Enhancing Scalability in Best-Effort Hardware Transactional Memory Systems

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

Dept. Computer Architecture, University of Malaga, Spain

Abstract

Current industry proposals for hardware transactional memory focus on best-effort solutions where hardware limits are imposed on transactions. These designs can efficiently execute transactions but they may abort due to different hardware and operating system limitations, with a significant impact on performance. For instance, transactions cannot survive capacity overflows, exceptions, interrupts, operating system events like page faults, migrations, context switches, and so on. To deal with these events, best-effort hardware transactional memory systems usually provide a software fallback path to execute a non-transactional version of the code.

In this paper we propose hardware implementation solutions to make transactions survive some of such limitations, in order to improve the performance and scalability of transactional applications in best-effort systems. First, we propose a hardware irrevocability mechanism that works either when hardware capacity overflows occur or in high contention scenarios. It anticipates capacity overflows and reduces the abort count. This mechanism avoids writing a fallback code, simplifying the programming of the transactional application. Second, we propose a two-phase abort mechanism to support both the execution of privileged mode code inside transactions and the interaction of this code with the irrevocability mechanism. Third, we propose a privileged-aware cache replacement policy to reduce capacity overflows in the presence of privileged code.

We evaluate our proposals with all the benchmarks of the STAMP transactional suite and carry out a performance comparison with a fallback-based hardware transactional memory system, after considering different fallback codes, showing significant performance benefits for requester-wins and requester-stalls conflict resolution policies.

Keywords: Hardware transactional memory, Best-effort, Irrevocability, Privileged mode code, Cache replacement policy, Requester-wins, Requester-stalls

1. Introduction

Transactional Memory (TM) [1] was first presented in 1993 [2] as a non-blocking synchronization mechanism for shared memory chip multiprocessors (CMPs). TM provides the programmer with the *transaction* construct that executes the code enclosed by it atomically and in isolation. The underlying system is in charge of maintaining those transactional properties with dedicated hardware and changes to the coherence protocol (hardware TM, or HTM).

Since Herlihy's seminal approach there have been several proposals exploring different designs of an HTM system [3, 4, 5, 6], and many others that have gained insight into the virtualization of the transactional hardware [3, 7, 8, 9] to support transactions of any size and duration.

However, it is not until recently that some processor manufacturers have included HTM support in their commercial off-the-shelf CMPs [10, 11, 12, 13]. Current industry proposals focus on best-effort solutions (BE-HTM) where hardware limits are imposed on transactions. These designs can efficiently execute transactions but they may abort due to different hardware and operating system (OS) limitations, with a significant impact on performance. For instance, transactions cannot survive capacity overflows, exceptions, interrupts, OS events like page faults, migrations, context

Email addresses: quislant@uma.es (Ricardo Quislant), eladio@uma.es (Eladio Gutierrez), zapata@uma.es (Emilio L. Zapata), oplata@uma.es (Oscar Plata)

switches, and so on. To deal with these events, 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 three hardware mechanisms to help transactions running in BE-HTM systems survive some hardware and OS limitations and to enhance their performance. Our proposals are the following:

- A hardware irrevocability mechanism that operates whenever a transaction aborts a given number of times, either because of conflicts with other concurrent transactions or due to hardware capacity overflows (a BE-HTM system relies on caches to store transactional information and a transactional cache block replacement implies losing such information). Irrevocability [7, 14] is a transactional execution mode that ensures transaction forward progress. We show how this mechanism can be easily implemented by tailoring the coherence protocol, and discuss its benefits over a software fallback path in terms of performance and ease of use. We study and evaluate irrevocability in the context of eager-eager requester-wins/stalls BE-HTM systems. Our irrevocability mechanism is able to anticipate a cache block replacement and to stall other concurrent transactions instead of aborting them.

- We propose a two-phase abort mechanism to support the execution of privileged mode code within transactions, such as that of exceptions or interrupts. Under this mechanism, the release isolation part of the abort (first phase) is separated from the part of restoring the transaction's checkpoint (second phase). The first phase is executed in case the privileged mode code replaces a transactional block, although the irrevocability mechanism is first tried to prevent the transaction from aborting. The second phase is executed when the privileged mode code execution comes to an end. We rely on a transaction-aware OS to enforce certain invariants that ensures correctness.

- Since privileged mode code may interfere with ongoing transactions by replacing transactional cache blocks that might abort them, we propose a privileged-aware (PA) cache replacement policy that reduces the number of privileged-mode-accessible blocks, maximizing the number of blocks devoted to transactions. It proves to work well for the benchmarks that exhibit more aborts due to privileged mode code cache replacements.

We evaluate our proposals in a simulation environment with all the benchmarks of the STAMP transactional suite and carry out a performance comparison with a fallback-based HTM system, after considering different fallback codes, showing significant performance benefits for certain benchmarks.

The remainder of the paper is organized as follows. Next section describes the baseline architecture design of the BE-HTM system that we have used to implement our proposals. Section 3 presents the details of the hardware irrevocability mechanism and its implementation. In Section 4 we discuss how privileged mode code can be allowed within transactions. Section 5 describes the privileged-aware replacement policy to decrease the abort count due to cache block replacements when running privileged mode code. In Section 6 we discuss the experimental results obtained when evaluating our proposals using the Simics/GEMS simulator and the STAMP benchmark suite. Finally, Section 7 reviews the related work and Section 8 draws the conclusions.

2. Baseline BE-HTM Architecture

The solutions proposed in this work to reduce the impact on performance of the limitations inherent to a BE-HTM system were designed on a baseline architecture similar to that provided by the state-of-the-art processor manufacturers, such as Intel Haswell [13] and IBM Power8 [15].

Figure 1 shows the baseline architecture. 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 [16] and implements an eager conflict detection policy. As to the conflict resolution policy, we consider two of them: (i) 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; (ii) Requester-stalls, where the requesting transaction is nacked and stalled until the conflict vanishes. The stalled transaction aborts if it can cause a deadlock cycle because of other transactions stalling on it, similar to LogTM [6].

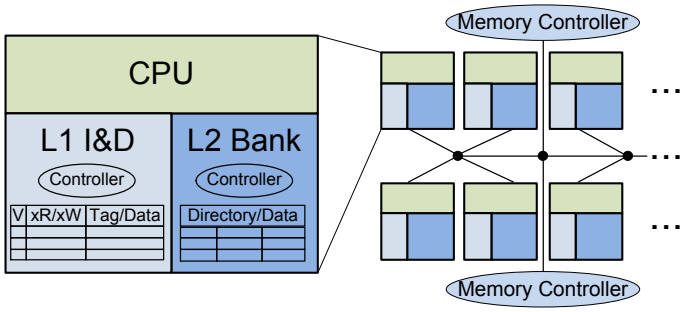

Figure 1: Baseline architecture of the BE-HTM system.

Causes of transaction aborts due to hardware/OS limitations are: a replacement of a transactional block in an L1 cache; an eviction of an L2 cache block that is marked as transactional in the L1 cache, due to the inclusion property of the cache coherence protocol; a change from user mode to privileged mode which can be caused by a hardware interrupt, a migration, a context change, a paging event, a system call,... mainly, OS events. To ensure forward progress of concurrent transactions, the user can provide a fallback code that executes whenever a transaction aborts a given number of times. We propose an alternative hardware irrevocability mechanism in Section 3 that entails certain benefits over the user fallback code.

The cache coherence protocol is a MESI directory protocol that holds a full bit vector of sharers. An L1 cache miss can cost up to four hops, although the last one is not in the critical path: a GET message from the requester to the directory, a FWD (forward) message from the directory to the L1 cache that owns the block, a message with the data between L1 caches (from forwarded to requester), and a message from the requester to unblock the directory, which stays in an intermediate state to forbid the access to the block while the miss is served [17]. The baseline protocol needs some 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 jeopardizes transaction isolation; therefore, transactions must be aborted on these type of evictions. Beside, L2 cache block replacements may abort a transaction because of the inclusion property.

- *L2 cache serves data of aborted transactions*: If a transaction requests data of a transaction that was aborted, 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 by comparing the ID of the GET message and the ID of the block in the directory. Then, the directory assumes that the transaction was aborted and now 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. Then, the L1 cache that owned the block receives a FWD message for a block that is no longer present due to an abort. In this case, the L1 cache informs the L2 cache and the L2 cache serves the data. This implies a five-hop access to the cache, including the final unblock from the requester to the L2 cache.

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 [18]. Nested transactions [19] are flattened and treated as if they were part of the outer one. Escape actions [19] 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.

Next sections describe the implementation solutions proposed in this paper to enhance transaction performance and scalability that are built atop the BE-HTM system presented here.

```
void beginTransaction() {
  if (_xbegin() == _XBEGIN_STARTED) {
    if (globalLock is free)        // Add lock to read-set
      return;                      // Execute transactionally
    _xabort(0xff);                 // 0xff means the lock's not free
  }
  // Come here following the transactional abort
  acquireLock(globalLock);
}

void endTransaction() {
  // If lock is free, assume that the lock's elided
  if (globalLock is free)
    _xend();                       // Commit
  else releaseLock(globalLock);
}
```

Figure 2: Fallback code example (from Intel).

3. 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 path is taken by way of a runtime system decision [12] or after a given number of transaction retries that can be set by the programmer [11, 13]. In this section, we describe a hardware irrevocability mechanism similar to that in [20], as an alternative to the software fallback path available in commercial systems.

Figure 2 shows the simplest code that Intel suggests as fallback code in its optimization manual [21]. If the transaction is successfully started, a global lock is checked to add it to the read-set of the transaction. If the lock is free, the transaction is started. If not, the transaction is aborted, since another transaction took the fallback path. In case the transaction is retried, the lock is acquired. Intel's manual suggests using a more detailed fallback code that distinguishes between different causes for transactional aborts. Certain abort causes would require retrying the transaction instead of taking the fallback path, for instance.

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

- 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: Figure 3 shows an execution scenario where a hardware irrevocability mechanism performs better than a software fallback code like the one in Figure 2. The fallback path version aborts transactional execution and retries the transaction acquiring the fallback lock. The other transactions running in the system are aborted as well, since they read the lock at the beginning[1]. Execution is rebooted and serialized. However, the hardware irrevocability mechanism does not discard that computation. 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.

The scenario in Figure 3 is optimistic. It considers no contention between the irrevocable transaction and the stalled ones, which would cause aborting either one or all of them, and the corresponding retries. Additionally, the fallback code causes a chain reaction, also called as *Lemming effect* [22], by which all transactions take the fallback even if they do not have to. Nonetheless, the figure depicts the potential of having a hardware irrevocability mechanism to deal with hardware capacity overflows and forward progress in best-effort HTM systems. Section 6 explores different fallback policies and quantifies the aforementioned benefits.

[1]The transaction that wrote the lock causes these aborts by means of strong isolation, and correctness is ensured as locks and transactions are not allowed simultaneously

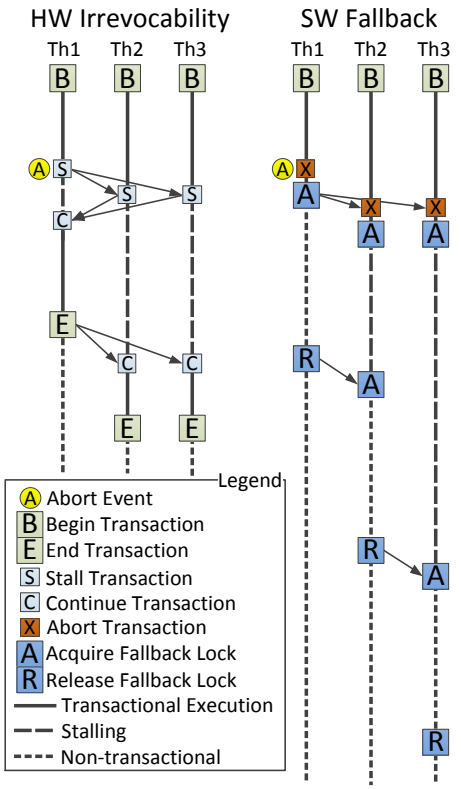

Figure 3: Execution scenario of hardware irrevocability vs. software fallback.

3.1. Implementation

The implementation of the hardware irrevocability mechanism proposed in this paper is very similar to that of [20], but in this case, the cache coherence protocol has to be modified to anticipate to capacity overflow events like L1 cache replacements and L2 cache evictions. [2]

The mechanism is token-based and only the core that owns the token can run an irrevocable transaction. Each core has a flag that indicates whether there is one irrevocable transaction running in the system or not (the I bit). Another flag in the core signals if the irrevocable transaction running in the system belongs to this core or another one, i.e. the core owns the token (the T bit). Along with the pair of bits (I,T), each core holds a counter (C) for the number of aborts allowed before the core asks for irrevocability.

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

- *(I,T) = (0,0)*: There is no irrevocable transaction running in the system and the token is not owned. Then, 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 stalls until the owner ends its transaction. If the token is received, its T bit is set to 1 and the controller broadcasts an irrevocability request message to the other cores so that they set the I bit to 1. The requester can safely continue its transaction in irrevocable mode, (I,T) = (1,1), after the request message is acknowledged by the other L1 cache controllers.

[2]The mechanism in [20] leaves the cache coherence protocol untouched since irrevocability is triggered by signature saturation, and cache evictions are not a problem as the conflict detection of the underlying HTM system is virtualized (based on Bloom filters).

Table 1: L1 cache coherence protocol modifications for irrevocability (highlighted in grey).

State	Events					
	L1 Replace ¬(xR∨xW) ∨ (I,T)=(1,1)	L1 Replace (xR∨xW)∧(C>1)	L1 Replace (xR∨xW)∧(C≤1)	L2 Replace ¬(xR∨xW) ∨ (I,T)=(1,1)	L2 Replace (xR∨xW) (I,T)=(1,0)∨(C>1)	L2 Replace (xR∨xW) (I,T)=(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: recycle mandatory queue
Zz: recycle request queue

- *(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 so the transaction of this core stalls. This value for (I,T) should not be found when a controller is going to ask for irrevocability, since the transaction should be stalled and it does not cause capacity overflows. However, if the stalled transaction aborts because of a conflict with the irrevocable and reaches the limit of aborts, the controller would find (I,T) = (1,0). Another possibility is that privileged code evicts a transactional block from the cache (see Section 4).

As shown in Figure 3, the hardware irrevocability mechanism is able to anticipate aborts due to capacity overflows, thus preserving the work done so far by the transaction. Such overflows or transactional data cache replacements would cause an abort in a conventional BE-HTM system, but the hardware irrevocability mechanism counts the abort and then it checks the counter. If the counter reaches the limit, the controller has to enter the irrevocability mode. The abort is then elided[3] and the transaction is stalled just before the replacement takes place, until the transaction is granted irrevocability. Then, the replacement can be performed safely.

We have modified the L1 cache coherence protocol to implement the anticipation to a block replacement. Table 1 shows the modifications made to the protocol highlighted in grey. L1 cache replacements are left untouched whenever either the block to be replaced is not transactional, ¬(xR∨xW), or the core is in irrevocable mode and owns the token, (I,T) = (1,1). Note the silent replacement when the block is in S state. However, if the block is transactional, xR∨xW, it depends on the controller counter (C) what to do. The counter is set to the number of transaction retries at the beginning of each transaction. If there is a transactional block replacement and the counter is greater than 1, the transaction aborts and the counter is decremented. Conversely, if the counter is lower than or equal to 1, the mechanism of irrevocability is triggered, and the mandatory queue recycled [4], i.e. the event is left to be executed later on. Should the core manage to get irrevocable, the controller's (I,T) bits will be set both to one when the L1 replacement event is re-executed. If not, the core will be stalled by continuously recycling the eviction event.

In case of L1 cache transactional block replacements due to L2 cache evictions (L2 Replace events in Table 1), if the core is running an irrevocable transaction, (I,T) = (1,1), the event is treated as a normal L2 cache replacement, just acknowledging the replacement and sending the data block if it was modified. However, if other transaction is running irrevocably, (I,T) = (1,0), the transaction in this core must be aborted to complete the eviction, thus avoiding a possible deadlock with the irrevocable one, which can be waiting for the L2 cache eviction to continue. Unlike L1 cache replacements, the transaction cannot be stalled waiting for irrevocability on an L2 cache replacement event. Therefore, the only situation in which a transaction asks for irrevocability on an L2 Replace event is when the counter is lower than or equal to 1 and there is no other irrevocable transaction in the system, (I,T) = (0,-).

There is a corner case in the protocol when we introduce the changes to implement the irrevocability mechanism. The L2 cache controller recycles a message in a queue whenever it cannot be served because the information needed

[3]However, aborts due to transactional data conflicts are not anticipated as it can cause deadlock if the conflict is with an irrevocable transaction.

[4]The cache controller comprises queues where coherence messages are buffered until they are served by the controller [17]. 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, like in [23].

to do so is in a block that is in a transient state. The transient state means that there is an ongoing memory transaction on that block, and the controller must wait for it to end before executing another memory transaction on the block. The problem is as follows. If a transaction gets irrevocable because of a transactional block replacement, all other cores stall their transactions and they stop requesting blocks. However, there may be a recycled memory transaction from a stalled requester in a L2 cache controller queue that can be re-executed even after its requester stalled due to irrevocability. If such a memory transaction requests the block whose replacement caused the irrevocability, it will no longer be in the transactional data set of the irrevocable transaction and the request will not be nacked as it should be. So the solution is nacking all forwarded requests during irrevocability.

3.2. Hardware Complexity

The token-based irrevocability mechanism requires one counter and two bits (I and T) per cache controller. The microcontroller logic has to be modified to implement the irrevocability operation procedures explained in the last section. Such procedures have to deal with new message types defined by the irrevocability protocol: a message to request the token, a message to send the token, a message to request irrevocability, a message to indicate the end of irrevocability and an acknowledge message. These messages comprise the type, and the source and destination core IDs, which is considerably less information than that of a coherence protocol message comprising the address and the value of a memory location. The messages go from L1 to L1 cache controller.

Regarding the implementation of the transactional block eviction anticipation, we have to modify the cache coherence protocol of the L1 cache and leave the L2 cache controllers untouched. The cache coherence protocol is a critical element of a multicore processor, difficult to specify and verify [23]. However, we only modify a small part of it which entails the cache block replacement events and their transitions. It only comprises the microcontroller logic to check if the block is transactional and if the counter get to 1. Plus the calls to the procedures of abort and irrevocability.

4. Allowing Privileged Mode Code Within Transactions

Best-effort HTM systems abort transactions whenever certain events occur amid them. Such events are exceptions, interrupts, page faults, TLB shootdowns, migrations, context switches, and others, that are usually managed by code executed in privileged mode. From the point of view of the HTM system, just a change from user mode to privileged mode is detected and what that code is going to do is unknown so the transaction aborts[5].

The occurrence of some of the aforementioned events can be minimized inside transactions by performing certain tasks in the initialization of the workload: One can traverse the working-set of the application to prevent page faults and TLB shootdowns; thread's affinity can be set to prevent migration, and so on. However, either keeping an interrupt from replacing a transactional cache block, or having the OS de-schedule a transactional application to schedule another transactional application are events more difficult to prevent.

We assume a transaction-aware OS [19, 24] where only one transactional application is allowed at a time. This hard invariant simplifies the way the BE-HTM system deals with privileged code. Thus, we allow mode changes inside transactions, unless they are migrations or page faults. It is the OS which would be in charge of aborting the transaction in such cases. Hardware and timer interrupts are allowed. For instance, the timer can trigger the OS scheduler and it might decide not to change the current process. In this context, the problem comes with the use of the L1 cache by the privileged code. If no transactional block is evicted, the privileged code goes unnoticed. Otherwise, the transaction should be aborted (and the fallback path taken) or irrevocability should be requested to anticipate the abort.

Nonetheless, aborting a transaction in the middle of privileged code execution would change the program counter to point to user mode transactional code, so we propose a *two-phase abort* where the reset of transactional state and the restoration of the transaction's checkpoint are performed separately:

[5]For example, a page fault might invalidate a transactional page from memory and then load it into another physical region of memory. Or a context switch might de-schedule a transactional thread by another transactional thread. Or an interrupt might replace a transactional cache block.

- *Release isolation*: Whenever a transactional block is going to be replaced by privileged code, the HTM system executes the first phase of the phase-wise abort and releases isolation of the transaction by flush-clearing all xR bits of the L1 cache and every pair of V and xW bits. This way, the block can be replaced and the privileged code can continue. Also, an *AbortedTransaction* bit is set.

- *Restore transaction's checkpoint*: Once the HTM system encounters the first transactional load/store of user code when the privileged code finished, the *AbortedTransaction* bit is checked, and if it is set, the checkpoint taken at the beginning of the transaction is restored. The two-phase abort is then completed. It should be noted that the transaction-aware OS is in charge of correctness and would perform any action to maintain it. So the HTM system should provide some kind of instructions like those in the Power ISA [10] to give architectural support to the OS.

As noted in Section 7, the one-transactional-application-at-a-time invariant can be relaxed by exposing both phases of the abort to the ISA. This way, the OS can decide to abort a transaction and schedule another transactional application instead. Let `tmrelease` be the instruction to release isolation of the ongoing transaction, and let `tmrestart` be the instruction to restore the transaction's checkpoint. Then, whenever there is a context switch from a transactional application to another transactional application the OS executes the `tmrelease` instruction, thus freeing the L1 cache for the use of the new transactional application. Subsequently, the transactional checkpoint is saved and the context of the scheduled application restored. The OS must hold information about the evicted application so that it executes the `tmrestart` instruction when re-scheduling it. The OS can know about an application being transactional if the compiler set a transactional bit in the executable header. If not, the two-phase abort would be executed whether or not the application to schedule is transactional.

4.1. Irrevocability and privileged mode code

The hardware irrevocability mechanism described in Section 3 needs certain modifications to interact with privileged mode code that should be noted:

- The mandatory queue (that holds the load/store requests from the CPU to the L1 cache) recycles user mode requests only. This way, if the controller's (I,T) bits are set to (1,0), the privileged mode requests can be served.

- Table 1 shows the modifications made to the cache coherence protocol to support irrevocability. Most of these changes remain the same, but now, a privileged mode load/store may trigger the events in the table, so the Abort actions must be changed by the first phase of the two-phase abort mechanism described above.

- Furthermore, if privileged mode code causes a replacement of an L1 cache transactional block and the counter reached the limit (L1 Replace $(xR \vee xW) \wedge (C \leq 1)$), the transaction asks for irrevocability. But if irrevocability is not granted, the transaction is aborted in order for the privileged code to continue (instead of being stalled as discussed in Section 3).

L2 cache replacements are treated the same as shown in Table 1. Actually, L1 cache replacements would be treated as L2 cache evictions when they are caused by privileged mode code.

5. Privileged-Aware Cache Replacement Policy

As mentioned in the previous section, if code in privileged mode is allowed within transactions, it might cause the replacement of a transactional block and, consequently, the transaction should be aborted because isolation is compromised.

In this section we propose a Privileged-Aware (PA) cache replacement policy that favours transactional cache blocks to the detriment of privileged mode code. The baseline system uses a pseudo LRU replacement policy that we opt to leave untouched. Instead, we decide when to use the *touch* procedure of the policy, whether touch the block or simply elide it. Thus, the cache controller has to check certain flags before touching a cache block. If the core is executing a transaction, the load/store request is in privileged mode and refers to the data L1 cache (the instruction L1 cache is not involved), the touch is elided.

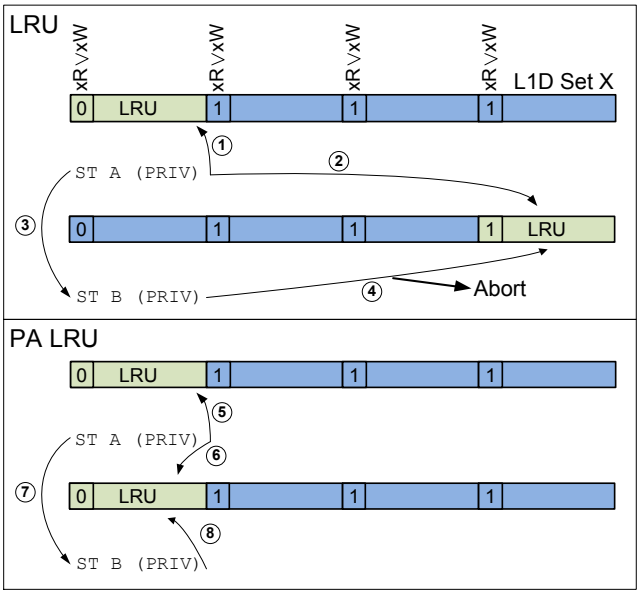

Figure 4: Normal LRU replacement policy versus Privileged-Aware (PA).

Figure 4 shows how the PA replacement policy works compared to a normal replacement policy. We have a set of the L1 data cache that already has three of its four blocks marked as transactional and the least recently used block is the first one, which is not transactional. Then, ① a privileged mode store is mapped to that set and stored in the LRU block; ②, the LRU block changes to the last block in the set; ③, another privileged mode store is mapped to the same set; ④, the store replaces a transactional block and the transaction is aborted. However, with the PA version, ⑤ the store is placed in the LRU block; ⑥, the touch is elided and the block stays as LRU; ⑦, the next store is stored in the same block ⑧ thus preventing the transaction from aborting.

The PA-replacement policy turns the cache into a direct mapped cache for privileged mode code, thus worsening its performance, but can reduce the number of transaction aborts.

As regards the implementation hardware complexity of our PA pseudo LRU policy, it only requires an AND gate whose inputs are the bit indicating that the processor is in a transactional region and the privilege mode bit of the memory request message (or the program status word – PSW). The output feeds the LRU logic that touches the least recently used bits. If the output is zero, the least recently used bits are updated as normally. If not, they are left the same. So, it does not affect the critical path of a cache access.

6. Experimental Evaluation

In this section we compare our proposals against the baseline BE-HTM with different software fallback path codes. Other systems like TCC [5] or OneTM [7] are not evaluated as they uses irrevocability in a different context.

6.1. Methodology

We have used Simics [25], a full system simulator, and the Wisconsin GEMS [26] 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, to include the best-effort HTM system outlined in Section 2, 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 and power modelling we have used Garnet [27], a detailed interconnect model that integrates Orion [28], a network power model.

Ruby adds pseudorandom delays to memory accesses to deal with variability in simulation experiments. Then, multiple runs of each experiment were carried out to obtain confident error bars [29].

Benchmarks

The benchmarks used for the evaluation of our proposals are all the codes of the Stanford's STAMP suite [30]. The codes were adapted to avoid unnecessary aborts:

- We used the pset_bind system call from Solaris OS to bind each thread of the application to a logical processor set. The psrset system administration command was used to create the logical processor sets in the simulated machine so that one real processor is assigned to one logical processor set. Binding threads to processors keeps the operating system from migrating such threads. However, the maximum number of threads is limited to 15 as one processor is assigned to the operating system.
- The benchmark memory footprint is traversed before starting simulation to avoid page faults inside transactions. Dynamic library functions used inside transactions were also called before entering transactions to let the linker fill in the Procedure Linkage Table (PLT). This way, we disable OS's dynamic linker interferences within transactions since the PLT has all the information about dynamic linking procedures.

Table 2 shows the parameters and characteristics of the benchmarks used in the evaluation. We have the number of transactions that successfully commits (# Xact), the percentage of time running transactions (% Time in Xact), and the average RS/WS (read set/write set) cardinality for each transaction, in cache blocks. Each transaction is identified by its ID, starting from 0 and numbered in order of appearance in the code. The count for each transaction is also shown (XactID(*avg*|RS|/ *avg*|WS|)Count). The values belong to the workloads running with 15 threads.

6.2. Software Fallback Implementation Results

In this section we explore different fallback code alternatives in order to choose the best to compare with the hardware irrevocability scheme described in Section 3.

Figure 2 shows the simplest version of a fallback code that includes neither a variable to specify the number of transaction retries nor the Lemming effect[6] avoidance. However, we have evaluated the fallback code shown in Figure 5 that includes the aforementioned optimizations. 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 at the point of the code where we want to resume

[6]If one transaction takes the fallback path, the others are aborted until there are no retries left, and wait for the fallback lock to be released, i.e. a complete serialization of the ongoing transactions is carried out.

Table 2: Workloads: Input parameters and transactional characteristics.

Bench	**Input**	**# Xact**	**% Time in Xact**	**XactID(*avg*|RS| / *avg*|WS|)Count**
Bayes	-v32 -r1024 -n2 -p20 -i2 -e2 -s1	733	95%	**0**(2/1)15 **1**(21/3)32 **2**(6/4)96 **3**(23/12)81 **4**(45/15)28 **5**(2/1)28 **6**(41/14)2 **7**(2/1)2 **8**(64/24)38 **9**(39/14)38 **10**(4/1)81 **11**(556/326)81 **12**(43/21)81 **13**(64/33)81 **14**(18/3)49
Genome	-g512 -s32 -n32768	19722	79%	**0**(109/2)2744 **1**(4/1)481 **2**(10/4)14911 **3**(6/4)481 **4**(5/2)1105
Intruder	-a10 -l16 -n4096 -s1	54933	82%	**0**(4/1)18321 **1**(24/7)18306 **2**(2/1)18306
Kmeans	-m15 -n15 -t0.05 -i random-n2048-d16-c16	8235	16%	**0**(8/2)6144 **1**(1/1)2046 **2**(2/1)45
Labyrinth	-i random-x32-y32-z3-n96	222	100%	**0**(3/1)111 **1**(321/218)96 **2**(11/4)15
SSCA2	-s14 -i1.0 -u1.0 -l9 -p9	93729	11%	**0**(1/1)15 **1**(1/1)15 **2**(3/2)93699
Vacation	-n4 -q60 -u90 -r16384 -t4096	4095	91%	**0**(66/11)3678 **1**(32/6)210 **2**(44/4)207
Yada	-a20 -i633.2	5406	99%	**0**(9/2)1329 **1**(1/0)1314 **2**(348/223)916 **3**(1/1)916 **4**(9/1)916 **5**(1/1)15

```
1  localRetries = 0;

2  void beginTransaction(&localRetries) {
3    TAKE_XACT_CHECKPOINT;                  // Return point on abort
4    localRetries++;                        // Increment xact retries

5    if (localRetries > RETRY_LIMIT) {      // Fallback?
6      while(!lockAcquire(globalLock));     // Take lock
7    } else {                               // Execute transactionally
8      while(lock != 0);                    // Avoid Lemming effect
9      BEGIN_XACT;
10     if(lock != 0)                        // Lock subscription
11       ABORT_XACT;
12   }
13 }

14 void endTransaction(localRetries) {
15   //If not retry limit, assume lock's elided
16   if (localRetries <= RETRY_LIMIT)
17     COMMIT_XACT;                         // Commit
18   else lockRelease(globalLock);
19 }
```

Figure 5: Fallback code with number of transaction retries and Lemming effect avoidance.

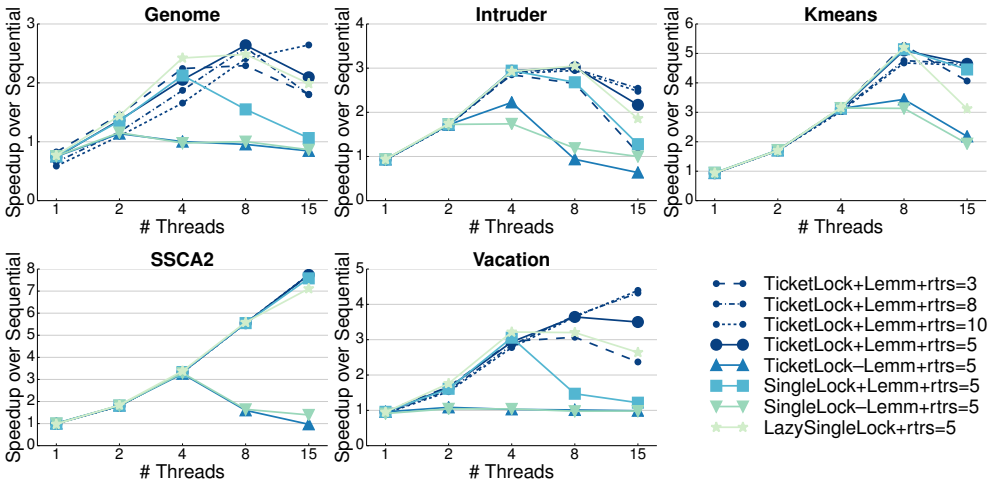

Figure 6: Speedup over the sequential application for different fallbacks and parameters (Lemm: lemming effect avoidance, rtrs: number of retries) with the requester-wins conflict resolution policy.

the transaction after abort, but it does not start transactional bookkeeping; (ii) `BEGIN_XACT` begins the transaction by isolating and keeping track of every memory access issued from that point on. Then, we can have non-transactional code between the two primitives to check whether we have to take the fallback path or not.

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 simple 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), which is also called lock subscription. 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) is similar to that in Figure 2 but takes into account the number of transaction retries.

Figure 6 depicts the speedup results obtained for those STAMP benchmarks that scale to some extent. We have

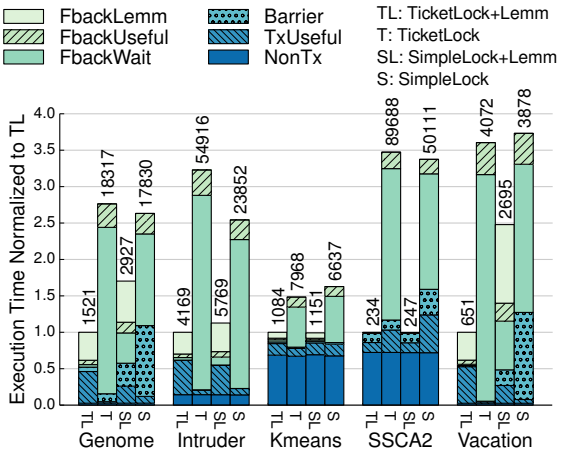

Figure 7: Cycle breakdown of the different fallback alternatives with 5 retries, 8 threads and the requester-wins conflict resolution policy. The number of transactions that take the fallback path is shown over the bars.

used the HTM version with requester-wins and privileged mode support (next sections discusses the other configurations). The fallback code used is that of Figure 5, with and without Lemming avoidance (±Lemm), and we have added another characteristic to it, which is a two-variable (ticket and turn) *ticket lock* [31] instead of the single spin lock. The lazy-subscription fallback lock proposed in [32] is also shown, where the lock subscription is moved from after the beginning of the transaction (line 10) to before the commit (between line 16 an 17)[7]. Also, the retry limit has been set to 5, which is a frequently used value [11, 13]. 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. Therefore, we can enhance the Lemming effect avoidance loop by waiting not only when a thread has acquired the lock (`lock!=0`) but also when there are more threads waiting to enter the critical section (the ticket is greater than the turn). 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 2), 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.

Figure 7 shows the execution time normalized to that of the ticket lock version with Lemming effect avoidance (lower is better) for 8 threads and 5 retries. The time is broken down into non-transactional cycles (NonTx), transactional cycles (TxUseful), cycles that threads spend waiting at barriers (Barrier), cycles spent waiting to enter the fallback path (FbackWait), cycles waiting at the Lemming avoidance loop (FbackLemm) and fallback execution cycles (FbackUseful). The number of transactions that take the fallback path is shown on top of the bars. We can see the effect of the enhanced Lemming avoidance loop with the ticket lock. The number of transactions that take the fallback decreases because the number of aborts decreases as well. It should be noted the increase in barrier cycles

[7] Calciu et al.'s lazy subscription has the premature commit problem if there are indirect jumps that would jump directly to the commit instruction without performing the subscription. It can lead to unsafe execution but this is not the case with STAMP.

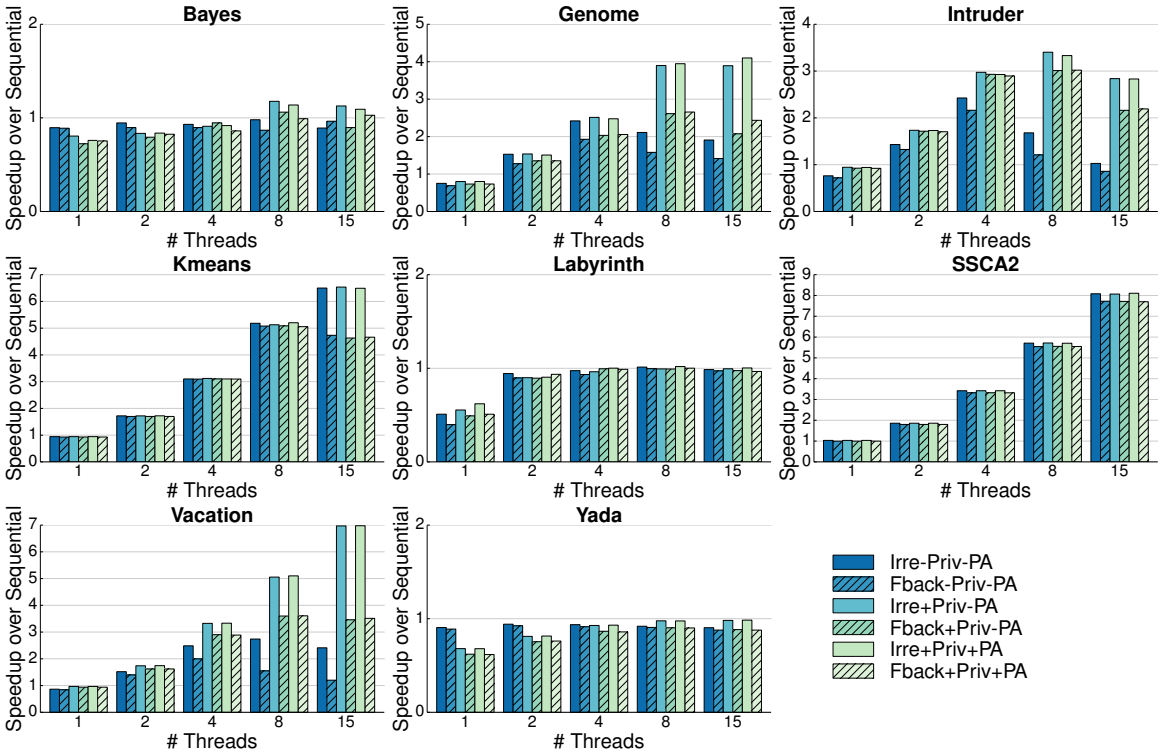

Figure 8: Speedup of the requester-wins policy with (+) or without (-) hardware irrevocability (Irre), software fallback (Fback), privileged mode support (Priv) and privileged-aware cache replacement policy (PA).

for the simple lock due to the load imbalance.

Implication. The ticket lock holds information about how many threads are waiting to take the fallback so the Lemming effect avoidance loop can be improved by waiting until all fallbacks have been executed. Thus, we can prevent that transactions beginning right after the Lemming loop are aborted by those waiting to take the fallback path. We recommend using a ticket lock instead of a simple one in high contended codes that spend much time in transactions. Also, the number of retries should increase with the number of threads to deal with the increased contention.

6.3. Requester-Wins Results

In this section we discuss the results yielded by the three proposals described in the paper with the requester-wins conflict resolution policy, and we compare them with the system with the best fallback solution of the previous section.

Figure 8 shows the speedup, with respect to the sequential version of the application, of the baseline BE-HTM system with the requester-wins policy, with or without the hardware irrevocability mechanism (Irre), the privileged mode code support (Priv), the privileged-aware cache replacement policy (PA) and the software fallback path (Fback). The hardware irrevocability mechanism counter has been set to 5, as well as the retry counter of the fallback. The fallback code version used was the one with Lemming effect avoidance and ticket spin lock, as concluded in the previous section.

From the results in Figure 8 we can classify the STAMP benchmarks in three different categories: (i) the group made up of Bayes, Labyrinth and Yada; (ii) Kmeans and SSCA2; and (iii) Genome, Intruder and Vacation.

Bayes, Labyrinth, 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

Table 3: Percentage of time in privileged mode for the system with requester-wins and irrevocability.

Bench	Privileged Code %				
	1 thread	2 threads	4 threads	8 threads	15 threads
Bayes	4.20	2.33	1.40	0.75	0.48
Genome	0.53	0.71	0.92	1.84	3.95
Intruder	0.86	1.10	2.20	3.62	7.18
Kmeans	0.02	0.34	0.18	0.38	0.38
Labyrinth	0.09	0.11	0.08	0.07	0.11
SSCA2	0.02	0.02	0.04	0.07	0.09
Vacation	1.08	1.40	1.70	3.22	3.10
Yada	1.00	0.68	0.43	0.33	0.31

(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 (or privileged mode incursions in case of -Priv), that may be persistent. This can be avoided by maintaining different retry counters as stated in Nakaike et al. [33] where they adapt the number of retries depending on the cause of abort by maintaining three counters. One for aborts because of the fallback lock, a second one for persistent aborts, and a third one for other aborts, called transient counter. In any case, hardware irrevocability performs slightly better than the fallback path as the former can anticipate the last abort. Also, the version not allowing privileged code is better for one thread[8] as transactions abort earlier and the counters get the retry limit earlier as well.

Although the irrevocable mechanism is better than the fallback one, these benchmarks do not scale because they exhibit large transactions in average. As it can be seen in Table 2, Bayes has transaction ID11, Labyrinth transaction ID1, and Yada transaction ID2, which are numerous and overflow the L1 cache. Table 4 shows the number of irrevocable transactions and its cause, and the majority of them are due to L1 replacements. The PA replacement policy does not seem to have an impact on these benchmarks as they do not exhibit many irrevocable transactions due to privileged mode code incursion. Also, Table 4 shows 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).

Kmeans, SSCA2

These two benchmarks scales well and behave similarly either by using hardware irrevocability or the software fallback path. Furthermore, there is no appreciable difference in allowing privileged mode code inside transactions. This is because of the short time spent in transactions that amounts to 16% for Kmeans and just 11% for SSCA2.

The size of transactions in Kmeans and SSCA2 is small in average, therefore the fallback path or hardware irrevocability is barely taken. Actually, Table 4 shows 0 irrevocable transactions due to L1 and L2 replacements, and privileged mode either. 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

[8]Except for Labyrinth, whose percentage of time in privileged mode is almost 0 (Bayes: 4,2% and Yada: 1%) for one thread, see Table 3.

Table 4: Average number of irrevocable transactions (in bold) for the requester-wins system with irrevocability and privileged mode support, broken down into those due to L1 or L2 replacements and which of these are due to privileged mode code, and those due to conflicts.

Bench	# Irrevocable Xacts (Due to L1 / Priv)(Due to L2 / Priv)(Due to Conflicts)				
	1 thread	2 threads	4 threads	8 threads	15 threads
Bayes	**91**(91/0)(0/0)(0)	**116**(94/1)(0/0)(22)	**135**(86/6)(0/0)(49)	**148**(68/3)(0/0)(80)	**179**(37/7)(3/0)(139)
Genome	**801**(801/0)(0/0)(0)	**869**(823/1)(0/0)(46)	**1203**(1120/1)(0/0)(83)	**1195**(1017/32)(0/0)(178)	**1679**(1358/235)(21/0)(301)
Intruder	**25**(25/4)(0/0)(0)	**316**(35/1)(0/0)(281)	**1055**(22/1)(0/0)(1033)	**3794**(36/5)(0/0)(3759)	**10562**(110/28)(1/0)(10450)
Kmeans	**0**(0/0)(0/0)(0)	**141**(0/0)(0/0)(141)	**400**(0/0)(0/0)(400)	**970**(0/0)(0/0)(970)	**1815**(0/0)(0/0)(1815)
Labyrinth	**79**(79/1)(0/0)(0)	**89**(76/0)(0/0)(13)	**97**(76/2)(0/0)(21)	**122**(57/1)(0/0)(64)	**160**(44/2)(0/0)(116)
SSCA2	**0**(0/0)(0/0)(0)	**37**(0/0)(0/0)(37)	**127**(0/0)(0/0)(127)	**283**(0/0)(0/0)(283)	**515**(0/0)(0/0)(515)
Vacation	**201**(201/0)(0/0)(0)	**348**(338/0)(0/0)(10)	**249**(217/0)(0/0)(33)	**347**(280/0)(0/0)(68)	**433**(301/0)(0/0)(132)
Yada	**705**(705/13)(0/0)(0)	**876**(720/10)(0/0)(156)	**1021**(702/15)(0/0)(319)	**1245**(710/26)(0/0)(535)	**1557**(628/30)(57/2)(872)

Table 5: Average number of aborts of the requester-wins system with irrevocability and its fallback counterpart (+Priv).

Bench	Aborts(IRRE/FBACK)		
	4 threads	8 threads	15 threads
Bayes	653/728	788/1212	1040/1585
Genome	5022/7942	5582/10821	8217/16321
Intruder	11455/10428	35861/39628	77299/96499
Kmeans	2193/2074	5425/6537	10296/19185
Labyrinth	435/631	617/783	797/931
SSCA2	657/575	1583/2140	3208/5486
Vacation	1272/2924	1773/6221	2357/9874
Yada	4651/9128	5895/12561	7600/13643

the fallback version, or not so slight for Kmeans and 15 threads, as the irrevocability mechanism stalls the transactions instead of aborting them. Table 5 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.

Genome, Intruder, Vacation

For this group of benchmarks we obtain considerable benefits by using the HTM system with hardware irrevocability and privileged mode support over the fallback configurations. They are benchmarks with medium and small-sized transactions (Genome and Vacation) or that are more contended (Intruder). This characteristics can be noted in the number of irrevocable transactions that are due to replacements or conflicts in Table 4. Also, the inclusion of another source of contention, like the privileged mode code, harms performance seriously. Table 3 shows that the percentage of privileged mode code in these benchmarks increases with the number of threads.

The hardware irrevocability mechanism not only performs better due to anticipating the last abort but also reduces the number of aborts by stalling non-irrevocable transactions instead of aborting them. Table 5 shows that the number of transaction aborts for the system with irrevocability is usually lower than that of its software fallback counterpart, and the amount of wasted work is larger for the fallback path case. Specially for Genome and Vacation with 15 threads, where the abort reduction is more than 50%.

As far as the PA replacement policy is concerned we get a performance benefit for Genome only. Although the percentage of time in privileged code in Table 3 increases, the reduction in irrevocable transactions (50% for 15 threads) cancels out the performance loss. As it can be seen in Table 4, Genome is the only benchmark that exhibits a considerable amount of irrevocable transactions due to L1 replacements that are caused by privileged mode code (up to 235 for 15 threads).

Summarizing, the BE-HTM system with irrevocability and privileged mode code support speeds up the execution 2x over the fallback path counterpart for Genome and Vacation, and it is around 25% better for Intruder and Kmeans and 15 threads. For the rest of the benchmarks we get the same or slightly better speed-up with the proposed mechanisms than using the software fallback path.

Implication. Privileged mode code execution (due to hardware interrupts, OS events, etc.) aborts user mode transactions in BE-HTM systems, which is another source of contention very detrimental to performance. Particularly when it is summed to the normal contention of a multithreaded application. Allowing privileged mode code execution within transactions is very important to have a scalable HTM system. On the other hand, the hardware irrevocability system is proved to outperform the software fallback path when it comes to providing forward progress guarantees to an HTM system. The number of transaction aborts is reduced to more than 50% in some cases and it can anticipate the last abort before switching to irrevocable mode. Finally, for those codes that exhibit a high number of aborts due to privileged mode code when allowed inside transactions, the PA cache replacement policy is a straightforward solution that reduces the number of aborted transactions at the expense of slowing the privileged code down.

6.4. Requester-Stalls Results

Figure 9 shows a speedup comparison between the requester-wins system discussed in the last section and the requester-stalls system. All bars have privileged code enabled within transactions while the irrevocability mechanism, the software fallback and the PA replacement policy are switched on and off.

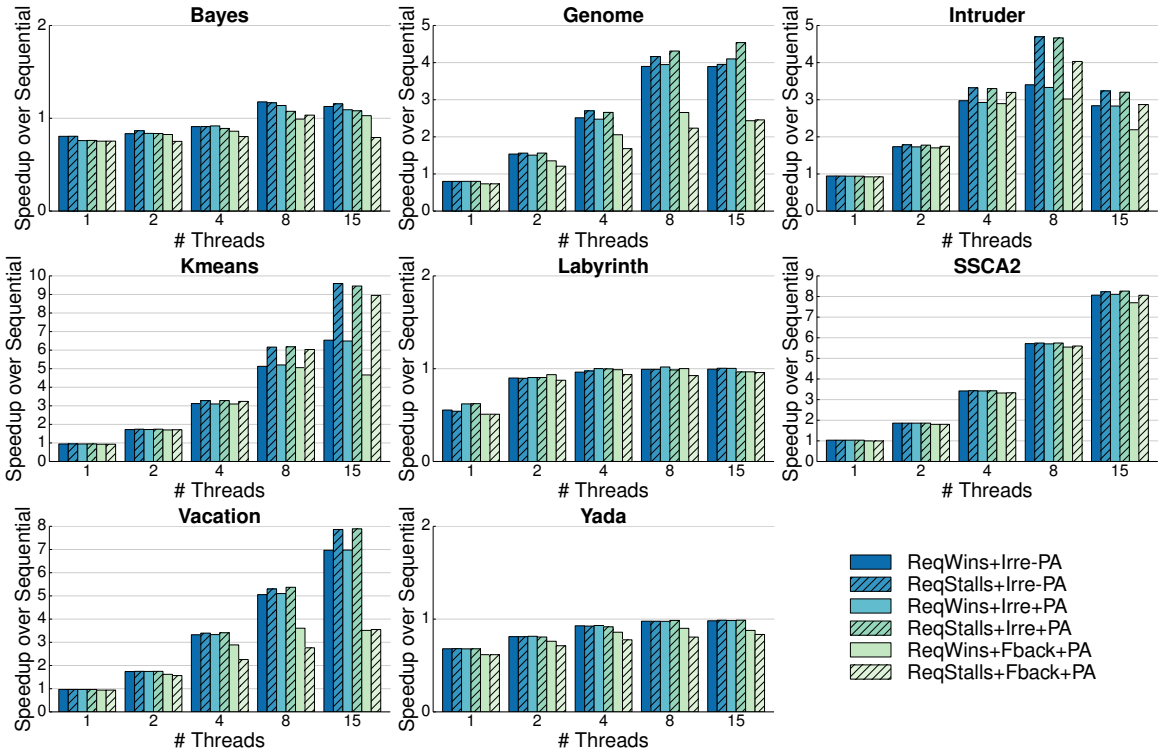

Figure 9: Speedup of the system with requester-wins and requester-stalls conflict resolution policies, combined with irrevocability (Irre) and software fallback (Fback). Privileged mode is allowed and the privileged-aware replacement policy (PA) is compared.

The results suggest another classification of the benchmarks under analysis depending on how much the requester-stalls policy improves the performance with respect to the requester-wins one. We still have the group of benchmarks that does not scale which comprises Bayes, Labyrinth and Yada. There is not a noticeable difference in performance by using the requester-stalls policy with these benchmarks. However, it can be seen a slight slowdown in the software fallback. The reason is that the thread that is trying to get the fallback lock is stalled on request until the ongoing transactions commit (or abort) and erase the lock from their read set. This fact encourages parallelism, but when the benchmark is as contended as the benchmarks in this group it is detrimental to performance since the thread taking the fallback is delayed while transactions are wasting time. The system with irrevocability does not slowdown the execution because the ongoing transactions are stalled whenever one of them gets irrevocable.

The rest of the benchmarks forms a group that experiences a performance gain because of the requester-stalls policy. The system using the software fallback path benefits from the increased concurrency as one thread stalls before taking the fallback lock until the ongoing transactions end. Also, the requester-stalls conflict resolution policy usually favours those transactions that have done more work. Such transactions usually have larger read and write sets and, consequently, it is more likely that other transactions conflict with them, but they are stalled on request, they do not win. Thus, the requester-stalls policy with irrevocability gets the best results and increases performance up to roughly 40% for Kmeans and 15 threads, and Intruder with 8 threads. Genome and Vacation show around 15% of improvement, and SSCA2 is less sensitive to the policy because of the small transactions. As regards the PA cache replacement policy, Genome with ReqStalls+Irre+PA yields up to 18% of performance improvement with respect to ReqWins+Irre-PA. PA favours large transactions as it is more likely that privileged mode code get in their way. As requester-stalls favours large transactions as well, both policies fit together properly.

Figure 10 depicts the cycle breakdown of the execution time normalized to the requester-wins system with irrevocability (WI) for such a configuration, the one with requester-stalls (SI) and their fallback counterparts (WF and SF) with 15 threads. The PA cache replacement policy is enabled. We can see that the time spent in arbitrating the irre-

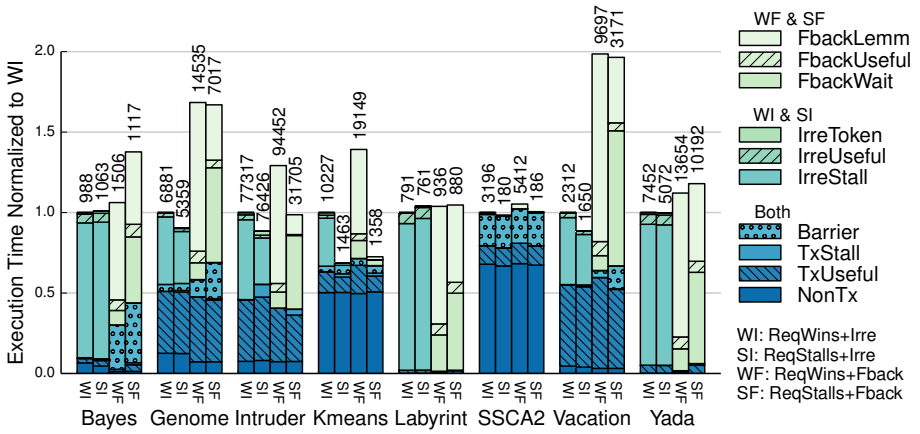

Figure 10: Cycle breakdown comparing requester-wins and requester-stalls policies as well as irrevocability and fallback with 15 threads. The number of aborts is shown on top of the bars.

vocability token (IrreToken) is negligible. The requester-wins policy shows stalled transaction cycles (IrreStall) but it is due to irrevocability. For the software fallback we have broken down the cycles as in Figure 7. It should be noted how the requester-stalls policy increases the FbackWait cycles and decreases the FbackLemm ones. The requester-wins policy allows acquiring the fallback lock by aborting the ongoing transactions so that they wait at the Lemming effect avoidance loop. However, the requester-stalls policy stalls the thread that want to acquire the lock (FbackWait cycles increases) until the ongoing transactions end. Such a reduction in transaction aborts (see the number of aborts in Figure 10) and the increased parallelism enhance the performance of the benchmarks with small transactions like Kmeans and Intruder.

Implication. The requester-stalls conflict resolution policy does not improve the performance of the codes that do not scale but can help to those that have small transactions because of the increased parallelism and the abort reduction. For the benchmarks with medium-sized transactions the improvement is not so noticeable and it may pose the question of whether it is worth implementing it or not.

6.5. Optimal Retry Count

The experimental results discussed so far comprise a retry count of 5 for every configuration of irrevocability/-fallback and number of threads. However, Figure 6 shows that a higher number of retries enhance the performance of certain benchmarks when the number of threads increases. Diegues et al. [34] propose a self-tuning mechanism to identify the optimal retry count, thus trying to solve this issue (see Section 7).

In this section we evaluate our proposals using the best retry configuration found in Section 6.2 to see how it affects the results. Figure 11 shows the results of our irrevocability mechanism (Irre) and the fallback path with ticket lock and Lemming effect avoidance (Fback) with 15 threads and 5 and 10 retries. Privilege mode is allowed (+Priv), the privilege-aware cache replacement policy is switched on and off (PA) and the conflict resolution policy is requester-wins. We can see how the gap between the irrevocability mechanism and the software fallback path shrinks with the increase in the number of retries. This is mainly due to the reduction in the number of transactions that get irrevocable or take the fallback. As we give more opportunities for the conflicting transactions to retry, they eventually arrange themselves in a way that can commit. Our proposals enhance the fallback mechanism of transactions, so the fewer the transactions that get irrevocable the lesser the gain.

6.6. Network Interconnect Power Requirements

As stated in Section 3.2, the irrevocability mechanism is implemented mainly in the L1 controllers without modifications to the CPU and cache memories. Consequently, L1 controllers have to deal with new messages introduced by the token-based irrevocability protocol with the corresponding network traffic. In this section we measure the power implications of this traffic in the network interconnect.

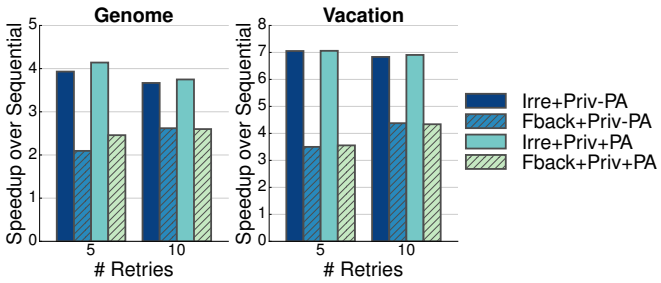

Figure 11: Speedup of irrevocability (Irre) and software fallback (Fback) for 15 threads, requester-wins and 5 and 10 retries.

Table 6: Interconnect dynamic power requirements for the different configurations and 15 threads. RW: Requester-Wins, RS: Requester-Stalls, I: Irrevocability, F: Fallback, P: Privileged mode, PA: Privileged-aware cache replacement policy. The static power of the network is 4.0212W.

	Dynamic Power in Watts (Routers + Links)							
Bench	**RW+I-P-PA**	**RW+F-P-PA**	**RW+I+P-PA**	**RW+F+P-PA**	**RW+I+P+PA**	**RW+F+P+PA**	**RS+I+P+PA**	**RS+F+P+PA**
Bayes	1.1977	1.1977	1.1983	1.1979	1.1983	1.1982	1.2024	1.2135
Genome	1.2257	1.2196	1.2631	1.2329	1.2682	1.2397	1.2795	1.2678
Intruder	1.2344	1.2336	1.2910	1.2750	1.2918	1.2765	1.3228	1.3498
Kmeans	1.2841	1.2918	1.2842	1.2912	1.2830	1.2917	1.2924	1.2940
Labyrinth	1.1960	1.1948	1.1962	1.1951	1.1962	1.1950	1.1962	1.1984
SSCA2	1.3825	1.3766	1.3817	1.3774	1.3829	1.3763	1.3842	1.3791
Vacation	1.2282	1.2146	1.2835	1.2454	1.2838	1.2459	1.3007	1.2895
Yada	1.2039	1.2041	1.2081	1.2043	1.2084	1.2043	1.2086	1.2189

Table 6 shows the dynamic power requirements of the network interconnect for all the benchmarks with 15 threads. All requester-wins configurations are shown along with the requester-stalls ones that allow privileged code inside transactions and use the PA replacement policy. The static power of the network is 4.0212W.

Overall, the software fallback path requires slightly less power than irrevocability. The irrevocability mechanism broadcasts messages to ask for the irrevocability token and to communicate the start and end of irrevocability. However, such messages consist of small packets with source and destination IDs which do not cause significant increase in network traffic and power. The lock subscription may cause high network traffic as well, particularly for benchmarks that spend most time in transactions. The lock invalidations and requests carry the lock address and value (larger packets), although these messages are not broadcast.

When comparing the configurations that allow privileged mode on transactions with those that do not, we find a more noticeable increase in power for the former due to the enhanced parallelism. The latter configurations, though, run for shorter before being aborted by privileged mode and gets earlier to the fallback path, thus serializing the execution and diminishing the network traffic.

However, the power delay product (PDP) given by multiplying the execution time of the benchmarks by the power values of Table 6 is similar to the execution time. This makes the power increase of the proposed mechanisms negligible compared with the software fallbacks.

6.7. Coexistence of Software Fallback and Hardware Irrevocability

IBM's Blue Gene/Q [12] has a system-provided retry mechanism that cannot be substituted by a user's software fallback. The user only set the number of retries via an environmental variable. However, it would not be very difficult to provide a way for the user to switch between software fallback and hardware irrevocability. An environmental variable would convey this information to the system, and the system would decide at the retry threshold whether to abort the transaction and execute the software fallback, or to execute the hardware irrevocability protocol.

This approach enforces that either all transactions take the fallback path or all of them use the irrevocability mechanism. But we might want to have some transactions with software fallback path and others using irrevocability in some situations. The interaction between irrevocable transactions and transactions in the fallback path should be avoided since neither of them are tracking transactional information nor can be aborted. Therefore, the system must

ensure that no transaction gets irrevocable when another one is in the fallback path and vice-versa. The implications of this coexistence for the implementation solutions proposed in this work are worth exploring in future research.

7. Related Work

Irrevocability in the context of HTM was first proposed in TCC [5] to deal with overflowed transactions. In the literature, irrevocability has been used to support operations that cannot be rolled back within transactions (like I/O and system calls), and to ensure forward progress due to contention and capacity overflows. Blundell et al. [7] introduces OneTM-Serialized as a system where overflowed transactions gets irrevocable and serializes the system to ensure forward progress. They implement the irrevocability mechanism in a log-based HTM context where the irrevocable transaction can be aborted after serialization since transactional data can be recovered from the log. They use 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 irrevocability mechanism. We implement irrevocability with a token-based scalable mechanism distributed through the core cache controllers, very similar to [20], but in the context of a best-effort HTM system, comparing its performance with a software fallback path. Also, our irrevocability mechanism has an abort counter to ensure forward progress on high contention scenarios.

Some leading computer manufacturers include BE-HTM support in their multicore processors. Intel Haswell [13] provides a HTM system similar to that of our baseline system in that transactional bookkeeping is kept in the L1 cache. However, Haswell seems to extend the read set beyond the L1 cache up to the L3 cache [35], as transactions tend to exhibit asymmetrical data sets [36]. Haswell provides a software fallback path to deal with transaction aborts. In terms of architectural support for exceptional situations like interrupts, exceptions, operating system events, etc. Haswell simply aborts the ongoing transaction.

On the other hand, IBM is deploying different BE-HTM systems. Blue Gene/Q [12] has a different approach. It 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, Blue Gene/Q implements an irrevocable mode by means of 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 irrevocable mechanism can anticipate an abort and initiates the irrevocable mode without wasting the work done so far by the transaction.

IBM System z [11] is another approach to a BE-HTM system. System z relies on a gathering store cache to hold transactional stores and on private L1 and L2 caches to hold the read set of transactions, although the read set size is extended from the L1 size to the L2 size by an LRU extension bit-vector that holds the L1 evicted blocks. The sets whose bit is set will detect a conflict even when the requested block is not actually transactional. System z also offers the possibility of having a software fallback code to ensure forward progress. The system aborts on interrupts, operating system events, and other exceptional situations.

The IBM Power architecture [10] has been also extended in order to be HTM-enabled. In contrast to its siblings, the Power architecture tries to solve certain interactions between the HTM system and the ISA to provide a robust system that supports simple system calls and debugging within transactions. Interrupts are allowed and they change the transaction mode to suspended. In this mode, memory accesses are performed non-transactionally, a new transaction cannot be started, and the access to transactional data by the non-transactional code causes the suspended transaction to abort. However, the transaction abort handling (by a fallback path or by co-opting the HTM system via an OS service) is deferred until transactional execution is resumed to prevent the code handling the interrupt from being routed to the abort handler. The ISA also has an instruction to reclaim the HTM facility that resets memory and registers to their pre-transactional state, just in case the OS decides to schedule a different thread. When the suspended thread is rescheduled the abort fallback handler is executed. We propose a similar deferred abort handling but we opt for asking for irrevocability in case the interrupt code tries to access a transactional block. If irrevocability is granted, the transaction is not aborted thus saving the work done so far. Otherwise, the first-phase of the abort releases the HTM facility. The two-phase abort can be exposed to the ISA to provide architectural support to the OS and thus relaxing the one-transactional-application-at-a-time invariant described in Section 4.

Diestelhorst et al. [37] propose OS-transparent suspend/resume by means of transactional resurrection, a mechanism that exposes all the register file to the abort handler so that it can resume execution inside the transaction. Then,

transactions are aborted on exceptions, interrupts, syscalls,... and the abort fallback handler can restore the transaction's register state the same as it was at the point of the abort, which adds complexity to the fallback programming. Other works[8, 38] deal with transactional pause/resume instructions to interact with the operating system, to de-schedule transactions or to allow interrupts. However, these systems rely on a fully virtualized HTM facility that involves high implementation complexity.

Regarding how to hide BE-HTM limitations from the programmer, Dalessandro et al. [39] propose a hybrid approach where the fallback path is a software TM (STM) system. Using the NOrec algorithm for STM, which does not require an ownership record per transactional datum, only a single global sequence lock for concurrency control, their Hybrid NOrec does not incur the overhead of instrumenting each access of hardware transactions. However, the single global sequence lock serializes the commit. Diegues et al. [34] expose the problem of getting different performance results depending on the configuration of the fallback path. Thus, they propose a self-tuning fallback code to try to adapt to the characteristics of the workload. They show how a benchmark can yield different performance when using different number of retries. Their solution uses different runtime learning techniques and gets performances of only 5% under the optimal static configuration.

8. Conclusions

Best-effort hardware transactional memory (BE-HTM) systems are being included in chip multiprocessors of several hardware manufacturers. Most of them do not have support for transactions to survive interrupts, operating system events and capacity overflows. Further, contention scenarios may lead to live-lock or never-ending execution. To ensure forward progress in those situations, hardware manufacturers provide the user with a software fallback path, which often comprises a global lock as suggested by the manufacturer.

In this paper we propose various hardware implementation solutions to enhance the performance and scalability of a BE-HTM and to release the programmer from the burden of writing fallback path code. We propose a hardware irrevocability mechanism as alternative to a fallback handler, that anticipates capacity overflows and stalls the rest of transactions running in the system, thus discarding as little transactional work as possible.

We propose a two-phase abort that, along with certain invariants, simplifies the support for privileged mode code amid transactions. Allowing privileged mode within transactions implies adapting the irrevocability mechanism to be activated whenever such code causes a capacity overflow. And so, we propose a privileged-aware cache replacement policy to mitigate the transactional cache evictions due to privileged mode code.

The experiments were conducted in a simulation environment comprising Simics/GEMS and all the workloads of the STAMP benchmark suite. We compared our proposals with a state-of-the-art BE-HTM system with fallback path, after evaluating different fallback alternatives. Results show significant performance improvements for certain benchmarks, and slightly for others, but the proposals never incur slowdown with respect to the fallback counterpart.

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, R. Rajwar, Transactional Memory, 2nd edition, Morgan & Claypool Publishers, 2010.

[2] M. Herlihy, 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] C. Ananian, K. Asanovic, B. Kuszmaul, C. Leiserson, S. Lie, Unbounded transactional memory, in: 11th Int'l. Symp. on High-Performance Computer Architecture (HPCA'05), 2005, pp. 316–327.

[4] L. Ceze, J. Tuck, J. Torrellas, C. Cascaval, Bulk disambiguation of speculative threads in multiprocessors, in: 33th Ann. Int'l. Symp. on Computer Architecture (ISCA'06), 2006, pp. 227–238.

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

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

[7] C. Blundell, J. Devietti, E. C. Lewis, 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, 2007, pp. 24–34.

[8] R. Rajwar, M. Herlihy, K. Lai, Virtualizing transactional memory, in: 32th Ann. Int'l. Symp. on Computer Architecture (ISCA'05), 2005, pp. 494–505.

[9] L. Yen, J. Bobba, M. Marty, K. Moore, H. Volos, M. Hill, M. Swift, D. Wood, LogTM-SE: Decoupling hardware transactional memory from caches, in: 13th Int'l. Symp. on High-Performance Computer Architecture (HPCA'07), 2007, pp. 261–272.
[10] H. W. Cain, M. M. Michael, B. Frey, C. May, D. Williams, 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.
[11] C. Jacobi, T. Slegel, D. Greiner, Transactional Memory Architecture and Implementation for IBM System z, in: 45th Ann. Int'l. Symp. on Microarchitecture (MICRO'12), 2012, pp. 25–36.
[12] A. Wang, M. Gaudet, P. Wu, J. N. Amaral, M. Ohmacht, C. Barton, R. Silvera, 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.
[13] R. M. Yoo, C. J. Hughes, K. Lai, 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.
[14] A. Welc, S. Bratin, A.-R. Adl-Tabatabai, Irrevocable transactions and their applications, in: 20th ACM Symp. on Parallelism in Algorithms and Architectures (SPAA'08), 2008, pp. 285–296.
[15] 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, B. S. Onge, Verification of Transactional Memory in POWER8, in: 51st Ann. Design Automation Conf. (DAC'14), 2014, pp. 1–6.
[16] M. M. K. Martin, C. Blundell, E. Lewis, Subtleties of Transactional Memory Atomicity Semantics, IEEE Computer Architecture Letters 5 (2) (2006) 17.
[17] D. J. Sorin, M. D. Hill, D. A. Wood, A Primer on Memory Consistency and Cache Coherence, 1st Edition, Morgan & Claypool Publishers, 2011.
[18] A. Shriraman, S. Dwarkadas, Refereeing Conflicts in Hardware Transactional Memory, in: 23rd Int'l . Conf. on Supercomputing (ICS'09), 2009, pp. 136–146.
[19] M. J. Moravan, J. Bobba, K. E. Moore, L. Yen, M. D. Hill, B. Liblit, M. M. Swift, 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), 2006, pp. 359–370.
[20] R. Quislant, E. Gutierrez, E. L. Zapata, O. Plata, Improving Signature Behavior by Irrevocability in Transactional Memory Systems, in: 26th Int'l. Symp. on Computer Architecture and High Performance Computing (SBAC-PAD'14), 2014, pp. 120–127.
[21] Intel 64 and IA-32 Architectures Optimization Reference Manual. Chapter 12.3: Developing an Intel TSX Enabled Synchronization Library (Sep 2014).
[22] D. Dice, M. Herlihy, D. Lea, Y. Lev, V. Luchangco, W. Mesard, M. Moir, K. Moore, D. Nussbaum, Applications of the Adaptive Transactional Memory Test Platform, in: 3rd Workshop on Transactional Computing (TRANSACT'08), 2008.
[23] D. J. Sorin, M. Plakal, A. E. Condon, M. D. Hill, M. M. K. Martin, D. A. Wood, Specifying and verifying a broadcast and a multicast snooping cache coherence protocol, IEEE Trans. Parallel and Distributed Systems 13 (6) (2002) 556–578.
[24] L. Baugh, C. Zilles, An Analysis of I/O And Syscalls In Critical Sections And Their Implications For Transactional Memory, Int'l Symp. on Performance Analysis of Systems and Software (ISPASS'08) (2008) 54–62.
[25] P. Magnusson, M. Christensson, J. Eskilson, D. Forsgren, G. Hallberg, J. Hogberg, F. Larsson, A. Moestedt, B. Werner, B. Werner, Simics: A full system simulation platform, IEEE Computer 35 (2) (2002) 50–58.
[26] M. Martin, D. Sorin, B. Beckmann, M. Marty, M. Xu, A. Alameldeen, K. Moore, M. Hill, D. Wood, Multifacet's general execution-driven multiprocessor simulator GEMS toolset, ACM SIGARCH Computer Architecture News 33 (4) (2005) 92–99.
[27] N. Agarwal, T. Krishna, L.-S. Peh, 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), 2009, pp. 33–42.
[28] A. Kahng, B. Li, L.-S. Peh, 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), 2009, pp. 423–428.
[29] A. R. Alameldeen, D. A. Wood, Variability in architectural simulations of multi-threaded workloads, in: 9th Int'l Symp. on High-Performance Computer Architecture (HPCA'03), 2003, pp. 7–18.
[30] C. Minh, J. Chung, C. Kozyrakis, K. Olukotun, STAMP: Stanford Transactional Applications for Multi-Processing, in: IEEE Int'l Symp. on Workload Characterization (IISWC'08), 2008, pp. 35–46.
[31] J. M. Mellor-Crummey, M. L. Scott, Algorithms for scalable synchronization on shared-memory multiprocessors, ACM Trans. on Computer Systems 9 (1) (1991) 21–65.
[32] I. Calciu, T. Shpeisman, G. Pokam, M. Herlihy, Improved Single Global Lock Fallback for Best-effort Hardware Transactional Memory, in: 9th Workshop on Transactional Computing (TRANSACT'14), 2014.
[33] T. Nakaike, R. Odaira, M. Gaudet, M. M. Michael, 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.
[34] N. Diegues, P. Romano, Self-Tuning Intel Transactional Synchronization Extensions, in: 11th Int'l. Conf. on Autonomic Computing (ICAC 14), 2014, pp. 209–219.
[35] B. Goel, R. Titos-Gil, A. Negi, S. A. Mckee, P. Stenstrom, Performance and Energy Analysis of the Restricted Transactional Memory Implementation on Haswell, in: 28th Int'l Symp. on Parallel and Distributed Processing (IPDPS'14), 2014, pp. 615–624.
[36] R. Quislant, E. Gutierrez, O. Plata, E. Zapata, Hardware signature designs to deal with asymmetry in transactional data sets, IEEE Trans. on Parallel and Distributed Systems 24 (3) (2013) 506–519.
[37] S. Diestelhorst, M. Nowack, M. Spear, C. Fetzer, Between All and Nothing — Versatile Aborts in Hardware Transactional Memory, in: 10th Workshop on Transactional Computing (TRANSACT'15), 2015.
[38] C. Zilles, L. Baugh, Extending Hardware Transactional Memory to Support Non-busy Waiting and Non-transactional Actions, in: 1st Workshop on Transactional Computing (TRANSACT'06), 2006.
[39] L. Dalessandro, F. Carouge, S. White, Y. Lev, M. Moir, M. L. Scott, M. F. Spear, Hybrid NOrec: A Case Study in the Effectiveness of Best Effort Hardware Transactional Memory, in: Int'l. Conf. on Architectural Support for Programming Languages and Operating Systems (ASPLOS'11), 2011, pp. 39–52.