetries = 0;        //User-tailorable
ret: localRetries++;          //User-tailorable
     if( localRetries > 5 )   //User-tailorable
       while(!go_irrevocable());       //Spin
     else
       xbegin(ret);     // Pass label for abort
     ...                      //Transaction body
     xcommit();
```

The user has to take into account that self-aborts cannot be used inside irrevocable transactions, in the same way it cannot be used inside fallbacks. The compiler can warn about this situation.

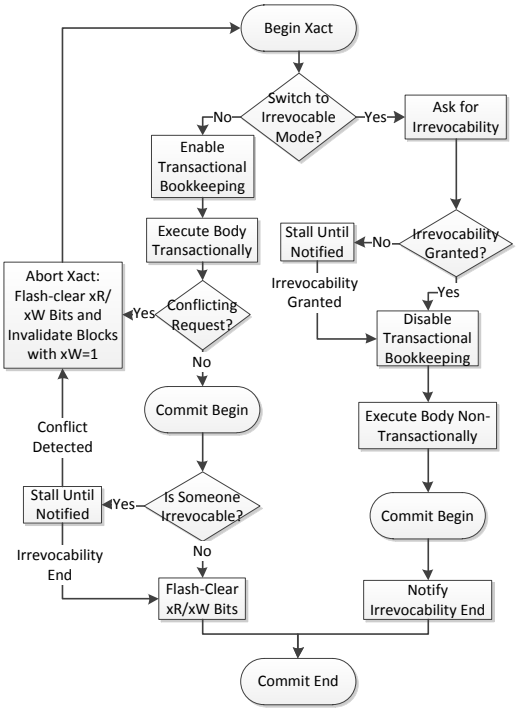

Fig. 2. Flowchart of transaction execution with lazy irrevocability.

4.1 Correctness

The lazy irrevocability mechanism described in Figure 2 ensures a correct execution as three properties are fulfilled:

- Transactional isolation: Is the property by which transactional stores are visible only by the transaction that performs them.
- Strong isolation: Defines the relationship between transactional and non-transactional code. With strong isolation, if non-transactional code writes a location which is in the data set of a transaction, the transaction is aborted.
- Delayed commit: the lazy irrevocability mechanism delays the commit of a transaction whenever there is an irrevocable transaction running in the system.

Figure 3 shows all four execution cases depending on when a transaction and an irrevocable transaction begin and commit. The first case shows an irrevocable transaction that begins after the normal transaction in the second thread (`Th2`). However, the transaction in `Th2` happens to commit before the irrevocable transaction ends. In this case, we should abort `Th2`'s transaction since it could have loaded from a location (`LD Y` in the figure) written by the irrevocable transaction (`ST Y`) and commits earlier, which compromises serializability [23] (ignore variable `X`, which is used in the next section). Unlike the lazy single global lock fallback described in Section 2, we do not abort the transaction. Instead, we stall it by delaying the commit until the irrevocable one ends. In the worst case scenario, the transaction will be aborted later than with the lazy fallback because of a conflict with other transactions, or with the irrevocable one due to strong isolation. Hopefully, the transaction can commit thus enhancing parallelism.

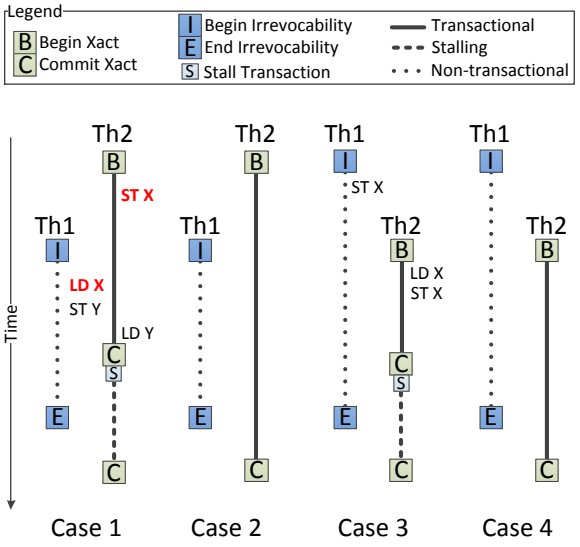

Fig. 3. Execution cases to show correctness of lazy irrevocability.

Case 2 is a correct scenario both with the lazy single global lock fallback and with lazy irrevocability. Actually, we translate Case 1 scenarios to Case 2 scenarios by artificial adaptation with the delayed commit.

The same adaptation happens with Cases 3 and 4. In both cases the normal transaction begins after the irrevocable one, but in Case 3 the normal transaction commits before the irrevocable ends. Again, serializability could be broken if, as shown in Figure 3, `Th1` modifies `X` before `Th2`'s transaction reads it. Delaying the commit of `Th2`'s transaction solves the problem.

Last but not least, the lazy irrevocability mechanism solves a serious problem in BE-HTM systems with lazy subscription fallbacks [8]. A transaction running in parallel with the fallback code may read an inconsistent location from the fallback. The inconsistent read might cause a spurious write to another location used by the transaction as the target of a jump. If such a target is a commit instruction, the transaction may commit incorrectly without performing the lock subscription. However, our lazy irrevocability mechanism implements the subscription in the commit instruction thus solving this hazard.

4.2 Potential Optimizations

It should be noted that further optimizations can be included in a system with lazy irrevocability, as we know that an irrevocable transaction always commits before any other transaction in the system. For example, Case 1 in Figure 3 shows a RAW conflict between `Th1` and `Th2` on variable `X`. However, such a RAW conflict is actually a WAR one when using lazy irrevocability, that can be removed by serving the old value of `X` to `Th1`.

`LD X` is highlighted because it will abort `Th2`'s transaction: `Th1` sends a GET message to the directory, which forwards the message to `Th2`. `Th2` checks the `xW` bit of `X` and finds a conflict. Next, `Th2` aborts its transaction and notifies the directory, which serves the old data. Such a procedure could be optimized if the `Th2`'s transaction is not aborted, since we know that it is going to commit later than the irrevocable one. However, implementing

such an optimization might imply profound changes to the cache coherence protocol, as we might have a cache block modified by two cores at the same time.

4.3 Irrevocability communication protocol

We propose a token-based implementation of the irrevocability communication protocol where only the core that owns the token can run irrevocably. 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. whether the core owns the token or not (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 requests irrevocability when C is 0.

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

- *(I,T) = (0,0)*: There are not irrevocable transactions running in the system and the token is not owned. If we have to request irrevocability, 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. On the other hand, a transaction is committed normally if (I,T) = (0,0).
- *(I,T) = (0,1)*: The core owns the token, so it can request irrevocability directly, as no one else is in irrevocable mode.
- *(I,T) = (1,0)*: Someone else is running an irrevocable transaction. A transaction that reached the retry limit would stall if this value for the (I,T) bit pair is found. Likewise, when a transaction attempts to commit and finds (1,0) it has to delay the commit until it finds (0,0).

The special case in which several transactions ask for the token at the same time is arbitrated by the network queues and the request queue of the core that owns the token. Figure 4 shows a core with the queues that communicates the CPU with the cache controller, mandatory and data queues, and the cache controller with the network, request and response queues. Four cores are shown, of which 2 and 4 send a token request at the same time, ①. The first point of arbitration is the router of the network that happens to route first the message from Core 2, ②. Consequently, the token request message from Core 2 is enqueued before that from Core 4 into the request queue of Core 1, ③, which owns the token as its T bit is set to one, (0,1). Next, the controller of Core 1 sends the token to Core 2, sets T to 0 and ignores the token request from Core 4. Note that Core 2 and 4 do not know which core owns the token, so request messages are actually broadcast.

4.4 Discussion

Using a software lock might be an alternative to our token-based communication protocol. However, lock contention has an effect on parallelism (see Section 7.3.1) and involves inundating the network interconnect with request and invalidation packets that include the address of the lock. Such packets are much larger than the packets needed to communicate irrevocability in our protocol, which only convey the source and destination processor IDs. Furthermore, the software lock (and the counter) has to be allocated in the L1 cache in order to be checked to implement the delayed commit, with the potential risk of evicting transactional data that would abort the transaction.

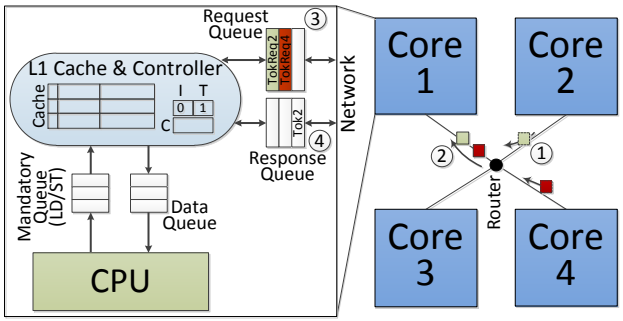

Fig. 4. Arbitration of the irrevocability token.

The lock, though, solves the problem of descheduling an irrevocable transaction as it is allocated in the virtual memory space of the application and can be identified by its address. However, if a lazy irrevocable transaction is descheduled, the (I,T) bits and the counter are saved with the thread context. The operating system may schedule a thread of another transactional application with its own (I,T) bits. If such a transaction is irrevocable as well, it may commit and notify the end of irrevocability. Therefore, the other threads that belong to other applications would change their (I,T) bits accordingly, when they should not. A solution might hold a process identifier per thread context associated to the (I,T) bits, similar to the speculation identifiers in BlueGene/Q [3], which associate a memory operation with a transaction to allow SMT (not assumed in our baseline system). BG/Q allows 128 transactions IDs which are reclaimed via hardware scrubbing when transactions commit. If no IDs are available transactions cannot begin. As regards our communication protocol, it should be modified to include the process ID in the packets in order to compare it before performing any action. Such modifications are not included in this paper, and are left for future work.

5 Lazy Irrevocability with Overflow Anticipation

There is an optimization to the lazy irrevocability mechanism that cannot be carried out by a software fallback. This is the overflow anticipation, and consists of requesting irrevocability just before a transactional block eviction. If the irrevocability is granted the transaction is able to continue without having to abort.

Figure 5 shows an execution scenario comparing the hardware lazy irrevocability mechanism with overflow anticipation, with a lazy subscription software fallback. Thread 1, `Th1`, finds a hardware overflow and has to evict a transactional block. In the particular case of lazy irrevocability with overflow anticipation (Figure 5a), the thread asks for irrevocability before evicting the block, and if it is granted the transaction can continue in irrevocable mode and is not aborted. When it comes to the fallback (Figure 5b), `Th1`'s transaction is aborted and reexecuted acquiring

TABLE 1
L1 cache coherence protocol modifications for lazy irrevocability with overflow anticipation (highlighted in gray).

State	Events					
	L1 Replace ¬(xR∨xW)	L1 Replace (xR∨xW)∧(C>1)	L1 Replace (xR∨xW)∧(C≤1)	L2 Replace ¬(xR∨xW)	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, Zz: recycle mandatory and request queue, respectively.
(#,#): pair of bits (I,T).

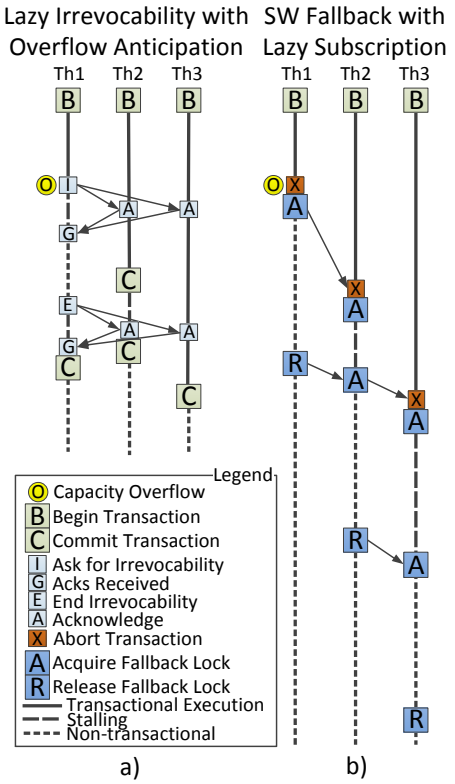

Fig. 5. Execution scenario of lazy irrevocability with anticipation (a) vs lazy subscription fallback (b).

the fallback lock. The other transactions running in the system continue as the subscription is lazy. `Th2`'s transaction happens to commit before `Th1`'s, so `Th2`'s is aborted on lock subscription. After aborting, the transaction waits for the lock to be released. With lazy irrevocability (Figure 5a), `Th2` stalls until receiving the message of end of irrevocability. `Th3`'s transaction commits after `Th1`'s and `Th2`'s but, in the case of the fallback (Figure 5b), `Th3` finds that the lock was just acquired by `Th2` and is aborted.

The scenario in Figure 5 is somewhat pessimistic for the software fallback but shows the potential of a lazy irrevocability mechanism and the benefits of the overflow anticipation. The hardware irrevocability mechanism with overflow anticipation preserves computation whenever it is possible.

5.1 Implementation

The implementation of the overflow anticipation requires slight modifications to the cache coherence protocol since we have to intervene the actions performed when a replacement event is triggered by the L1 cache. Table 1 shows such modifications highlighted in gray. L1 cache replacements of non-transactional blocks, $\neg(xR \vee xW)$, are executed without modification. However, if the block is transactional, $xR \vee xW$, the controller counter (C) is checked to see if the transaction has reached the limit of aborts. If not, the transaction is aborted and the counter decremented. The block state changes to invalid, I. Note that a cache replacement may not be a persistent event. Therefore, the transaction is retried C times before changing to irrevocable mode. On the other hand, if the counter reached the limit, $C \leq 1$, we ask for irrevocability. The comparison is made with less or equal to anticipate the last abort. The message that caused the L1 replace event is recycled in the mandatory queue until irrevocability is granted, thus stalling the processor. One important step which is performed by the irrevocability protocol is the following: once the transaction changes to irrevocabile mode, xR and xW bits are flash-cleared as if it was a transaction commit. In this manner, the recycled message that caused the transactional L1 replacement now triggers a non-transactional L1 replacement, and subsequent replacements are triggered likewise.

Regarding L1 transactional block replacements due to L2 cache evictions and the inclusion property, Table 1 shows such events named as L2 Replace. Non-transactional L2 replacements are left untouched. The replacement is acknowledged and the data invalidated. The data is sent as well if it was modified. However, when the data to be evicted by L2 is transactional in the L1 cache, $xR \vee xW$, we abort the transaction and the counter is decremented if either the counter did not reach the limit of retries or there is another transaction in irrevocable mode, $(1,0)$. The latter condition, which is not present in L1 cache replacements, is needed to abort those transactions that reached the limit of retries. We cannot stall transactions on L2 replacements since we do not know if such replacements are due to an irrevocable transaction that depends on the L2 eviction to continue. If the transaction is stalled waiting for irrevocability, we would risk reaching a deadlock scenario. Therefore, the thread asks for irrevocability only when no one in the system is in irrevocable mode and the transaction reached the limit of retries, $(0,-) \wedge (C \leq 1)$. Irrevocability might not be granted due to another transaction that asked for irrevocability at the same time, of which we have not been informed yet. Once we get informed, the (0,-) bits change to (1,0), and the recycled message triggers the event that aborts the transaction.

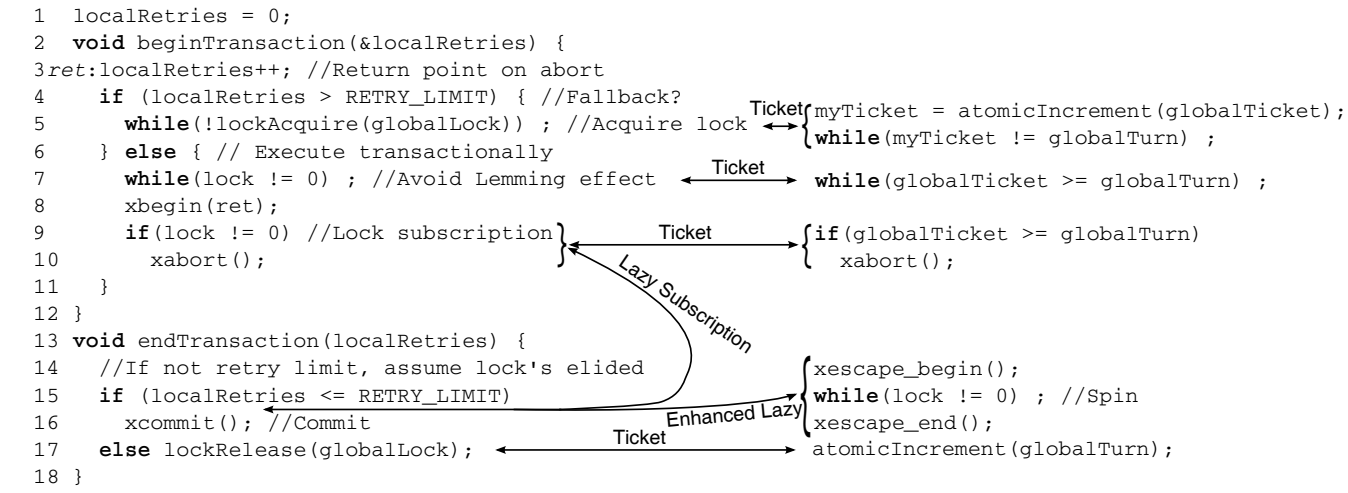

Fig. 6. Software fallback code variants.

6 Software Fallback Alternatives

In this section we survey the software fallback alternatives found in the literature [7], [16], [17] and propose two optimizations (ticket lock and escape spinning) to get a comprehensive set of fallbacks to compare with our lazy irrevocability proposals.

Figure 6 shows the code of a single lock fallback path with retry limit and lemming effect avoidance [16], [17]. The lemming effect occurs due to the lock subscription inside transactions, and causes a complete serialization of all transactions running in the system. When a transaction reaches the limit of retries and acquires the fallback, the ongoing transactions abort and retry. The lock is already taken when they subscribe to it, therefore they abort again until they reach the retry limit. Finally, all transactions end up by executing the fallback. The solution to avoid such an effect is described below.

To start a transaction we use the alternative provided by Haswell where the jump to the abort handler is exposed to the user (`ret` label in line 3). Thus, we can insert non-transactional code after beginning the transaction in order to check whether the transaction has to take the fallback path or not.

The procedure to begin a transaction (lines 2–12) starts off by taking a checkpoint and incrementing the local variable that holds the number of retries (line 1) of the transaction. Then, if the transaction reached the retry limit, defined by a global variable (`RETRY_LIMIT`), the thread tries to acquire a global spin lock (line 5). Otherwise, the thread executes transactionally (lines 7–11). Line 7 shows the solution to avoid the aforementioned lemming effect. It consists of having the thread waiting in a loop until the global lock is released. In this manner, the fallback aborts the other transactions in the system only once, and transactional execution is resumed after the fallback execution. Next, the `xbegin(ret)` primitive starts the transactional bookkeeping and, immediately after, the if statement subscribes the transaction to the lock (lines 9–10). Note that a explicit abort has to be placed in the lock subscription in order to abort the transactions that begin after the fallback. On the other hand, the procedure to commit a transaction (lines 13–18) checks the local retry variable to know whether the fallback lock was acquired. If not, the transaction commits normally. Otherwise, the lock is released.

An enhancement to the fallback code in Figure 6 is the lazy subscription proposed in [7]. The subscription code (lines 9–10) is moved to the end of the transaction, between lines 15 and 16, thus allowing transactions in parallel with the fallback code. The isolation and sandboxing provided by the HTM facility, along with the subscription abort ensures correctness. With lazy subscription, the lemming effect avoidance code in line 7 would not be necessary since transactions are not immediately aborted after beginning. However, a long fallback might cause the same effect. Next, we describe a ticket lock lemming effect avoidance that alleviates this problem.

We propose two enhancements to the fallback codes discussed above. The first one solves a problem that arises from using a single lock in the fallback with early subscription and lemming effect avoidance. When more than one transaction reached the limit of retries and one of them acquires the fallback lock, the others wait for the lock to be released. In addition, the transactions that have not reached the limit wait at the lemming loop. Then, we have a group of threads waiting to begin fallbacks and another waiting to begin transactions. Once the lock is released, the former contends for the lock while the latter begins the transactions. After lock contention one thread acquires the lock and aborts all transactions. We propose to enhance the accuracy of the lemming effect avoidance loop to solve this problem by means of a *ticket lock* [24]. The ticket lock uses two variables to maintain the turn and the ticket that grant access to a critical section. A thread that wants to enter the critical section takes a ticket and waits for its turn. The ticket is incremented atomically (line 5, right). The lock subscription changes to accommodate the ticket lock (lines 9–10, right), and the lock release is achieved by incrementing the turn atomically (line 17, right). Finally, the lemming effect avoidance loop (line 7, right) now checks if the ticket is greater or equal than the turn, which waits for all the threads waiting to execute the fallback path.

We also propose an enhanced lazy subscription fallback path as counterpart of our lazy irrevocability mechanism without anticipation (anticipation cannot be implemented in software) described in Section 4. Essentially, it is an escaped spinning subscription. The standard lazy subscription mechanism aborts the transaction if the lock is taken. However, we wait for the lock to be released to commit after the fallback, which ensures correctness and tries to discard as little work as possible. The wait has to be escaped (`xescape_begin()` and `xescape_end()` instructions be-

tween lines 15 and 16, right) so that the transaction is not aborted due to strong isolation. The transaction could also be aborted by either a conflict or a capacity overflow due to having to allocate the lock variable into the cache. It should be noted that escape actions are not provided by all commercial HTM systems, e.g. Haswell.

7 Evaluation

7.1 Methodology

We use the full system simulator Simics [11] along with the Wisconsin GEMS [12] module for Simics. Simics runs an unmodified instance of Solaris 10, and we have modified Ruby, the multiprocessor memory system timing simulator included in GEMS, by implementing the best-effort HTM system outlined in Section 3, 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. L2 cache is unified, shared and divided into 16 banks of 512KB each. L2's associativity is 8-way with 64B blocks. The directory keeps a full bit-vector of sharers. For the network time modelling we have used Garnet [25], a detailed interconnect model that integrates Orion [26], a network power model.

Each thread of a benchmark is bound to a processor to prevent migrations. Consequently, the maximum number of threads is limited to 15 as one processor is assigned to the operating system.

Ruby adds pseudorandom delays to memory accesses to include variability in simulations. Then, multiple runs of each experiment were carried out to obtain 1% error bars [27].

The benchmarks used for the evaluation are the Stanford's STAMP suite [13], version 0.9.9. Table 2 shows the parameters and characteristics of the benchmarks: the number of transactions that successfully commit (# Xact), the percentage of time running transactions (% Time in Xact), and the average read/write set cardinality (RS/WS), in cache blocks.

7.2 Software Fallback Results

Figure 7 shows the speedup over the sequential application for the STAMP benchmarks that scale better in the baseline system described in Section 3, with the software fallback alternatives discussed in Section 6. All experiments were conducted with the retry limit set to five, which is a frequently used value [2], [4]. We can see that the single lock with lemming effect avoidance yields poor results for the majority of benchmarks when the number of threads increases. More threads means more contention, and several threads may reach the retry limit at the same time, which unveils the inaccuracy of the lemming effect avoidance loop when using a single lock. Figure 8 shows the percentage of transactions that takes the fallback path, and the ticket lock shows a significant reduction with respect to the single lock due to the accurate lemming effect avoidance loop. Kmeans-high is the only case where the ticket lock seems to increase the number of transactions taking the fallback. However, it does not translate into a performance degradation due to the low percentage of time that the benchmark stays inside transactions (see Table 2).

Regarding the lazy subscription single lock, it performs very similarly to our early subscription ticket lock with lemming effect avoidance. Nevertheless, the lazy single lock experiences a slowdown with 8 and 15 threads. Figure 8 shows an increased percentage of transactions that take the fallback compared with our ticket lock approach. The lazy subscription encourages parallelism, but transactions are aborted if they end before the fallback. In addition, aborted transactions are retried in the hope that the fallback is finished, and they might be aborted again. To tackle such a problem we try our ticket lock lemming effect avoidance loop in conjunction with lazy subscription, and we can see that it improves the performance as the number of threads increases. The lazy ticket lock allows the ongoing transactions in parallel with the fallback, but if the transactions abort they are stalled in the lemming loop. Figure 8 shows a significant reduction in transactions that take the fallback for the lazy ticket lock with lemming effect avoidance.

Last but not least, we propose an alternative to the lazy single lock fallback in systems that provide escape actions. Escape actions allow a spinning lock subscription that waits for the fallback lock to be released, instead of aborting if the fallback has not ended yet. This fallback yields the best results as can be seen in Figure 7, with noticeable improvements for Genome and Vacation. Figure 8 shows how the number of transactions that take the fallback path is also reduced, except for Intruder. Intruder shows more contention with high-threaded loads, so delaying transaction commit may cause more aborts due to conflicts. Other factors such as capacity overflows due to the fallback lock may have an influence. However, performance is scarcely affected.

Implication: The ticket lock lemming effect avoidance fallback with lazy subscription should be used instead of either a single lock approach or the lazy single lock solution, whenever the BE-HTM system does not support escape actions. Otherwise, the lazy single lock fallback with escaped spin should be used in a system with escape actions.

7.3 Irrevocability Results

In this section we discuss the results obtained with our hardware irrevocability mechanisms, with and without overflow anticipation, compared with the results of the escaped spin lazy single lock software fallback path, which yields the best results of the fallbacks evaluated in the last section. The escaped spin lazy single lock fallback is the software counterpart of our irrevocability mechanism without anticipation. The fallback uses lock synchronization instead of our irrevocability communication protocol, and the BE-HTM system must support escaped actions to escape the lazy spinning subscription to the lock. Figure 9 shows the results for the aforementioned proposals along with the lazy ticket lock with lemming avoidance, the best fallback without escape actions. The hardware irrevocability mechanisms' retry counter have been set to five like that of the fallback variants.

The results show three benchmarks that do not scale: Bayes, Labyrinth and Yada. Their parallel performance is similar to that of the sequential. In fact, with one thread the results are even worse than the sequential. This performance degradation with one thread is due to the number of retries before getting irrevocable or taking the fallback, which is five in this evaluation. With only one thread there are no aborts due to conflicts, but to capacity overflows (see Table 3). To ameliorate this issue we can set different retry counters as stated by Nakaike et al. [28], where the number of retries depends on the cause of abort. However, the number of counters and its maximum value are characteristics easily applicable to our hardware irrevocability mechanisms, and we have used only one counter for the sake of simplicity. In any case, the hardware irrevocability mechanism with overflow anticipation performs slightly better than the fallbacks as it can anticipate the last abort.

TABLE 2
Workloads: Input parameters and transactional characteristics.

Benchmark	Input	# Xact	Time in Xact	$avg\|\mathbf{RS}\|$	$avg\|\mathbf{WS}\|$
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-high	-m15 -n15 -t0.05 -i random-n2048-d16-c16	8235	46%	6.23	1.75
Kmeans-low	-m40 -n40 -t0.05 -i random-n2048-d16-c16	10980	12%	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-high	-n4 -q60 -u90 -r16384 -t4096	4095	95%	63.20	10.16
Vacation-low	-n2 -q90 -u98 -r16384 -t4096	4095	93%	48.17	8.60
Yada	-a20 -i633.2	5447	100%	62.45	38.21

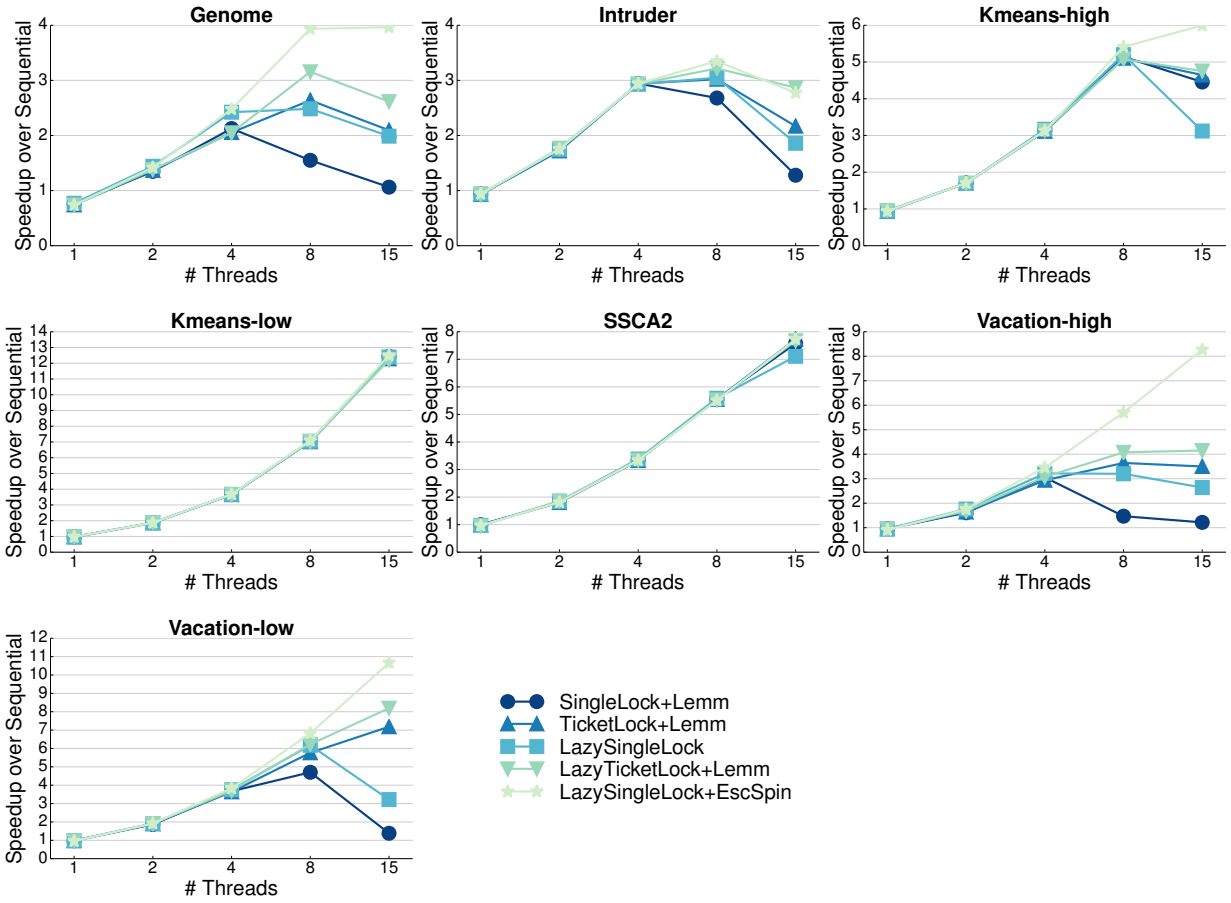

Fig. 7. Software fallbacks' speedup over sequential for the benchmarks that scale better. Lemm: lemming effect avoidance; EscSpin: Escaped spin.

Another group of benchmarks, composed of Kmeans-low, SSCA2 and Vacation-low, scales properly but shows little difference among the various serialization methods. Such benchmarks have small transactions in average, Kmeans-low and SSCA2 spend about 13% of time in transactions (see Table 2), and all of them exhibit low contention among transactions. Table 3 shows the number of irrevocable transactions for the system with irrevocability with overflow anticipation. The irrevocable transactions are broken down into those due to L1 transactional replacements, L2 evictions that cause L1 transactional replacements due to the inclusion property, and irrevocable transactions due to conflicts. For this group of benchmarks we can see that the percentage of irrevocable transactions is very low: 0.56%, 5.16% and 6.03% for SSCA2, Kmeans-low and Vacation-low respectively. Moreover, Kmeans and SSCA2 do not show irrevocabilities due to cache replacements, therefore the irrevocability with anticipation mechanism does not result in any enhancement. Conversely, Vacation shows more irrevocable transactions due to replacements, on account of its larger read set, but they are not enough to suppose an improvement. However, in the case of Vacation there is a noticeable improvement when using the parallelism encouraging mechanisms of irrevocability and fallback with escaped spin, instead of the more conservative lazy ticket lock with lemming effect avoidance. Again, we can observe a slight improvement of the irrevocability mechanisms over the escaped spin fallback due to the absence of lock contention (see Section 7.3.1).

TABLE 3
Number of irrevocable transactions for the irrevocability with anticipation mechanism, broken down into those due to L1 replacements, L2 replacements, and conflicts.

Bench	# Irrevocable Xacts (Due to L1 / Due to L2 / Due to Conflicts)				
	1 th	2 th's	4 th's	8 th's	15 th's
Bayes	**89**(89/0/0)	**117**(92/0/25)	**134**(74/0/60)	**211**(62/0/149)	**226**(34/0/191)
Genome	**796**(796/0/0)	**940**(895/0/44)	**1130**(1042/0/88)	**1119**(922/0/197)	**1434**(1165/0/268)
Intruder	**24**(24/0/0)	**265**(33/0/232)	**1144**(24/0/1120)	**6397**(42/0/6355)	**16619**(89/0/16530)
Kmeans-high	**0**(0/0/0)	**147**(0/0/147)	**378**(0/0/378)	**1043**(0/0/1043)	**1986**(0/0/1986)
Kmeans-low	**0**(0/0/0)	**26**(0/0/26)	**165**(0/0/165)	**291**(0/0/291)	**567**(0/0/567)
Labyrinth	**76**(76/0/0)	**88**(64/0/24)	**102**(62/0/41)	**106**(29/0/77)	**117**(22/0/96)
SSCA2	**0**(0/0/0)	**33**(0/0/33)	**124**(0/0/124)	**283**(0/0/283)	**527**(0/0/527)
Vacation-high	**204**(204/0/0)	**341**(338/0/4)	**253**(218/0/36)	**336**(268/0/68)	**428**(302/0/126)
Vacation-low	**58**(58/0/0)	**104**(100/0/4)	**81**(64/0/17)	**103**(69/0/33)	**247**(181/3/63)
Yada	**705**(705/0/0)	**1344**(674/0/671)	**1700**(609/0/1091)	**1863**(581/0/1282)	**1883**(540/2/1342)

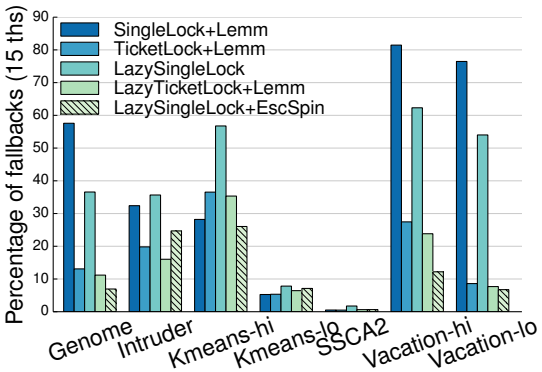

Fig. 8. Percentage of transactions that take the fallback path for 15 threads.

The last four benchmarks show a performance improvement when using the irrevocability mechanisms instead of the software fallbacks: Genome, Intruder, Kmeans-high and Vacation-high. Figure 10 shows a cycle breakdown of the execution time of the benchmarks for 15 threads. The results are normalized to the time of the irrevocability mechanism with overflow anticipation. We show the time spent outside transactions (NonTx), the useful transaction time (TxUseful), and the time spent in synchronization barriers (Barrier), which are used in a number of benchmarks to separate phases of the algorithm. These three parameters are common to fallback and irrevocability mechanisms. In addition, we break down the execution time of the specific mechanism. The irrevocability mechanisms show the time spent by threads waiting for the token to become irrevocable (IrreToken), the useful irrevocable time (IrreUseful), which is the execution time spent by irrevocable transactions, and the time that threads spent in the delayed commit (IrreCommit), i.e. the time of transactions waiting for the irrevocable transaction to end. The fallback variants have their counterparts. FbackWait is the same as IrreToken but waiting to acquire the lock. FbackUseful is the same as IrreUseful, and FbackSpin is the same as IrreCommit for the lazy single lock with escaped spin fallback. The latter accounts the time in the lemming loop for the lazy ticket lock with lemming effect avoidance fallback.

We can see in Figure 10 that irrevocability with and without anticipation can perform similarly one another, and better than the software irrevocability counterpart (LS), specifically for Intruder, Kmeans-high and Vacation-high with around a 14%, 25% and 13% of improvement respectively. This happens when the cause of transaction irrevocability is mainly based on conflicts, and capacity overflows are not dominant, as shown in Table 3. Vacation-high, though, shows a greater number of irrevocable transactions due to replacements, and the irrevocability with anticipation does perform better than the irrevocability without anticipation, but the difference is not significant. The following section explains why the software escaped spin fallback performs worse than its hardware counterpart due to lock contention (see the increased time in FbackSpin and FbackWait for these benchmarks). On the other hand, Genome shows a noticeable performance improvement of 28% with respect to the software escaped spin fallback by using irrevocability with overflow anticipation, and also compared with the irrevocability mechanism without anticipation. In this case, Genome's irrevocable transactions are mainly due to capacity overflows (see Table 3) as it has a numerous group of transactions with a large read set. In any case, our hardware irrevocability mechanisms do not incur any performance loss over the software alternatives.

7.3.1 *Lock Contention Effect*

We have seen that the irrevocability mechanism without anticipation can yield better results than its software fallback counterpart, even when they perform essentially the same steps to handle overflowed and conflicting transactions. However, the software fallback uses locks to ensure fallback synchronization whereas our irrevocability mechanism implements a token communication protocol. Lock contention has a detrimental effect on the performance of the fallback path that our communication protocol lacks.

Figure 11 shows the effect that lock contention has on the escaped spin lazy single lock fallback. There are three transactions in the system, in Cores 3, 12 and 15. The transaction in Core 3 is waiting (EscSpin) for the transaction in Core 15 to end the fallback. The transaction in Core 12 is waiting to acquire the lock (LockWait). Hence, the lock is shared (S) by the three cores and acquired by Core 15, so its value is 1. Note that the lock is mapped into Core 14's L2 cache directory.

Core 15 reaches the end of the fallback (FbackEnd) and releases the lock by setting it to 0 (ST Lock, 0). Then, Core 15's L1 cache requests ownership of the lock with a GETX request message ①. This message is sent to Core 14's L2 cache directory, which sends invalidation (INV) messages to the sharers of the lock ②. Cores 12 and 3 acknowledge the directory, and Core 15

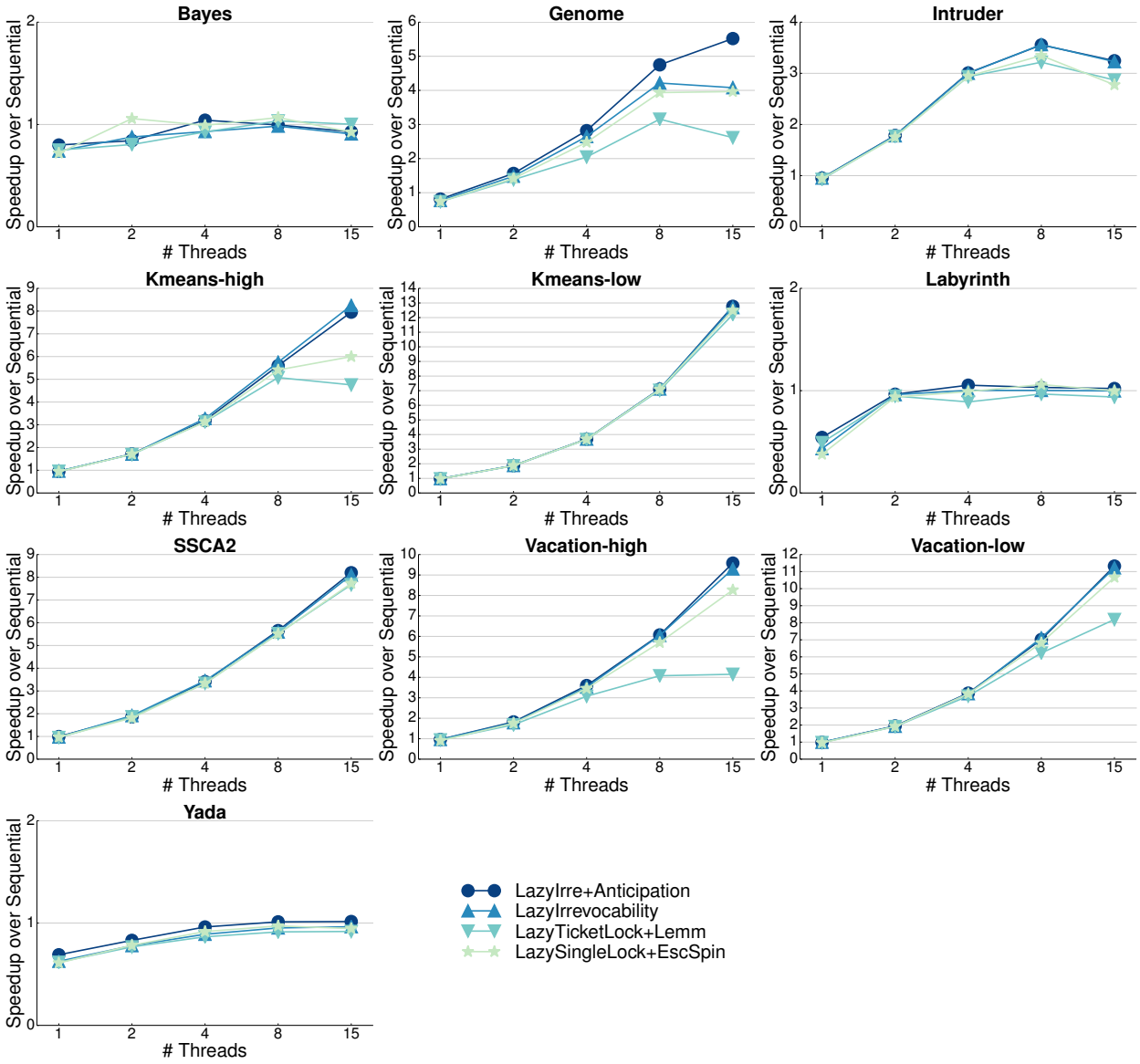

Fig. 9. Speedup over sequential of the irrevocability mechanisms with and without overflow anticipation, and the best software fallback variants. Lemm: lemming effect avoidance; EscSpin: Escaped spin.

releases the lock. In addition, Cores 12 and 3 request the lock from the directory ③, the former wants the ownership (GETX) and the latter just the value (GETS). The GETX message from Core 12 arrives before the GETS message from Core 3, and the directory forwards (FWD) this message to the owner of the lock ④. Core 15, which is the owner, sends the lock to Core 12 ⑤ which unblocks the directory and acquires the lock (Lock=1) to begin the fallback (FbackBegin). Finally, the GETS message from Core 3 reaches the directory ⑥, and it is forwarded to Core 12, the owner of the lock ⑦. Core 12 sends the data to Core 3 ⑦ and unblocks the directory, so all the instances of the block move to shared state (S). Core 3 receives the value of the acquired lock (1) and stays waiting in the loop until the fallback of Core 12 ends.

This lock contention effect hinders parallelism since it keeps transactions waiting when they should have committed. This fact sheds light on why a benchmark may exhibit more aborts using our irrevocability mechanisms than using the fallback. Figure 10 shows that Intruder aborts more transactions when using our irrevocability mechanisms. However, the execution time remains better than using fallbacks, as our proposals encourage parallelism. We can see that the time waiting for the token (IrreToken) and the time waiting to commit (IrreCommit) are reduced compared with the time waiting for the lock (FbackWait) and the time waiting to commit (FbackSpin). Our token communication protocol ensures that a transaction delaying the commit commits as soon as an *End Irrevocability* message is received. Unlike using locks, where the core has to request the lock to check its value, thus risking the lock contention effect. Such an effect is more probable as transactional contention increases.

Implication: Our hardware irrevocability mechanisms are an improvement over the software fallback path as they outperform the fallback in high contention and capacity overflow scenarios, and do not require the user to program a fallback path. If the main cause of irrevocability is capacity overflow, irrevocability with

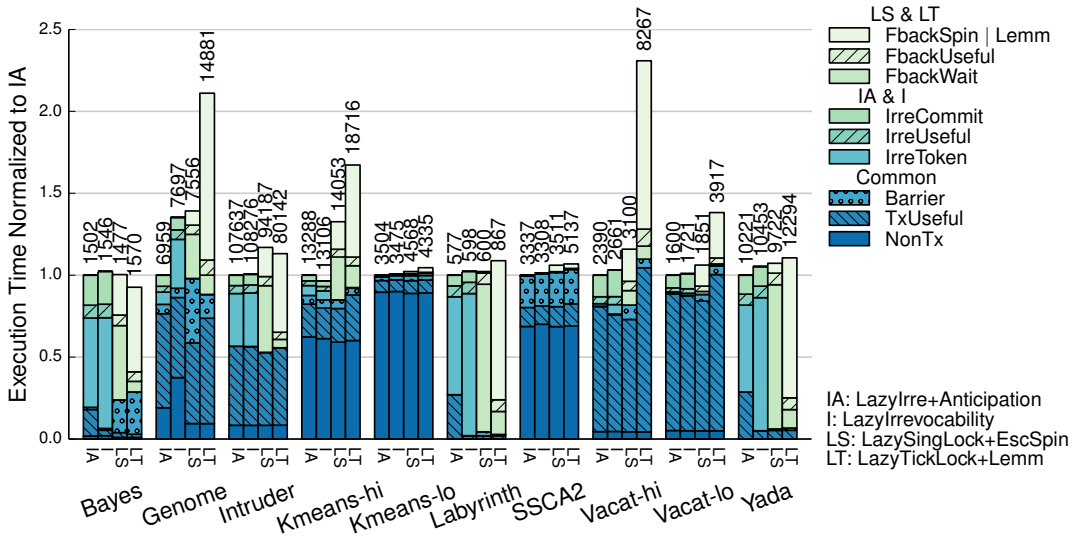

Fig. 10. Cycle breakdown of the execution time normalized to the system with irrevocability and overflow anticipation, for 15 threads. The number of aborts is shown on top of the bars.

TABLE 4
Network interconnect power requirements for 15 threads.

Bench	Power (W)				
	Static (Routers)	Dynamic (Routers + Links)			
		IA	I	LS	LT
Genome	4.02	1.2855	1.2596	1.2601	1.2438
SSCA2		1.3877	1.3849	1.3839	1.3776
Yada		1.2096	1.2090	1.2093	1.2092

overflow anticipation should be used. If conflicts and contention are predominant, the lazy irrevocability mechanism without anticipation suffices, as it does not require cache coherence protocol modifications and solves the lock contention effect.

7.4 Network Interconnect Power Requirements

Table 4 shows and excerpt of the system's network interconnect power requirements yielded by the Orion power model [26]. One significant benchmark from each of the groups identified in Section 7.3 is shown.

Lazy irrevocability with anticipation (IA) requires slightly more power than the lazy irrevocability without anticipation (I) and the software fallback alternatives (LS, LT). This is due to the increased parallelism. As far as the irrevocability communication protocol is concerned, it relies on broadcast to ask for the irrevocability token and to communicate the start and end of irrevocability. However, as we discuss in Section 4, such broadcast messages consist of small packets with source and destination IDs that do not imply a significant increase in network traffic and power. Moreover, a transaction becoming irrevocable should not be a frequent event.

Alternatively, the lock subscription of a software fallback may cause high network traffic. Although Figure 11 shows an scenario with only three transactions, it is highly probable that all cores are running transactions in applications that spend most time in transactions, such as the majority of those in Table 2. Therefore, the invalidations in ② would be more of a broadcast than a multicast, and such invalidations carry the address of the block to be invalidated, which is much larger than an irrevocability packet. More so, a core might have to request the block again, which comprises address and data fields.

In any case, by multiplying the power values of Table 4 by the execution time of the benchmarks (power delay product — PDP) we obtain values that make the power increase of the irrevocability mechanisms negligible compared with the software fallbacks.

7.5 Cache Size Sensitivity

As the cache is usually the holder of transactional information in BE-HTM systems, we have studied the effect of the cache size in our proposals. Each core of our baseline system has a private 32KB 4-way data L1 cache with a read and a write transactional bit per block. Figure 12 shows the speedup results obtained using a private 256KB 8-way data L1 cache for the three benchmarks which do not scale with the baseline system.

Overall, the three benchmarks still do not scale. Bayes shows a slight speedup improvement as the number of threads increases. The overflow anticipation seems to have no effect in performance as the number of overflowed transactions is reduced significantly due to the larger cache. However, there still are a few transactions that overflow the cache (specifically 5) and cause a slowdown with one thread. Although there are fewer overflowed transactions the overflow is detected later due to the larger cache. The same happens with Yada, but the number of overflowed transactions remains the same, and the slowdown is worse with one thread. The lazy irrevocability with anticipation is more effective in this case.

Labyrinth, though, does not exhibit overflowed transactions with the larger cache configuration, and it performs the same as sequential with one thread. However, it does not scale due to conflicts. Labyrinth is a transactional adaptation of Lee's algorithm [29]. The main transaction makes a copy of a grid to find the shortest path between two points, to finally add the path to the grid. The latest step of adding the path to the grid aborts all transactions that read the grid. There is an optimization to this transaction that consists of early-releasing the transactional read

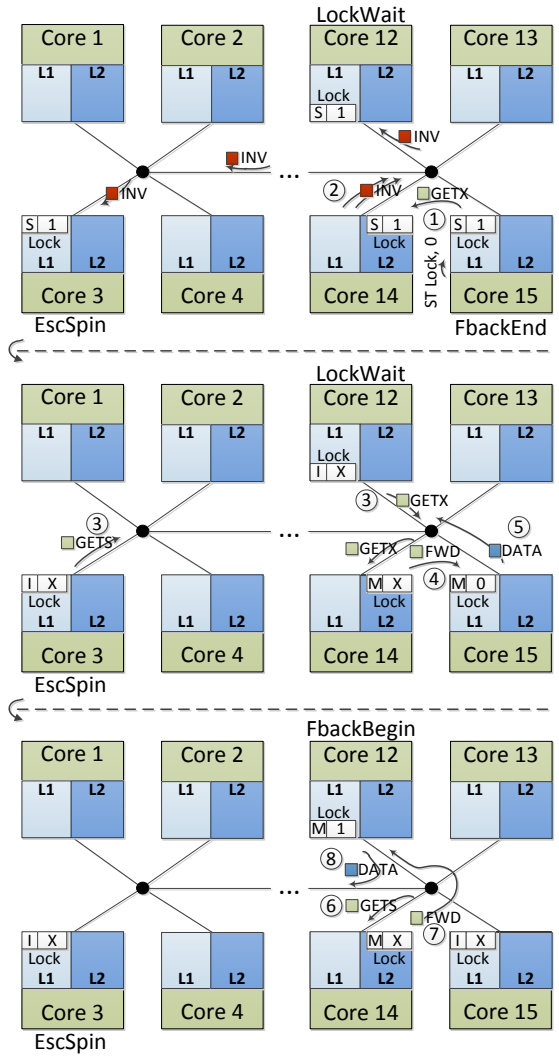

Fig. 11. Lock contention effect on escaped spin lazy single lock fallbacks.

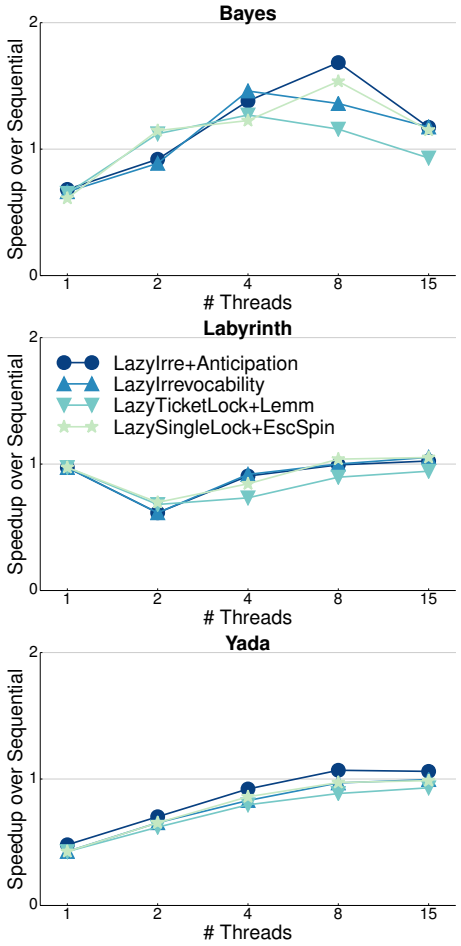

Fig. 12. Speedup with L1 caches of size 256KB and associativity 8.

set [30] just after making the copy of the grid [31]. However, when the path is added to the grid we can find that some point of the path collides with other path calculated by other transaction, and we must explicitly abort our transaction. Explicit aborts should not be used by neither irrevocable transactions nor software fallback paths as they cannot be rolled back, and early-release is still not provided by commercial BE-HTM.

8 Conclusions

Recently, a number of chip multiprocessors provide best-effort transactional extensions that need a software fallback path to overcome hardware limitations and to ensure forward progress of transactional applications. In this paper, we propose a new kind of hardware irrevocability that ensures forward progress and hides hardware limitations to users so that they do not have to provide a software fallback path, thus simplifying TM programmability.

Our lazy hardware irrevocability mechanism is based on the concept of lazy subscription used in software fallbacks, where the subscription to the lock is performed at the end of the transaction to encourage parallelism. Our mechanism modifies the begin and commit transactional constructs to ask for irrevocability and wait for an irrevocable transaction to end, respectively, without need for modification of the cache coherence protocol. We compare the lazy irrevocability mechanism with its software fallback counterpart, a lazy single lock with escaped spinning proposed by us. The hardware irrevocability outperforms the software counterpart in a number of benchmarks, mainly due to the use of our irrevocability communication protocol instead of a lock. The lock causes a contention effect which lacks our protocol, which also solves the unsafe execution problem associated with lazy subscription fallbacks.

In addition, we propose a lazy irrevocability mechanism with overflow anticipation that makes slight modifications to the cache coherence protocol to anticipate the replacement of transactional cache blocks. We ask for irrevocability before the block is replaced to avoid losing the work done so far by the transaction. This approach cannot be implemented in a software fallback path and performs better than the fallback when the main cause of irrevocability is capacity overflow.

We also propose a software fallback that uses a ticket lock with enhanced lemming effect avoidance that outdoes the current fallback proposals for those BE-HTM systems not providing escape actions.

The evaluation is carried out with the Simics/GEMS simulator along with the STAMP benchmark suite, and we obtain speedups from 14% to 28% over the fallback path approaches.

9 Acknowledgements

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] 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)*, pp. 289–300, 1993.

[2] 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)*, pp. 19:1–19:11, 2013.

[3] 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)*, pp. 127–136, 2012.

[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)*, pp. 25–36, Dec. 2012.

[5] A. Adir, C. Meissner, A. Nahir, R. R. Pratt, M. Schiffli, B. St. Onge, B. Thompto, E. Tsanko, A. Ziv, D. Goodman, D. Hershcovich, O. Hershkovitz, B. Hickerson, K. Holtz, W. Kadry, A. Koyfman, J. Ludden, and B. S. Onge, "Verification of Transactional Memory in POWER8," in *51st Ann. Design Automation Conf. (DAC'14)*, pp. 1–6, 2014.

[6] 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)*, pp. 285–296, june 2008.

[7] L. Dalessandro, F. Carouge, S. White, Y. Lev, M. Moir, M. L. Scott, and M. F. Spear, "Hybrid NOrec: A Case Study in the Effectiveness of Best Effort Hardware Transactional Memory," *ACM SIGPLAN Notices*, vol. 47, p. 39, jun 2012.

[8] 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.

[9] 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.

[10] M. J. Moravan, J. Bobba, K. E. Moore, L. Yen, M. D. Hill, B. Liblit, M. M. Swift, and D. A. Wood, "Supporting Nested Transactional Memory in logTM," in *12th Int'l. Conf. on Architectural Support for Programming Languages and Operating Systems (ASPLOS'06)*, pp. 359–370, 2006.

[11] 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.

[12] 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.

[13] 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)*, pp. 35–46, 2008.

[14] 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)*, pp. 225–236, 2013.

[15] M. Machado Pereira, M. Gaudet, J. Nelson Amaral, and G. Araujo, "Study of hardware transactional memory characteristics and serialization policies on Haswell," *Parallel Computing (available online)*, dec 2015.

[16] 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.

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

[18] Y. Afek, A. Levy, and A. Morrison, "Programming with hardware lock elision," *ACM SIGPLAN Notices*, vol. 48, pp. 295–296, aug 2013.

[19] 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)*, pp. 102–113, 2004.

[20] 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)*, ISCA '07, pp. 24–34, 2007.

[21] K. Moore, J. Bobba, M. Moravan, M. Hill, and D. Wood, "LogTM: Log-based transactional memory," in *12th Int'l. Symp. on High-Performance Computer Architecture (HPCA'06)*, pp. 254–265, 2006.

[22] A. Shriraman and S. Dwarkadas, "Refereeing Conflicts in Hardware Transactional Memory," in *23rd Int'l . Conf. on Supercomputing (ICS'09)*, pp. 136–146, 2009.

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

[24] J. M. Mellor-Crummey and M. L. Scott, "Algorithms for scalable synchronization on shared-memory multiprocessors," *ACM Trans. on Computer Systems*, vol. 9, pp. 21–65, Feb. 1991.

[25] N. Agarwal, T. Krishna, L.-S. Peh, and N. Jha, "GARNET: A detailed on-chip network model inside a full-system simulator," in *IEEE Int'l. Symp. on Performance Analysis of Systems and Software (ISPASS'09)*, pp. 33–42, April 2009.

[26] A. Kahng, B. Li, L.-S. Peh, and K. Samadi, "ORION 2.0: A fast and accurate noc power and area model for early-stage design space exploration," in *Design, Automation & Test in Europe (DATE'09)*, pp. 423–428, April 2009.

[27] A. R. Alameldeen and D. A. Wood, "Variability in architectural simulations of multi-threaded workloads," in *9th Int'l Symp. on High-Performance Computer Architecture (HPCA'03)*, pp. 7–18, 2003.

[28] 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)*, pp. 144–157, 2015.

[29] C. Y. Lee, "An Algorithm for Path Connections and Its Applications," *IRE Transactions on Electronic Computers*, vol. EC-10, pp. 346–365, sep 1961.

[30] M. Herlihy, V. Luchangco, M. Moir, and W. N. Scherer III, "Software Transactional Memory for Dynamic-sized Data Structures," in *22nd Ann. Symp. on Principles of Distributed Computing (PODC'03)*, pp. 92–101, 2003.

[31] I. Watson, C. Kirkham, and M. Lujan, "A Study of a Transactional Parallel Routing Algorithm," in *16th Int'l. Conf. on Parallel Architecture and Compilation Techniques (PACT '07)*, pp. 388–398, 2007.

Ricardo Quislant received the MSc degree in computer engineering from the University of Granada, and the PhD from the University of Malaga, Spain, in 2006 and 2012, respectively. Currently, he is working as a researcher in the Department of Computer Architecture at the University of Malaga. His main research interests include computer memory system and high-performance computing, with special regard to transactional memory.

Eladio Gutierrez received the MSc and PhD degrees in telecommunication engineering both from the University of Malaga, Spain, in 1995 and 2001 respectively. Since 2003 he has been an associate professor in the Department of Computer Architecture at the University of Malaga. His research interests include parallel architectures and algorithms, automatic parallelization, engineering education and graphics processing units.

Emilio L. Zapata received the MSc degree in physics from the University of Granada and the PhD degree in physics from the University of Santiago de Compostela, Spain, in 1973 and 1983, respectively. Since 1991, he has been a full professor in the Department of Computer Architecture at the University of Malaga. His main research interests include numerical and audiovisual applications, high-performance architectures, and compilation techniques for parallel computers.

Oscar Plata received the MSc and PhD degrees in physics from the University of Santiago de Compostela, Spain, in 1985 and 1989, respectively. Currently, he is a full professor in the Department of Computer Architecture at the University of Malaga, Spain. His research interests include high-performance computing and compiler techniques for parallel architectures.