TMbarrier: Speculative Barriers Using Hardware Transactional Memory

Manuel Pedrero
Dept. Computer Architecture
University of Malaga, Spain
mpedrero@uma.es

Eladio Gutierrez
Dept. Computer Architecture
University of Malaga, Spain
eladio@uma.es

Oscar Plata
Dept. Computer Architecture
University of Malaga, Spain
oplata@uma.es

***Abstract*— Barrier is a very common synchronization method used in parallel programming. Barriers are used typically to enforce a partial thread execution order, since there may be dependences between code sections before and after the barrier. This work proposes TMbarrier, a new design of a barrier intended to be used in transactional applications. TMbarrier allows threads to continue executing speculatively after the barrier assuming that there are not dependences with safe threads that have not yet reached the barrier. Our design leverages transactional memory (TM) (specifically, the implementation offered by the IBM POWER8 processor) to hold the speculative updates and to detect possible conflicts between speculative and safe threads. Despite the limitations of the best-effort hardware TM implementation present in current processors, experiments show a reduction in wasted time due to synchronization compared to standard barriers.**

***Keywords* - Hardware Transactional Memory; Speculation; IBM POWER8; Parallel Computing**

I. Introduction

Modern multicore processors include several cores sharing the physical memory. Exploiting these resources may not be a trivial task, since the programmer have to deal with challenges associated with parallel programming. These difficulties usually derive from data dependencies arising when several threads need to access to the shared memory. The coordination between cooperating threads is typically carried out with synchronization primitives, such as mutexes or barriers, that prevent threads from reaching sections of the code until certain conditions are met. It is well known that these approaches are pessimistic as they may block the execution of one or more threads until they can continue its execution safely.

Every time a thread blocks due to synchronization it stops making useful work for the application. A goal of programmers must be maximizing the exploitation of the available hardware by keeping a high degree of concurrency while preserving correctness.

As multicore processors have become mainstream, efforts have been made to make parallel programming easier to use. Transactional Memory (TM) [1], [2] is one of the proposals oriented to simplify multithreaded programming. It was proposed as an alternative to lock-based mechanisms to coordinate concurrent threads. TM introduces the concept of transaction, a section of code that runs assuring atomicity and isolation with respect to other transactions that may be executing concurrently. TM has been an active research topic for the last two decades. Recently major processor manufacturers have included hardware best-effort solutions in their architectures [3], [4]. Apart from hardware designs (HTM), many software approaches (STM) have been also proposed [2].

TM can be leveraged to support thread-level speculation (TLS), which is especially interesting in the parallelization of legacy code [5], [6], [7], [8]. Speculation allows to optimistically execute sections of code in parallel by deferring memory updates until all the threads have executed the section without data conflicts. When a conflict is detected, the conflict manager must act accordingly to preserve correctness (for instance, discarding all the speculative work and re-executing the section sequentially). The TM conflict detection and version management methods can be used to support speculation. Usually, this involves enforcing a total order among transactions to preserve correctness in case of a conflict.

The TM paradigm does not necessarily assume any specific transaction execution order as it has been devised as a substitute for locks. Nevertheless several TM proposals introduce a total ordering [9], [10], [11]. In this case, each transaction is launched with an unique, increasing order identifier. When a transaction finishes its execution, it has to wait until every previous transaction has committed before it can release its changes to shared memory. This enforces a serialization of the transactional commit phase, that may hinder performance.

Many applications need only to enforce an ordering between certain phases of an algorithm. In these scenarios, barriers are often used to synchronize these different phases [12]. As this is a common abstraction, environments for parallel programming usually include support for barriers. Since TM is intended to be used in conjunction with parallel libraries, it does not include specifically a definition for barriers.

In this work, TMbarrier is proposed, a transactional barrier intended to be used in transactional codes. The idea behind TMbarrier is to allow a thread to continue speculatively its execution after a barrier without waiting for the rest of the threads. This proposal is aimed at substituting standard barriers, in order to reduce the time wasted waiting in barriers.

The design of the transactional barrier relies in HTM features present in processors like IBM POWER8. We have made an implementation of TMbarrier using the HTM primitives of such processor and conducted an experimental evaluation. Our results show that TMbarrier can reduce significantly the time wasted in barriers, improving the performance in the tested applications despite the current limitations of the HTM. We also suggest that our approach could profit some extra HTM features not present in the processor.

In summary, this paper makes the following contributions:

- A new transactional barrier, TMbarrier, that leverages TM to speculate after a barrier.
- A new primitive, `tmbarrier`, to be introduced to the TM API.
- An implementation of TMbarrier using the best-effort HTM offered by the IBM POWER8 processor.

II. Limitations of Commercial HTM

In the past years, HTM has reached commercial processors. Intel introduced transactional extensions (TSX) in the Haswell architecture and IBM added transactional facilities to the POWER8 processor. Both of them feature a basic support for TM, including hardware support for conflict detection, store buffering and an execution rollback mechanism in the case of a conflict. These designs are categorized as *best-effort*, because the system does not guarantee the eventual commit of a hardware transaction.

Limitations of these HTM systems basically derive from their implementation based on the cache memory system that behaves as the speculative storage for transactional data sets. On the one hand, the cache limited capacity can give rise to transaction aborts when their data sets are larger than the available storage (4MB for reads and 22KB for writes in Intel TSX [6], and 8KB for each set in IBM POWER8). On the other hand, as conflicts are detected at cache-block level, false conflicts can arise if different data used within transactions are mapped to the same cache block (false sharing). As the size of the cache block is large compared to typical variable sizes (64 bytes in Intel Haswell and 128 bytes in IBM POWER8), the impact of these conflicts is often not negligible [8].

Additionally, these HTM implementations are implicit, that is, when the processor enters in transactional mode, all memory accesses within the transaction are stored marked as transactional, taking part of the data set and consequently susceptible to cause aborts. Also, pure nested transactions are not supported. Nesting are usually implemented by flattening the nested transactions into a large outer one. Lastly, the conflict manager, that determines which transaction wins in case of conflict [13], is not configurable.

A remarkable feature of the IBM HTM is the ability to disable the transactional execution within the transaction. This is called the *suspended mode*. All memory accesses in suspended mode are registered as non-transactional, but can still induce aborts with any transaction, including the one in suspended mode. Although this feature is mainly intended for debugging, a careful use of it can provide a communication mechanism between transactions without the risk of aborts. As this feature will be used as a basis for TMbarrier design, our focus is set hereinafter on the IBM POWER8.

Algorithm 1 Transaction start in best-effort HTM

```
 1 function TM_BEGIN( )
 2     while True do
 3         while IsLocked(fallbackLock) do                ▷ Avoid lemming effect
 4         end while
 5         $tx.status \leftarrow$ HtmStart                       ▷ HTM xact. call
 6         if $tx.status ==$ Success then                      ▷ Hardware path
 7             $tx.mode \leftarrow$ HW
 8             if IsLocked(fallbackLock) then ▷ Add fallbackLock to rset
 9                 HtmAbort                                 ▷ HTM abort call
10             end if
11             Break
12         else                                             ▷ Abort treatment
13             $tx.retries = tx.retries - 1$
14             if $tx.retries == 0$ then                      ▷ Software path
15                 Lock(fallbackLock)
16                 $tx.mode \leftarrow$ SW
17                 Break
18             end if
19         end if
20     end while
21 end function
```

A. Best-effort HTM

Best-effort HTM transactions are not guaranteed to commit [3]. There are several events that may trigger the abort of a transaction, and some of them always will preclude a hardware transaction to finalize. For this reason, best-effort HTM requires a software fallback path provided by the programmer.

Typically, this is solved by using a global lock, as shown in algorithms 1 and 2. To start a transaction, the HTM start procedure (`HtmStart`) is called, which returns whether the transaction was initiated or not. If successful, the transaction needs to be subscribed to the global lock `fallbackLock` by reading it, thus adding it to its read set. If the transaction cannot start (or it was aborted), it continues through the *else* path in line 12, where the counter `tx.retries` is decremented. If this counter reaches zero, the system assumes that the transaction cannot continue in hardware mode, so it tries to lock `fallbackLock` and switches to an irrevocable software path. This action causes all the active hardware transactions to abort, because the lock was previously read in line 8.

When the transaction ends (algorithm 2), it checks whether it is in either the hardware or software path. In the case of being in the hardware path, the HTM ending procedure (`HtmEnd`) is invoked, that tries to commit changes to memory. If the transaction is in the software path instead, all the updates to memory have been done in non-transactional mode, so it is only needed to release the global lock. In any case, the `tx.retries` counter is reset for the transaction.

Line 3 in algorithm 1 is intended to avoid a harmful scenario, *lemming effect* [14], [15], that happens when the software path causes all the other transactions to abort and exhaust their retries, resulting in a cascade of switches to the software path.

III. Transactional Barriers

Consider the general linear recurrence equation extracted from the Livermore Loops [16] in algorithm 3. The inner loop can be parallelized as shown in algorithm 4 [17]. In

Algorithm 2 Transaction end in best-effort HTM

```
function TM_END( )
    if $tx.mode$ == HW then
        HTMEND                                  ▷ HTM commit call
    else
        UNLOCK(fallbackLock)
    end if
    $tx.retries \leftarrow$ MAXRETRIES              ▷ Restore retries
end function
```

Algorithm 3 Livermore Loop 6: General linear recurrence

```
for i = 1 to N do
    for k = 0 to i do
        $w[i] = w[i] + b[k][i] * w[(i-k)-1]$
    end for
end for
```

Algorithm 4 Livermore Loop 6: Parallel version

```
for t = 0 to N - 2 do
    for k = tid*chunk to (tid+1)*chunk do
        if $k < (N-t)$ then
            $w[t+k+1] = w[t+k+1] + b[k][t+k+1] * w[t]$
        end if
    end for
    BARRIER
end for
```

this parallel version, the inner loop distributes the workload by blocks, and the exploitable parallelism decreases in each iteration of the outer loop. A barrier is mandatory before each iteration of the outer loop, as the threads have to access to updated values in w. The barrier synchronizations can have a considerable impact in the performance due to time spent in them and the conservative assumptions made by the compiler [18].

TM can be leveraged to avoid such barriers by enclosing iterations in transactions and by establishing a total order of precedence among them based on the original sequential code. This approach have been used to enable TLS with TM [6]. The rationale behind this is that threads could execute the majority of the work in parallel and only transaction commits are serialized. However, in practice this serialization may result in a high overhead.

With the aim of mitigating such an overhead, the total order could be relaxed to a partial order in those parallel codes where dependences are solved with a barrier synchronization. Taking again the example of algorithm 4, if the inner loop body is enclosed in a transaction, transactions executed between the same barrier calls can be committed in any order. In this way, we can define a *partial ordering* that enforces that all transactions *before* the barrier call are required to commit before any other transaction *after* the barrier can try to confirm its changes to global memory. This partial order keeps the sequential equivalence of the code, while enabling more opportunities for parallel exploitation, since all the transactions before the barrier could commit without being serialized.

Our proposal is to introduce a new primitive, hereinafter `tmbarrier`, that provides some information to a given TM system in order to enable a dynamic partial ordering at runtime. When a thread reaches `tmbarrier`, the thread switches to a *speculative* mode by opening a new transaction upon the `tmbarrier` call. Since this moment, all updates carried out by the thread are buffered by the TM system until either a conflict is detected (a) or all the remaining threads have reached the `tmbarrier` (b). In case (a), the transaction rolls back to the barrier point (i.e., it is aborted), discarding any speculative work. In case (b), the transaction is allowed to commit, and the thread will switch again to a non-speculative mode, thus updating the speculative data in memory. In our proposal, the speculation window only can be extended between two `tmbarrier` calls (i.e., if a speculative thread enters `tmbarrier`, it will be blocked until becomes non-speculative or aborts).

A. Ordering

To enable the partial ordering in TMbarrier, each thread holds an epoch number in its metadata (`tx.order`). A global variable (`glOrder`) synchronizes the real epoch. Epochs are initialized to 1 at the beginning of the application, and they are incremented as follows: `tx.order` is updated when the corresponding non-speculative thread reaches a `tmbarrier`; `glOrder` is updated when the last non-speculative thread exits a `tmbarrier`. A transaction can only commit when its `tx.order` matches `glOrder`.

These epochs allow the system to identify transactions executing before and after a barrier in each thread. Hereinafter, we refer to speculative transactions as those started after the barrier. Since local epoch in speculative transactions do not match `glOrder`, they are not allowed to commit until all the threads before the barrier have reached it, so that `glOrder` has been updated.

B. Nested transactions

The scheme outlined so far considers only a single speculative transaction per thread, because no transaction is allowed to commit until `glOrder` is updated to match speculative epochs. To improve the speculative execution efficiency, a maximum of `specMax` transactions per thread are allowed to execute after a `tmbarrier` by leveraging nested transactions [19]. The TM system will open an *outer* transaction after the `tmbarrier`, so subsequent transactions are started as *inner* nested ones up to a maximum of `specMax`. Note that this outer transaction is instrumental while nested ones are those already present in the code. By doing so, the condition of a single speculative transaction after the `tmbarrier` is kept, but this *outer* transaction can enclose several *inner* ones, allowing to increase the degree of speculation.

C. Using TMbarrier

Figure 1 illustrates how TMbarrier work. Consider three concurrent threads executing a code with `glOrder` n. Thread 1 executes a transaction with `tx.order` n and reaches a `tmbarrier` call. Instead of blocking, it increments its local epoch and opens a outer transaction (depicted in red) after exiting the barrier. It continues the execution and starts new (inner) speculative transactions with `tx.order` $n+1$. The

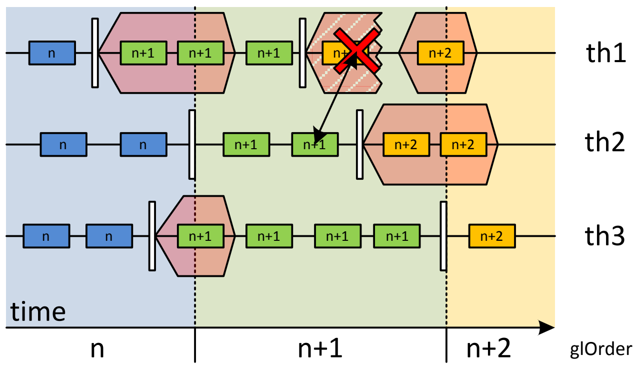

Fig. 1. Using TMbarrier. Colored rectangles represent transactions, blank bars represent barriers. Red hexagons are outer transactions enclosing several inner transactions. Each n inside a trasaction is its local epoch (`tx.order`), while bottom n represents the global epoch (`glOrder`).

TABLE I
THREAD STATES FOR TMBARRIER

	non-xact	hw-xact	sw-xact	spec	spec-xact
non-xact	-	start	max. retries	tmbarrier	-
hw-xact	cmt, abort	-	-	-	-
sw-xact	commit	-	-	-	-
spec	spec. commit	-	-	-	start
spec-xact	-	-	-	commit	-

same occurs with thread 3. Thread 2 is the last one in reaching the `tmbarrier`, so it updates `glOrder` to $n+1$ and it does not switch to speculative mode, since its local epoch matches `glOrder`. After that, when thread 1 finishes its second transaction after the barrier, it checks that its local epoch also matches `glOrder` and commits the outer transaction, thus releasing its speculative data to the global memory.

The same is done by thread 3. Thread 1 encounters a `tmbarrier` again, but this time its speculative transaction detects a conflict with a non-speculative transaction in thread 2, so it aborts both the inner and outer transaction, discarding all speculative work and returning its execution just after exiting the second `tmbarrier`. As thread 3 has not reached `tmbarrier` yet, thread 1 reopens the outer transaction and re-executes speculatively. This time, after committing an inner transaction, thread 1 can switch again to non-speculative mode, since all threads have reached the second `tmbarrier`. The same occurs to thread 2, that speculates two inner transactions before switching again to non-speculative mode.

IV. TMBARRIER DESIGN

When designing TMbarrier, one of the goals was to test the possibilities provided by state-of-the-art commercial HTM implementations. The algorithms 1 and 2 has been used as a basis to implement the `tmbarrier` primitive.

A. Thread states

In a best-effort HTM, three exclusive states are considered for a given thread: (1) *non-xact*: a thread that is not executing a transaction, (2) *hw-xact*: a thread executing a hardware transaction, and (3) *sw-xact*: a thread executing a software fallback path. In our proposal of TMbarrier two more states are introduced, namely: (4) *spec*: a thread that is executing speculative code after a `tmbarrier` call, but this code is not originally within a transaction, and (5) *spec-xact*: a thread that is executing a hardware speculative transaction after a `tmbarrier` call. Table I shows those states and the events that trigger transitions among them.

A thread always starts execution in *non-xact* state. When a transaction starts, the thread changes to *hw-xact*, and continues in this state until it aborts or commits, returning then to *non-xact*. If a transaction aborts too many times, it transitions from *non-xact* state to *sw-xact* by acquiring the fallback lock (line 15, algorithm 1). A transaction in this state is irrevocable, consequently it must end by committing its changes and returning to *non-xact* state.

When a thread in *non-xact* state encounters a `tmbarrier` call, speculation begins by opening an outer transaction and switching to *spec* state. In this state, the beginning of any (inner) transaction will change the state to *spec-xact*. A speculative transaction is always hardware, that is, it cannot take the fallback path. Whenever a speculative transaction commits or aborts, the thread and global epochs are checked in order to end the speculation as soon as possible. If epochs match the thread switches from *spec* to *non-xact* state by committing the outer transaction (spec. commit transition).

B. `tmbarrier` *primitive*

The `tmbarrier` primitive uses an internal counter to keep track of the number of threads that have executed the primitive. Its initial value is the number of threads involved and it is decremented atomically each time a thread enters `tmbarrier`. When the last thread enters `tmbarrier`, the global epoch `glOrder` is atomically incremented, allowing speculative threads to commit its changes. Otherwise the thread switches its mode to *spec* according to table I and opens a outer transaction. Finally, if a thread enters `tmbarrier` in *spec* state, it is halted in TM suspended mode until it eventually commits (when its epoch matches `glOrder`) or aborts due to a conflict.

C. Starting transactions

Transaction start is similar to algorithm 1. However, in the case of an abort (lines 12-19), if the thread is speculative it is prevented to switch to *sw-xact* state. Instead, it decrements `specMax` and restarts the `tx.retries` counter. This preserves the correctness of the code, as no transaction in *spec* state commits until all threads executing before the barrier have ended. The reason for decrementing `specMax` is to lessen the risk of abort, as the number of inner transactions after `tmbarrier` is reduced. After updating both counters, the thread determines if it can switch to the *non-spec* mode by checking if `glOrder` matches `tx.order` (see table I).

D. Committing transactions

Transaction commit is based on algorithm 2, but now it is checked if the transaction is in *spec* state when it reaches the commit phase. If this is the case, transaction enters in the TM suspended mode to check if glOrder matches tx.order. All transactional accesses to glOrder must be done in the TM suspended mode to prevent that all speculative threads abort when glOrder is updated.

If epochs match, transaction performs a double commit, one to exit the inner transaction, and another one to exit the outer transaction and to switch to *non-spec* state. If epochs do not match, the thread performs a single commit to exit the inner transaction, and a thread-local counter is incremented until reaching specMax. If speculation can continue, no further action is needed: the next call to transaction start will open another inner transaction. If specMax is reached, the thread will be halted in the TM suspended mode until its epoch matches glOrder and can commit its changes (or abort due to a conflict).

E. Dealing with POWER8 HTM limitations

Some specific features of the best-effort HTM implementation in POWER8 should be taken into account so as to implement the TMbarrier efficiently.

First, the HTM is implicit and it detects conflicts with cache block granularity, so it is necessary to map glOrder and the fallback lock into two separate cache blocks. The fallback lock variable is always included in the read set of every transaction. Nevertheless glOrder must not be registered in any transactional set, as the increment of glOrder would make any active transactions abort. While transactions in *hw-xact* state could eventually commit, all speculative work after tmbarrier would be lost as transactions in *spec-xact* state cannot commit until glOrder is updated.

Also, all thread-local metadata have to be aligned to a separate cache block. Additionally they should be packed in an single cache block to avoid false sharing. Altogether, our implementation uses a single cache block (128 Bytes) per thread to store transaction metadata, and two additional blocks to store global metadata.

The TM suspended mode have to be handled with special care, because it may lead to non-intuitive effects [13]. For example, incrementing a speculative transaction counter within a hardware transaction in suspended mode can cause an abort if another metadata located in the same cache block has been read, because there is a non-transactional access to a block marked transactional in the thread. A way to tackle this problem is by forbidding any writes in suspended mode. Another less restrictive way is by isolating data that is accessed transactionally from data that is updated in suspended mode in separate cache blocks. In our proposal, the TM suspended mode is only used when checking if thread epoch matches glOrder, so no writes in this mode are made. We also collect some statistics, but only outside a transaction (e.g., causes of aborts and commit paths after a successful commit).

Nested transactions in POWER8 HTM are handled by flattening inner transactions into the corresponding outer one. This feature has two disadvantages compared to other more advanced techniques [19]. First, a conflict in an inner transaction triggers the abort of all the transactions enclosed by the same outer transaction. Second, memory accesses performed by any inner transaction are accumulated in the same data set, which is shared by all the inner transactions, increasing the abort probability. To address this limitation the maximum number of speculative transactions inside an outer transaction is tuned dynamically. Besides, if a speculative transaction aborts due to exceeding the maximum data set supported by the hardware, specMax is immediately decremented to reduce the footprint of the outer transaction.

TMbarrier design should prevent a conflict involving a speculative transaction from causing a loss of non-speculative work. This requires a custom TM conflict manager with an *abort-speculative* policy ensuring that any conflict between speculative and non-speculative transactions is solved by aborting the speculative transaction. Unfortunately, as POWER8 HTM conflict manager is hardcoded, a transaction before tmbarrier can abort because of a conflict with a speculative transaction.

V. Experimental Evaluation

The performance of the proposed TMbarrier method has been evaluated using two representative benchmarks. The first one is a pure barrier microbenchmark from [20] to test the transactional barriers in the absence of conflicts. This benchmark consists of a sequence of steps synchronized by a barrier. In each step only half of the threads do actual work resulting in load imbalance. The second benchmark is a general linear recurrence from the Livermore Loops. It exhibits a high time spent in barriers and a relatively low amount of enforced dependences [18].

Experiments have been conducted in the system described in table II. Reported results for each profile correspond to the fastest of 30 executions in order to rule out system effects such as thread scheduling. Each thread was mapped onto a physical core by means of taskset to prevent that several threads share HTM physical resources which would increase the abort rate.

Several profiles have been considered in the evaluation:

- *P8*: it corresponds to POWER8 HTM with a basic scheme as shown in algorithms 1 and 2. In this profile, the barrier synchronization has been implemented with standard OpenMP barriers.
- *P8-NoBar*: it is a similar scheme to *P8* but all barriers have been removed. Although this profile yields incorrect results, it can be used as an upper reference for the achievable performance.
- *P8-Ordered*: it stands for an ordered HTM version of the *P8* profile where each transaction commit is blocked until all the previous transactions have committed, as described in section I. Barriers have been removed because synchronization is guaranteed by the total order.
- *P8-OrdWTB*: it refers to the TMbarrier proposal implemented using the POWER8 HTM. In this profile barriers have been implemented with a tmbarrier call as described in section III. All threads in this profile can speculate up to 8 transactions per thread after a barrier.

A. Barrier microbenchmark

This barrier microbenchmark [20] is intended to illustrate a best-case scenario for TMbarrier. In the benchmark, computations are organized in steps. In each step several threads make some work and synchronize themselves with a barrier at the end of the step. The workload is not balanced among the threads. Instead, only half of the spawned threads works in each step. Computations do not involve data dependences. As the workload scales linearly with the number of threads, efficiency has been calculated [1], which informs about the overhead introduced by the synchronization method.

Figure 2 (left) shows the efficiency obtained with default configuration from [20], as well as in other scenarios with less work per step. The upper bound set by the idealized profile *P8-NoBar* is in the range 90%-100% depending on the workload. Results show that our proposal, *P8-OrdWTB*, is able to keep a good efficiency in all these scenarios, whereas the other two alternatives are significantly penalized by the synchronization overhead. Notice that in this benchmark, idle threads can benefit from TMbarrier as they can continue executing the next step speculatively, thus exploiting parallelism in presence of imbalanced loads.

To analyze how well TMbarrier leverages speculation, figure 2 (right) breaks down the different commit paths followed by transactions in *P8-OrdWTB*. As relative breakdowns were not sensitive to workload only the default configuration is shown. Four different paths are distinguished: (1) *Non Spec.* refers to transactions finished before reaching a `tmbarrier` call (*hw-xact* state); (2) *Spec. Normal* refers to speculative transactions (after the barrier) that have executed at least one inner transaction and have performed a double commit (inner plus outer) to exit speculation; (3) *Spec. Barrier* refers to transactions that have reached the barrier in *spec* state, and have waited for the rest of transactions running before the barrier; and (4) *Spec. Max.* refers to speculative transactions that have reached the limit of speculation.

Path (1) is determined by the transactions committed by the last thread arriving to each barrier, which has no opportunity to speculate. This is a lower limit that basically derives from the algorithm itself and take part of its critical execution path. In this case the amount remains almost constant with respect to 1-thread scenario because speculative transactions do not abort as the benchmark has no data dependences.

Also, there is a large percentage of the transactions that need to wait in the next barrier (3). This is explained because each thread only executes a single transaction in each step, and synchronizes with a barrier immediately afterwards. In our implementation, threads in *spec* state block in `tmbarrier`. This is also the reason for the absence of commits that reaches the speculation limit (4). Finally, *Spec. Normal* (2) is almost absent with two threads, but grows linearly with the number of threads, because there are more in-flight speculative transactions that will fulfill the commit condition when they have finished.

[1]efficiency $= \frac{\text{throughput(N threads)}}{N \times \text{throughput(sequential)}}$, for N threads with throughput $= \frac{\text{\#steps}}{\text{exec. time}}$

TABLE II
EVALUATION PLATFORM

Parameter	Description
Processor	IBM POWER8 3.5GHz
Cores	10 physical with SMT 8
Memory	512GB
OS	Ubuntu Server 16.04.1 LTS, Kernel 4.4.0-47 ppc64le
Compiler	GNU gcc v5.4

All profiles exhibit a very low abort rate regardless of the number of threads and the workload. This is expected due to optimistic nature of the microbenchmark, but confirms that *P8-OrdWTB* does not introduce extra conflicts.

B. Recurrence

Results for this kernel with different chunk sizes (number of iterations enclosed by a single transaction) are shown in figure 3. Chunk sizes have been selected in the range of interest where the number of transactions are minimized while maintaining a low abort ratio in order to maximize the speedup for the P8 profile. The kernel presents a limited amount of parallelism even when eliminating barrier synchronization. These results agree with [17], that points out that the maximum achievable performance with the best software barriers is around 2x in a 16-core processor. We have measured an upper limit of about 3x by eliminating all barriers (*P8-NoBar*). In profile *P8-Ord* the penalty introduced by total order limits the speedup to 1.4x in the best scenario. Also, the overhead in this implementation grows rapidly with the number of threads. Profile *P8* that uses standard barriers reaches about 1.5x speedup with 8 threads and the best chunk configuration. Results are not much better than *P8-Ord* due to the synchronization overhead and the performance is very dependent on the chunk size.

P8-OrdWTB increases this speedup up to 2x but most importantly, it consistently outperforms P8 with standard barriers in non-optimal chunk configurations. This is explained because the adaptative speculation of TMbarrier permits to execute more transactions with smaller chunks, and also to reduce the time in the barriers with larger chunks. The upper limit given by *P8-NoBar* is about 3x with 10 threads, which shows the limited amount of exploitable parallelism found in this code. Reasons for this behavior is the memory-bound nature of this benchmark, and the reduction of the amount of operations in the inner loop with successive iterations of the outer loop.

Regarding aborts, the Transaction Commit Rate (TCR) [21] is shown in figure 3 (down). This metric is the ratio between the number of committed transactions and the total number of started transactions. A maximum value of 1 indicates that all transactions committed without aborting.

Profile *P8* keeps aborts near zero because transactions in the inner loop do not have true data dependences (conflicts may be caused only by memory location aliases). Barriers prevent conflicts due to true dependences. *P8-NoBar* also has a very low number of aborts because none of the transactions have to wait due to synchronization, and the resulting data reference

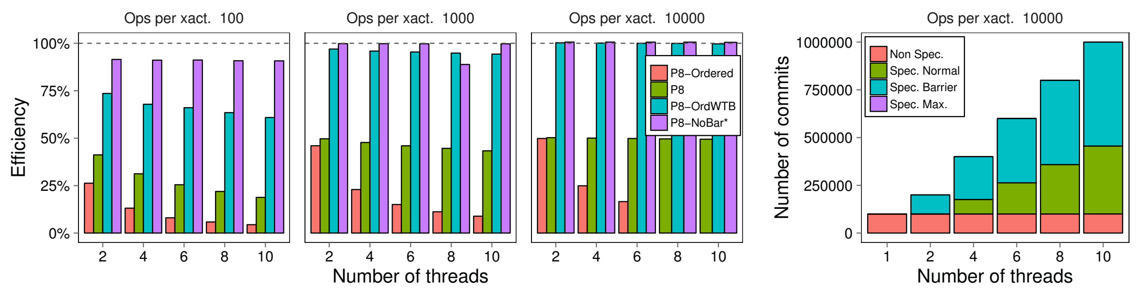

Fig. 2. Microbenchark: efficiency over sequential execution (left) and commit paths (right)

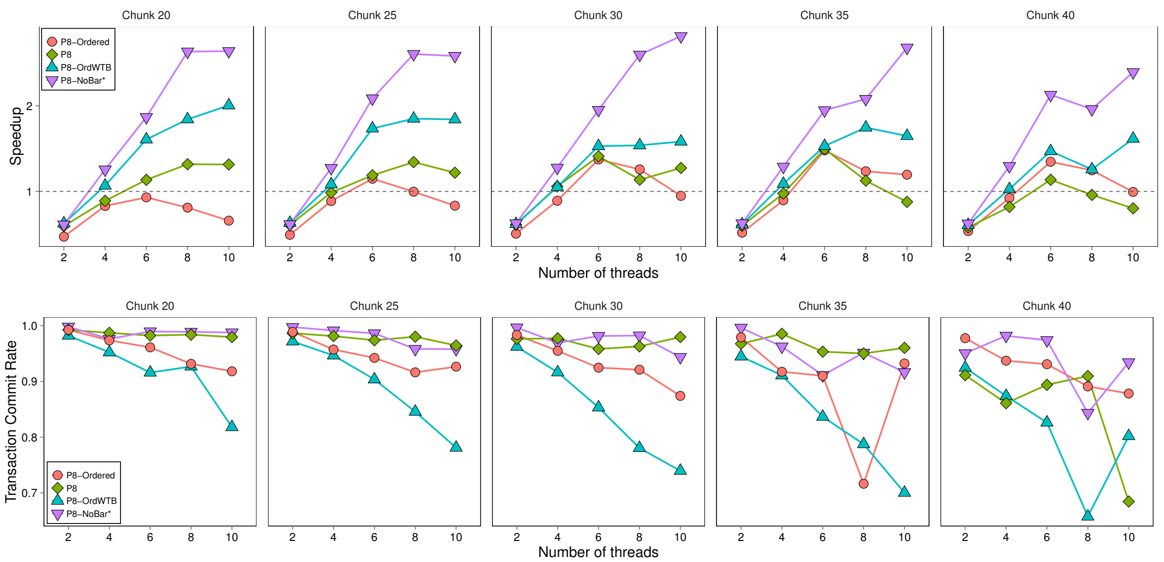

Fig. 3. Recurrence: speedup over sequential execution (top) and TCR (bottom)

pattern exhibits a very low conflict probability even in the absence of barriers.

The profile *P8-OrdWTB* has a lower TCR, that decreases when using more threads and with larger chunk sizes. Those aborts are mainly due to conflicts caused by those enforced dependences between transactions before and after the barrier [17]. In fact, allowing speculative execution beyond a barrier leads to new memory access patterns, that may produce new conflicts. This enforces us to adjust the degree of speculation when the number of aborts is too high.

In cases with lower TCR, a noticeable percentage of the aborts are caused by conflicts with non-transactional code. This is due to the fallback lock acquisition when a transaction running before the barrier has aborted too many times. When a transaction acquires this lock, all the active transactions abort.

For larger chunks of iterations (enclosed by a single transaction), aborts due to cache capacity may also occur, as nested transactions are supported via flattening, increasing the probability of exceeding the maximum transactional set size.

In this benchmark, the ratio of successfully committed speculative transactions with respect to the total number of transactions is about 2%, with variations due to the number of threads, size of the transaction and frequency of `tmbarrier` calls. Note that although this is not a high rate, a single speculative commit is enclosing up to eight inner transactions.

VI. Related work

Leveraging TM to exploit optimistic parallelism using Thread-Level Speculation (TLS) or Speculative Multithreading (SpMT) is not a new topic [22], [6], [23], as TM support for conflict detection and update buffering is very convenient to TLS designs. An analysis of the usage of HTM for TLS, using Intel TSX, is presented in [6]. Based on results over some programs of SPEC CPU2006, the authors conclude that the main cause of performance loss are transaction aborts due to memory conflicts, and suggest future improvements in HTM implementations, such as data-forwarding, multi-version cache and word-level conflict detection. This agrees with [24], which studies the exploitable speculative parallelism in SPEC2006 finding interesting parallelization opportunities using TLS.

Applying TLS strategies to already parallelized codes can involve to speculate after synchronization directives such as barriers or locks. In [25] authors propose a hardware design that enables this kind of speculation using a safe thread to guarantee forward progress and monitor for conflicting accesses to preserve correctness.

OpenTM [26], which extends OpenMP with a TM API, considers a new set of directives meant to express non-blocking synchronization. OpenTM supports a total order in transactions and in transactional loops, but does not allow OpenMP synchronization clauses within transactions as TM strong isolation could be not guaranteed. The proposal requires *virtualized transactions*, that are not bounded by execution time, memory footprint or other limitations. Nevertheless, this feature is not supported by current best-effort HTM implementations.

Focused on barriers, [18] studies the effect of barrier synchronization and misspeculation past barriers in a set of parallel programs. The goal is to find TLS opportunities in codes which feature both considerable time spent in synchronization and infrequent interprocessor dependences. The authors propose a new solution based on the advanced load address table (ALAT) present in Itanium processors to detect misspeculation and to speculate past barriers.

Another approach to speculate in synchronization directives by combining OpenMP and HTM is proposed in [20]. This approach is meant to be used with non-transactional code, but also uses a transaction after a barrier to buffer speculative updates. Nevertheless it is not intended to leverage escape actions nor already transactional code. For this reason, a single post-barrier synchronization point has to be manually specified, forcing the transaction to abort if it cannot commit when reaching it. Authors reported a best-case success rate of around 40% for speculative transactions in the same barrier microbenchmark analyzed in this paper. The same configuration yields a nearly perfect success rate in our proposal, as shown in Figure 3 right, where successful speculative transactions scale linearly with the number of threads. This work does not consider some limitations of real HTMs, namely read/write set sizes and illegal instructions into the speculative transaction. Also, this solution does not exploit nested transactions nor several speculation levels, as it cannot support multiple synchronization points.

VII. Conclusion

This work proposes TMbarrier, a transactional barrier designed to reduce the wasted time due to barriers by using speculation. TMbarrier is designed using existing best-effort HTM implementations present in commodity processors. The proposal is focused on already transactional codes that make use of barriers, though it can be extended to non-transactional applications. It is only necessary to replace standard barriers with the proposed transactional barrier. The performance of TMbarrier has been analyzed experimentally using a microbenchmark and a recurrence kernel from the Livermore Loops. Both codes exhibit a high amount of time spent in barrier synchronizations and a low number of enforced dependences between data before and after the barrier. Results show that parallel codes using the proposed transactional barriers outperform the corresponding versions using standard barriers. Moreover, the proposed implementation can benefit from future advances in best-effort HTMs, such as larger transactional read/write sets, word-level conflict detection and a better transaction nesting support.

Acknowledgments

This work has been supported by the Government of Spain under project and TIN2016-80920-R, and the Junta de Andalucía under project P12-TIC-1470.

References

[1] M. Herlihy, J. Eliot, and B. Moss, "Transactional memory: Architectural support for lock-free data structures," in *ISCA*, 1993, pp. 289–300.

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

[3] H. Lee, G. Guthrie, D. Williams, M. Michael, G. Frey, W. Starke, C. May, R. Odaira, and T. Nakaike, *Transactional memory support in the IBM POWER8 processor*. IBM Journal R&D, 2015.

[4] R. M. Yoo, C. J. Hughes, K. Lai, and R. Rajwar, "Performance evaluation of intel transactional synchronization extensions for high-performance computing," in *SC*, 2013.

[5] M. DeVuyst, D. M. Tullsen, and S. W. Kim, "Runtime parallelization of legacy code on a transactional memory system," in *HiPEAC*, 2011.

[6] R. Odaira and T. Nakaike, "Thread-level speculation on off-the-shelf hardware transactional memory," in *IISWC*, 2014, pp. 212–221.

[7] J. Barreto, A. Dragojevic, P. Ferreira, R. Filipe, and R. Guerraoui, "Unifying thread-level speculation and transactional memory," in *Middleware*, 2012, pp. 187–207.

[8] J. Salamanca, J. N. Amaral, and G. Araujo, "Evaluating and improving thread-level speculation in hardware transactional memories," in *IPDPS*, 2016, pp. 586–595.

[9] L. Hammond, V. Wong, M. K. Chen, B. D. Carlstrom *et al.*, "Transactional memory coherence and consistency," in *ISCA*, 2004, pp. 102–114.

[10] P. Felber, C. Fetzer, and T. Riegel, "Dynamic performance tuning of wordbased software transactional memory," in *PPoPP*, 2008, pp. 237–246.

[11] M. Pedrero, E. Gutierrez, S. Romero, and O. Plata, "A comparative analysis of STM approaches to reduction operations in irregular applications," *Journal of Computational Science*, vol. 17, pp. 630–638, 2016.

[12] R. K. Karmani, N. Chen, B.-Y. Su, A. Shali, and R. Johnson, "Barrier synchronization pattern," in *ParaPLOP*, 2009.

[13] IBM, *Power ISA Version 2.07, Book II*. IBM, 2013.

[14] D. Dice, M. Herlihy, D. Lea, Y. Lev, V. Luchangco *et al.*, "Applications of the adaptive transactional memory test platform," in *TRANSACT*, 2008.

[15] R. Quislant, E. Gutierrez, E. L. Zapata, and O. Plata, "Lazy irrevocability for best-effort transactional memory systems," *IEEE Trans. Parallel Distrib. Syst.*, vol. 28, no. 7, pp. 1919–1932, 2017.

[16] J. T. Feo, "An analysis of the computational and parallel complexity of the Livermore Loops," *Parallel Computing*, vol. 7, no. 2, pp. 163–185, 1988.

[17] J. Sampson, R. Gonzalez, J. F. Collard, N. P. Jouppi, M. Schlansker, and B. Calder, "Exploiting fine-grained data parallelism with chip multiprocessors and fast barriers," *IEEE Micro*, pp. 235–246, 2006.

[18] V. Nagarajan and R. Gupta, "Speculative optimizations for parallel programs on multicores," in *LCPC*, 2009, pp. 323–337.

[19] M. J. Moravan, J. Bobba, K. E. Moore, L. Yen, M. D. Hill *et al.*, "Supporting nested transactional memory in logTM," in *ASPLOS*, 2006, pp. 359–370.

[20] L. Bonnichsen and A. Podobas, "Using transactional memory to avoid blocking in OpenMP synchronization directives," in *IWOMP*, 2015, pp. 149–161.

[21] M. Ansari, C. Kotselidis, K. Jarvis, M. Luján, C. C. Kirkham, and I. Watson, "Advanced concurrency control for transactional memory using transaction commit rate." in *Euro-Par*, 2008, pp. 719–728.

[22] L. Porter, B. Choi, and D. M. Tullsen, "Mapping out a path from hardware transactional memory to speculative multithreading," in *PACT*, 2009, pp. 313–324.

[23] C. von Praun, L. Ceze, and C. Cascaval, "Implicit parallelism with ordered transactions," in *PPoPP*, 2007, pp. 79–89.

[24] V. Packirisamy, A. Zhai, and et. al., "Exploring speculative parallelism in SPEC2006," in *ISPASS*, 2009, pp. 77–88.

[25] J. F. Martínez and J. Torrellas, "Speculative synchronization: applying thread-level speculation to explicitly parallel applications." in *ASPLOS*, 2002, pp. 18–29.

[26] W. Baek, C. C. Minh, M. Trautmann, C. Kozyrakis, and K. Olukotun, "The OpenTM transactional application programming interface," in *PACT*, 2007, pp. 376–387.