Primus Inter Pares*: Improving Parallelism in Hardware Transactional Memory

ABSTRACT

Hardware transactional memory (HTM) is supported by recent processors from Intel and IBM. HTM is attractive because it can enhance concurrency while simplifying programming. Today's HTM systems rely on existing coherence protocols, which implement a requester-wins strategy. This, in turn, leads to very poor performance when transactions frequently conflict, causing them to resort to a non-speculative fallback path. Often, such a path severely limits concurrency.

In this paper, we propose very simple architectural changes to the existing requester-wins HTM architectures. These changes permit higher levels of concurrency when transactions cannot make progress and require a fallback path. The idea is to support a special mode of execution in HTM, called power mode, which can be used to enhance conflict resolution between regular and so-called power transactions. Our idea is backward-compatible with existing HTM code, imposing no additional cost on transactions that do not use the power mode. In addition, it supports dynamic undesired data sharing detection, indicating when transactions whose data sets should be disjoint, are not. Using extensive evaluation of micro- and STAMP benchmarks in a transactional memory simulator and real hardware-based emulation, we show that our technique significantly improves performance of the baseline that does not use power mode, and performs similarly or better than state-of-the-art related proposals that require mode substantial architectural changes.

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 [18] and IBM [7, 19, 33]. 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*, that code block appears to have executed instantaneously, while if it *aborts*, 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, 37]. However, the experience shows that even with a moderate level of conflicts between hardware transactions, the performance of HTM substantially deteriorates [11, 12, 14, 37]. That is because most existing HTM implementations piggy-back on cache coherence protocols [17], which mostly implement a *requester-wins* policy: if one transaction requests exclusive access to a cache line held by another, the earlier transaction aborts and restarts. Thus, data conflicts cause repetitive transactional aborts, 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 [13]), and taking this path aborts any concurrent speculative transactions, even when there is no actual data conflict between the speculative and non-speculative threads.

This paper's contribution is to propose a strikingly simple mechanism that allows HTM to continue speculating 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, in order to support power mode, each coherence request is augmented with one bit indicating whether the request is coming from a core speculating on HTM. The power transaction replies with a NACK to coherence requests from other transactions, causing the requester to abort and allowing the power transaction to proceed. 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, they might repeatedly conflict and abort each other. Once one of those transactions decides to abandon speculation, it will choose to execute under lock, causing 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;

*From Latin: First Among Equals.

all other operations working on different nodes would be able to seamlessly continue their speculation. As we describe later in the paper, a simple software or hardware mechanism can be put in place to ensure that only one transaction enters the power mode at a time.

We note that power transactions are "backward-compatible" with existing HTM systems in the sense that allowing 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.

We used two approaches to evaluate the utility of power transactions. First, using software emulation of a hardware power transaction implementation, we conducted experiments to compare the relative performance of a number of micro- and STAMP [23] 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 transaction automatically aborts all speculative transactions, fails to exploit substantial opportunities for concurrent execution.

Second, we added the power mode support to SuperTrans [29], a transactional memory simulator built from SESC [32], that was recently enhanced to more accurately simulate best-effort HTM similar to Intel TSX [28]. Using SuperTrans, we compared the utility of power mode to two variants of PleaseTM, a recent related proposal for improving parallelism in HTM [28], as well as to the baseline implementation that does not use power mode. Using STAMP benchmarks [23], we show that power mode not only provides non-trivial speedup above the baseline implementation (confirming our emulation-based study), but also performs similarly or better than both variants of PleaseTM despite requiring less architectural changes.

An additional, secondary benefit of supporting power mode is that it provides a lightweight mechanism for dynamic transactional *undesired data sharing detection*. Transactional programming can introduce new kinds of performance challenges. A transactional undesired data sharing occurs when two transactions that are believed to have disjoint data sets 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. As explained below, power mode can be used to detect and flag such unexpected synchronization conflicts, giving the programmer a powerful new tool for performance debugging.

2. RELATED WORK

In Intel Haswell [18] and its successors, as well as in IBM Power 8 [7], 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 *transactional lock elision* [13] (TLE), 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]. If the speculative lock elision fails (typically, after a few retries), 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 papers show that TLE is very effective when most transactions succeed, but its benefit fades once the lock is acquired often [11, 14, 37]. In order to keep our usage examples of power transactions concrete, we focus on the use of lock as the alternative path, effectively enhancing the standard TLE technique [13]. We note, though, that power mode is equally helpful in reducing the use of any non-speculative fallback path, such as the one implemented using software transactional memory (STM) [8,22], lock-free techniques [20], etc.

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 [4]. One of the variants of OneTM, called OneTM-Concurrent, supports concurrent execution of non-overflown transactions and non-transactional code with one overflown transaction. In order 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 architectured register, saved and restored on every context switch. Being designed to enhance existing (bounded) HTM architectures, power mode does not require any of those complications.

In the context of software transactional memory, Ni et al. [27] 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 [27], "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". The control over execution modes and conflict resolution between transactions in [27] 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 [17], numerous attempts have been made to improve and extend it (e.g., [1, 21, 24, 25, 31] to give just a very few examples). The scope of this paper 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 [1, 21, 24,

36], running transactions across context switches [31, 36], supporting nested transactions [25, 36], etc.) is typically different from the one considered in this paper.

Perhaps the most relevant prior work to this paper is the recent paper by Park et al. on a PleaseTM mechanism for requester-wins HTM systems [28]. In PleaseTM, hardware transactions insert 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 upon receiving a coherence request. In order to continue its execution, the 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, it slows down the responding transaction, and would do that over and over again with every retry. In opposite to PleaseTM, power mode allows resolution of conflicts at the time the request is received, without slowing down the responding transaction. Furthermore, transactions in PleaseTM can still repeatedly abort each other, as they respond with pleading bits to each other's requests. At the same time, power provides more definitive control over which transaction would receive priority over others. We believe that for those reasons, power transactions provide better performance than PleaseTM, as demonstrated in Section 5.

In another relevant paper, Armejach et al. consider a few hardware and hybrid (software and hardware) techniques to improve performance of requester-wins HTM [2]. The most relevant technique to our work is perhaps the one called DRW (delayed requester-wins). 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 seems to require hardware changes that are much more substantial than supporting power mode transactions.

Baugh et al. describe how fine-grained memory protection [38] can be used to build a strongly-atomic hybrid transactional memory [3]. The idea is to allow transactions running using software transactional memory to protect memory locations they are reading or writing by setting auxiliary protection bits added to each cache line; hardware transactions or non-transactional code that attempt to access protected locations receive a protection fault, and back off or abort. Supporting fine-grained memory protection requires many intrusive architectural changes that have to make sure that protection bits stay associated with the data throughout the memory hierarchy. Furthermore, this idea is designed for hybrid transactional memory that has an STM component and is not directly applicable to hardware transactional memory systems. Finally, to make use of the fine-grained memory protection, one needs to instrument every transactional access (executed by STM) and invoke special instructions (for setting protection bits) for every such access; power-mode transactions do not require any of those.

3. POWER-MODE TRANSACTIONS

We first describe the common mechanism required to support power transactions regardless of how transactions enter the power mode. Next, we discuss the details of supporting software-controlled entry into the power mode, followed by details on hardware-controlled entry. 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 [18]. Note that the exit method from power mode does not require any special treatment, i.e., power mode transactions commit or abort exactly as regular transactions. Finally, we discuss variations in our design along with their impact on the properties of power mode transactions.

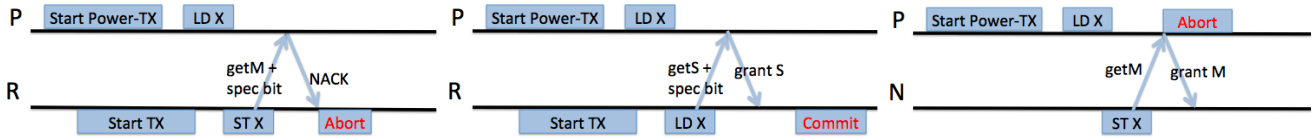
3.1 Common Mechanism

Supporting the power mode requires each hardware thread to support 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-mode transaction (via one of the mechanisms described in the subsequent sections) and reset when the thread completes its 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, the cache controller logic remains unchanged. We note that in order 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.



(a) R conflicting with P (b) R not conflicting with P (c) N conflicting with P

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

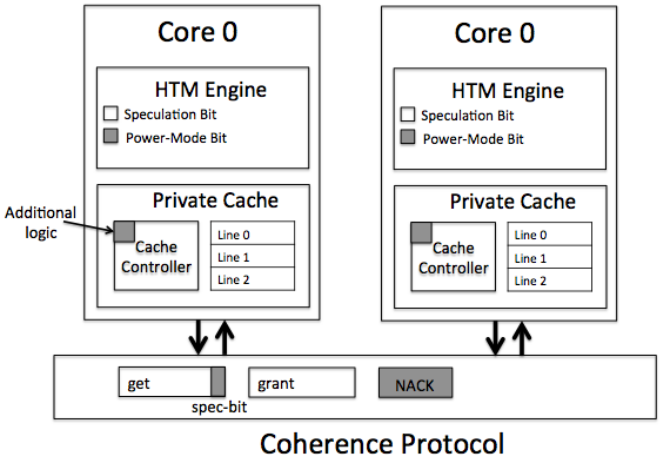


Figure 2: Architectural changes required to support power-mode transactions.

Specifically, when the NACK is received and the receiving thread is speculating 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-mode 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-mode transaction (P), a thread running another (regular or power-mode) transaction (R), and a thread running a 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 papers on computer architecture considered adding a NACK message (e.g., [5,21,36] to give a few examples). As opposite to most of that work, however, the very limited use of NACKs in our case does not result in any additional change in the coherence protocol, such as new coherence states.

The required changes in an HTM architecture are summarized in Figure 2 with new or modified components shown in grey. We truly believe that all the proposed changes are simple, if not trivial. Even comparing to the relatively lightweight PleaseTM mechanism [28], we consider the changes required to support power mode to be less intrusive, and thus more feasible.

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

3.2 Software-controlled Entry

In order to allow software to control which transaction(s) would run in a power mode, one new instruction should be added to the 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.

It is the responsibility of the programmer to ensure (or not) that only one transaction switches into the power mode at a time. Following is one particular mechanism to achieve that, integrated with a common implementation of the TLE mechanism [13]. In this mechanism, 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 escalates to power mode if it repeatedly fails to commit. The atomic compare-and-swap (CAS) (Line 28) 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 transactions do not access the powerFlag flag. Thus, starting (and committing) a power-mode 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 and lock in standard TLE as well. 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, we discount this attempt by decrementing the ntrials counter (Line 17). 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 lock becomes available (Line 31).

3.3 Hardware-controlled Entry

```
initially : (global) lock = 0;
            (global) powerFlag = −1;
            (thread local) tID = unique thread ID;
```

Lock procedure:

```
1  ntrials = 0;
2  while (true) {
3    // start a regular or power−mode transaction
4    // according to the value of 'powerFlag'
5    if ((powerFlag != tID && begin_htm()) ||
6        (powerFlag == tID && begin_power_htm())) {
7      if (isLocked(&lock)) self−abort();
8      return;
9    }
10   if (! self−aborted) {
11     if (powerFlag == tID) {
12       // exit the power mode and fall to the lock
13       powerFlag = −1;
14       break;
15     } else {
16       // avoid the lemming effect
17       ntrials −−;
18     }
19   }
20   // increase the counter for the number
21   // of non−power mode trials
22   if (++ntrials >= MAX_TLE_TRIALS) {
23     // if we exhausted the number of non−power
24     // mode trials, check if the 'powerFlag' flag is
25     // available and try to set it.
26     // Note: for TLE we would simply break here
27     // and fall to the lock
28     if (powerFlag == −1) CAS(&powerFlag, −1, tID);
29   }
30   // wait for the lock to become available
31   if (isLocked(&lock)) wait;
32 }
33 // we failed to commit a transaction, grab the lock
34 Lock(&lock);
```

Unlock procedure:

```
1  if (!isLocked(&lock)) {
2    commit_htm();
3    if (powerFlag == tID) powerFlag = −1;
4  } else {
5    Unlock(&lock);
6  }
```

Figure 3: TLE using power mode transactions

Along (or instead of) a software-controlled entry into power mode, the HTM engine itself may control when the 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-mode transaction at a time, either system-wide or for each process. There are many ways to implement such functionality, which requires the ability to arbitrate concurrent requests from multiple hardware threads. One option, similar to the proposal made in [4] 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 [4], 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 [18] 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, it may try to atomically set the transaction status word, and if succeeded, it shall set its power-mode bit, and run a power-mode transaction. Upon 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 mode transaction is in progress, 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-mode 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-mode transactions against conflicts with non-transactional threads (but not with other power-mode 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-mode thread can refuse invalidations. 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 easily be generalized to encompass multiple levels or 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 [35]. 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 as long as the number of power mode levels is not larger than the number of bits that can be stored in the transaction status word.

As mentioned in Section 3.1, when a transaction receives 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 unexpected data sharing between transactions. That is, to test whether two transactions have disjoint data sets, run them concurrently in power mode, and if one aborts with the power-mode conflict abort code, then the transactions' data sets are not disjoint, and there is a possibly unexpected data sharing.

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 [9], 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 a hardware transaction is comparable to running that transaction in software[1]. This is also echoed by results of single-thread performance in various papers [11, 26, 37]. 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.

In order 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 [9, 16]. We instrument all memory accesses in transactions to read and update ownership records as appropriate, leveraging a recently introduced compiler support for a completely atomatic 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 exists. 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).

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

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, 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 benefits of power mode. In the following subsection, we expand on implementation aspects of our framework.

4.1.2 Implementation Details

The GCC compiler [15] (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 alternatives, such as TLE, STM, or lock synchronization, as well as the opportunity to provide customized instrumentation barriers and functions to be called when transactions 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 its cache lines, and aborts if it finds a metalock held in a conflicting mode. If the metalock does not conflict, 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, doubling the number of accessed cache lines. However, regular transactions access the additional (metalock) cache

[2] The 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.

lines only for reading, which does not stress Haswell HTM, whose read set capacity is relatively large [26]. 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 records 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 δ elapsed since the transaction stated, switch to locking mode, and spin for another duration δ, effectively charging twice for the transaction so far. Once a power mode transaction switches to the lock (simply by setting a Boolean flag), as in standard TLE, all regular transactions are aborted and wait for the lock to become available again. By spinning after lock acquisition, 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.

The mapping between an address (or more precisely, a cache line) and its corresponding metalock uses a fast pseudo-uniform hash function described in [34]. In our framework, we used very large arrays of 4M words representing metalocks 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.

Our experiments were run on an Intel Haswell (Core i7-4770) 4-core hyper-threaded machine (8 hardware threads in total). Before starting measurements, all threads were set to spin for a few seconds to allow the system to warm up. Our goal was to compare standard TLE [13] with one that makes use of power mode 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.

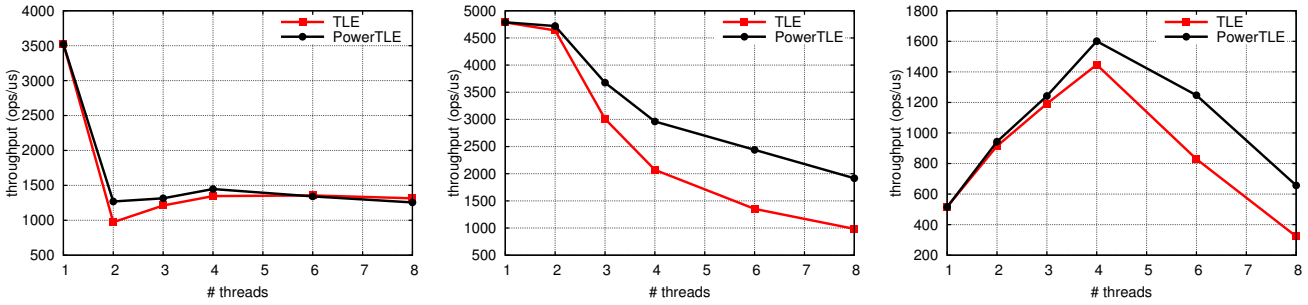
Each critical section was attempted ten times using regular transactions before reverting to lock (in TLE) or power mode (in PowerTLE). As demonstrated in Figure 3, a power mode transaction is tried only once (or more, in case of self-abort indicating that the lock is taken). We note, however, that other, 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, 14], it falls out of scope of this paper.

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 ten runs performed in the same configuration. The breakdown of operations between different modes of executions, e.g., regular transactions, power transactions, etc., 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 some power mode transaction was running (denoted as C TXs).

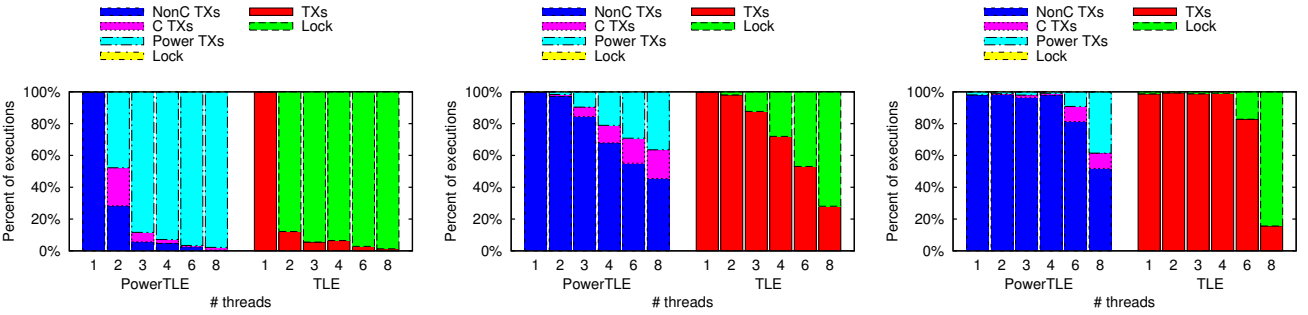
For the experiment reported in Figure 4 (a), the queue is initialized with 100K elements, and all threads run a total number of 100K RemoveMin operations, divided equally among the participating threads. We measure the time from the start till the last thread is done with its operations, and calculate throughput by dividing the total number of performed operations (100K) by this time. In this particular workload, all threads compete with each other over the minimal element in the queue. Not surprisingly, except for two threads, power mode does not increase throughput, 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 very few regular transactions manage to complete, while the majority of operations is executed using a lock (in TLE) or power mode transactions (in PowerTLE). The case of two threads is slightly different, and shows that substantial portion of regular transactions completes concurrently with a power-mode transaction. Indeed, this is the only point where PowerTLE beats TLE by a large gap (cf. Figure 4 (a)). We believe this happens because a regular transaction (running a very short RemoveMin operation) manages to "sneak in" without any contention while another thread transitions into power mode and before the actual data conflict occurs. In TLE, all transactions are aborted at the moment the lock is acquired, and thus many transactions do not have enough time to complete when another thread switches to lock. When we increase the number of threads, this benefit of PowerTLE fades as regular transactions conflict with each other.

In the experiment reported in Figure 4 (b), the queue is initialized with 100K elements, and each thread runs loop iterations for 5 seconds, 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 Insert operation in power mode, other threads can proceed concurrently to apply their non-conflicting operations. As a result, at 8 threads, PowerTLE achieves almost 2x more throughput than TLE. Figure 5 (b) shows that, indeed, some portion of regular transactions manages to compete concurrently with a power mode transaction, and this portion grows with the number of threads. Interestingly, the portion of regular transactions completing non-concurrently with a power mode transaction is also larger for PowerTLE than the portion of transactions in TLE. We attribute that to the decreased



(a) RemoveMin only (b) 50% RemoveMin, 50% Insert (c) Insert only

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



(a) RemoveMin only (b) 50% RemoveMin, 50% Insert (c) Insert only

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

lemming effect [13] that the power mode transaction has comparing to lock, as the former does not abort all transactions but only those conflicting with it.

The benefit of PowerTLE over TLE increases even further 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 100K insert operations, divided equally among threads. At 8 threads, PowerTLE achieves more than 2x throughput of TLE. 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 present results of a set microbenchmark implemented on top of AVL trees. The AVL tree implementation is similar to the one found in OpenSolaris. In all experiments, each thread runs iterations for 5 seconds, 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).

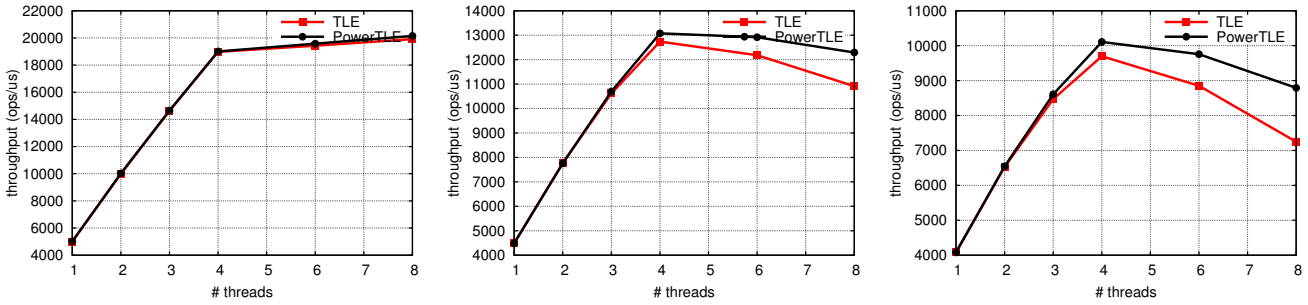
Figure 6 (a) shows results for the read only workload where all threads perform only Find operations. Here, the vast majority of operations succeed without any retries, and thus power mode is not used. The breakdown of execution modes shows that, indeed, virtually all operations succeed using regular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

The workloads in Figure 6 (b) and (c) include update operations. Specifically, the former shows results for an experiment in which 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 (Figures 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.

4.4 STAMP

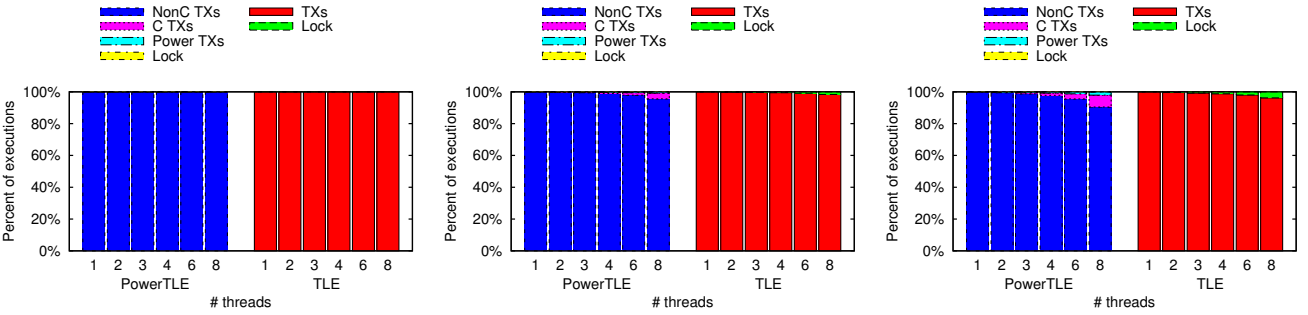
This section presents results measured with the STAMP benchmarking suite [23], which is used extensively in transactional memory research[3]. For each benchmark, we used a

[3] We used a version of STAMP available at `https://github.com/mfs409/stamp`.



(a) Read only (b) 60% Find, 20% Insert, 20% Remove (c) 20% Find, 40% Insert, 40% Remove

Figure 6: AVL tree-based set throughput. Higher is better.



(a) Read only (b) 60% Find, 20% Insert, 20% Remove (c) 20% Find, 40% Insert, 40% Remove

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

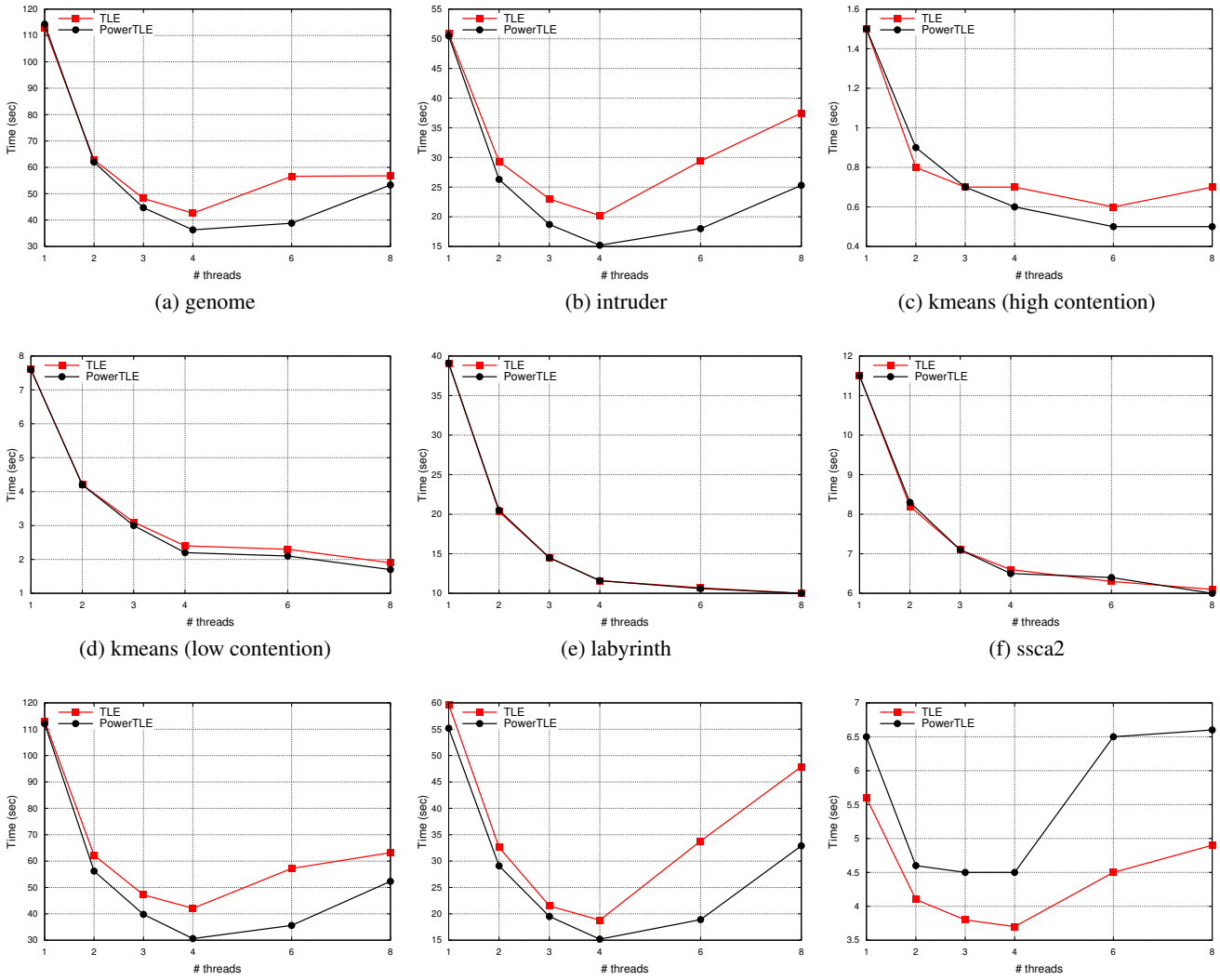
standard ('native') set of command line parameters. Figure 8 shows running time reported by each benchmark, averaged over ten runs. We omit the results for one of the STAMP benchmarks (namely, bayes) due to extremely high variance (which was also observed by others [28, 37]).

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. Interestingly, 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 8 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. In fact, the only place where PowerTLE improves slightly over TLE in kmeans-low is when a small fraction of critical sections fail to the lock (in TLE) or revert to power mode (in PowerTLE) as the number of threads grows. 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 is executed using power mode 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 mode transactions, PowerTLE achieves essentially the same results as TLE for this benchmark.

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 mode 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 [6]. Exploring the impact of this optimization is in our future work.



(a) genome (b) intruder (c) kmeans (high contention)

(d) kmeans (low contention) (e) labyrinth (f) ssca2

(g) vacation (high contention) (h) vacation (low contention) (i) yada

Figure 8: STAMP running time measurements. Lower is better.

5. SIMULATOR-BASED EVALUATION

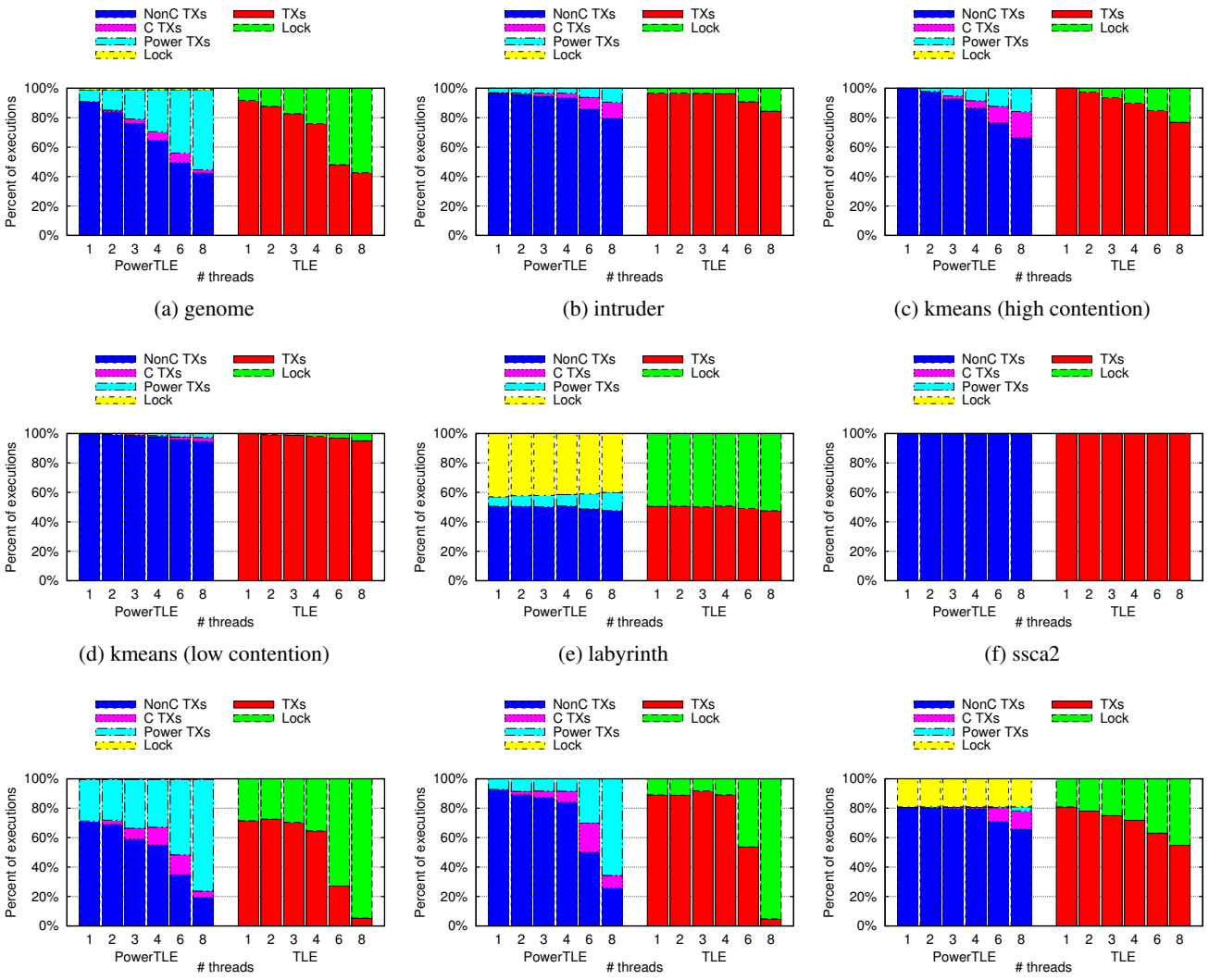
In addition to the framework discussed above, we added support for power mode into SuperTrans [29], a transactional memory simulator built on top of SESC [32]. As reported in [28], SuperTrans was enhanced with a best-effort HTM support similar to Intel TSX.

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 a 8 by 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.

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

We simulate a system with 16 threads running STAMP [23] 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 [28]. For the latter, we use two variations called ResponderWins and MoreReadsWins. In the former, the requester running a hardware transaction and receiving a line with the plea bit set, aborts its transaction. In the latter, each core tracks the number of cache lines read transactionally and includes this counter along with the plea bit. The requester



(a) genome (b) intruder (c) kmeans (high contention)

(d) kmeans (low contention) (e) labyrinth (f) ssca2

(g) vacation (high contention) (h) vacation (low contention) (i) yada

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

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 [28] and implemented in SuperTrans by their authors.

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, and performs better or on par with both 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 at least 30%) improve performance of 3 benchmarks, while not harming the performance of any other benchmark by more than 10%.

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

Benchmark	Baseline	RequesterWins	MoreReadsWins	PowerTLE
genome	1.000	0.912	0.859	0.928
intruder	1.000	0.847	1.180	1.100
kmeans (low)	1.000	0.999	0.999	1.003
kmeans (high)	1.000	0.527	0.527	0.667
labyrinth	1.000	0.950	1.126	0.782
ssca2	1.000	1.002	1.003	1.014
vacation (low)	1.000	0.994	1.011	1.047
vacation (high)	1.000	0.948	0.971	1.035
yada	1.000	0.952	0.978	0.521
mean	1.000	0.903	0.962	0.900

Table 1: Relative performance of STAMP (lower is better).

Benchmark	Baseline	RequesterWins	MoreReadsWins	PowerTLE
genome	1.3	1.1	0.9	0.0
intruder	15.2	9.3	20.3	0.1
kmeans (low)	0.0	0.0	0.0	0.0
kmeans (high)	5.6	0.0	0.0	0.0
labyrinth	25.4	26.3	26.8	0.0
ssca2	0.0	0.0	0.0	0.0
vacation (low)	0.0	0.0	0.0	0.0
vacation (high)	0.4	0.1	0.3	0.0
yada	28.1	17.1	16.8	0.0

Table 2: Percent of transactions that fall to the lock.

HTM is a promising tool to ease the development and accelerate the performance of concurrent code. Most existing HTM implementations rely on existing 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 [11,12,14,37]. The second property means that in order to guarantee progress, concurrent programs must include a non-speculative fallback path. This path is typically implemented by using a lock [13]; 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 paper, we aim to alleviate these issues through the introduction of special power transactions. These transactions receive priority in conflict resolution with other, regular transactions. We show that supporting power transactions requires very simple, almost trivial changes to existing best-effort requester-wins HTM implementations. Furthermore, our 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 how power transactions improve parallelism between transactions. This, in turn, leads to significant benefits for HTM that supports power transactions over the one that does not.

7. REFERENCES

[1] C. S. Ananian, K. Asanovic, B. C. Kuszmaul, C. E. Leiserson, and S. Lie. Unbounded transactional memory. In *Proceedings of the International Symposium on High-Performance Computer Architecture (HPCA)*, pages 316–327, 2005.

[2] Adrià Armejach, Ruben Titos-Gil, Anurag Negi, Osman S. Unsal, and Adrián Cristal. Techniques to improve performance in requester-wins hardware transactional memory. *ACM Trans. Archit. Code Optim.*, 10(4):42:1–42:25, 2013.

[3] Lee Baugh, Naveen Neelakantam, and Craig Zilles. Using hardware memory protection to build a high-performance, strongly-atomic hybrid transactional memory. In *Proceedings of the International Symposium on Computer Architecture (ISCA)*, pages 115–126, 2008.

[4] Colin Blundell, Joe Devietti, E. Christopher Lewis, and Milo M. K. Martin. Making the fast case common and the uncommon case simple in unbounded transactional memory. In *Proceedings of the International Symposium on Computer Architecture (ISCA)*, pages 24–34, 2007.

[5] Jayaram Bobba, Kevin E. Moore, Haris Volos, Luke Yen, Mark D. Hill, Michael M. Swift, and David A. Wood. Performance pathologies in hardware transactional memory. In *Proceedings of the International Symposium on Computer Architecture (ISCA)*, pages 81–91, 2007.

[6] Trevor Brown, Alex Kogan, Yossi Lev, and Victor Luchangco. Investigating the performance of hardware transactions on a multi-socket machine. In *Proceedings of the ACM Symposium on Parallelism in Algorithms and Architectures (SPAA)*, pages 121–132, 2016.

[7] Harold W. Cain, Maged M. Michael, Brad Frey, Cathy May, Derek Williams, and Hung Le. Robust architectural support for transactional memory in the power architecture. In *Proceedings of the International Symposium on Computer Architecture (ISCA)*, pages 225–236, 2013.

[8] Luke Dalessandro, François Carouge, Sean White, Yossi Lev, Mark Moir, Michael L. Scott, and Michael F. Spear. 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*, pages 39–52, 2011.

[9] Peter Damron, Alexandra Fedorova, Yossi Lev, Victor Luchangco, Mark Moir, and Daniel Nussbaum. Hybrid transactional memory. In *Proceedings of the International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, pages 336–346, 2006.

[10] Dave Dice, Alex Kogan, and Yossi Lev. Refined transactional lock elision. In *ACM SIGPLAN Workshop on Transactional Computing (TRANSACT)*, 2015.

[11] Dave Dice, Alex Kogan, and Yossi Lev. Refined transactional lock elision. In *Proceedings of the Symposium on Principles and Practice of Parallel Programming (PPOPP)*, 2016.

[12] Dave Dice, Alex Kogan, Yossi Lev, Timothy Merrifield, and Mark Moir. Adaptive integration of hardware and software lock elision techniques. In *ACM Symposium on Parallelism in Algorithms and Architectures (SPAA)*, pages 188–197, 2014.

[13] David Dice, Yossi Lev, Mark Moir, and Daniel Nussbaum. 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)*, pages 157–168, 2009.

[14] Nuno Diegues, Paolo Romano, and Luís Rodrigues. Virtues and limitations of commodity hardware transactional memory. In *Proceedings of the 23rd International Conference on Parallel Architectures and Compilation (PACT)*, pages 3–14, 2014.

[15] GNU. Transactional memory in GCC. Retrieved from `https://gcc.gnu.org/wiki/TransactionalMemory` on 3 August 2015.

[16] Tim Harris and Keir Fraser. Language support for lightweight transactions. In *Proceedings of the Conference on Object-oriented Programing, Systems, Languages, and Applications (OOPSLA)*, pages 388–402, 2003.

[17] Maurice Herlihy and J. Eliot B. Moss. Transactional memory: architectural support for lock-free data structures. In *Proceedings of the 20th Annual International Symposium on Computer Architecture (ISCA'93)*, pages 289–300. ACM Press, 1993.

[18] Intel Corporation. Transactional Synchronization in Haswell. Retrieved from `http://software.intel.com/en-us/blogs/2012/02/07/transactional-synchronization-in-haswell/`, 8 September 2012.

[19] Christian Jacobi, Timothy Slegel, and Dan Greiner. Transactional memory architecture and implementation for IBM System Z. In *Proceedings of the IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pages 25–36, 2012.

[20] Yujie Liu, Tingzhe Zhou, and Michael F. Spear. Transactional acceleration of concurrent data structures. In *Proceedings of Symposium on Parallelism in Algorithms and Architectures (SPAA)*, pages 244–253, 2015.

[21] M. Lupon, G. Magklis, and A. Gonzalez. Fastm: A log-based hardware transactional memory with fast abort recovery. In *Proceedings of the International Conference on Parallel Architectures and Compilation Techniques (PACT)*, pages 293–302, 2009.

[22] Alexander Matveev and Nir Shavit. 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)*, pages 59–71, 2015.

[23] Chi Cao Minh, JaeWoong Chung, Christos Kozyrakis, and Kunle Olukotun. STAMP: stanford transactional applications for multi-processing. In *Proceedings of the International Symposium on Workload Characterization (IISWC)*, pages 35–46, 2008.

[24] Kevin E. Moore, Jayaram Bobba, Michelle J. Moravan, and Mark D. Hill an David A. Wood. Logtm: Log-based transactional memory. In *Proceedings of the IEEE Symposium on High-Performance Computer Architecture (HPCA)*, pages 258–269, 2006.

[25] Michelle J. Moravan, Jayaram Bobba, Kevin E. Moore, Luke Yen, Mark D. Hill, Ben Liblit, Michael M. Swift, and David A. Wood. Supporting nested transactional memory in logTM. In *Proceedings of the International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, pages 359–370, 2006.

[26] Takuya Nakaike, Rei Odaira, Matthew Gaudet, Maged Michael, and Hisanobu Tomari. 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)*, 2015.

[27] Yang Ni, Adam Welc, Ali-Reza Adl-Tabatabai, Moshe Bach, Sion Berkowits, James Cownie, Robert Geva, Sergey Kozhukow, Ravi Narayanaswamy, Jeffrey Olivier, Serguei Preis, Bratin Saha, Ady Tal, and Xinmin Tian. 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)*, pages 195–212, 2008.

[28] S. Park, M. Prvulovic, and C. J. Hughes. PleaseTM: Enabling transaction conflict management in requester-wins hardware transactional memory. In *Proceedings of the IEEE International Symposium on High Performance Computer Architecture (HPCA)*, pages 285–296, 2016.

[29] J. Poe, C. B. Cho, and T. Li. 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*, pages 159–166, 2008.

[30] Ravi Rajwar and James R. Goodman. Speculative lock elision: Enabling highly concurrent multithreaded execution. In *Proceedings of the ACM/IEEE International Symposium on Microarchitecture (MICRO)*, pages 294–305, 2001.

[31] Ravi Rajwar, Maurice Herlihy, and Konrad Lai. Virtualizing transactional memory. In *Proceedings of the International Symposium on Computer Architecture (ISCA)*, pages 494–505, 2005.

[32] Jose Renau, Basilio Fraguela, James Tuck, Wei Liu, Milos Prvulovic, Luis Ceze, Smruti Sarangi, Paul Sack, Karin Strauss, and Pablo Montesinos. SESC simulator, January 2005. http://sesc.sourceforge.net.

[33] Amy Wang, Matthew Gaudet, Peng Wu, José Nelson Amaral, Martin Ohmacht, Christopher Barton, Raul Silvera, and Maged Michael. Evaluation of Blue Gene/Q hardware support for transactional memories. In *Proceedings of the International Conference on Parallel Architectures and Compilation Techniques (PACT)*, pages 127–136, 2012.

[34] Thomas Wang. Integer hash function, 2007. Retrieved from `http://web.archive.org/web/20071223173210/http://www.concentric.net/~Ttwang/tech/inthash.htm`, 3 August 2015.

[35] Roel Wieringa. *Design Methods for Reactive Systems: Yourdan, Statemate, and the UML*. Morgan Kaufmann Publishers, Boston, 2003.

[36] 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 *Proceedings of the International Symposium on High Performance Computer Architecture (HPCA)*, pages 261–272, 2007.

[37] Richard M. Yoo, Christopher J. Hughes, Konrad Lai, and Ravi Rajwar. Performance evaluation of Intel® transactional synchronization extensions for high-performance computing. In *International Conference for High Performance Computing, Networking, Storage and Analysis (SC)*, 2013.

[38] Pin Zhou, Feng Qin, Wei Liu, Yuanyuan Zhou, and Josep Torrellas. iWatcher: Efficient Architectural Support for Software Debugging. In *Proceedings of the International Symposium on Computer Architecture (ISCA)*, pages 224–235, 2004.

8. APPENDIX A

Figures 10 and 11 show some details of our implementation, 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.

Entering power mode is protected by a simple test-test-set lock (Line 24) augmented with a sequence number (Line 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 the metalock is "locked" by a power mode transaction (see Lines 44 and 76). Thus, once the sequence number is incremented at the end of the power mode transaction, any regular transaction starting and reading this number afterwards can deduce that all metalocks have been released.

The power mode transaction stores the current sequence number into the corresponding metalock word (Lines 49 and 84). We use an if-statement (Lines 48 and 83) 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 mode 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 mode transaction performs its read; otherwise, a power mode 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 mode transaction for read, but it cannot acquire ownership of cache lines accessed by a power mode 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 mode transaction for read

```
#define NUM_META_LOCKS (4 * 1024 * 1024)
#define CACHE_LINE_SIZE (64)

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

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

// These macros translate from an address to a
// read/write meta lock protecting the cache line
// where the address belongs to.
#define ADDR_TO_READ_LOCK(addr)
  \
    (&rMetadata[fast_hash(
  \
        addr&~(CACHE_LINE_SIZE-1),
  \
        NUM_META_LOCKS-1)])
#define ADDR_TO_WRITE_LOCK(addr)
  \
    (&wMetadata[fast_hash(
  \
        addr&~(CACHE_LINE_SIZE-1),
  \
        NUM_META_LOCKS-1)])

struct State {
  uint64_t powerFlag;
  uint64_t seqNumber;
  bool isLocked;
  uint64_t rMetadata[NUM_META_LOCKS];
  uint64_t wMetadata[NUM_META_LOCKS];
  uint32_t uniqRCacheLines;
  uint32_t uniqWCacheLines;
  uint64_t lastPowerModeStartTime;
  ...
} g_State;

struct ThreadInfo {
  bool myPowerFlag;
  uint64_t localSeqNumber;
  ...
}
```

Figure 10: Implementation details for power mode support

```
T read_barrier (void *addr) {
  ThreadInfo *tx = getThreadInfo();
  if (!tx->myPowerFlag) {
    uint64_t seqNumber = tx->localSeqNumber;
    if (*ADDR_TO_WRITE_LOCK(addr) >= seqNumber)
      htm_abort();
  } else {
    if (*ADDR_TO_READ_LOCK(addr) < g_State.seqNumber) {
      *ADDR_TO_READ_LOCK(addr) = g_State.seqNumber;
       membarstoreload();
       if (!g_State.isLocked &&
           ++g_State.uniqRCacheLines > READ_CAPACITY) {
           // switch to the lock-based execution;
           // this aborts all regular transactions,
           // and forces them to wait for the lock
           // to become available again
           g_State.isLocked = true;
           membarstoreload();
           // calculate how much time we wasted
           // so far on this power mode transaction
           uint64_t timer = read_hw_clock();
           uint64_t delta = timer -
               g_State.lastPowerModeStartTime;
           // charge this amount of time for the
           // re-execution under lock
           while (read_hw_clock() - timer < delta);
         }
     }
  }
  return *addr;
}

void write_barrier (void *addr, T val) {
  ThreadInfo *tx = getThreadInfo();
  if (!tx->myPowerFlag)) {
    uint64_t seqNumber = tx->localSeqNumber;
    if (*ADDR_TO_READ_LOCK(addr) >= seqNumber)
      htm_abort();
    if (*ADDR_TO_WRITE_LOCK(addr) >= seqNumber)
      htm_abort();
  } else {
    int seqNumber = *ADDR_TO_WRITE_LOCK(addr);
    if (seqNumber < g_State.seqNumber)) {
      *ADDR_TO_WRITE_LOCK(addr) = g_State.seqNumber;
       if (!g_State.isLocked &&
           ++g_State.uniqWCacheLines > WRITE_CAPACITY) {
            /* same as Lines 53-66 */
       }
     }
  }
  *addr = val;
}
```

Figure 11: Instrumentation barriers used to implement the libitm interface of GCC

and for write, respectively. As described above, we use these numbers to emulate capacity aborts by power mode transactions and re-execution under lock. Based on data in [26], 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 mode transaction, and spin for that amount of time, charging the lock-based execution for running the prefix of the (effectively, aborted) power mode transaction (Lines 61–66). Note that once the power mode 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-mode transaction, as it would access the same memory locations and read same values. As a result, the emulation of time cost required to abort a power mode transaction and re-execute it under lock is realistic. Note that once the execution of the power mode transaction continues under lock, it goes through same barriers, to keep the cost of memory access comparable across all execution modes.

The implementation of procedures for starting and ending a transaction is very similar to the one shown in Figure 3, with the calls to begin and commit power mode transactions (Line 34 in Lock procedure and Line **??** in Unlock procedure, respectively) being omitted. Other differences in the code are related to how regular transactions handle a lock-based execution, which is identical to standard TLE. Specifically, when a regular transaction begins (following the call to `begin_htm()` in Line 6 in Figure 3), it checks whether the isLocked field in g_State is set and aborts if so. This check also effectively subscribes the transaction to this flag, so that any subsequent change to the flag (i.e., when a power mode transaction switches to the lock) will abort the transaction. Also, to mitigate the lemming effect, the regular transaction waits for the flag to be unset before making another trial (i.e., between Line 29 and Line 32 in Figure 3).