Concurrent Irrevocability in Best-Effort Hardware Transactional Memory

Rubén Titos-Gil[1], Ricardo Fernández-Pascual[1], Alberto Ros[1] and Manuel E. Acacio[1]

Abstract—Existing best-effort requester-wins implementations of transactional memory must resort to non-speculative execution to provide forward progress in the presence of transactions that exceed hardware capacity, experience page faults or suffer high-contention leading to livelocks. Current approaches to irrevocability employ lock-based synchronization to achieve mutual exclusion when executing a transaction non-speculatively, conservatively precluding concurrency with any other transactions in order to guarantee atomicity at the cost of degrading performance. In this work, we propose a new form of *concurrent irrevocability* whose goal is to minimize the loss of concurrency paid when transactions resort to irrevocability to complete. By enabling optimistic concurrency control also during non-speculative execution of a transaction, our proposal allows for higher parallelism than existing schemes. We describe the extensions to the instruction set to provide concurrent irrevocable transactions as well as the architectural extensions required to realize them on a best-effort HTM system without requiring any modification to the cache coherence protocol. Our evaluation shows that our proposal achieves an average reduction of 12.5% in execution time across the STAMP benchmarks, with 15.8% on average for highly contended workloads.

Index Terms—Parallel programming, multicore architectures, transactional memory

1 Introduction and Motivation

Hardware Transactional Memory (HTM) was originally proposed in the early 1990s with the promise of simplifying parallel programming by overcoming the major difficulties associated with conventional locking techniques, namely priority inversion, convoying, and difficulty of avoiding deadlock [1]. More than a decade after, the omnipresence of multicore architectures, and consequently, the urgency for making parallel programming widely accessible, brought HTM into the spotlight again. During those years important research efforts were done to solve many of the pitfalls that the first-generation HTM systems had, among them transaction virtualization to allow for cache victimization, unbounded nesting, thread suspension/migration, paging, etc. It was not until the early years of this decade that processor manufacturers began to deploy hardware support for transactional memory on their chips [2], [3]. Unfortunately, HTM support in commercially available processors is very rudimentary and is therefore far from what was supposed to be an HTM system for a general audience.

Particularly, HTM support in current processors is best-effort: The architecture does not guarantee that a speculative transaction will ever succeed [4]. Therefore, a non-speculative alternative software path, often called *fallback path*, must be combined with the HTM support to ensure forward progress in circumstances that otherwise would cause livelock because of insufficient speculative buffering capacity, page faults, high contention, etc. Atomicity among transactions is then guaranteed through pessimistic concurrency control, reverting to execution of transactions in mutual exclusion in the same way lock-based programs do. In this way, no matter if intermediate values are evicted to shared levels of the cache hierarchy, or program execution gets interrupted by the kernel (page fault, interrupt or system call), no other speculative transactions can observe the intermediate state until the non-speculative transaction (henceforth also referred to as *irrevocable*) has completed and released the lock.

Depending on their implementation details and the characteristics of each workload, best-effort HTM systems may need to resort more or less frequently to the fallback path. How read and write sets are tracked, how speculative values are handled or how conflicts are managed, will partly determine whether a given transaction can commit in hardware (*i.e.*, within the speculative mechanisms of the HTM substrate), or must fall back to non-speculative execution to make progress. For instance, the size and associativity of the buffers used for speculative versioning (typically the L1 data cache) impose a hard limit on the maximum write set size. Similarly, HTM designs with eager conflict detection and requester-wins resolution (like current implementations of HTM in Intel chips) need to resort to mutual exclusion through the fallback path to escape livelocks.

Conversely, the way transactional programs are written also influences the ability of transactions to commit within the hardware bounds. Workloads with fine-grain transactions and low contention rarely take the fallback path and when they do, the slowdown suffered is anyways low—small amount of work is discarded, and short time is spent waiting for an irrevocable transaction to complete. On the other hand, programs written with a coarse-grain synchronization style in mind—precisely the approach promoted by TM as part of its promise to simplify parallel programming—are more likely to contain long-running transactions that put pressure on the speculative buffering capacity, and may be more prone to exhibit contention. In requester-wins best-effort HTM implementations, these workloads may need to resort to the fallback path rather often. Examples of these workloads are found in the STAMP benchmarks [5], where threads spend a significant amount of cycles in transactions that eventually must fall back to irrevocability in order to make progress, as it can be observed in

• [1]*Dept. Ingeniería y Tecnología de Computadores, Universidad de Murcia, 30100 Murcia (SPAIN)*
E-mail: {rtitos, rfernandez, aros, meacacio}@ditec.um.es

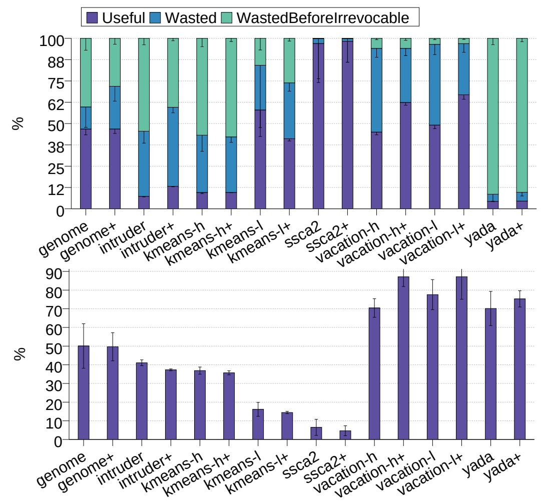

Fig. 1: Distribution of transactional cycles (top). Percentage of aborted transactional cycles directly caused by the irrevocability mechanism (bottom).

Fig. 1-top. For the requester-wins best-effort HTM system that acts as baseline in this work, Fig. 1-top breaks down in three categories the cycles spent executing speculative transactions (cycles in the abort handler are also included), for 16-thread runs: cycles in committed transactions (Useful), cycles in aborted transactions that eventually commit in a later retry (Wasted), and cycles in aborted transactions that eventually end up taking the non-speculative path (WastedBeforeIrrevocable). We can see how in benchmarks such as genome, intruder, kmeans-h and yada, threads waste a very significant fraction of speculative transactional cycles before falling back to non-speculative, irrevocable execution. The reader is referred to Section 4 for further details about our methodology.

The software implementation of irrevocability recommended by chip manufacturers [4] uses a single global lock, commonly referred to as the *fallback lock*, which must be read by all speculative transactions as soon as speculation begins. This approach, known as *eager subscription*, ensures that an irrevocable transaction never executes concurrently with other speculative transactions. When the lock is acquired, all subscribed speculative transactions are aborted as a result of the conflict. For the baseline HTM system used throughout this work, Fig. 1-bottom shows that the work discarded because of the acquisition of the fallback lock is considerable in several STAMP benchmarks. The figure shows the fraction of aborted cycles caused by the fallback lock (including both conflict-induced aborts and explicit aborts of speculative transactions that find the lock held upon subscription), over the total number of aborted cycles. We can see that in benchmarks such as vacation-h (medium), vacation-l (medium) and yada (medium) about 80% of all aborted cycles are caused by transactions resorting to irrevocability. The results shown in Fig. 1 reveal the importance of minimizing the performance penalty of resorting to the fallback path.

The rationale behind this work is that precluding concurrency between irrevocable and speculative transactions is sufficient to guarantee atomicity, but is not always necessary, as illustrated by the following example: consider a parallel program that accesses two distinct shared data structures, A and B, respectively, through transactions T_A (with high contention) and T_B (low contention). In this scenario, when a speculative instance of T_A decides to acquire the lock after repeated conflict-induced aborts, all other transactions are aborted, including all instances of T_B, even though T_B does not pose a risk to the atomicity of T_A, given their disjoint read-write sets. Provided that irrevocable transactions have some way to detect and prevent conflicting accesses from speculative transactions, the performance degradation of taking the fallback path can be diminished by bringing in parallelism. Ideally, the transition to irrevocable execution shall not immediately give up the optimistic concurrency control of transactional memory, but instead only revert to the purely conservative, pessimistic approach of locking when strictly necessary to preserve atomocity.

Our key observation to realize this idea is that the conflict detection hardware that a best-effort HTM already has in place can be leveraged to detect races between speculative and non-speculative transactions. With minor architectural extensions, the performance drop of resorting to the fallback path can be largely reduced through the use of *on-demand* mutual exclusion, so that the acquisition of the fallback lock is elided as long as those situations that threaten atomicity do not arise. To the best of our knowledge, our work is the first to enable concurrency between irrevocable and speculative transactions using eager lock subscription. To achieve this goal, we propose a hardware-software co-design that combines existing HTM mechanisms (ability to track read-write sets and detect conflict-, capacity- or interrupt-induced aborts) with traditional synchronization primitives (locks). Following up the example above, our proposal allows threads executing transaction T_B to commit concurrently without interference from contention experienced by concurrent instances of T_A: after repeated aborts, T_A eventually resorts to our proposed hardware support for irrevocability so that one instance of T_A is guaranteed to make progress at a time while side-stepping the acquisition of the fallback lock in the common case.

In this way, we present a best-effort HTM design that is able to decouple non-speculative execution from mutual exclusion, two different concepts that so far have been invariably combined when implementing irrevocability. To do so, we propose the concept of *concurrent irrevocable transaction* (CIT), present its programming interface, and describe the architectural extensions required to support CITs on a best-effort HTM system without requiring any modification to the cache coherence protocol. We coined the term *concurrent irrevocability* to describe the ability of our approach of executing non-speculative transactions in concurrency with any number of speculative transactions, without imposing any particular order in their commits. This enables maximum parallelism among threads also when one takes the fallback path, allowing it to co-exist with other running transactions for as long as its progress or atomicity is not at risk. Through detailed full-system simulation using Gem5 [6], we show that our proposal achieves an average performance improvement of 12.5% across STAMP [5], with 15.8% on average for benchmarks with high contention.

The rest of the manuscript is organized as follows. Relevant background and related works are discussed in Section 2. Then, we present our approach for concurrent irrevocable transactions in Section 3. Section 4 describes our simulation environment and detailed results are shown and analyzed in Section 5. Finally, Section 6 contains the main conclusions of this work and avenues for future work.

2 Background and Related Work

The typical software implementation of irrevocability when using the Intel *Restricted Transactional Memory* (RTM) instructions is shown in Listing 1. Following the *Transactional Synchronization Extensions* (TSX) recommendations [4], all speculative transactions perform eager subscription on a single global lock, which is invariably present in their read set: immediately after the transaction begin instruction (`xbegin` in RTM), transactions must check the value of the lock (line 6) and only proceed if the lock is free. This way, when a thread determines that it must resort to non-speculative execution, it achieves mutual exclusion by acquiring the lock (line 15), simultaneously meeting the two necessary conditions to maintain weak atomicity: i) No other transaction can execute non-speculatively since it must first acquire the fallback lock, which would be locked for as long as another non-speculative transaction executes; and ii) the write to the lock variable causes the immediate abort of all other speculative transactions due to a transactional conflict on a block in their read set. Race conditions with newly started speculative transactions not yet subscribed to the lock, are resolved by explicitly aborting when the fallback lock is found acquired (line 8). Note that atomicity would be at risk if speculative transactions were allowed to run concurrently with a non-speculative transaction, as the latter would not be able to prevent conflicting accesses made by the former. Thus, after a speculative transaction has been aborted, the thread must wait for the fallback lock to be unlocked before it can retry the transaction.

Listing 1: Recommended software implementation of wrappers to begin/commit a transaction on Intel RTM.

```
1  void beginTransaction() {
2    int ret, nretries = 0;
3    do {
4      ret = _xbegin();
5      if (transactionHasStarted(ret)) {
6        if (!isLocked(fallbackLock))
7          return; // Execute speculatively
8        else _xabort();
9      }
10     // Fallback path after abort starts here
11     while (isLocked(fallbackLock)) idle();
12     ++nretries;
13   } while (canSucceedOnRetry(ret) &&
14            !tooManyRetries(nretries));
15   acquireLock(fallbackLock);
16   // Execute non-speculatively
17 }
18 void commitTransaction() {
19   if (isLocked(fallbackLock))
20     releaseLock(fallbackLock);
21   else
22     _xend();
23 }
```

Lazy subscription [7] allows some overlap between irrevocable and speculative transactions by delaying the subscription to the lock until immediately before the transaction attempts to commit (*e.g.*, before the `xend` instruction) [8]. Lazy subscription can lead to a variety of incorrect behaviors [9] that hinder its use in existing HTM, as a consequence of letting speculative transactions read memory locations in an inconsistent state (*i.e.*, the intermediate values produced by the non-speculative transaction). Ensuring that a transaction lazily subscribes to the correct lock can only be accomplished through specific hardware support.

Quislant *et al.* [10] realize lazy subscription in a best-effort HTM design that employs a dedicated broadcast-based token protocol (*i.e.*, a hardware lock) to force correct subscription and arbitrate commit order and entry into irrevocable mode. In that work, committing speculative transactions that find an ongoing irrevocable transaction are not aborted; instead, they wait and keep detecting conflicts until the irrevocable transaction has completed. Its main drawbacks are the additional complexity in the communication fabric required and the challenges to support multiprogramming or multiple locks for the fallback path, since the existence of at most one irrevocable transaction (system-wide) is hardwired.

The reader should note the implicit pessimism inherent in lazy subscription when compared to our proposal: a speculative transaction about to commit may not have any overlapping with a concurrent non-speculative transaction, but it must abort *just in case*. As opposed to our proposal, lazy subscription forces a specific commit order in which all speculative transactions must invariably serialize *after* the irrevocable one, since the latter is unable to detect conflicts. In contrast, our proposal builds atop eager lock subscription, avoids all the pitfalls caused by observing inconsistent state of lazy subscription, and it enables concurrency between speculative and irrevocable transactions without imposing any particular commit order among them. The atomicity of a concurrent irrevocable transaction is guaranteed by leveraging conflict detection hardware so that accesses coming from speculative transactions are serviced at a later point.

Introducing the ability to prevent conflicting accesses into requester-wins best-effort HTM systems has also been considered in recent work [11] with the aim of improving performance under contention. By extending the coherence protocol to support negative acknowledgments (*nacks*), Dice *et al.* propose selective inversion of the conflict resolution policy to *requester-loses* so as to prioritize one speculative transaction over the rest and allow it to make forward progress despite conflicting with others, thus resorting less often to the fallback path. In this way, when a regular transaction repeatedly aborts due to conflicts, it re-executes with higher priority instead of directly resorting to non-speculative execution. Nonetheless, these high priority transactions are still speculative and thus a fallback path is required to guarantee progress of transactions that cannot be accommodated in hardware. Thus, *power transactions* only improve parallelism during contention, whereas our proposal also benefits large transactions that exceed hardware capacity limits or encounter page faults, by allowing concurrency during the fraction of the irrevocable transaction executed before such events. Furthermore, our concurrent irrevocable transactions can be implemented without *nacks*, as opposed to power transactions. In this regard, chip manufacturers willing to modify the coherence protocol may opt for other conflict resolution policies such as *requester-stalls* (like LogTM [12] does), though in this case some scheme of deadlock avoidance becomes necessary.

Reducing the cost of aborts in best-effort HTM systems has also been the subject of a recent work by Park *et al.* [13], where the authors propose a hardware-software solution to salvage the aborting transaction's useful work in some scenarios by trapping to a software routine on abort signal and executing the appropriate *pre-abort handler* while the transaction is paused (before speculation is discarded). On its part, Mohamedin *et al.* [14] also take a hardware-software approach at coping with resource limitations of best-effort HTM: rather than falling back to mutual exclusion for transactions that run into hardware limits, they propose a *slow path* in which a transaction that cannot commit in hardware due to time or space constraints is split into sub-transactions executed within

B	Begin Transaction (XBEGIN)	Up	Power lock release
C	Commit Transaction (XEND)	Ta	Token acquire
Lf	Acquire Fallback Lock	Tr	Token release
Li	Acquire Irrevocable Lock	Cnf	Conflict signal
B’	Retry Transaction (XBEGIN)	Cap	Insufficient capacity signal
Bi	Begin Irrevocable Transaction (IXBEGIN)	Data	Data response
C’	Early Commit	🟪	Transactional
Uf	Release Fallback Lock	🟩	Irrevocable concurrent
Ui	Release Irrevocable Lock	🟦	Irrevocable non concurrent
A	Abort Transaction	🟩	Transactional power
Lp	Power lock acquire	🟥	Waiting
Bp	Begin Power Transaction (IXBEGIN)		

Fig. 2: Legend for diagrams in Figures 3 and 4.

hardware bounds while employing software instrumentation to maintain atomicity of the original transaction.

Several hybrid schemes that combine HTM support with software algorithms have also been proposed in order to allow concurrency between speculative transactions and the fallback path. In [15], Afek *et al.* propose a lock-elision algorithm that provides concurrency between speculative and non-speculative transactions, by splitting the critical section executed on the fallback path into segments of dynamically-adjustable size, which are executed as speculative transactions. In this scheme, the single global lock of the fallback path is replaced with fine-grained locking to detect conflicts with the speculative transactions. Dice *et al.* also take a similar approach in [16], where they propose algorithms that rely on existing compiler support to allow threads to speculate concurrently on HTM along with a thread holding the lock, at the cost of an additional read/write set instrumentation.

3 Concurrent Irrevocability

In this section, we describe our proposed scheme of hardware-supported *Concurrent Irrevocable Transaction* (henceforth referred to as CIT). We present it as an extension to the Intel RTM instruction set, and as such the underlying best-effort HTM system follows the TSX specifications, where transactions can abort because of conflicts, lack of capacity, page faults or interrupts.

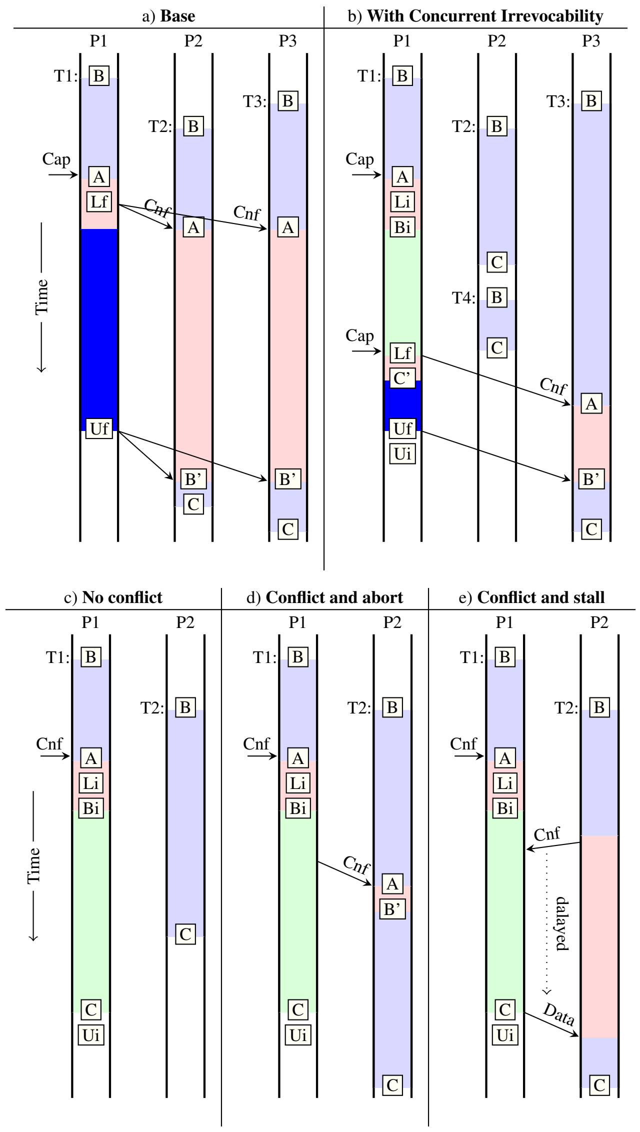

Fig. 3: Behavioral differences between Baseline HTM and CIT (a,b). Handling of contention in CIT (c,d,e). Legend in Fig. 2.

3.1 Overview

CITs differ from speculative transactions in the following aspects:

- A CIT is *not speculative*. It runs in transactional mode to reuse the same hardware mechanisms as a speculative transaction —mainly, the ability to track read-write sets for conflict detection— but it always executes to completion, even in the presence of conflicts, page faults, interrupts or evictions of speculatively modified (SM) blocks from L1 cache. Coherence traffic stemming from CITs is not marked as speculative, unlike messages generated by speculative transactions, which have slightly different protocol behavior to support speculative versioning in L1 caches (*i.e.*, conditional invalidation of SM blocks on abort).
- A CIT performs an *early commit* in response to those events that would have raised the abort signal in speculative transactions. An early commit consists of two steps: first, the processor writes the 64-bit value 1 to an address specified as operand at the beginning of the CIT. The expected use is that such address corresponds to the fallback lock that all speculative transactions must subscribe to in a best-effort HTM system, as described in Listing 1. When the write that acquires the lock completes (*i.e.*, all subscribed speculative transactions have aborted), the actions of a regular transaction commit are carried out (clear read/write sets, etc.) and execution continues non-transactionally.
- Page faults do not abort a CIT. Instead, upon detection of a page fault or interrupt, a CIT self-initiates an early commit as described above. Interrupt/fault processing begins after the lock has been acquired and the core is back to non-transactional execution.
- Conflicting requests do not raise the abort signal. The L1 cache controller running in CIT mode monitors coherence traffic to detect remote conflicting requests and delays the response to accesses from concurrent speculative transactions until the core running the CIT exits transactional mode.

Fig. 3 a) and b) compare the behavior of existing best-effort HTMs (*Base*) against our proposal, when a given transaction (*T1*) aborts because of speculative buffering capacity limits (*Cap*). In the baseline, the fallback lock is immediately acquired, causing conflict-induced aborts of all concurrent speculative transactions subscribed to the lock (*T2* and *T3*), which must wait until the lock is released (red fraction) to restart their execution. In our scheme, the redesigned software abort handler (shown in Listing 2) makes *T1* first acquire an *irrevocability lock* (*Li*) to ensure that no other CIT exists, and then re-execute as a CIT (*Bi*). This allows other concurrent speculative transactions (*T2*, *T3* and *T4*) to coexist with *T1*, allowing unlimited commits (*T2*) and new transactions to begin (*T4*), for as long the CIT runs within the hardware limits and without encountering operating system events. Upon abort signal, the CIT self-initiates an early commit (*C'*), the fallback lock (*Lf*) is acquired (aborting *T3*), and from then on *T1* continues execution in mutual exclusion (blue fraction in *T1*) until the non-speculative transaction completes, releases both fallback and irrevocable locks (*Uf*, *Ui*) and lets *T3* restart.

Fig. 3 c) to e) show some of the key characteristics of our proposal: in c) we can see the ability of a CIT to run concurrently with other speculative transactions whose read-write sets are disjoint, making our design better at tolerating contention. CITs only abort truly conflicting speculative transactions, as shown in d), overcoming the pathological effect of *friendly fire* [17] seen in the underlying requester-wins HTM system. Furthermore, as shown in e), CITs capture a desirable behavior of requester-stalls systems like LogTM [12], [18], which can resolve conflicts without the need to discard the work done up to the offending access.

Fig. 4 compares the key behavioral differences between CIT and the most closely related proposals from the literature, namely lazy irrevocability (*LazyIrr*) [10] and power transactions (*Power*) [11]. In Fig. 4 a) and b) we can see that while neither scheme precudes concurrency during irrevocable transactions, the approach adopted by LazyIrr limits available parallelism since it must conservatively block *all* committing speculative transactions to ensure that the lazy irrevocable transaction commits before. In contrast, CIT does not impose any particular commit order, allowing an unlimited number of concurrent speculative transactions. As depicted in Fig. 4 c) and d), the main difference between CIT and Power lies in the non-speculative nature of CITs, whereas power transactions are elevated-priority yet still speculative transactions that require a non-speculative fallback path (similar to Base) to deal with page faults or capacity limits.

3.2 Application Binary Interface (ABI)

The redesigned wrapper functions to provide concurrent irrevocability transparently to application programmers are shown in Listing 2. As we can see, the `beginTransaction` function is nearly identical to that in Listing 1 except for lines 16 and 17. The common part of both code snippets works as follows: the value returned by `xbegin` (*ret*) is checked to distinguish between newly started and aborted transactions. In the latter case, the hardware has just restored the architectural state at `xbegin`, and set the program counter to the instruction following `xbegin` plus a PC-relative immediate offset to the abort handler (typically set to 0). The returned code also contains the abort status bits which indicate whether the transaction may not succeed on retry (*e.g.*, for capacity-induced aborts). In this case, or if the maximum number of retries has been exceeded, the abort handler decides to resort to non-speculative execution. From this point on, our implementation of the fallback path differs from that in Listing 1. Rather than immediately acquiring `fallbackLock`, the acquisition of the `irrevocabilityLock` (line 16) ensures that at most one CIT exists in the program. Note that the role of the fallback lock remains unchanged (used for eager subscription, acquired by non-speculative transactions to instantly achieve mutual exclusion). Then, a new instruction called `ixbegin` executes (line 17) whose memory operand is the address of `fallbackLock`, *i.e.* a lock variable, currently unlocked (*i.e.*, a value of zero) which all speculative transactions must subscribe to. The following subsection describes the ISA extensions that enable CITs (`ixbegin` and `xend`) in detail.

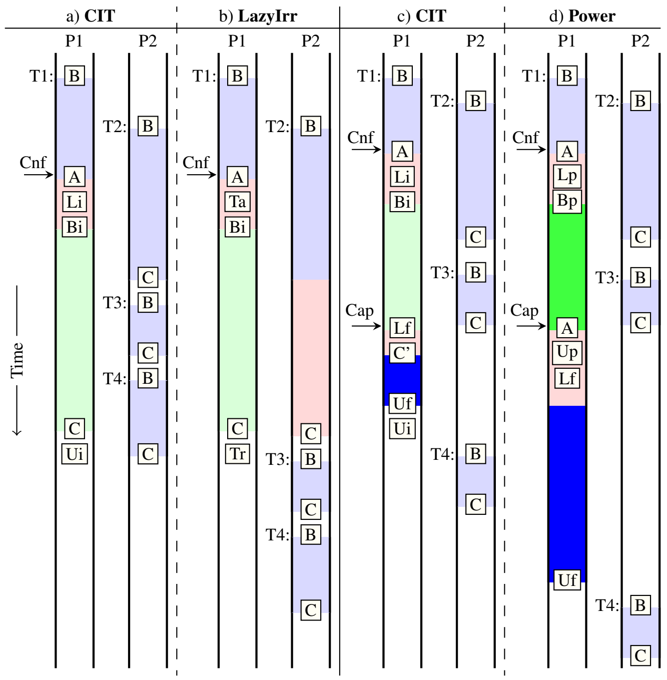

Fig. 4: Behavioral differences between CIT and *Power* and between CIT and *LazyIrr*. Legend in Fig. 2.

Listing 2: Implementation of the fallback path using the proposed ISA extensions for concurrent irrevocability.

```
1  void beginTransaction() {
2    int ret, nretries = 0;
3    do {
4      ret = _xbegin();
5      if (transactionHasStarted(ret)) {
6        if (!isLocked(fallbackLock))
7          return; // Execute speculatively
8        else _xabort();
9      }
10     // Fallback path after abort starts here
11     while (isLocked(fallbackLock)) idle();
12     ++nretries;
13   } while (canSucceedOnRetry(ret) &&
14            !tooManyRetries(nretries)));
15   // Execute non-speculatively
16   acquireLock(irrevocabilityLock);
17   ixbegin(fallBackLock);
18 }
19
20 void commitTransaction() {
21   int ret = _xend(tag);
22   if (ret == _XEND_LOCKED) {
23     releaseLock(fallbacklock);
24     releaseLock(irrevocabilityLock);
25   } else if (ret == _XEND_UNLOCKED) {
26     releaseLock(irrevocabilityLock);
27   } else (ret == _XEND_COMMITTED) {
28     // Nothing to do
29   }
30 }
```

The `commitTransaction` function is a bit different from that in Listing 1 as a result of the *try-or-else* semantics of `xend` required to support concurrent irrevocability. Depending on the returned commit status code, different actions need to be taken at commit time, if any. Speculative transactions return `_XEND_COMMITTED`, indicating that no further steps are needed. Otherwise, the committing transaction is a CIT that may or may not have acquired the fallback lock. If the CIT performed an early commit (as indicated by the returned value `_XEND_LOCKED`), the fallback lock needs to be released. In either case, the irrevocability lock also needs to be unlocked.

It is important to note that, unlike previously proposed schemes of irrevocability found in the HTM literature [10], ours does not require special hardware support to ensure that a single irrevocable transaction exists. Moreover, our design does not limit the number of CIT that can coexist at hardware level. As a result, the hardware complexity to support CITs is clearly lower than that of prior work, as a consequence of our hardware-software codesign that uses existing synchronization mechanisms (locks) in conjunction with slightly augmented hardware support. Listing 2 represents the simplest use case for CITs, where all speculative transactions are subscribed to the same fallback lock. Though in this case a single *irrevocabilityLock* is used to ensure that at most one CIT can execute at a time, the ABI does not impose any limits on how many CITs can execute concurrently, opening up opportunities for software optimizations in the abort handler. The general requirement for correctness is that programmers must allow at most one CIT per fallback lock. Following this rule, multiple fallback locks could be employed (conveniently protected by their respective irrevocability lock) if the programmer or compiler can ascertain which transactions never have overlapping accesses. In this hypothetical scenario, transactions could safely subscribe to different locks without risking atomicity violations, taking further advantage of CITs. Note that in the event that different fallback locks were mistakenly used for two transactions that end up having conflicting accesses, atomicity would be at risk when either thread takes the fallback path and executes the transaction non-speculatively, since the acquisition of one fallback lock by one will not cause the abort of speculative instances of the other (subscribed to another lock). In such circumstances, the speculative transaction could, for instance, read an intermediate update by the non-speculative thread and still be able to commit, hence violating atomicity.

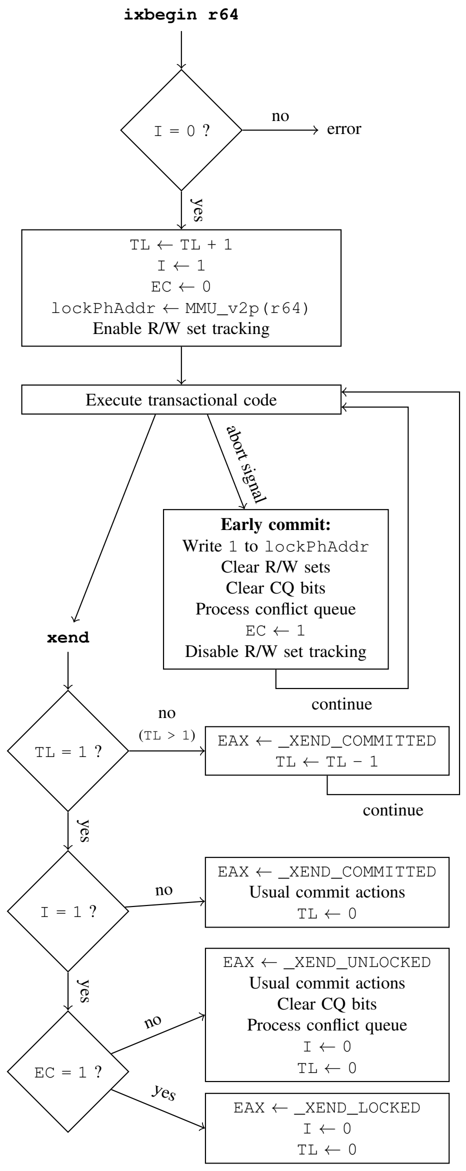

Fig. 5: Flow diagram for `ixbegin` and `xend`.

3.3 ISA extensions

The behavior of the instructions that support concurrent irrevocability is illustrated in Fig. 5 and described next.

`ixbegin r64`. This instruction takes a 64-bit register operand that contains the virtual address of a 64-bit value. It increments the *transaction level* (*TL*), sets the *irrevocable* bit ($I = 1$) to indicate that the CPU is running an irrevocable transaction and resets the *early commit* bit ($EC = 0$). Both *I* and *EC* bits are extra bits added by our proposal, as explained in Section 3.4. The *TL* register is used in RTM to support nested transactions through flattening: TL is respectively incremented and decremented by `xbegin` and `xend`, so that a value greater than zero indicates that the CPU is executing a speculative transaction. The role of TL is slightly changed by our design: $TL > 0$ means the CPU is executing a transaction, but its nature may be either speculative ($I = 0$) or irrevocable ($I = 1$). When in irrevocable mode, $EC = 0$ means that concurrency with speculative transactions is allowed. While in a CIT ($I = 1$ and $EC = 0$), the transactional status bit of each in-flight instruction is set (as if in a speculative transaction) to leverage read-write set tracking for detection of remote conflicting accesses as well as overflow of speculatively modified blocks from L1 cache. Unlike `xbegin`, `ixbegin` does not take a register file checkpoint nor does it need a 32-bit relative offset that points to the abort handler, since CITs are guaranteed to commit in hardware. Instead, `ixbegin` takes the virtual memory address contained in its register operand and sends it to the memory management unit (MMU). The instruction can commit once the translated physical address is stored in a dedicated control register (`lockPhAddr` in Fig. 5). Subsequent executions of `xbegin` with $I = 1$ simply increment *TL* as in RTM so as to support transaction nesting; however, it is an error if *I* was already set before executing `ixbegin`. If the abort signal is raised with $I = 1$, the CPU self initiates early commit by signalling the MMU to automatically perform a write access with a value of 1 to the physical address stored in the `lockPhAddr` register. When such

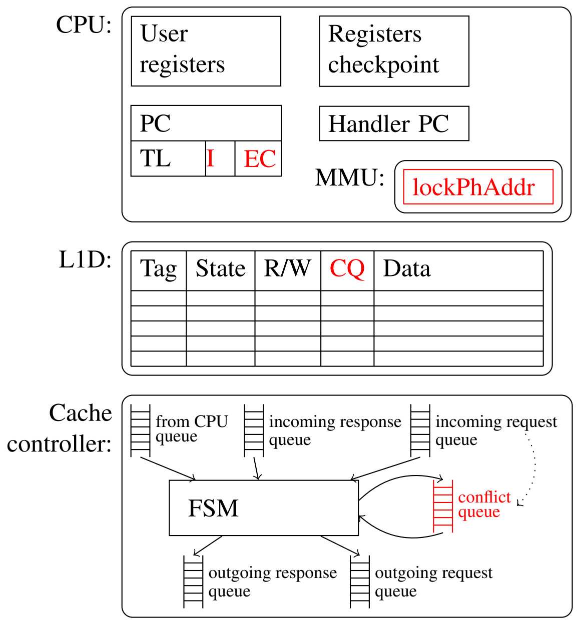

Fig. 6: Hardware changes to support CIT (highlighted in red).

hardware-generated store completes in cache (*i.e.*, the fallback lock has been acquired), the CPU clears the read/write sets and sets the *early commit* (*EC*) bit, disabling read/write set tracking.

`xend`. We extend the behavior of this RTM instruction so that it is used to commit the transaction currently running on the CPU, whether it is a speculative ($I = 0$) or an irrevocable ($I = 1$) transaction. As specified by the RTM interface, this instruction does not take any input operand nor does it return any value/set any flags. We propose to extend its interface so that the outcome of the commit operation is returned via `EAX` register in a similar way to how `xbegin` works. As shown in Fig. 5, the CPU checks TL upon execution of `xend`. If $TL > 1$ (nested transaction commit), then no actions are required other than decrementing TL, and `_XEND_COMMITTED` is returned. If $TL = 1$, then the CPU is about to commit the outermost transaction, which in turn may be speculative or irrevocable: if $I = 0$, the typical actions to commit speculative updates are performed, and `_XEND_COMMITTED` is placed in `EAX`. If $I = 1$, then the CPU is running an irrevocable transaction, and the *EC* bit indicates whether it has encountered an event that made it commit early. If $I = 1$ and $EC = 0$, then CPU proceeds in exactly the same way as for speculative transactions (*i.e.*, read-write sets are cleared), the *I* bit is cleared and `_XEND_UNLOCKED` is returned in EAX. If `xend` finds $I = 1$ and $EC = 1$, it means the irrevocable transaction performed an *early commit*. No commit actions are required other than clearing the *I* bit and setting the value of `EAX` with `_XEND_LOCKED` to indicate that the memory quadword at the address passed to `ixbegin` was set to 1.

3.4 Hardware implementation

The hardware changes to support CITs are ilustrated in Fig. 6 and summarized below.

CPU status. An additional control register is needed to keep the physical address obtained once the memory operand passed to `ixbegin` is translated by the MMU. Apart from this, the CPU needs to maintain two extra bits, the *irrevocable bit* (*I*) and the *early commit bit* (*EC*), as part of the transactional status, which is considered together with the transaction level (TL) when handling `xbegin` and `xend`, as described above.

Coherence protocol. CITs are implemented without behavioral modifications to the coherence protocol (*i.e.*, no additional states), and thus may result more compelling to chip manufacturers than prior proposals that rely on *nacks* [11] or dedicated communication protocols [10]. Our design introduces a new bit in both coherence request and response messages: requests must carry a *speculative* bit similar to [11], and responses a *plea* bit similar to [19]. The *speculative* bit is set when the originating core is running a speculative transaction (*i.e.*, $TL > 0$ and $I = 0$), and its use is described in Section 3.7. The use of the *plea* bit is explained in Section 3.8. Through these bits, our design seemingly maintains the same *requester-wins* nature of existing HTM implementations and guarantees that all requests *eventually* get serviced, while selectively using a *best-effort requester-stalls* policy under the hood for conflicting speculative requests.

L1 cache controller. To realize such *dual* conflict resolution policy (*i.e.*, requester-wins in regular speculative transactions and requester-stalls in CITs) while avoiding deadlocks, L1 cache controllers are augmented with an additional message queue called *conflict queue*, while the per cache entry transactional metadata in L1 cache, *i.e.*, the *speculatively modified* (SM) bit, is extended in one bit: the *conflict queued* (CQ) bit. The conflict queue is used to maintain coherence messages received from remote speculative transactions: conflicting requests are inserted into the conflict queue when the CPU is running a CIT ($I = 1$), and removed from it for regular processing by the cache controller either upon execution of `xend` with $TL = 1$, or because of an abort signal that causes an early commit. The behavior of the L1 cache controller running speculative transactions ($I = 0$) remains unmodified. When in irrevocable mode ($I = 1$), if the L1 cache controller receives a conflicting coherence request from a speculative transaction, it sets the CQ bit of the requested cache block, moves the coherence request to the conflict queue and does not send any response. Non-transactional conflicting requests received by CITs are serviced as if no conflict was detected and thus violate the atomicity of the CIT, as explained in Section 3.6.

L1 cache entry. The additional CQ bit acts as a protocol-level deadlock avoidance mechanism, as it is used to detect write misses and replacements during the execution of the CIT, which cannot be processed by the directory until the coherence controller running the CIT resumes processing the messages delayed in its conflict queue. To this end, the CQ bit keeps track of those L1 cache blocks for which an outstanding conflicting request exists in the conflict queue. As a result of the response being delayed, the directory entry for the cache block remains in a transient coherence state and it cannot process subsequent coherence requests to the same block address — including those from the L1 cache running the CIT itself. To break deadlocks, the CQ bit is checked upon L1 upgrade misses (*i.e.*, stores to S-state lines in MESI) as well as on replacements of E- or M-state blocks: if $CQ = 1$, then the CIT self-initiates an early commit and begins the acquisition of the lock. Once the auto-generated store access to the lock has completed, an early commit takes place: the read and write sets are cleared along with the CQ bits in L1 cache, and the L1 cache controller begins processing the messages buffered in the conflict queue. The coherence message resulting from the miss or replacement of the CQ block that triggered the early commit is processed as usual: it will be eventually processed by the directory after the delayed conflicting remote request has obtained data/invalidation from the early-committed CIT.

Out-of-order core considerations. When CIT support is implemented on out-of-order cores, raising the abort signal while $I = 1$ causes a squash of all inflight instructions, much like a regular abort. Once the reorder buffer (ROB) and the load queue are empty, the autogenerated store to the lock address is inserted into the store queue (SQ) and immediately sent to cache, since its effective target address and stored value are readily available. The SQ contains stores already retired from the ROB awaiting writeback to cache, which are not suppressed when $I = 1$, but rather cleared from their transactional status to avoid setting the SM bit in L1 cache. Note that the autogenerated store access must overtake any pending stores to correctly deal with early commits caused by accesses to CQ blocks: it is possible that a preceding retired store has targeted a CQ block in S-state, in which case the conflicting store will not be able to complete until the early commit is complete. Once the lock is acquired, the remote conflicting request buffered in the conflict queue will be responded, the directory unblocked and then the overtaken store to the former CQ block will complete. In spite of such reordering, the memory consistency model is not affected, since the store to the fallback lock does not have any particular ordering with respect to those program stores that are awaiting writeback.

Read signatures. When signatures are used for read-set tracking, conflicts may be detected on requests to blocks that are no longer cached in L1, either due to false positives or to previous silent evictions. In these cases, the CQ bit is unavailable and thus the response cannot be delayed; instead, the CIT self-initiates an early commit and once completed, the request gets serviced.

Replacement algorithm. To minimize early commits caused by the eviction of a CQ block, the L1 replacement logic is made aware of the CQ bits in the cache set, so that a CQ entry is only selected as a victim when no other entry has the CQ bit unset.

3.5 Interactions with Virtual Memory

Programmers of TM runtimes/libraries using CITs must be aware of the interactions with the virtual memory system, since the processor expects the virtual memory address pointed by the operand passed to `ixbegin` (*i.e.*, the lock variable) to remain mapped at the same physical memory address for the duration of the transaction: the CPU must be able to write to that page immediately after it self-initiates early commit, without kernel intervention. To achieve this goal, we opt for a hardware-software codesign approach.

On the software level, the kernel needs to provide user processes with support for ensuring that some virtual memory pages remain mapped in physical memory and keeping a fixed physical address. The existing *mlock* system call only guarantees that the page will be physically present in RAM, but not necessarily in a fixed frame. Linux kernel developers have released patches that pave the way to make memory-pinning available to user space through a *mpin* system call [20], which would be called at TM runtime initialization for the page containing the fallback lock.

On the hardware level, the execution of `ixbegin` causes its memory operand to be passed to the memory management unit (MMU) for address translation—but no access is performed at this point. The `ixbegin` instruction can only retire once the translated physical address has been stored in an internal CPU control register. Hardware page table walks triggered by a TLB miss at this point can cause page faults, since the parts of the multi-level page table required to get to the appropriate page table entry may not be paged-in. Since such page faults occur in non-transactional code, they are handled as usual by trapping to the OS and re-executing the faulting instruction after returning from kernel code. Note that the CPU does not actually enter transactional (irrevocable) state until `ixbegin` retires.

3.6 Weak atomicity guarantees

In spite of early commits that allow remote conflicting requests to complete, atomicity of a CIT is always guaranteed by the acquisition of the lock that causes all remote speculative transactions to abort, including those that temporarily breached isolation of the irrevocable transaction. Note that the write to the lock upon early commit becomes globally visible before the delayed conflicting requests are processed, which means that all speculative transactions are already in the process of aborting (because of a conflict on the lock) by the time their conflicting coherence requests receive data or the last acknowledgment from the early-committed irrevocable transaction. Should a remote conflicting request be non-transactional, atomicity of the CIT would be violated. Nonetheless, such inability to provide strong atomicity for non-speculative transactions is a common limitation of all best-effort HTM implementations proposed to date, including those commercially available: in spite of their inability to detect conflicting accesses between a non-speculative transaction holding the fallback lock and non-transactional accesses, the weak atomicity model offered by current HTMs is sufficient for most codes [21].

3.7 Programs not compliant with the CIT ABI

As stated by its ABI, programmers using CITs are required to subscribe all speculative transactions with potential data races to the same fallback lock. The hardware does not restrict how many CITs can run concurrently at any time: it is the programmer's responsibility to ensure that a single CIT can run at any time.

Programs not compliant with such ABI may produce incorrect results. The atomicity of a CIT is only guaranteed with respect to other speculative transactions, but not with respect to non-transactional accesses nor other CITs (as explained above, weak atomicity is a well-known limitation of any best-effort HTM). However, in no case would this lead to a deadlock, as CITs are never allowed to stall conflicting memory accesses coming from a non-speculative requester (*i.e.*, another CIT or a non-transactional thread): only those conflicting requests coming from speculative transactions are subject to CIT's *under the hood* requester-stalls policy. Such selection of conflict policy can be done based on the *speculative* bit annotation that coherence request messages introduced by our proposal. Since coherence messages originating from CITs are not marked as speculative, a mutual race between two or more CITs can never result in a deadlock. There are other examples of programs where CITs are incorrectly used, thus threatening atomicity. For instance, if a conflicting request from a speculative transaction not subscribed to the (same) lock is stalled by a CIT, and the latter then performs an early commit, the stalled request will be responded and the data used by a transaction that has not been aborted as a result of the acquisition of the lock, therefore violating atomicity. Notwithstanding, the aforementioned examples of incorrect use of CIT are largely irrelevant to application developers, which will simply continue to link their codes to the TM runtime/library. Only developers of TM runtimes/libraries using TM must pay attention when leveraging hardware support for CIT when implementing the abort handler.

3.8 CIT-specific hardware capacity limits

A critical part of our design is how to handle the uncommon case of conflict queue fill-up to guarantee deadlock freedom in spite of running into such capacity limits. Empirical observation running the STAMP benchmarks in our simulation environment shows that the number of distinct simultaneously conflicting addresses seen by any transaction is small, in most cases less than sixteen. This number not only depends on the workload characteristics, but also on the type and number of processing cores —modern processing cores can maintain many speculative memory accesses in flight and thus cause more simultaneous conflicts than less aggressive cores. Regardless of these factors, the hardware must ensure deadlock-freedom and preserve atomicity amonst transactions at all times. Our proposal allows conflicting requests that cannot be accommodated in the conflict queue to be serviced as if no conflict had occurred (*i.e.*, falling back to *requester-wins* policy), and handles this uncommon case using a solution similar to the *plea* bit scheme proposed by Park *et al.* [19]: coherence response messages are augmented with an extra bit annotation that simply forces the abort of the requester. Such additional annotations do not entail any behavioral change to the coherence protocol, and are simply transported in *ack*/data messages so that the destination L1 cache controller raises the abort signal when such bit is found set. Non-transactional conflicting requests are always immediately responded and thus in such cases the atomicity of the CIT is not guaranteed (the *plea* bit is anyways ignored by the requesting cache). Note that this is no different from data races that may occur between a non-transactional thread and a non-speculative transaction holding the fallback lock in existing HTM systems.

3.9 Limitations

As shown in Fig. 5, the `ixbegin` instruction cannot be used while already executing an irrevocable transaction ($I = 0$). That is, a CIT cannot be nested within another CIT. However, a system that uses CITs can easily support nested transactions, since `xbegin` is supported within CITs (it increases TL as within speculative transactions). The runtime must ensure using `xbegin` to start a nested transaction when the outer one is already irrevocable.

`ixbegin` receives the address of the fallback lock at the beginning of the transaction and the system will automatically write a 1 to that address when the CIT needs to become exclusive. This means that the fallback lock used by a system using CITs needs to be a single word and should be acquired that way. The rest of the locks used by the system are not affected (i.e., mutexes from pthreads or other libraries can be used).

4 Simulation Environment

We have extended the widely used Gem5 simulator [6] with transactional memory support, in order to model a variety of HTM implementations ranging from best-effort solutions to *full-blown* virtualized designs like LogTM [18]. The latter approaches the charateristics of an ideal HTM implementation and it serves as an upper performance bound that helps to quantify how our proposed design bridges the gap between best-effort requester-wins HTM designs and more complex implementations.

The full-system simulation mode of Gem5 is employed to capture the effects of *RTM-unfriendly* operating system events on the performance of transactional workloads. We use the detailed timing model for the memory subsystem provided by Ruby, combined with the detailed out-of-order CPU model known as *O3CPU*. Gem5 provides functional simulation of the x86-64 ISA and boots an unmodified Gentoo Linux with kernel version 3.2.24. We perform our experiments on a 16-core tiled CMP system, as described in Table 1. Each tile contains a processing core with private L1 instruction and data caches, and a slice of the shared L2 cache with associated directory entries. A 2-D mesh NoC is employed to interconnect the tiles. The L1 caches maintain inclusion with the L2. The private L1 caches are kept coherent through an on-chip distributed directory (associated with the L2 cache banks), which maintains presence-bit vectors of sharers and implements the MESI protocol.

TABLE 1: System parameters.

MESI Directory-based CMP	
Core Settings	
Cores	16 out-of-order, 4-way width
Load queue / Store Queue	72 / 56
Memory Settings	
L1 I&D caches	Private, 32KiB, split
	8-way, 1-cycle hit latency
L2 cache	Shared, 8 MiB, unified
	16-way, 24(tag)+12(data)-cycle latency
Memory	3GB, 200-cycle latency
Network Settings	
Topology and Routing	2-D mesh (4×4), X-Y
Flit size	16 bytes
Message size	5 flits (data), 1 flit (control)
Link latency / bandwidth	1 cycle / 1 flit per cycle

TABLE 2: HTM systems evaluated.

Base	Baseline, perfect read signature, SM-bits in L1 cache
CIT	CIT with 16-entry conflict queue
LazyIrr	Lazy irrevocability [10] with *magic* token
Power	Power transactions [11]
Power+	Power transactions [11], with report of power-induced aborts.
LogTM	LogTM-SE, perfect signatures, 8-entry log filter [18]

HTM systems. Table 2 summarizes the HTM systems evaluated in Section 5. Our baseline is an RTM-like best-effort design that uses *speculatively modified* (SM) bits in the L1 data cache to track write sets, and a *perfect signature* to track read sets. In this way, it can maintain much larger read-sets than write-sets, following the features seen in commercial Intel chips [22], where write-sets cannot exceed L1 size (32 KiB) but read-sets can reach several megabytes. Furthermore, in order to model a baseline that uses the L1 data cache for speculative versioning like existing best-effort HTM implementations do, the standard MESI coherence protocol is augmented with speculative versioning support: handling of silent invalidation of M-state SM blocks on abort, and ensuring that M-state blocks are always written back to the shared cache level upon the first speculative write.

We implement support for *concurrent irrevocable transactions* (*CIT*) on top of the HTM baseline described above, without changing the coherence protocol. For the evaluated benchmarks and inputs, a conflict queue of sixteen entries suffices to hold pending conflicting requests without running into capacity issues.

We compare CIT to two recent proposals that are most related, whose aim is also to improve performance of best-effort HTM designs. We have faithfully implemented power transactions (*Power*) as described by Dice *et al.* [11], where software-controlled entry into power mode was employed and the coherence protocol was modified to support *nacks*. Additionally, we consider an optimized flavour of power transactions (*Power+*) which overcomes a pathological performance that we observed in *Power*. In particular, its cause was the heuristic proposed by the authors to mitigate the *lemming effect* [23]: if a regular transaction aborts as a result of a conflict and finds that a power transaction exists (the *power flag* is set), then it assumes that its *killer* was the power transaction and it does not count this abort towards the maximum number of retries threshold for switching to power mode. If the heuristic turns out to be wrong (*i.e.*, the *killer* was a regular transaction concurrent with a power transaction), then transactions may take many more retries to switch to power mode than they ought to. As shown in Section 5, this unexpected behavior results in Power performing worse than Base in several benchmarks. Instead of relying on a heuristic, in Power+ the abort handler is provided with precise information about power-induced aborts, using an additional abort status bit in the value returned via `EAX` to indicate whether the conflict was with a power transaction.

Moreover, we model an idealized implementation of the lazy irrevocability mechanism (*LazyIrr*) proposed by Quislant *et al.* [10], where the irrevocability token is magically handled with zero-latency of acquisition and no network traffic generated, through a global shared flag directly accessed by all simulated CPUs on `xbegin`/`xend`. Therefore, the results presented for this system do not account for the overheads associated to the proposed token communication protocol.

For fair comparison across best-effort designs evaluated, all of them check the abort status *retry* bit returned on abort, which indicates whether the transaction may succeed on retry. Furthermore, they follow the Intel RTM specification in regard to interrupts/faults in speculative transactions, as opposed to [10].

We compare the relative performance of the aforementioned best-effort designs, against LogTM-SE [18], a popular HTM system that provides *virtualized* transactions of any footprint or duration. LogTM-SE uses a requester-stalls policy to resolve conflicts through a timestamp-based scheme of deadlock avoidance.

Benchmarks. The STAMP transactional benchmarks [5] with recommended small and medium ('+') inputs are used as workloads. The results presented in our evaluation are for 16-thread runs. Fig. 7 shows the scalability up to 16 threads for our baseline and LogTM, where we can see the poor parallel performance of bayes and labyrinth in all HTM systems. The large write-set size of their transactions in both codes, well above 32 KiB for medium inputs, leads to poor performance when executed in best-effort HTM systems due to capacity-induced aborts, as it can be seen in Fig. 7. Moreover, execution time of bayes varies significantly in different executions, as it implements a hill climbing search algorithm that, depending on the thread interleaving, can execute a different number of transactions for the same input [3], [24], as clearly seen in Fig. 7. In labyrinth, the reason that lies behind its poor scalability is the lack of hardware support for *early release* [3], [25], which prevents its main data structure from being removed from the read set after privatization, so that any thread attempting to commit its work invariably aborts all other concurrent transactions. Morover, the even more pathological performance of LogTM in labyrinth is explained by the large write set size of transactions that repeatedly abort, which not only makes the undo log compete for cache resources with program data, but also causes very expensive software rollbacks.

All the results correspond to the parallel part of the applications and we have accounted for the variability of parallel applications. For each workload-configuration pair we gather average statistics over 10 randomized runs, by adding a random jitter of up to 1 extra cycle to the DRAM response time. Threads were pinned to cores in order to avoid migration. The results shown in the following section are normalized to the baseline system.

5 Performance Evaluation

To showcase the potential performance gains of using CIT in programs where threads encounter varying levels of contention depending on the data accessed, we have developed a microbenchmark with mixed contention levels, whose pseudocode is given in Listing 3 (the `atomic` construct delimits a transaction). With the Base HTM system, frequent conflict-induced aborts experienced by threads executing transaction *(a)* (line 3) repeatedly force threads attempting to execute transaction *(b)* (line 9) to wait on the fallback lock, even though the read-write sets of *(a)* and *(b)* are invariably disjoint. Results are shown in Fig. 8 using *numTrans=1024*, *maxTransLoop=16*, *innerLoops=32* and *numHighContTransPerLoop=8* with random indices over two arrays of 8 and one million 64-bit integers, respectively. On the left side, we can see the scalability of all HTMs considered running such microbenchmark, and on the right the normalized execution time for the 16 threads. Execution time is categorized as follows: holding the fallback lock (*HasLock*); executing speculative transactions that commit (*Committed*); waiting on the fallback lock to be released before retrying a speculative transaction (*WaitForRetry*); and the rest (*Other*) including aborted transactions, non-transactional execution and handling of page faults or interrupts.

We can see that the Base system does not scale well beyond 4 threads, while the rest continue scaling up to 8 threads (Power and Power+) or even 16 (CIT, LazyIrr and LogTM). The factor limiting the scalability of Base is the increase in the *WaitForRetry* time, which CIT largely avoids by allowing concurrency between one thread executing *(a)* irrevocably and several other threads executing *(b)* speculatively. LogTM and especially Power also avoid increasing the *WaitForRetry* time but at the cost of doing additional work (more aborts and more expensive aborts), while our zero-overhead implementation of LazyIrr (as explained in section 4) achieves almost the same scalability in this microbenchmark at the cost of higher complexity.

Listing 3: Pseudocode of the synthetic microbenchmark.

```
1  for (i=0; i < numTrans; i += transLoop) {
2    for (j=0; j < numHighContTransPerLoop; j++) {
3      atomic { // (a) High contention
4        computeNewValues(smallArray, ...);
5      }
6    }
7    transLoop = random() % (maxTransLoop + 1);
8    for (k=0; k < transLoop; k++) {
9      atomic { // (b) Low contention
10       for (j=0; j < innerLoops; j++) {
11         computeNewValues(largeArray, ...);
12       }
13     }
14 }
```

Fig. 9 compares the relative execution time of all six HTM systems considered in this study for all STAMP benchmarks. For the reasons mentioned in Section 4, bayes and labyrinth are disregarded in the rest of this section and excluded from the calculation of *Average**. Furthermore, *Selected Average* also excludes kmeans-l and ssca2 since they barely resort to the irrevocability mechanisms. The role that each source of overhead plays on the performance of each HTM design is depicted by

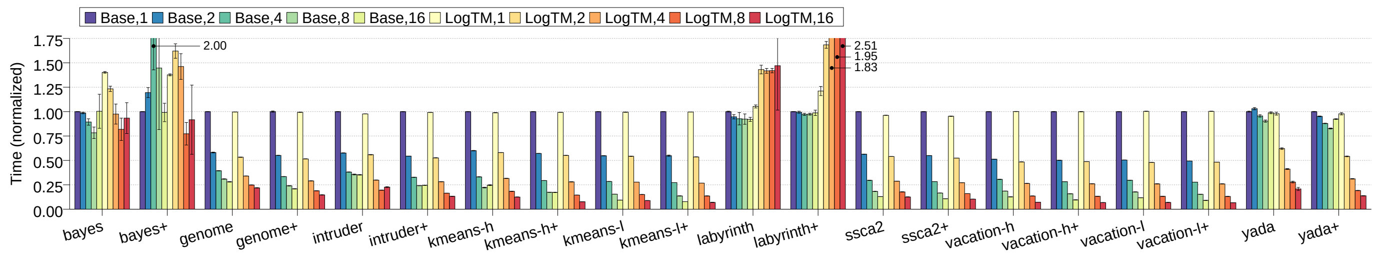

Fig. 7: Scalability of baseline and LogTM HTM systems up to 16 threads.

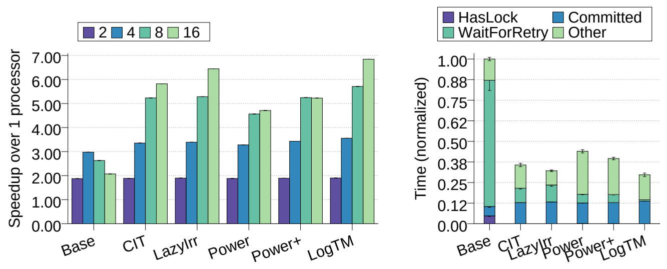

Fig. 8: Execution time for synthetic microbenchmark.

Fig. 10, which breaks down execution time into disjoint components. For brevity, only data for medium inputs ('+') are shown (small input breakdowns are similar). Each execution cycle is attributed to one of the following categories: non-transactional and barrier (*NonTransactional+Barrier*); handling of page faults and interrupts (*Kernel*); holding the fallback lock (*HasLock*); for LazyIrr and LogTM, blocked in speculative transactions due to an ongoing irrevocable transaction —LazyIrr— or a conflict —LogTM— (*Stalled*); executing speculative transactions (*Committed* and *Aborted*); waiting on the fallback lock to be released before retrying a speculative transaction (*WaitForRetry*); for LogTM, restoring the undo log and backing off (*Undo+Backoff*); and cycles in non-speculative transactions that did not preclude concurrency, including both CIT and lazy irrevocable transactions (*ConcurrentIrrevocable*). To quantify the additional parallelism achieved by CIT, LazyIrr and Power in comparison to Base, Fig. 11 shows the fraction of committed cycles from speculative transactions that are (either partially or entirely) concurrent with an irrevocable (or power) transaction. Naturally, this percentage is always 0 in Base. Fig. 12 categorizes aborted transactional cycles per cause of abort, providing a valuable characterization of the interaction between benchmarks and HTM systems that helps this analysis. In this figure, aborted cycles due to conflicts on the subscribed fallback lock (including explicit aborts when the lock is found held) are accounted for in a category of its own (*FallbackLock*). Note how LogTM does not abort transactions for any reason other than conflicts, unlike best-effort HTMs.

Fig. 13 shows the distribution of committed transactional cycles per type of transaction: in regular hardware transactions (*concurrent speculative*); cycles in CITs before early commit, and in lazy irrevocable transactions (*concurrent non-speculative*); cycles holding the fallback lock (*non-concurrent non-speculative*). Cycles spent in power transactions are categorized separately, though conceptually they are part of *concurrent speculative*. We also separate cycles spent stalling before a successful commit, which may occur in LazyIrr while waiting for a lazy irrevocable transaction to end, or in LogTM as a result of its requester-stalls policy. The goal of this plot is to act as an indicator of how much of the useful transactional work executed in mutual exclusion in Base is moved into other concurrent components by CIT and the related works. In Base, all committed transactions are either concurrent speculative or non-concurrent non-speculative. In LogTM, all transactions are concurrent speculative, although the highest priority transaction as per its timestamp could be considered irrevocable as it cannot be aborted by any other. Note how, in spite of its indisputable best performance, committed transactions in LogTM generally take longer than those in best effort HTMs, due to the overhead of logging, particularly in benchmarks with large write sets like yada where the log takes part of available cache resources.

5.1 CIT vs Base

In Fig. 9 we can see that CIT achieves an average reduction in execution time with respect to Base of 15.8% for the selected benchmarks (12.5% overall), with improvements of up to 45.8% in a highly contended benchmark such as kmeans-h+. The reason for such performance gains lies behind CIT's ability to side-step nearly all acquisitions of the fallback lock while still providing irrevocability, thus allowing both speculative and non-speculative work done in parallel in scenarios where Base must resort to mutual exclusion to guarantee atomicity of the irrevocable transaction. Fig. 11 gives an idea of all the valid work that Base discards every time that the fallback lock is acquired, which CIT is able to salvage. We can see how in intruder, up to 56% of all committed transactional cycles belong to transactions that run concurrently with an irrevocable transaction, thus explaining the overall performance improvement seen for this benchmark. As we can observe in Fig. 10, for all benchmarks except yada, CIT virtually eliminates the *HasLock* component seen in Base, which in turn dramatically reduces the serialization of threads because of the fallback lock (*WaitForRetry*), by 38% on average. Fig. 13 shows how nearly all transactional work done in mutual exclusion in Base gets done concurrently in CIT (again, except in yada), either speculatively or non-speculatively, resulting most visible in benchmarks where conflicts are the primary source of aborts (genome, intruder and kmeans-h). As we can observe in Fig. 12, the fraction of aborted cycles due to the fallback lock is reduced in such benchmarks. As a result of the additional concurrency permitted among speculative transactions, in many benchmarks the aborted cycles seen in CIT increase with respect to Base.

The case of yada deserves particular attention: CIT retains around 40% of *HasLock* cycles seen for Base (Fig. 10), because of page faults affecting its very long running main transaction (*regionRefine*): our results for LogTM indicate that such transaction takes on average 90,000 cycles to complete, including handling of eventual page faults. For CIT, experiments reveal that around

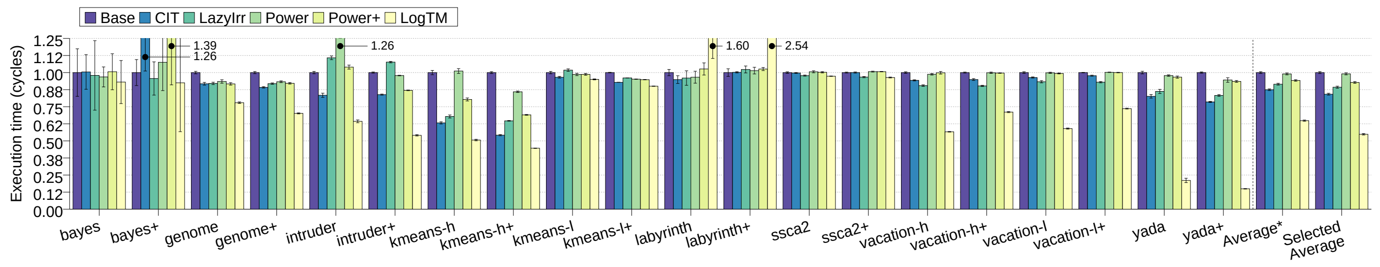

Fig. 9: Normalized execution time. *Average** does not include bayes nor labyrinth.

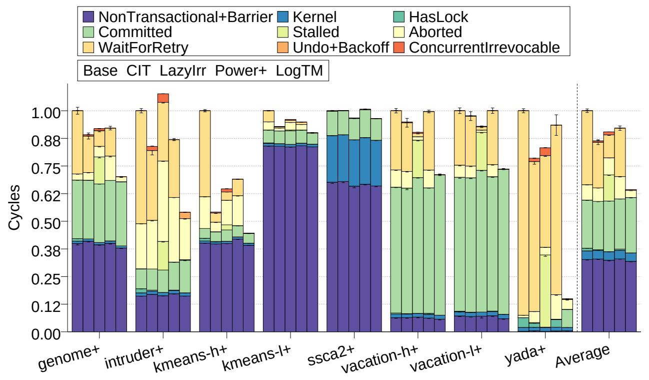

Fig. 10: Normalized execution time breakdown.

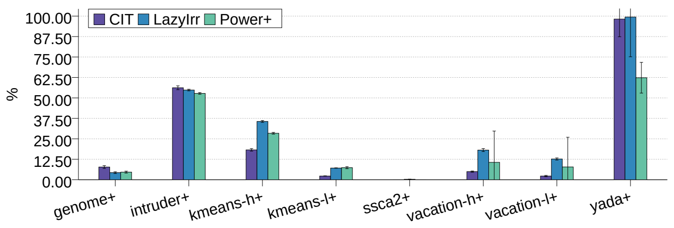

Fig. 11: Percentage of committed cycles from speculative transactions that run concurrently with an irrevocable/power transaction.

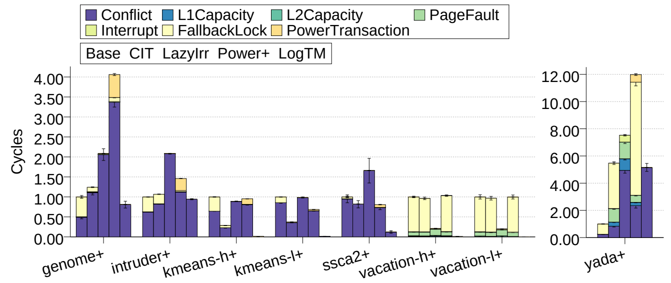

Fig. 12: Normalized aborted cycles, by cause.

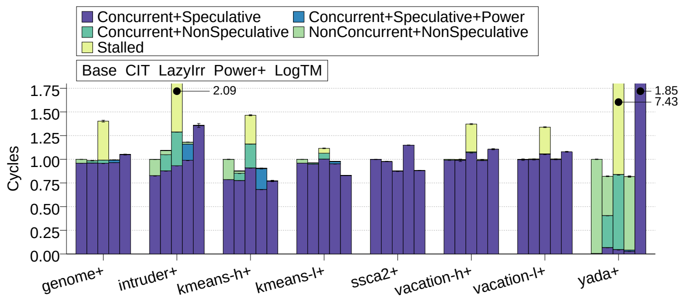

Fig. 13: Distribution of committed cycles, per type of transaction.

90% of the successful executions of yada's main transaction are non-speculative (explaining its poor scalability in all best-effort HTM systems considered), and 12% of its aborts are caused by page faults. Additionally, the fraction of irrevocable transactional cycles executed after a page fault is considerable, explaining why in Fig. 13 we see that still 50% of all committed transactional cycles are still done while holding the fallback lock (*NonConcurrent+NonSpecultive*). Note that CIT increases the aborted cycles caused by page faults, as seen in Fig. 12, which is a consequence of the additional concurrency: in Base, many speculative transactions that are bound to suffer a page fault do not reach the faulting access because of another transaction becoming irrevocable. In CIT, the same speculative transactions can make further progress in parallel with an irrevocable one, only to end up reaching the faulting access and discard more work than Base. Fig. 10 and Fig.13 show that in Base almost all transactional cycles are executed non-speculatively (in mutual exclusion), while CIT performs around 7% of all transactional cycles in concurrent speculative transactions, achieving a reduction in execution time of 21% for yada+. The predominance of non-speculative transactions throughout its execution explains why nearly 100% of all committed speculative transactions are partly or entirely concurrent with a CIT (see Fig. 11).

5.2 CIT vs Power+

This section compares our proposal against Power+, an improved version of Power which addresses the pathological scenario explained in Section 4. As it can be seen in Fig. 10, the performance improvements of Power are very limited and in some benchmarks it is even slower than Base. On the contrary, Power+ performs substantially better than Power and never worse than Base. CIT clearly outperforms Power+ by around 7% on average, due to its combination of irrevocability and concurrency. Because power transactions are speculative, their dual-priority mechanism can only be applied after conflict-induced aborts. Upon page-fault- or capacity-induced aborts, Power+ falls back to non-speculative execution in mutual exclusion for the whole extent of the transaction, just like Base. This inherent limitation of Power+ becomes clearly visible in vacation, for which Fig. 9 shows identical performance to Base, since in this benchmark most aborted cycles are due to page faults, as depicted in Fig. 12. In the same circumstances, CIT allows concurrency for the fraction of the transaction executed up to the early commit, allowing other concurrent speculative transactions to commit before the fallback lock gets acquired, and explaining the performance improvement of CIT compared to Base. According to Fig. 11, in vacation-h+ around 5% of all committed speculative transactional cycles are concurrent with a CIT. Note how, because most concurrent irrevocable transactions eventually resort to mutual exclusion in CIT (upon page fault), the fraction of aborted cycles caused by the acquisition of the fallback lock remains similar to Base in Fig. 12. In regard to yada, page faults are also the reason behind the limited performance gains of Power+ over Base. Those transactions that are bound to suffer a page fault and also suffer repeated conflict-induced aborts before reaching the faulting access, switch to power mode only to discover that irrevocability is required in the end. In this manner, running in power mode becomes futile given the unavoidable switch to irrevocability, which in Power+ takes place much later than in Base or CIT. This effect is visible in the extra aborted cycles (nearly 12× increase over Base shown in Fig. 12), both due to the work done in power mode up to the page fault, and by all other conflicting transactions aborted in favor of an elevated-priority transaction that eventually must abort too.

5.3 CIT vs LazyIrr

When compared to LazyIrr, Fig. 9 shows that CIT's average performance is 4.6% better, even though our idealized implementation of lazy irrevocability does not take into account the latency and network traffic overheads incurred by the *irrevocability token* protocol proposed by Quislant *et al.* In sight of these results, and considering the additional hardware complexity of LazyIrr, CIT reveals itself as a more cost-effective alternative at improving HTM performance through concurrency during irrevocable execution. CIT does so while maintaining eager lock subscription, whereas LazyIrr requires dedicated communication circuitry to overcome the many threats of lazy subscription. Fig. 10 shows that CIT clearly outperforms LazyIrr in benchmarks with frequent conflict-induced aborts (genome, intruder, kmeans-h), where the difference in execution time stems partly from CIT's ability to discard less work by resolving conflicts through stalls rather than aborts, and partly from the appearance of the *Stalled* component in LazyIrr: in CIT speculative transactions may commit without having to wait for an ongoing concurrent irrevocable transaction, whereas in LazyIrr they must stall in order to guarantee its atomicity.

The slowdown relative to the baseline seen in intruder is a quantitative proof of LazyIrr's limits to concurrency in workloads that mix both long- and short-running transactions with high contention. In intruder, a long transaction (*decoder_process*, average duration of 900 cycles in Base) frequently becomes irrevocable and can cause long stalls on other threads executing smaller transactions (*e.g.*, *getComplete*, duration of 100 cycles in Base/CIT), which in turn exposes them to further conflicts as a result of their increased duration. The larger *Aborted* component in LazyIrr in Fig. 10 is due to the increase in the number of aborts of the two shorter transactions compared to Base/CIT, a direct result of their longer duration (*e.g.*, *getComplete* shows a 3× increase in average duration and 6× increase in aborts). Even though LazyIrr nearly halves the number of aborts of *decoder_process*, the adverse effects on the shorter transactions outweigh such an improvement and cause the slowdown with respect to Base seen in Fig. 10. In contrast, CIT does not impose any limits to concurrency.

The marginal improvement seen for LazyIrr over CIT in ssca2 is an artifact of our ideal implementation of LazyIrr: even though none of the HTMs evaluated resort to irrevocability, the overhead of software lock subscription penalizes Base and CIT in comparison to LazyIrr, which magically subscribes to a hardware lock at no cost. The result is that, because of the small transaction size, LazyIrr executes around 30% fewer instructions in each committed transaction than Base or CIT (20 vs 30 instructions).

LazyIrr only achieves a significant improvement over CIT in one benchmark (vacation), which stems from LazyIrr's ability to handle page faults occurring in irrevocable transactions in parallel with other speculative transactions. Since page faults do not result in the fallback lock being acquired in LazyIrr, discarded work in vacation is largely reduced (*i.e.*, LazyIrr eliminates the *Fallback-Lock* component in Fig. 12 with respect to Base, CIT or Power+). As it can be observed in Fig. 11, the consequence of handling page faults in parallel is that for vacation-h, in LazyIrr 18.1% of speculative transactional cycles committed are concurrent with an irrevocable transaction, while in CIT this happens only for around 5.0% of cycles. As opposed to LazyIrr, best-effort HTM systems using eager subscription (Base, CIT and Power+) must handle page faults in mutual exclusion. Although out of the scope of this work, the reader must note that page fault handling could be easily moved out of the critical path of irrevocable transactions in best effort HTM systems with eager subscription, if only the hardware returned the faulting virtual address to the abort handler.

By design, LazyIrr never resorts to mutual exclusion and thus all committed transactional cycles are concurrent (either speculative or non-speculative), as seen in Fig. 13. However, in LazyIrr committed speculative transactions may have been blocked at commit time until a concurrent lazy irrevocable transaction completes: *Stalled* cycles are responsible for the increment in the total cycles spent in committed transactions seen in Fig. 13. Such useful stalled cycles largely dominate committed transactional cycles in yada (as a result of its long running transactions), and also represent an important fraction in genome, intruder, kmeans-h and vacation. These potentially long stalls introduced at commit time are challenging to programmers willing to avoid wasting computational power in scheduled threads that are indeed idle.

6 Conclusions and Future Work

This work proposes Concurrent Irrevocable Transactions (CITs) for best-effort requester-wins HTM. CITs increase the concurrency among transactions by eliminating the need or at least deferring the need of making irrevocable transactions non-concurrent (*i.e.*, avoiding the acquisition of the fallback lock). We show that CIT improves performance with respect to the baseline architecture 15.8% on average in contended benchmarks and 12.5% on average across the STAMP benchmark suite. CIT increases concurrency more effectively than Power Transactions because CIT can execute at least with partial concurrency even transactions affected by page faults or capacity-induced aborts. Also, CIT outperforms Lazy Irrevocability because no other transactions can commit while a lazy irrevocable transaction executes, introducing stall time and reducing concurrency, a limitation which CIT does not have despite its lower complexity.

The ISA and hardware changes needed by CIT are minimal but enable significant performance gains, as shown in this work. As future work, we expect to further increase concurrency between transactions using the same hardware extensions with the help of program annotations and/or the compiler, by allowing the execution of several irrevocable transactions concurrently as long as it can be proven beforehand that they access only disjoint data.

Acknowledgments

This work has been supported by the Spanish MCIU and AEI, as well as European Commission FEDER funds, under grant "RTI2018-098156-B-C53".

References

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

[2] C. Jacobi, T. Slegel, and D. Greiner, "Transactional memory architecture and implementation for IBM System z," in *45th IEEE/ACM Int'l Symp. on Microarchitecture (MICRO)*, 2012, pp. 25–36.

[3] R. Yoo, C. Hughes, K. Lai, and R. Rajwar, "Performance evaluation of intel transactional synchronization extensions for high performance computing," in *ACM/IEEE Conf. on Supercomputing (SC)*, 2013.

[4] Intel Corporation, "Intel 64 and IA-32 architectures optimization reference manual, chapter 15: Intel TSX recommendations," pp. 15:1–15:30, 2019.

[5] C. Cao Minh, J. Chung, C. Kozyrakis, and K. Olukotun, "STAMP: Stanford transactional applications for multi-processing," in *IEEE Intl. Symposium on Workload Characterization*, 2008, pp. 35–46.

[6] N. Binkert, B. Beckmann, G. Black, S. K. Reinhardt, A. Saidi, A. Basu, J. Hestness, D. R. Hower, T. Krishna, S. Sardashti *et al.*, "The gem5 simulator," *ACM SIGARCH Computer Architecture News*, vol. 39, no. 2, pp. 1–7, 2011.

[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," in *16th Int'l Conf. on Architectural Support for Programming Language and Operating Systems (ASPLOS)*, 2011, pp. 39–52.

[8] I. Calciu, T. Shpeisman, G. Pokam, and M. Herlihy, "Improved single global lock fallback for best-effort hardware transactional memory," in *9th ACM SIGPLAN Workshop on Transactional Computing*, 2014.

[9] D. Dice, T. L. Harris, A. Kogan, Y. Lev, and M. Moir, "Pitfalls of lazy subscription," in *Proceedings of the 6th Workshop on the Theory of Transactional Memory (WTTM)*, 2014.

[10] R. Quislant, E. Gutiérrez, E. L. Zapata, and O. G. Plata, "Lazy irrevocability for best-effort transactional memory systems," *IEEE Transactions on Parallel and Distributed Systems (TPDS)*, vol. 28, no. 7, pp. 1919–1932, 2017.

[11] D. Dice, M. Herlihy, and A. Kogan, "Improving parallelism in hardware transactional memory," *ACM Transactions on Architecture and Code Optimization (TACO)*, vol. 15, no. 1, pp. 9:1–9:24, 2018.

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

[13] S. Park, C. J. Hughes, and M. Prvulovic, "Transactional pre-abort handlers in hardware transactional memory," in *27th Int'l Conf. on Parallel Architectures and Compilation Techniques (PACT)*, 2018, pp. 33:1–33:11.

[14] M. Mohamedin, R. Palmieri, A. Hassan, and B. Ravindran, "Managing resource limitation of best-effort htm," *IEEE Transactions on Parallel and Distributed Systems (TPDS)*, vol. 28, no. 8, pp. 2299–2313, 2017.

[15] Y. Afek, A. Matveev, O. R. Moll, and N. Shavit, "Amalgamated lock-elision," in *Int'l Symp. on Distributed Computing (DISC)*, 2015, pp. 309–324.

[16] D. Dice, A. Kogan, and Y. Lev, "Refined transactional lock elision," *ACM SIGPLAN Notices*, vol. 51, no. 8, p. 19, 2016.

[17] J. Bobba, K. E. Moore, L. Yen, H. Volos, M. D. Hill, M. M. Swift, and D. A. Wood, "Performance pathologies in hardware transactional memory," in *34th Int'l Symp. on Computer Architecture (ISCA)*, 2007, pp. 81–91.

[18] L. Yen, J. Bobba, M. R. Marty, K. E. Moore, H. Volos, M. D. Hill, M. M. Swift, and D. A. Wood, "LogTM-SE: Decoupling hardware transactional memory from caches," in *13th Int'l Symp. on High-Performance Computer Architecture (HPCA)*, 2007, pp. 261–272.

[19] S. Park, M. Prvulovic, and C. J. Hughes, "Pleasetm: Enabling transaction conflict management in requester-wins hardware transactional memory," in *22nd Int'l Symp. on High-Performance Computer Architecture (HPCA)*, 2016, pp. 285–296.

[20] J. Corbet. (2014) Locking and pinning. [Online]. Available: https://lwn.net/Articles/600502/

[21] L. Dalessandro and M. L. Scott, "Strong isolation is a weak idea," in *4th ACM SIGPLAN Workshop on Transactional Computing*, 2009.

[22] B. Goel, R. Titos-Gil, A. Negi, S. A. McKee, and P. Stenstrom, "Performance and energy analysis of the restricted transactional memory implementation on haswell," in *28th Int'l Parallel and Distributed Processing Symp. (IPDPS)*, 2014, pp. 615–624.

[23] D. Dice, Y. Lev, M. Moir, and D. Nussbaum, "Early experience with a commercial hardware transactional memory implementation," in *14th Int'l Conf. on Architectural Support for Programming Language and Operating Systems (ASPLOS)*, 2009, pp. 157–168.

[24] A. Dragojevic and R. Guerraoui, "Predicting the scalability of an STM," in *5th ACM SIGPLAN Workshop on Transactional Computing*, 2010.

[25] 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 Int'l Symp. on Computer Architecture (ISCA)*, 2015, pp. 144–157.

Rubén Titos-Gil received MS and PhD degrees in Computer Science from the University of Murcia, Spain, in 2006 and 2011, respectively. As a PhD student, he was awarded a FPU scholarship from the Spanish Government. After holding post-doctoral positions at Chalmers, Sweden, and at the Barcelona Supercomputing Center, Spain, in 2015 he rejoined the University of Murcia where he has since served as an adjunct professor. His research focuses on hardware support for synchronization in parallel processors.

Ricardo Fernández-Pascual received the MS and PhD degrees in computer science from the Universidad de Murcia, Spain, in 2004 and 2009, respectively. In 2004, he joined the Computer Engineering Department as a PhD student with a fellowship from the regional government. In 2006, he joined the Computer Engineering Department of the Universidad de Murcia where he is currently an associate professor. His research interests include general computer architecture, memory hierarchies for parallel processors, and performance simulation.

Alberto Ros Alberto Ros is Associate Professor at the University of Murcia, Spain. He received the Ph.D. degree in computer science from the same university, in 2009, after being granted with a fellowship from the Spanish government to conduct the Ph.D. studies. He hold post-doctoral positions at the Universitat Politècnica de València and at Uppsala University. He has co-authored more than 60 research papers in international journals and conferences. His research interests include cache coherence protocols, memory hierarchy designs, and memory consistency for multicore architectures.

Manuel E. Acacio is a Full Professor of computer architecture and technology at the University of Murcia, Spain. Dr. Acacio obtained his PhD degree in Computer Science in March 2003. Before, in the summer of 2002, he worked as a summer intern at IBM TJ Watson. Currently, Dr. Acacio leads the Computer Architecture & Parallel Systems research group at the University of Murcia. He is author of about 100 papers in refereed international conferences and journals. As well, he has served as a committee member of numerous international conferences. His research interests are focused on the architecture of multiprocessor systems. From April 2011 to April 2015, Dr. Acacio served as an associate editor of IEEE TPDS Int'l Journal, since August 2016 he is member of the editorial board of MPDI Computers Int'l Journal. Since September 2018, he serves as academic editor in the editorial board of Hindawi Scientific Programming journal. He is also member of the board of distinguished reviewers of ACM TACO Int'l Journal since May 2014.