Improving Parallelism in Hardware Transactional Memory

DAVE DICE, Oracle Labs

MAURICE HERLIHY, Brown University and Oracle Labs

ALEX KOGAN, Oracle Labs

Today's hardware transactional memory (HTM) systems rely on existing coherence protocols, which implement a requester-wins strategy. This, in turn, leads to poor performance when transactions frequently conflict, causing them to resort to a non-speculative fallback path. Often, such a path severely limits parallelism. In this article, we propose very simple architectural changes to the existing requester-wins HTM implementations that enhance conflict resolution between hardware transactions and thus improve their parallelism. Our idea is compatible with existing HTM systems, requires no changes to target applications that employ traditional lock synchronization, and is shown to provide robust performance benefits.

CCS Concepts: • **Computer systems organization** → **Multicore architectures**; • **Theory of computation** → **Parallel computing models**;

Additional Key Words and Phrases: Hardware transactional memory, cache coherence protocols, requester-wins, lock elision

ACM Reference format:

Dave Dice, Maurice Herlihy, and Alex Kogan. 2018. Improving Parallelism in Hardware Transactional Memory. *ACM Trans. Archit. Code Optim.* 15, 1, Article 9 (March 2018), 24 pages.

<https://doi.org/10.1145/3177962>

1 INTRODUCTION

Hardware transactional memory (HTM) supports a model of concurrent programming where the programmer specifies which code blocks should be atomic but not how that atomicity is achieved. Some form of HTM is currently supported by processors from Intel [22] and IBM [8, 23, 43]. Transactional programming models are attractive, because they promise simpler code structure and better concurrency compared to traditional lock-based synchronization.

An atomic code block is called a *transaction*. HTM executes transactions *speculatively*: If an attempt to execute a transaction *commits*, then that code block appears to have executed instantaneously, while if it *aborts*, then that code has no effect, and control passes to an abort handler; a condition code usually indicates why the transaction failed.

As long as transactions do not conflict on the shared data they access, and as long as pathologies such as capacity aborts and unsupported instructions are avoided, HTM has been shown to achieve nearly linear scalability [11, 24, 47]. However, the experience shows that even with a moderate level of conflicts between hardware transactions, the performance of HTM substantially

Author's address: A. Kogan (corresponding author), Oracle Labs, 35 Network Drive, Burlington, MA 01803, USA; email: alex.kogan@oracle.com.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.

© 2018 ACM 1544-3566/2018/03-ART9 \$15.00

<https://doi.org/10.1145/3177962>

deteriorates [11, 15, 47]. That is because most existing HTM implementations piggy-back on cache coherence protocols [20], which mostly implement a *requester-wins* policy: If one transaction requests exclusive access to a cache line held by another, then the earlier transaction aborts and restarts. Thus, data conflicts cause repetitive transactional aborts [34], which in turn force the execution to proceed through a slower, non-speculative path. This path typically employs locks (e.g., in the very popular transactional lock elision (TLE) method [14, 37]), and taking this path aborts any concurrent speculative transactions, even when there is no actual data conflict between speculative and non-speculative threads.

This article's contribution is to propose enhancing existing requester-wins HTM systems with a strikingly simple mechanism that allows threads to continue speculating on HTM despite repetitive aborts. The idea is to elevate the status of a transaction that fails to commit so that any conflict between this and other, non-elevated, transactions can be resolved in the favor of the former. Thus, the elevated transaction, which we call a *power* transaction or running in a *power mode*, can execute *speculatively in parallel* with transactions it does not conflict with, while impeding progress only of transactions that it does conflict with. In a nutshell, to support power mode, each coherence request is augmented with one bit indicating whether the request is coming from a thread speculating on HTM. The power transaction replies with a negative acknowledgment (NACK) to coherence requests from other transactions, causing the requester to abort and allowing the power transaction to proceed. At the same time, regular transactions not conflicting with power transaction(s) can run in parallel with the latter.

As an example where this additional parallelism may be beneficial, consider a binary search tree where each tree operation is run in a separate transaction. If two operations try to modify the same node in the tree, then they might repeatedly conflict and abort each other. Once one of those transactions decides to abandon speculation and execute under lock, it will cause all other transactions, including those that access a completely different set of nodes in the tree, to wait for its completion. With our proposal, however, that transaction will switch into the power mode and thus stop the other operation on the same node from aborting it again; all other operations working on different nodes would be able to seamlessly continue their speculation. As described later in the article, a simple software or hardware mechanism can be put in place to ensure that only one transaction enters the power mode at a time. Such a mechanism is required for performance only, not correctness; that is, multiple power transactions can coexist in the same system as long as they do not conflict and abort each other if they do.

We note that power transactions are compatible with existing HTM systems in the sense that providing hardware transactions the ability to escalate to power mode will not break existing code. Moreover, support for power transactions imposes no additional cost on transactions that do not use the power mode. In other words, the performance of any software that uses (or not) regular transactions would remain unchanged even after the power mode support is introduced. Furthermore, power transactions can be employed without modifying target applications, e.g., in every case where TLE is applicable. In fact, when the entry to the power mode is controlled by hardware, the existence of the power mode can be completely hidden from the programmer.

We used two approaches to evaluate the utility of power transactions. First, exploiting the recent support for compiling transactional programs in GCC [17], we emulated hardware power transactions in software running on top of a real HTM implementation (namely Intel Haswell TSX). We conducted experiments to compare the relative performance of a number of micro- and STAMP [29] benchmarks with and without power transactions. (As described below, our software emulation is conservative, in the sense that it tends to understate the advantage of power mode.) With one exception, every benchmark tested yielded improved performance under power mode.

These experiments imply that the standard dual-path code structure, in which any non-speculative path automatically aborts all speculative transactions, fails to exploit substantial opportunities for concurrent execution. Note that the power mode is not useful when all transactions conflict or when none of them conflict. We stress that the potential benefit of the power mode is in unleashing the parallelism in workloads where *some* transactions conflict with each other, allowing those transactions that do not conflict to keep running. The fact that this parallelism exists in various workloads is not trivial, but our performance evaluation clearly demonstrates that it does.

Second, we added the power mode support to SuperTrans [36], a transactional memory simulator built from SESC [41], that was recently enhanced to more accurately simulate best-effort HTM similar to Intel TSX [34]. Using SuperTrans, we compared the utility of power mode to several variants of PleaseTM [34], a recent related proposal for improving parallelism in HTM, as well as to the baseline implementation that does not use power mode. Using STAMP benchmarks [29], we show that power mode not only provides non-trivial speedup above the baseline implementation (confirming our emulation-based study) but also performs better than all other evaluated variants of PleaseTM despite requiring less architectural changes.

The rest of the article is organized as follows. We discuss related work in Section 2. The required hardware support for power transactions is detailed in Section 3, along with other implementation details, such as techniques for transitioning between power and regular modes. Section 4 and Section 5 present the evaluation of the effectiveness of power transactions in emulation-based settings and using the SuperTrans simulator, respectively. We conclude the article with some remarks in Section 6. Extended implementation details of our emulation are deferred to Appendix A.

2 RELATED WORK

In Intel Haswell [22] and its successors, as well as in IBM Power 8 [8], hardware transactions are best effort: No transaction is guaranteed to commit. Transactions may abort because of data conflicts, cache overflow, or cache associativity issues. Transactions must not execute certain instructions, such as I/O instructions and system calls.

In these systems, progress is usually guaranteed by combining HTM with some form of locking. Perhaps the simplest and most widely used such technique is the transactional lock elision (TLE) [14, 37], where the critical section associated with a lock is first attempted speculatively, transactionally reading but not writing the lock state. TLE is attractive, because it can be enabled, without any changes to the target application, at the level of a library providing lock implementations while preserving the semantics provided by the lock based synchronization [11, 14]. If the speculative lock elision fails (typically, after a few retries), then the thread acquires the lock and re-executes the critical section non-speculatively. TLE provides the same progress guarantees as regular locking, but it has a non-trivial cost: Once the lock has been acquired, all concurrent speculative transactions will fail and wait until the lock is released, even if there are no actual data conflicts. As a result, numerous articles show that TLE is very effective when most transactions succeed, but its benefit fades once the lock is acquired often [11, 15, 47]. To keep our usage examples of power transactions concrete, we focus on the use of locks as the alternative path, effectively enhancing the standard TLE technique [14, 37]. We note, though, that power mode is equally helpful in reducing the use of any fallback path, such as the one implemented using software transactional memory (STM) [9, 28], lock-free techniques [26], and so on.

The use of a special execution mode for (software or hardware) transactions has been previously explored in related contexts. Blundell et al., for instance, design a system called OneTM that supports unbounded hardware transactions [5]. One of the variants of OneTM,

called OneTM-Concurrent, supports concurrent execution of non-overflowed transactions and non-transactional code with one overflowed transaction. To support this mode of execution, OneTM-Concurrent requires, among other architectural changes, additional metadata storage and management in memory controllers as well as an additional architected register, saved and restored on every context switch. Power mode, being designed to enhance existing (bounded) HTM implementations, does not require any of those complications.

In the context of software transactional memory, Ni et al. [33] describe an STM runtime library that supports multiple modes of executions. One of the modes, called *obstinate*, is a software equivalent of power transactions. Citing from Reference [33], “a transaction running in obstinate mode always wins all conflicts with other transactions—regular transactions are allowed to run concurrently with the obstinate one, but the obstinate transaction has the highest conflict resolution priority of all transactions in the system.” As expected from any STM system, however, the control over execution modes and conflict resolution between transactions in Reference [33] is done entirely in software in a dedicated contention manager module of the system.

Since the introduction of the HTM design by Herlihy and Moss [20], numerous attempts have been made to improve and extend it (e.g., References [3, 27, 30, 31, 40, 46] to give just a very few examples). The scope of this article precludes elaborating on all these efforts. We note, however, that, broadly speaking, the architectural changes required by most of them go far and beyond the ones needed to support the power mode. Furthermore, the prime goal they pursue (e.g., supporting hardware transactions with unbounded capacity [3, 27, 30, 46], running transactions across context switches [40, 46], supporting nested transactions [31, 46], and so on.) is typically different from the one considered in this article.

Perhaps the most relevant prior work to this article is the recent publication by Park et al. on a PleaseTM mechanism for requester-wins HTM systems [34]. In PleaseTM, hardware transactions insert a plea bit (or bits) into their responses to coherence requests. These plea bits are considered by the requester and allow supporting alternative conflict resolution schemes. For instance, a requester running a hardware transaction and receiving a response with the plea bit set may abort its transaction, effectively achieving a responder-wins conflict resolution strategy for hardware transactions. By keeping track of the number of transactionally read cache lines and encoding this information in a number of plea bits, PleaseTM allows a scheme where a transaction with more lines read wins a conflict. While requiring several architectural changes, PleaseTM does not modify the cache coherence protocol itself, meaning that a pleading transaction releases a cache line on receiving a coherence request. Consequently, to ensure atomicity, the pleading transaction needs to re-request the cache line and validate the line data when the line is re-acquired. As a result, even when a requesting transaction decides to abort (i.e., accept the plea), it slows down the responding transaction and would do that over and over again with every retry. This mechanism also puts pressure at the coherence bus, especially when the requesting transaction runs on a different socket. Power mode differs from PleaseTM in that it allows resolution of conflicts at the time the request is received, without slowing down the responding transaction and without increasing coherence traffic. Furthermore, transactions in PleaseTM can still repeatedly abort each other as long as they respond with pleading bits to each other’s requests. At the same time, power mode provides more definitive control with respect to which transaction would receive priority over others.

In another relevant article, Armejach et al. consider a few hardware and hybrid (software and hardware) techniques to improve performance of requester-wins HTM [4]. Perhaps the most relevant technique to our work is the one called delayed requester-wins (DRW). The idea behind DRW is to allow the exclusive owner of a cache line to delay response to conflicting requests, thus increasing the chance for its transaction to complete. Delayed conflicting requests are queued at

the exclusive owners' caches and are considered when the transaction ends (by commit or abort). To avoid deadlocks, DRW associates timeouts with buffered requests and conservatively handles a request when its timer expires. The requirement to manage the buffers of incoming conflicting requests and their associated timers call for hardware changes that are much more elaborated than supporting power transactions.

The need to prioritize transactions to enhance conflict resolution was also considered in References [38, 39]. The authors use timestamps (composed of a logical clock incremented after every successful transaction and processor ID) and append them to all transactional requests. If a thread receives a cache line request with a higher timestamp, then it delays the response until after its transaction is completed. Deferring processing of incoming requests has certain amount of complexity as just mentioned while relying on logical clocks favors threads with fewer transactions, not necessarily those that had repeated conflicts.

3 POWER TRANSACTIONS

We first describe the common mechanism required to support power transactions regardless of how transactions enter the power mode (Section 3.1). Next, we discuss the details of supporting a software-controlled entry into the power mode, followed by details on a hardware-controlled entry (Section 3.2 and Section 3.3, respectively). We note that those two entry methods are complementary, i.e., we envision that some architectures may provide both methods, alike the support for HLE and RTM in Intel TSX [22]. Note that exiting power mode does not require any special treatment, i.e., power transactions commit or abort exactly as regular transactions. We also discuss variations of our design along with their impact on the properties of power transactions (Section 3.4), as well as outline certain limitations in our design (Section 3.5).

3.1 Common Mechanism

Supporting the power mode requires each hardware thread to maintain a distinctive speculation status that can be encoded in one bit of a thread state. That is, in addition to the status indicating that a given hardware thread is speculating on HTM, we need to store a bit of information (that we call the power-mode bit), which, when set, indicates that the speculating thread is running in power mode. This bit will be set when a hardware thread starts a new power transaction (via one of the mechanisms described in the subsequent sections) and reset when the thread completes that transaction (either through commit or abort).

In addition, we require to add a speculation status bit as a simple payload to coherence request messages. This bit indicates whether the request is coming from a thread speculating on HTM (either in power or regular modes). It is ignored by the coherence hardware and is simply passed to cache controllers, which in turn can take it into consideration when preparing the corresponding coherence response.

Cache controllers are modified so that when the following three conditions hold, they respond with a special NACK message: (1) the speculation bit in the incoming request is set, (2) the power-mode bit of the target thread is set, and (3) the request is to invalidate or downgrade transactionally-held data. If any of those conditions does not hold, then the cache controller logic remains unchanged. We note that to support (regular, non-power-mode) HTM, the cache controller already implements logic to consider the speculation state of the target as well as whether the request is to invalidate or downgrade transactionally held data (so that the hardware transaction run by the target thread can be aborted). Thus, the additional complexity of considering the speculation bit in the request payload is trivial.

Another modification in the cache controller is related to the treatment of the NACK coherence response message. Specifically, when the NACK is received and the receiving thread is speculating

Fig. 1. Interactions between a thread running a power transaction (P) and another thread running a (regular or power) transaction (R) or non-transactional code (N).

Fig. 2. Architectural changes required to support power transactions.

on HTM (either in power or regular modes), the current transaction is aborted; a special abort code may be used to specify that the abort occurred due to a data conflict with a power transaction. Otherwise, the NACK response is ignored. This can happen only if the hardware transaction that issued the coherence request, which resulted in the NACK response, has been aborted while awaiting that response. Figure 1 illustrates possible interactions between a thread running a power transaction (P), a thread running another (regular or power) transaction (R), and a thread running non-transactional code (N).

The only modification to the cache coherence protocol is supporting a special NACK response message (if it does not already support one). We note that numerous previous articles on computer architecture considered adding a NACK message (e.g., References [6, 27, 30, 46]). As opposed to most of that work, however, the very limited use of NACKs in our case does not result in any additional changes in the coherence protocol, such as managing timestamps and adding deadlock-avoidance logic, or in coherence messages, such as including timestamps in every coherence request message.

The required changes in an HTM architecture are summarized in Figure 2 with new or modified components shown in gray. We believe that all the proposed changes are simple (albeit with some caveats, as discussed in Section 3.5). It is hard to compare in a quantitative way the amount of hardware changes required by various proposals to existing architectures, especially given that those proposals often modify different components of the architectures. Yet, adding support for power mode stands out for its simplicity and is substantially less intrusive when compared to any related proposal known to us as described in Section 2. This makes supporting power mode more feasible in real systems.

Note that the common mechanism as described so far does not limit or control the number of power transactions that can coexist in the system. Such control can be provided through entry mechanisms described in the subsequent sections.

3.2 Software-Controlled Entry

To allow software to control which transaction(s) would run in a power mode, one new instruction should be added to the instruction set architecture (ISA). This instruction should be virtually identical to the one used to begin a (regular) hardware transaction but use a different opcode, which would instruct the core executing it to set the power-mode bit of the corresponding hardware thread. As mentioned above, there is no need to add a new instruction(s) for completing the power mode transaction.

With the software-controlled entry to power mode, it is the responsibility of the programmer to ensure (or not) that only one transaction switches into the power mode at a time. As a common case, at most one transaction accessing shared data would run in the power mode. However, we stress that this is merely a performance consideration rather than correctness. In fact, as we discuss in Section 3.4, there are cases (e.g., performance debugging) in which having multiple transactions in power mode is desirable.

Following is one particular approach for ensuring that there is at most one transaction in the power mode. This approach is integrated with a common implementation of the TLE mechanism [14, 37] and assumes that the target application uses locks to synchronize access to shared data inside critical sections. As mentioned in Section 2, TLE can be enabled *without any changes* to the target application, e.g., by interposing the library providing the lock implementation. The power mode preserves this property—using a TLE mechanism with power transactions, such as the one described below, does not require modifying target applications as well.

In the TLE mechanism enhanced to exploit power transactions, the entry into power mode is protected by a lock. For simplicity, we use a spin lock, but other locks are possible (e.g., a queue lock for fairness). Figure 3 shows pseudo-code for such an enhanced TLE mechanism. Here, a transaction elevates into the power mode if it repeatedly fails to commit. The atomic compare-and-swap (CAS) instruction (Line 24) ensures that only the thread that sets the powerFlag flag to its thread ID will enter the power mode. We note that using thread IDs can be avoided, e.g., by using a thread-local flag that tells the current thread whether it is the one that entered the power mode. Notice that the code does not access the powerFlag flag inside a hardware transaction. Thus, starting (and committing) a power transaction does not abort regular transactions.

When using power mode, regular transactions may be subject to the *lemming effect* [13] arising when one transaction enters power mode and forces the rest to follow; this effect exists with (regular) transactions in standard TLE as well [13]. One way to mitigate the lemming effect is to give less (or even zero) weight for retries happening while the powerFlag flag is set. That is, if an attempt to use a regular transaction fails and powerFlag is set, then we discount this attempt by decrementing the ntrials counter (Line 16). We note that the pseudo-code in Figure 3 also includes a standard anti-lemming optimization in TLE, in which a transaction is retried only when the underlying lock becomes available (Line 28) [24].

3.3 Hardware-Controlled Entry

Along with (or instead of) a software-controlled entry into power mode, the HTM engine itself may control when a regular transaction switches into the power mode. In this case, no ISA extensions are required. In fact, the availability of power mode for hardware transactions may be completely hidden from the programmer in this case.

A simple hardware-based scheme can be put in place to ensure that there is only one power transaction at a time, either systemwide or for each process. There are many ways to implement such functionality, which requires the ability to arbitrate concurrent requests from multiple

```

initially : (global) lock = available ;
            (global) powerFlag = -1;
            (thread local) tID = unique thread ID;

```

Lock procedure:

```

1  ntrials = 0;
2  while (true) {
3      // start a regular or power transaction according to the value of 'powerFlag'
4      if ((powerFlag != tID && begin_htm()) ||
5          (powerFlag == tID && begin_power_htm())) {
6          if (isLocked(&lock)) self-abort();
7          return;
8      }
9      if (!self-aborted) {
10         if (powerFlag == tID) {
11             // exit the power mode and fall to the lock
12             powerFlag = -1;
13             break;
14         } else if (powerFlag != -1) {
15             // avoid the lemming effect
16             ntrials--;
17         }
18         // increase the counter for the number
19         // of non-power mode trials
20         if (++ntrials >= MAX_TLE_TRIALS) {
21             // if we exhausted the number of non-power mode trials, check if the 'powerFlag' flag is
22             // available and try to set it.
23             // Note: for TLE we would simply break here and fall to the lock
24             if (powerFlag == -1) CAS(&powerFlag, -1, tID);
25         }
26     }
27     // wait for the lock to become available
28     while (isLocked(&lock)) { pause; }
29 }
30 // we failed to commit a transaction, grab the lock
31 Lock(&lock);

```

Unlock procedure:

```

1  if (!isLocked(&lock)) {
2      commit_htm();
3      if (powerFlag == tID) powerFlag = -1;
4  } else {
5      // release the underlying lock
6      Unlock(&lock);
7  }

```

Fig. 3. TLE using power transactions.

hardware threads. One option, similar to the proposal made in Reference [5], is to add a shared (between all hardware threads) transaction status word, which resides in a fixed location in the virtual address space of each process. This word acts as a mutex lock, i.e., the thread enters the power mode only if it atomically sets the value of the word and exits that mode when it atomically resets it. Unlike the proposal in References [5], however, regular transactions do not need to monitor this word and perform any special logic for conflict detection when it is set.

Considering the HLE mechanism in Intel TSX [22] as an example, when the hardware thread encounters a lock instruction with the opcode prefix that allows speculation, it may start

speculation using a regular transaction. If aborted, then it may try to atomically set the transaction status word, and if it succeeds, it shall set its power-mode bit and run a power transaction. On completion (either abort and commit), it shall reset the power-mode bit and the shared transaction status word. If the thread fails to set the transaction status word, which means that another power transaction is in progress, then it may retry with a regular transaction or, if the preset retry policy instructs so, execute the lock instruction non-speculatively.

3.4 Variations

There are a few interesting extensions for the common mechanism discussed in Section 3.1. First, we may omit including the speculation status bit in the coherence request messages. A power transaction then will send NACKs in response to all invalidation and downgrading requests, not just requests from transactions. A transactional thread (regular or power) that receives a NACK simply aborts, and a non-transactional thread backs off (pauses) and resends its request. This approach has some advantages: It alleviates the need to introduce a new bit into coherence messages' payload, and it protects power transactions against conflicts with non-transactional threads (but not with other power transactions). The principal disadvantage is that care must be taken to avoid denial-of-service vulnerabilities, perhaps by limiting the duration during which a power transaction can refuse invalidations or by throttling (at the hardware level) the rate at which transactions can enter the power mode. Furthermore, the need to introduce a back-off mechanism into the cache controller logic may complicate the support for power mode.

Second, the power mode support could be easily generalized to encompass multiple levels of power transactions. Instead of a single power-mode bit, each hardware thread state may include a power-mode counter indicating the level at which the thread is running a power transaction. The payload of cache coherence messages is respectively enhanced to include this counter. Higher-priority transactions refuse invalidation and downgrading requests from lower-priority transactions, effectively providing a kind of transactional priority system, which may be a start toward adapting transactional programming to reactive systems [45]. It is straightforward to enhance both software and hardware-controlled entry mechanisms discussed above to climb through the levels of power mode before resorting to a non-speculative execution. It should be noted that in the software-controlled entry, a new ISA instruction for starting a power mode transaction should include a level argument. Along with that, no further changes are required for the hardware-controlled entry beyond increasing the number of transaction status words to match the number of power mode levels.

As mentioned in Section 3.1, when a transaction receives a NACK and aborts, it may specify a special abort code providing indication to the programmer of a conflict with a power transaction. Taking this a step further, we may use a different abort code to indicate that the recipient of the NACK *was running in the power mode* as well. This abort code provides a way to detect undesired data sharing between transactions. A transactional undesired data sharing occurs when two transactions that are believed to have disjoint datasets actually have a data conflict. Such hidden conflicts can result from false sharing, from hidden data accessed by library calls, or from performance counters and related structures. Such conflicts can cause transactions to abort more often than expected, adversely affecting system performance. To test whether two transactions have disjoint datasets, run them concurrently in power mode, and if one aborts with the power-mode conflict abort code, then the transactions' datasets are not disjoint, and there is a possibly unexpected data sharing. Note that the special abort code would allow detection and, potentially, a repair of both true and false sharing issues, facilitating recent work that employs hardware performance counters for that purpose [16]. Exploring this potential benefit of supporting the power mode is left as future work.

3.5 Limitations

Supporting the power mode requires augmenting the cache coherence protocols with NACK messages. While directory-based coherence protocols are easily amendable for such a modification (as they already typically use positive acknowledgment messages), it might be more challenging to implement NACKs in a coherence protocol that is based on snooping. Those protocols rely on the shared bus to determine the global order. Once a requester observes its invalidation message on its own input queue from the shared bus (e.g., when it tries to perform a store), it can safely infer that its writes will be ordered correctly in the global order. Therefore it can proceed with the store, without waiting for an acknowledgment (or needing one). Although the details of cache coherence protocols used in modern systems are scarce, based on the publicly available information (e.g., Reference [21]), some of the systems featuring HTM are directory based, at least partially, and thus this issue does not apply to them.

Another challenge in implementing NACKs even in a directory-based coherence protocol is associated with the deferred invalidation optimization, also known as the Scheurich's optimization (SO) [42]. With this optimization, the cache controller may respond (with an acknowledgment) to the incoming invalidation request as soon as that request comes in, deferring the invalidation itself until later. In particular, when issuing the response, the controller might not have access to the information needed to determine that it should send a NACK instead.

We could not find a definitive answer as to whether this kind of optimization is used by any real system that implements HTM. Along with that, Hechtman and Sorin note that "...one important limitation of SO is that it does not apply to locks or flags. That is, using SO to defer an invalidation to a lock or flag does not help performance and, in fact, can hurt performance and potentially lead to deadlock" [19]. McKenney also points out to issues arising if invalidation deferrals are in place (see References [35], Section C.4). Therefore, we believe that even if an architecture uses this optimization, it has to be in a limited set of scenarios. Even if those scenarios involve HTM, one way to deal with that is to disable this optimization only when the power-mode bit is set at the core.

We note that the above limitations also apply to most, if not all, numerous prior articles that suggested using negative acknowledgements (plus far more radical changes, as briefly mentioned in Section 3.1) in coherence protocols; none of those articles, to the best of our knowledge, have fully addressed those concerns. Unlike the prior work, the power mode employs negative acknowledgements in a limited set of circumstances and in particular does not require additional logic for managing timestamps and/or avoiding deadlocks. Thus, we believe that supporting power mode is feasible with a small, contained change to the (directory-based) coherence protocol.

4 EMULATION-BASED EVALUATION

We have evaluated the utility of power mode with two complementary approaches. In this section, we describe our attempt to emulate power mode transactions in software (i.e., running them without HTM), while regular transactions run on top of HTM. This approach is inspired by work on hybrid transactional memory systems [10] and, in particular, by the implementation of refined TLE [11]. We note that this is not our intent to compare power transactions to hybrid TMs (which use a software-only code path) but rather to evaluate if and how the existence of power mode support can increase the parallelism of hardware transactions and ultimately improve performance of the existing HTM implementations. Our second evaluation approach is based on a transactional memory simulator and is described in Section 5.

4.1 Framework

4.1.1 High-Level Idea. Our experience shows that the time required for a successful execution of an atomic block using a hardware transaction is comparable to running that block in software.¹ This is also echoed by results of single-thread performance in various articles [11, 32, 47]. We leverage this fact to emulate power mode transactions in software, while using an actual HTM implementation (Intel Haswell, in our case) to execute regular transactions.

To mimic the behavior of a hardware power mode implementation and resolve data conflicts between power and regular transactions in favor of the former, we utilize software “metalocks.” These metalocks are implemented with the use of ownership records, or orecs, commonly used in the design of software and hybrid transactional memory systems [10, 18]. We instrument all memory accesses in transactions to read and update ownership records as appropriate, leveraging a recently introduced compiler support for a completely automatic instrumentation process. Thus, both power and regular transactions run on the instrumented path. A power transaction acquires ownership on memory words it accesses by writing into ownership records. Regular transactions check (by reading ownership records) that their memory accesses are not conflicted with those made by the concurrent power mode transaction, if such exist. The ownership records are designed in a way that regular transactions are aborted only when an actual conflict exists (as they should). In particular, a regular transaction is *not* aborted when it reads the same data as a power transaction does. Furthermore, a simple mechanism is put in place to clear all ownership records at once when the power mode transaction is completed, either by abort or commit (details are provided in Appendix A).

Our framework effectively adds the power mode to an *existing* HTM implementation, leveraging all of its properties for running and managing (regular) hardware transactions. In the emulated system, all instructions but loads and stores have absolutely the same latency as provided by the native platform. Load and store instructions are slowed down due to the use of instrumentation. We note that both power and regular transactions are slowed down, so the *relative* performance of these transaction types using the software metalocks is a way to estimate their relative performance in a hardware implementation. Indeed, because power transactions, unlike regular transactions, might be required to write each time they read (to acquire corresponding metalocks), our estimation is conservative, favoring the relative performance of regular transactions. Despite the impacts of instrumentation, which depend on the number of loads and stores in a critical section [11], we believe that the ability to exploit an actual HTM implementation as well as the ability to use arbitrary benchmarks make our framework an interesting tool able to provide important insights on performance of power mode. In the following subsection, we expand on implementation aspects of our framework.

4.1.2 Implementation Details. The GCC compiler [17] (starting from version 4.8) provides the libitm interface for transactional programs. The compiler translates critical sections implemented as atomic transactions into two distinct code paths: instrumented and uninstrumented. The instrumented code path includes calls to *instrumentation barriers*, functions invoked on each transactional memory access. The libitm library provides instrumentation barriers for a few standard synchronization mechanisms, such as TLE, STM, or lock synchronization, as well as the opportunity to provide customized instrumentation barriers and functions to be called when transactions

¹This is true at least as long as transactions are not tiny, so the overhead of starting and committing a hardware transaction is negligible.

commit or abort. For our framework, however, we used our own custom implementation of the libitm interface to reduce instrumentation overheads.²

We associate two metalocks with each cache line accessed by a transaction, one for read access and another for write access. A power transaction (run without HTM) acquires metalocks for the cache lines it accesses, according to the access mode desired (read or write) by writing a value into the corresponding metalock. A regular transaction (run with HTM) reads the metalocks associated with the cache lines it accesses and aborts if it finds a metalock held in a conflicting mode. If the metalock does not conflict, then the transaction proceeds to access the intended data. This scheme emulates power mode semantics, ensuring that any conflicting (and only conflicting) request by a regular transaction for data accessed by a power mode transaction is refused, causing that regular transaction to abort.

The instrumentation increases every transaction's data footprint, i.e., the number of accessed cache lines. However, regular transactions access the additional (metalock) cache lines only for reading, which typically does not stress Haswell HTM, whose read set capacity is relatively large [32]. Although power transactions access metalocks for writing, they do not use HTM and thus are not limited by its write capacity.

Power transactions sustain conflicts with regular transactions, but they can abort for other reason, and in particular, due to capacity limitations. A direct way to emulate capacity aborts for a power transaction is to detect when a capacity limit is reached, roll back that transaction, and restart it using locks. This direct approach, however, requires logging each transaction's write set and reverting its memory updates on abort, further increasing instrumentation overhead. Instead, we opted for the following less-intrusive emulation. First, each power mode transaction takes a timestamp when it begins its execution. Second, we track the number of cache lines accessed by a power mode transaction. Once this number goes beyond a preset limit, we calculate the time period δ elapsed since the transaction started, switch to the locking mode, and spin for another time period δ , effectively charging for the time the power transaction has spent so far. When a power mode transaction switches to the locking mode (simply by setting a Boolean flag), all regular transactions are aborted (as in standard TLE) and wait for the lock to become available again. By spinning after lock acquisition (for the time period δ), we "charge" for the time required to re-execute the same atomic block without actually rolling back the changes made by the power mode transaction and without reapplying them under lock. A reader interested in further implementation details, including the metalock structure and the pseudocode of instrumentation barriers, is referred to Appendix A.

Our experiments were run on an Intel Haswell (Core i7-4770) four-core hyper-threaded machine (eight hardware threads in total). Our goal was to compare standard TLE [14, 37] with one that makes use of power transactions (henceforth *PowerTLE*). The pseudo-code for PowerTLE is provided in Figure 3. To evaluate the benefit of the additional concurrency provided by power mode, and to reduce the impact of other unrelated factors, such as the cost of instrumentation or transactions' increased memory footprints, we used exactly the same instrumentation barriers for TLE as well. We emphasize that the prime difference between TLE and PowerTLE in our framework is the ability of the latter to use a power transaction that runs concurrently with regular transactions as long as those two kinds of transactions do not actually conflict on shared data.

In addition to TLE, we compare PowerTLE to other related techniques that aim to prioritize certain transactions in case of conflicts by modifying (in software only) the baseline TLE mechanism.

²The library implementing the libitm interface in GCC is dynamically linked to an executable, resulting in an expensive function call for every memory access on the instrumented path. Our custom implementation of the libitm interface supports static linkage with the target executable.

Fig. 4. Skip list-based priority queue throughput. Higher is better.

The first technique is called Hourglass and uses so-called *toxic* transactions [25]. Specifically, once a transaction experiences multiple aborts, it becomes toxic (by setting a global flag), which means that other transactions cannot start their execution as long as the toxic transaction does not complete (or reverts to the lock, clearing the flag). Note that the toxic transaction does not abort running (normal) transactions, but those would stall on the next retry if they fail to commit and as long as there exists a toxic transaction. The second technique, called software-assisted conflict management (SCM), uses an auxiliary lock for transactions that repeatedly abort due to conflicts [2]. That is, after certain number of aborts due to data conflict, the transaction would acquire the auxiliary lock (and block if it is unavailable) before the next retry.

To reduce the impact of memory management on the performance, we use a cache-index aware (and HTM-friendly) memory allocator [1]. Each critical section was attempted 10 times using regular transactions before reverting to lock (in TLE) or power mode (in PowerTLE). For Hourglass and SCM, we use half of that number as a threshold for a transaction to become toxic or acquire the auxiliary lock, respectively. As demonstrated in Figure 3, a power mode transaction is tried only once (or more, in case of self-abort indicating that the lock is taken). We note, however, that other, more sophisticated retry policies that use power mode can be put into place. Although finding an optimal lock elision retry policy is an interesting question by itself [12, 15], it falls out of the scope of this article.

4.2 Skip List-Based Priority Queues

Figure 4 shows throughput results of a priority queue microbenchmark that uses a standard skip list implementation as an underlying data structure. The results shown are the average of 10 runs performed in the same configuration. Before starting measurements, all threads were set to spin for a few seconds to allow the system to warm up. The breakdown of operations between different modes of executions, e.g., regular transactions, power transactions, and so on, is presented in Figure 5. For PowerTLE, we report separately regular transactions completed without any power mode transaction running concurrently with them (denoted as NonC TXs) and those completed while a power mode transaction was running (denoted as C TXs).

For the experiment reported in Figure 4(a), the queue is initialized with 5M elements, and all threads run a total number of 5M RemoveMin operations, divided equally among the participating threads. We measure the time from the start until the last thread is done with its operations and calculate throughput by dividing the total number of performed operations (5M) by this time. In this particular workload, all threads compete with each other over the minimal element in the queue. Not surprisingly, power mode improves throughput only slightly (over the baseline TLE), since a power mode transaction conflicts with every other regular transaction and thus aborts them. This is echoed by results in Figure 5(a) showing that only few regular transactions manage

Fig. 5. Breakdown of execution modes for operations in skip list-based priority queue benchmark. “C TXs” (“NonC TXs”) stand for regular transactions executed concurrently (non-concurrently, respectively) with a power transaction.

to complete, while the majority of operations is executed using a lock (in TLE) or power mode transactions (in PowerTLE). The more graceful reduction in contention provided by Hourglass proves itself useful in achieving the best throughput in this workload.

In the experiment reported in Figure 4(b), the queue is initialized with 5M elements, and each thread runs loop iterations for 5s, where in each iteration it chooses randomly to remove a minimal element or insert a random element into the queue. Here the increased concurrency provided by power mode starts to take effect as the number of threads increases. This is because when a thread runs, e.g., an RemoveMin operation in power mode, other threads can proceed concurrently to apply their non-conflicting Insert operations. Figure 5(b) shows that, indeed, some portion of regular transactions manages to complete concurrently with a power transaction, and this portion grows with the number of threads. Note that both Hourglass and SCM fail to improve the performance of TLE. This is because they do not distinguish between conflicting operations, i.e., once a transaction Tx becomes toxic (or acquires the auxiliary lock), it blocks retries of all other transactions (or at least all transactions that attempt to acquire the auxiliary lock), regardless of whether they conflict with Tx or not.

The benefit of PowerTLE over TLE and other alternatives is also apparent when we consider only Insert operations that are less likely to conflict with each other compared to RemoveMin operations. Figure 4(c) shows the results of the experiment where the queue is initially empty and all threads perform a total number of 5M Insert operations, divided equally among threads. When the number of threads grows, the improved concurrency of PowerTLE becomes evident with the increase in the portion of regular transactions executed while a power mode transaction was running (cf. Figure 5(c)).

4.3 AVL Tree-Based Sets

In this section, we discuss results of a set microbenchmark implemented on top of AVL trees. The AVL tree implementation is similar to the one found in OpenSolaris OS. In all experiments, each thread runs iterations for 5s, and in each iteration it chooses an operation and a key. The operations are randomly selected from a given workload distribution, while the key is randomly selected from a given range from 0 to 511. The set is initialized to contain half of the given key range (256 keys).

Figure 6(a) shows results for the read only workload where all threads perform only Find operations. Here, the vast majority of operations succeed without any retries, and thus power mode is not used. Thus, all compared lock elision mechanisms yield similar performance. The breakdown of execution modes shows that, indeed, virtually all operations succeed using regular transactions (cf. Figure 7(a)). This is not surprising, as the operations do not conflict with each other.

Fig. 6. AVL tree-based set throughput. Higher is better.

Fig. 7. Breakdown of execution modes for operations in AVL tree-based set benchmark.

The workloads in Figure 6(b) and (c) include update operations. Specifically, in the former threads perform 60% Find operations, while in the latter threads perform 20% Find operations; the rest is divided equally between Insert and Remove. Here, as the number of threads grows, some transactions fall back to the lock (in TLE) as they experience conflicts on data they access. As a result, the benefit of increased concurrency provided by PowerTLE becomes more significant as the number of threads and/or the portion of update operations increases. The breakdown of execution modes for these workloads (Figure 7(b) and (c), respectively) confirms that as the number of threads increases, more regular transactions manage to complete concurrently with a power transaction in PowerTLE rather than falling to the lock as they would with TLE. In those workloads, Hourglass proves useful and achieves performance comparable to that of PowerTLE. At the same time, SCM does not help much.

4.4 STAMP

This section presents results measured with the STAMP benchmarking suite [29], which is used extensively in transactional memory research.³ For each benchmark, we used a standard (“native”) set of command line parameters. Figure 8 shows running time reported by each benchmark, averaged over 10 runs. We omit the results for one of the STAMP benchmarks (namely bayes) due to extremely high variance (which was also observed by others [34, 47]).

The results in Figure 8 show that power mode can be very helpful in certain cases, while it is harmful in one particular case. Specifically, in five cases (genome, intruder, kmeans-high, vacation-high, and vacation-low), PowerTLE beats TLE by substantial margin, while it harms the performance of yada. Only in one of those five case (kmeans-high), SCM and Hourglass

³We used a version of STAMP available at <https://github.com/mfs409/stamp>.

Fig. 8. STAMP running time measurements. Lower is better.

manage to improve over TLE and achieve performance on par with PowerTLE. In all other four cases (genome, intruder, vacation-high, and vacation-low), SCM and Hourglass either harm or match the performance of TLE (and thus perform substantially worse than PowerTLE). Notably, in all three cases where PowerTLE performs on par with or only slightly improves over TLE (kmeans-low, labyrinth, and ssca2), the TLE variant exhibits scalability up to eight threads, thus limiting the benefits of power mode.

The breakdown of execution modes for critical sections of various STAMP benchmarks is presented in Figure 9 and sheds some light on the performance of PowerTLE compared to TLE. First, just like in the case of microbenchmarks reported in Sections 4.2 and 4.3, power mode appears to be helpful when substantial amount of transactions fail to lock (in TLE), and these transactions manage to commit using power mode. This happens in all five cases where PowerTLE beats TLE.

Second, in two of the three cases where PowerTLE and TLE perform almost the same (kmeans-low and ssca2), the vast majority of critical sections execute using regular transactions only. Along with that, the case of labyrinth shows a different picture (cf. Figure 9(e)). Despite almost half of critical sections being executed using locks (in TLE), only a small portion of them are executed using power transactions (in PowerTLE), suggesting that the majority of those transactions fail due to capacity reasons. These results suggest that most of the time in this particular benchmark is spent outside of critical sections, explaining why despite the overhead of failed power transactions, PowerTLE achieves essentially the same results as TLE for this benchmark.

Fig. 9. Breakdown of execution modes for critical sections in STAMP benchmarks.

Finally, while yada shows a similar pattern to labyrinth (i.e., executions fail to commit using power mode transactions and therefore switch to lock), its running time is more sensitive to the performance of its critical sections. Here, the cost of failed power transactions is detrimental to the performance of PowerTLE. This benchmark shows that power transactions, like any kind of speculative execution, are effective only when speculation is mostly successful. We note, though, that a relatively straightforward optimization in PowerTLE that might eliminate performance degradation in yada is to avoid using the power mode if a regular transaction fails due to capacity. This optimization should be used with care, as at times transactions that fail due to capacity do manage to commit if retried [7]. Exploring the impact of this optimization is in our future work.

5 SIMULATOR-BASED EVALUATION

In addition to the framework discussed above, we added support for power mode into SuperTrans [36], a transactional memory simulator built on top of SESC [41]. As reported in Reference [34], SuperTrans was enhanced with a best-effort HTM support similar to Intel TSX.

In our evaluation, we use the default configuration file provided with the simulator, with minor configuration modifications for more realistic cache structure.⁴ Specifically, we model a CMP machine with 64 cores connected through an 8×8 mesh network. Each core has private 8-way associative 64KB L1 instruction and data caches, a private 16-way associative 256KB L2 cache and a shared L3 cache with 8MB capacity. The L1 caches have hit latency of 3 cycles, the L2 caches have hit latency of 18 cycles, and the L3 cache has hit latency of 34 cycles.

⁴We verified that our configuration modifications did not have any impact on the results.

Table 1. Relative Performance of STAMP (Lower Is Better)

Benchmark	Baseline	ResponderWins	MoreReadsWins	PowerTLE	ResponderWins+NACK	MoreReadsWins+NACK
genome	1.000	0.951	0.892	0.848	0.908	0.918
intruder	1.000	1.126	0.829	0.723	1.014	0.756
kmeans (low)	1.000	0.995	0.996	0.994	0.996	0.996
kmeans (high)	1.000	0.519	0.519	0.654	0.520	0.520
labyrinth	1.000	1.285	1.006	0.914	1.250	1.071
ssca2	1.000	0.998	1.001	1.013	0.998	0.999
vacation (low)	1.000	1.005	1.000	1.063	1.012	1.001
vacation (high)	1.000	0.960	0.934	1.029	0.959	0.940
yada	1.000	0.944	0.944	0.527	0.948	0.952
mean	1.000	0.976	0.902	0.863	0.956	0.906

We simulate a system with 16 threads running STAMP [29] with recommended inputs for a simulator environment. Note that these input sets are different from the native ones used in Section 4.4, as they are intended to produce shorter workloads that can be simulated in a reasonable time. Thus, some benchmarks might exhibit different contention patterns.

We modify the existing TLE implementation to use power transactions following the pseudocode in Figure 3. In line with the previous section, we refer to this modified implementation as PowerTLE. We compare PowerTLE to TLE running on top of the baseline (requester-wins, best-effort) HTM as well as on top of HTM modified according to the PleaseTM proposal [34]. For the latter, we use two variations called ResponderWins and MoreReadsWins. In the former variation, the requester running a hardware transaction and receiving a line with the plea bit set, aborts its transaction. In the latter variation, each core tracks the number of cache lines read transactionally and includes this counter along with the plea bit. The requester compares this counter to its own and aborts its transaction if it read less lines than the responder. Both these variations are discussed in detail in Reference [34] and implemented in SuperTrans by their authors. In addition, we compare PowerTLE to PleaseTM-based variants modified to use NACKs. In those variants, instead of inserting a plea bit (or bits), the transaction sends a NACK message to the requester.⁵

Table 1 summarizes the relative performance of all variants compared to the baseline HTM implementation. That is, for each variant, we divide the simulated running time of the benchmark (as reported in the “Time=” output line by each benchmark) to the one measured with the baseline. The simulator results show that PowerTLE outperforms TLE in most cases and performs better, on average, than all PleaseTM variants. In general, the average gains of PowerTLE over TLE are more modest compared to those measured with our emulation framework, in part due to the different workload settings. Yet PowerTLE is able to significantly (by 15–47%) improve the performance of five benchmarks, while not harming the performance of any other benchmark by more than 6%. We note that adding NACKs to PleaseTM is not helpful, in part because the original PleaseTM mechanism (implemented by its authors) is simulated similarly to the one that uses NACKs.

Table 2 provides details on the percentage of critical section executions that end up falling to the lock for each of the variants. Two observations can be drawn from these data. First, the percentage

⁵While working on those modifications, we discovered that the implementation of the original PleaseTM mechanism in the simulator is somewhat inaccurate. Specifically, in the simulator, the decision of which transaction to abort (among all conflicting transactions) is taken at the time the memory access is made rather than at the time the response from the transaction to which a plea has been sent is received (as the PleaseTM protocol prescribes). This is a subtle but quite important difference. That is because the simulated implementation of PleaseTM precludes certain scenarios that might harm its performance in the real world, as the pleading transaction does not release the ownership of the cache line (in the simulation) while sending the plea.

Table 2. Percentage of Critical Sections Executed under Lock

Benchmark	Baseline	ResponderWins	MoreReadsWins	PowerTLE	ResponderWins+NACK	MoreReadsWins+NACK
genome	1.3	1.3	1.0	0.0	1.1	1.2
intruder	14.4	19.3	8.6	0.2	19.7	8.3
kmeans (low)	0.0	0.0	0.0	0.0	0.0	0.0
kmeans (high)	6.1	0.0	0.0	0.0	0.0	0.0
labyrinth	26.8	29.9	31.2	0.0	29.9	28.1
ssca2	0.0	0.0	0.0	0.0	0.0	0.0
vacation (low)	0.0	0.0	0.0	0.0	0.0	0.0
vacation (high)	0.4	0.2	0.1	0.0	0.1	0.1
yada	30.3	16.6	16.6	0.0	16.3	17.0

of critical section executions falling to the lock in the baseline TLE is different, across all STAMP benchmarks, from the percentage presented in Figure 9 for the emulation-based evaluation. This suggests that the contention patterns are indeed different and explains the difference in the overall performance results. Second, PowerTLE eliminates virtually all failures to the lock. This property is important for several benchmarks, such as yada and kmeans-high, helping PowerTLE there to achieve better parallelism between hardware transactions that leads to impressive gains over TLE. We note, though, that the lack of failures to the lock does not translate to performance advantage for all benchmarks, as same transactions that fall to the lock in TLE might be unable to make progress in PowerTLE due to conflicts with a power mode transaction.

6 CONCLUSION

HTM is a promising tool to ease the development and accelerate the performance of concurrent code. Most existing HTM implementations rely on requester-wins cache coherence protocols and provide best-effort guarantees to concurrent transactions. The first property means that concurrent transactions abort frequently when data conflicts are common, as demonstrated by multitude of previous work [11, 15, 47]. The second property means that to guarantee progress, concurrent programs must include a non-speculative fallback path [22]. This path is typically implemented using a lock [14, 37]; once a thread switches to this path, all other transactions have to wait even if they do not conflict with the holder of the lock.

In this article, we introduced special power transactions with the aim of alleviating these issues. These transactions, running in a so-called power mode, receive priority in conflict resolution over other, regular transactions. We show that supporting power transactions requires simple changes to existing best-effort requester-wins HTM implementations. Our extensive experimental evidence using micro- and STAMP benchmarks, collected with emulation on top of a real HTM implementation as well as with a transactional memory simulator, demonstrates that power mode can improve parallelism between hardware transactions, leading to significant benefits for HTM that supports power transactions over the one that does not.

APPENDIX

A ADDITIONAL DETAILS ON EMULATION-BASED EVALUATION

Figures 10 and 11 provide additional implementation details of our emulation of the power mode, including the definition of metalocks and other auxiliary data structures (Figures 10) and the pseudo-code of read and write instrumentation barriers (Figure 11). Although we target the Intel Haswell architecture, the evaluation framework design is architecture independent and can be used with other HTM systems.

```

1 #define NUM_META_LOCKS (4 * 1024 * 1024)
2 #define CACHE_LINE_SIZE (64)

4 #define READ_CAPACITY (256)
5 #define WRITE_CAPACITY (64)

7 // fast pseudo-uniform hash function that maps a given key
8 // into a number between 0 and mask
9 uint64_t fast_hash(uintptr_t key, uint64_t mask) { ... }

11 // These macros translate from an address to a
12 // read/write meta lock protecting the cache line
13 // where the address belongs to.
14 #define ADDR_TO_READ_LOCK(addr) \
15     (&rMetadata[fast_hash( \
16         addr & ~(CACHE_LINE_SIZE - 1), \
17         NUM_META_LOCKS - 1)])
18 #define ADDR_TO_WRITE_LOCK(addr) \
19     (&wMetadata[fast_hash( \
20         addr & ~(CACHE_LINE_SIZE - 1), \
21         NUM_META_LOCKS - 1)])

23 struct State {
24     uint64_t powerFlag;
25     uint64_t seqNumber;
26     bool isLocked;
27     uint64_t rMetadata[NUM_META_LOCKS];
28     uint64_t wMetadata[NUM_META_LOCKS];
29     uint32_t uniqRCacheLines;
30     uint32_t uniqWCacheLines;
31     uint64_t lastPowerModeStartTime;
32     ...
33 } g_State;

35 struct ThreadInfo {
36     bool myPowerFlag;
37     uint64_t localSeqNumber;
38     ...
39 };

```

Fig. 10. Implementation details for power mode emulation.

```

41 T read_barrier(void *addr) {
42     ThreadInfo *tx = getThreadInfo();
43     if (!tx->myPowerFlag) {
44         uint64_t seqNumber = tx->localSeqNumber;
45         if (!ADDR_TO_WRITE_LOCK(addr) >= seqNumber)
46             htm_abort();
47     } else {
48         if (!ADDR_TO_READ_LOCK(addr) < g_State.seqNumber) {
49             *ADDR_TO_READ_LOCK(addr) = g_State.seqNumber;
50             membarstoreload();
51             if (!g_State.isLocked &&
52                 ++g_State.uniqRCacheLines > READ_CAPACITY) {
53                 // switch to the lock-based execution;
54                 // this aborts all regular transactions,
55                 // and forces them to wait for the lock
56                 // to become available again
57                 g_State.isLocked = true;
58                 membarstoreload();
59                 // calculate how much time we wasted
60                 // so far on this power mode transaction
61                 uint64_t timer = read_hw_clock();
62                 uint64_t delta = timer -
63                     g_State.lastPowerModeStartTime;
64                 // charge this amount of time for the
65                 // re-execution under lock
66                 while (read_hw_clock() - timer < delta);
67             }
68         }
69     }
70     return *addr;
71 }

73 void write_barrier(void *addr, T val) {
74     ThreadInfo *tx = getThreadInfo();
75     if (!tx->myPowerFlag) {
76         uint64_t seqNumber = tx->localSeqNumber;
77         if (!ADDR_TO_READ_LOCK(addr) >= seqNumber)
78             htm_abort();
79         if (!ADDR_TO_WRITE_LOCK(addr) >= seqNumber)
80             htm_abort();
81     } else {
82         if (!ADDR_TO_WRITE_LOCK(addr) < g_State.seqNumber) {
83             *ADDR_TO_WRITE_LOCK(addr) = g_State.seqNumber;
84             if (!g_State.isLocked &&
85                 ++g_State.uniqWCacheLines > WRITE_CAPACITY) {
86                 /* same as Lines 53-66 */
87             }
88         }
89     }
90     *addr = val;
91 }

```

Fig. 11. Instrumentation barriers used to implement the libitm interface of GCC.

The mapping between an address (or more precisely, a cache line) and its corresponding metalock uses a fast pseudo-uniform hash function described in Reference [44]. In our framework, we used very large arrays of 4M words representing metalocks (Lines 27 and 28) to reduce the chance that two cache lines will be mapped to the same metalock. Moreover, large arrays and a pseudo-uniform hash function mean that the chance that two cache lines accessed in the same transaction are mapped into adjacent metalock words is negligible, mitigating the chance for false sharing on the accessed metadata. As a result, it was not necessary to pad metalock words to avoid false sharing. Other fields in the State structure (cf. Figures 10); however, are properly padded (not shown for clarity).

Entering power mode is protected by a simple test-test-set lock (similar to the one shown in Figure 3) augmented with a sequence number (cf. Lines 24 and 25). The latter is incremented after every lock acquisition (that is, right after a transaction enters the power mode) and before lock release (that is, right before a power mode transaction commits). The sequence number serves the purpose of efficient release of all acquired metalocks. Specifically, an execution that uses a regular transaction stores the current sequence number in a thread-local variable (`localSeqNumber` in the `ThreadInfo` structure, Line 37) *before* starting on HTM (and thus any change to this number by a power mode transaction does not abort running regular transactions). Regular transactions use this number to check whether a metalock is “locked” by a power transaction (see Lines 44 and 76). Thus, once the sequence number is incremented at the end of a power transaction, any regular transaction starting and reading this number afterwards can deduce that all metalocks acquired by that power transaction have been released.

The power transaction (whose `myPowerFlag` is set) stores the current sequence number into the corresponding metalock word (Lines 49 and 83). We use an if-statement (Lines 48 and 82) to check whether the store is actually required to avoid writing the same value when the same cache line is accessed multiple times by a power transaction. (This if-statement also helps to keep track of the number of unique cache lines accesses for read and for write; the concrete use of these numbers is described later.) This optimization is more important for the read barrier, which requires a store-load memory fence (Line 50) to ensure that the metalock update becomes visible to regular transactions before the power transaction performs its read; otherwise, a power transaction may read inconsistent data. We note that in TSO architectures, such as Intel Haswell, the store-load memory fence is not required in the write barrier due to the total order on memory writes.

Notice that in the read instrumentation barrier, a regular transaction accesses a write metalock only (Line 45), while in the write barrier, it accesses both read and write metalocks (Lines 77 and 79). Thus, a regular transaction is able to share cache lines accessed by a power transaction for read, but it cannot acquire ownership of (i.e., write to) cache lines accessed by a power transaction for read or for write, as required.

The `uniqRCacheLines` and `uniqWCacheLines` fields of the `State` structure are used to keep track of the number of unique cache lines accessed by a power transaction for read and for write, respectively. As described above, we use these numbers to emulate capacity aborts by power transactions and re-execution under lock. Based on data in Reference [32], the read capacity of HTM in Intel Core i7-4770 machine (which is the machine we used for our evaluation) is several tens of thousands of cache lines, while the write capacity is a few hundreds of cache lines. Factors like cache associativity and hyper threading limit the effective capacity of hardware transactions. In fact, our experiments show that in some cases, transactions experience capacity aborts when they access only a few hundreds of cache lines for read and even less than that for write. As a result, we chose very conservative capacity limits for our evaluation (cf. Lines 4 and 5).

Taking the read barrier as an example, once the number of unique read cache lines goes beyond a threshold (Line 52), we switch to the lock-based execution by turning the `isLocked` flag on (Line 57). After that, we calculate how much time has passed since we started the power transaction, and spin for that amount of time, charging the lock-based execution for running the prefix of the (effectively, aborted) power transaction (Lines 61–66). Note that once the power transaction transitions into the lock-based execution, other, regular transactions are aborted and wait for the lock to become available again. Thus, the lock-based execution in a real system would take the same path as the power transaction, as it would access the same memory locations and read same values. As a result, the emulation of the time cost required to re-execute a power transaction under lock is realistic. Note that once the execution of the power transaction continues under lock, it

goes through same barriers to keep the cost of memory accesses comparable across all execution modes.

ACKNOWLEDGMENTS

The authors thank Sunjae Park for providing the implementation of and answering questions related to PleaseTM. The authors also thank Tim Harris, Jason Peck, and the anonymous referees for their valuable comments and helpful suggestions to improve the exposition of the material in the article.

REFERENCES

- [1] Yehuda Afek, Dave Dice, and Adam Morrison. 2011. Cache index-aware memory allocation. In *Proceedings of the International Symposium on Memory Management (ISMM'11)*. 55–64.
- [2] Yehuda Afek, Amir Levy, and Adam Morrison. 2014. Software-improved hardware lock elision. In *Proceedings of the ACM Symposium on Principles of Distributed Computing (PODC'14)*. 212–221.
- [3] C. S. Ananian, K. Asanovic, B. C. Kuszmaul, C. E. Leiserson, and S. Lie. 2005. Unbounded transactional memory. In *Proceedings of the International Symposium on High-Performance Computer Architecture (HPCA'05)*. 316–327.
- [4] Adrià Armejach, Ruben Titos-Gil, Anurag Negi, Osman S. Unsal, and Adrián Cristal. 2013. Techniques to improve performance in requester-wins hardware transactional memory. *ACM Trans. Archit. Code Optim.* 10, 4 (2013), 42:1–42:25.
- [5] Colin Blundell, Joe Devietti, E. Christopher Lewis, and Milo M. K. Martin. 2007. Making the fast case common and the uncommon case simple in unbounded transactional memory. In *Proceedings of the International Symposium on Computer Architecture (ISCA'07)*. 24–34.
- [6] Jayaram Bobba, Kevin E. Moore, Haris Volos, Luke Yen, Mark D. Hill, Michael M. Swift, and David A. Wood. 2007. Performance pathologies in hardware transactional memory. In *Proceedings of the International Symposium on Computer Architecture (ISCA'07)*. 81–91.
- [7] Trevor Brown, Alex Kogan, Yossi Lev, and Victor Luchangco. 2016. Investigating the performance of hardware transactions on a multi-socket machine. In *Proceedings of the ACM Symposium on Parallelism in Algorithms and Architectures (SPAA'16)*. 121–132.
- [8] Harold W. Cain, Maged M. Michael, Brad Frey, Cathy May, Derek Williams, and Hung Le. 2013. Robust architectural support for transactional memory in the power architecture. In *Proceedings of the International Symposium on Computer Architecture (ISCA'13)*. 225–236.
- [9] Luke Dalessandro, François Carouge, Sean White, Yossi Lev, Mark Moir, Michael L. Scott, and Michael F. Spear. 2011. Hybrid NOrec: a case study in the effectiveness of best effort hardware transactional memory. In *Proceedings of the 16th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS'11)*. 39–52.
- [10] Peter Damron, Alexandra Fedorova, Yossi Lev, Victor Luchangco, Mark Moir, and Daniel Nussbaum. 2006. Hybrid transactional memory. In *Proceedings of the International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS'06)*. 336–346.
- [11] Dave Dice, Alex Kogan, and Yossi Lev. 2016. Refined transactional lock elision. In *Proceedings of the Symposium on Principles and Practice of Parallel Programming (PPOPP'16)*.
- [12] Dave Dice, Alex Kogan, Yossi Lev, Timothy Merrifield, and Mark Moir. 2014. Adaptive integration of hardware and software lock elision techniques. In *Proceedings of the ACM Symposium on Parallelism in Algorithms and Architectures (SPAA'14)*. 188–197.
- [13] Dave Dice, Yossi Lev, Mark Moir, Daniel Nussbaum, and Marek Olszewski. 2009. Early experience with a commercial hardware transactional memory implementation. Technical report, Sun Labs.
- [14] David Dice, Yossi Lev, Mark Moir, and Daniel Nussbaum. 2009. Early experience with a commercial hardware transactional memory implementation. In *Proceedings of the International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS'09)*. 157–168.
- [15] Nuno Diegues, Paolo Romano, and Luís Rodrigues. 2014. Virtues and limitations of commodity hardware transactional memory. In *Proceedings of the 23rd International Conference on Parallel Architectures and Compilation (PACT'14)*. 3–14.
- [16] Ariel Eizenberg, Shiliang Hu, Gilles Pokam, and Joseph Devietti. 2016. Remix: Online detection and repair of cache contention for the JVM. In *Proceedings of the ACM Conference on Programming Language Design and Implementation (PLDI'16)*. 251–265.
- [17] GNU. 2015. Transactional memory in GCC. Retrieved August 3, 2015 from <https://gcc.gnu.org/wiki/TransactionalMemory>.

- [18] Tim Harris and Keir Fraser. 2003. Language support for lightweight transactions. In *Proceedings of the Conference on Object-oriented Programming, Systems, Languages, and Applications (OOPSLA '03)*. 388–402.
- [19] Blake A. Hechtman and Daniel J. Sorin. 2012. The limits of concurrency in cache coherence. In *Proceedings of the Workshop on Duplicating, Deconstructing and Debunking (WDDD'12)*.
- [20] Maurice Herlihy and J. Eliot B. Moss. 1993. Transactional memory: architectural support for lock-free data structures. In *Proceedings of the 20th Annual International Symposium on Computer Architecture (ISCA '93)*. 289–300. ACM Press.
- [21] Intel Corporation. 2017. Intel Xeon Processor Scalable Family Technical Overview. Retrieved November 20, 2017 from <https://software.intel.com/en-us/articles/intel-xeon-processor-scalable-family-technical-overview>.
- [22] Intel Corporation. 2012. Transactional Synchronization in Haswell. Retrieved September 8, 2012 from <http://software.intel.com/en-us/blogs/2012/02/07/transactional-synchronization-in-haswell/>.
- [23] Christian Jacobi, Timothy Slegel, and Dan Greiner. 2012. Transactional memory architecture and implementation for IBM System Z. In *Proceedings of the IEEE/ACM International Symposium on Microarchitecture (MICRO'12)*. 25–36.
- [24] Andi Kleen. 2014. Scaling existing lock-based applications with lock elision. *Communications of the ACM* 57, 3 (2014), 52–56. <https://dl.acm.org/citation.cfm?id=2576793>.
- [25] Yujie Liu and Michael Spear. 2011. Toxic transactions. In *Proceedings of the ACM SIGPLAN Workshop on Transactional Computing (TRANSACT'11)*.
- [26] Yujie Liu, Tingzhe Zhou, and Michael F. Spear. 2015. Transactional acceleration of concurrent data structures. In *Proceedings of the Symposium on Parallelism in Algorithms and Architectures (SPAA'15)*. 244–253.
- [27] M. Lupon, G. Magklis, and A. Gonzalez. 2009. Fastm: A log-based hardware transactional memory with fast abort recovery. In *Proceedings of the International Conference on Parallel Architectures and Compilation Techniques (PACT'09)*. 293–302.
- [28] Alexander Matveev and Nir Shavit. 2015. Reduced hardware norec: A safe and scalable hybrid transactional memory. In *Proceedings of the International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS'15)*. 59–71.
- [29] Chi Cao Minh, JaeWoong Chung, Christos Kozyrakis, and Kunle Olukotun. 2008. STAMP: stanford transactional applications for multi-processing. In *Proceedings of the International Symposium on Workload Characterization (IISWC'08)*. 35–46.
- [30] Kevin E. Moore, Jayaram Bobba, Michelle J. Moravan, Mark D. Hill, and David A. Wood. 2006. Logtm: Log-based transactional memory. In *Proceedings of the IEEE Symposium on High-Performance Computer Architecture (HPCA'06)*. 258–269.
- [31] Michelle J. Moravan, Jayaram Bobba, Kevin E. Moore, Luke Yen, Mark D. Hill, Ben Liblit, Michael M. Swift, and David A. Wood. 2006. Supporting nested transactional memory in logTM. In *Proceedings of the International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS'06)*. 359–370.
- [32] Takuya Nakaike, Rei Odaira, Matthew Gaudet, Maged Michael, and Hisanobu Tomari. 2015. Quantitative comparison of hardware transactional memory for Blue Gene/Q, zEnterprise EC12, Intel Core, and POWER8. In *Proceedings of the International Symposium on Computer Architecture (ISCA'15)*.
- [33] Yang Ni, Adam Welc, Ali-Reza Adl-Tabatabai, Moshe Bach, Sion Berkowits, James Cownie, Robert Geva, Sergey Kozhukov, Ravi Narayanaswamy, Jeffrey Olivier, Serguei Preis, Bratin Saha, Ady Tal, and Xinmin Tian. 2008. Design and implementation of transactional constructs for C/C++. In *Proceedings of the ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA'08)*. 195–212.
- [34] S. Park, M. Prvulovic, and C. J. Hughes. 2016. PleaseTM: Enabling transaction conflict management in requester-wins hardware transactional memory. In *Proceedings of the IEEE International Symposium on High Performance Computer Architecture (HPCA'16)*. 285–296.
- [35] Paul E. McKenney. 2017. Is Parallel Programming Hard, And, If So, What Can You Do About It? Retrieved November 20, 2017 from <https://www.kernel.org/pub/linux/kernel/people/paulmck/perfbook/perfbook.2011.01.02a.pdf>.
- [36] J. Poe, C. B. Cho, and T. Li. 2008. Using analytical models to efficiently explore hardware transactional memory and multi-core co-design. In *Proceedings of the International Symposium on Computer Architecture and High Performance Computing*. 159–166.
- [37] Ravi Rajwar and James R. Goodman. 2001. Speculative lock elision: Enabling highly concurrent multithreaded execution. In *Proceedings of the ACM/IEEE International Symposium on Microarchitecture (MICRO'01)*. 294–305.
- [38] Ravi Rajwar and James R. Goodman. 2002. Transactional lock-free execution of lock-based programs. In *Proceedings of the International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS'02)*. 5–17.
- [39] Ravi Rajwar and James R. Goodman. 2003. Transactional execution: Toward reliable, high-performance multithreading. *IEEE Micro* 23, 6 (2003), 117–125.

- [40] Ravi Rajwar, Maurice Herlihy, and Konrad Lai. 2005. Virtualizing transactional memory. In *Proceedings of the International Symposium on Computer Architecture (ISCA'05)*. 494–505.
- [41] Jose Renau, Basilio Fraguera, James Tuck, Wei Liu, Milos Prvulovic, Luis Ceze, Smruti Sarangi, Paul Sack, Karin Strauss, and Pablo Montesinos. 2005. SESC simulator, January 2005. Retrieved from <http://sesc.sourceforge.net>.
- [42] Christoph Scheurich and Michel Dubois. 1987. Correct memory operation of cache-based multiprocessors. In *Proceedings of the International Symposium on Computer Architecture (ISCA'87)*. 234–243.
- [43] Amy Wang, Matthew Gaudet, Peng Wu, José Nelson Amaral, Martin Ohmacht, Christopher Barton, Raul Silvera, and Maged Michael. 2012. Evaluation of Blue Gene/Q hardware support for transactional memories. In *Proceedings of the International Conference on Parallel Architectures and Compilation Techniques (PACT'12)*. 127–136.
- [44] Thomas Wang. 2015. Integer Hash Function, 2007. Retrieved August 3, 2015 from <http://web.archive.org/web/20071223173210http://www.concentric.net/Ttwang/tech/inthash.htm>.
- [45] Roel Wieringa. 2003. *Design Methods for Reactive Systems: Yourdan, Statemate, and the UML*. Morgan Kaufmann Publishers, Boston.
- [46] L. Yen, J. Bobba, M. R. Marty, K. E. Moore, H. Volos, M. D. Hill, M. M. Swift, and D. A. Wood. 2007. LogTM-SE: Decoupling hardware transactional memory from caches. In *Proceedings of the International Symposium on High Performance Computer Architecture (HPCA'07)*. 261–272.
- [47] Richard M. Yoo, Christopher J. Hughes, Konrad Lai, and Ravi Rajwar. 2013. Performance evaluation of Intel® transactional synchronization extensions for high-performance computing. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis (SC'13)*.

Received July 2017; revised November 2017; accepted December 2017