A case for Exiting a Transaction in the Context of Hardware Transactional Memory

Isuru Herath, Demian Rosas, Daniel Goodman, Mikel Luján, Ian Watson
Advanced Processor Technologies Group
The University of Manchester
United Kingdom
{herathh, rosasd, goodmand, lujanm, watson}@cs.man.ac.uk

ABSTRACT

Despite the rapid growth in the area of Transactional Memory (TM), there is a lack of standardisation of certain features. The behaviour of a transactional abort is one such feature. All hardware TM and most software TM designs treat abort as a way of restarting the current transaction. However an alternative representation for the same functionality has been expressed in some software transactional memories and programming languages proposals. These allow the termination of a transaction without restarting. In this paper we argue that similar functionality is required for hardware TM as well. We call this functionality *Exit_Transaction*, in which a programmer can explicitly ask the underlying TM system to move to the end of the transaction without committing it. We discuss how to extend a hardware TM system to support such a feature and our evaluation with two hardware TM systems shows that by using this functionality a speedup of up to 1.35X can be achieved on the benchmarks tested. This is achieved as a result of lower contention for resources and less false positives.

Categories and Subject Descriptors

C.1.4 [**Processor Architectures**]: Parallel Architectures; D.1.3 [**Programming Techniques**]: Concurrent Programming

Keywords

Transactional Memory

1. Introduction

Transactional Memory was initially proposed [11] as a direct generalisation of the *load-linked-store-conditional* instruction, in order to provide atomicity to more than a single memory location. Since then, several Hardware TM (HTM) systems have been proposed with different approaches for versioning and conflict detection (eg:TCC [7], Log-TM [15]). There have also been several attempts to standardise the syntax and semantics of TM [12, 14, 16, 21]. Recently, several chip manufactures unveiled proposals for hardware assisted transactional memory including Sun's Rock processor [4, 6], Azul [5] and AMD-ASF[1].

Despite all this work, there is a mismatch in the specification of *Abort_Transaction*. All the TM community seem to agree that if used when a conflict occurs an abort discards all the speculative operations and reinstate the processor to the state that it was in at the beginning of the transaction. However, when it becomes possible to directly invoke this function from user code two specifications can be found. One is to discard all the speculative operations and to restore the state as it was at the beginning of the transaction as in Log-TM [15], the majority of the TM proposals follow this approach. The other specification propose to discard all the speculative operations and to transfer the control to the end of the atomic block [16, 21]. All the proposed HTM systems appear to follow the first specification.

In this paper we argue that support for both functions is necessary for a hardware TM. We discuss how to extend a hardware TM system to provide the behaviour expressed by the second specification and evaluate the proposed functionality in two hardware TM systems.

In order to clarify the discussion, we name the functionality expressed by the first specification as *Restart_Transaction* and that of the second as *Exit_Transaction*. These are defined as,

Restart_Transaction: *Discard all the operations performed within the atomic section and restart the transaction.*

Exit_Transaction: *Discard all the operations performed within the atomic section and transfer the control to the end of the atomic region.*

The rest of the paper is organized as follows. Section 2 shows some code examples and discusses the need for the *Exit_Transaction* functionality. Section 3 contains related work and where possible how to use these approaches to achieve the objective of *Exit_Transaction*. The transformation of existing code segments to use the *Exit_Transaction* functionality is shown in section 4. Architectural support for *Exit_Transaction* is discussed in section 5. An evaluation of *Exit_Transaction* is presented in section 6 and the applicability of *Exit_Transaction* on other hardware TMs is discussed in section 7. Finally in section 8 we make our concluding remarks.

2. Motivation

Transactional Memory(TM) [11] provides optimistic concurrency for critical regions marked by programmers. The marking of critical regions is typically made either with the *atomic*{} keyword marking a block of code or with the *Begin_Transaction* and *End_Transaction* instruction pair. Once critical regions are marked in an application, the underlying TM system ensures that the Atomicity, Consistency and Isolation(ACI) properties are maintained for the marked regions. In a lazy-lazy TM system all the operations are performed speculatively within the critical section and atomically committed at the end. In a hardware TM system, this commit phase normally involves writing all the modi-

fied cache entries to the next level memory.

The usefulness of the speculative operations are not considered at the time of committing. This is because at the hardware level this usefulness cannot be determined. The only information available is a set of memory locations and their prospective values. On the other hand, at the programming language level the usefulness of a transaction can easily be extracted. Before continuing the discussion further, we need to establish a definition for the *usefulness* of a transaction. We define the *usefulness* as, *if an application contains a set of tasks and committing a transaction contributes to the reduction of the size of this set, then it is a useful transaction.*

We will now look at three examples, (a) a benchmark, (b) a micro-benchmark and (c) a real-life scenario.

2.1 Lee-TM [23]

Lee is a routing algorithm whose objective is to find a path from a given source point to a given destination point. The TM algorithm proposed by Watson *et al.* [23] is comprised of two phases: *expand* and *backtrack*. The *expand* routine starts from a source point and expands in all directions until it reaches its destination. Once it reaches its destination, it starts traversing backwards until it reaches the source, recording in the process the optimal path. The transaction in this algorithm, encompasses both the *expand* and the *backtrack* methods. If the *expand* method was able to reach its destination, it returns True, else it returns False. The pseudocode of the algorithm is shown in Code 1.

Code 1 Lee-TM pseudocode

```
Begin_Transaction;
bool isFound=expand();
if(isFound){
    backtrack();
}
End_Transaction;
```

If we apply the above definition of *usefulness* to this scenario, the set of tasks we have is to find paths from source points to destination points. If a committing transaction is to be considered useful, it should have found a path from a given source to a given destination, but finding a path requires the execution of both *expand* and *backtrack*, however the latter is executed only if the former returns True. As it is not guaranteed that *expand* always returns True a transaction is not guaranteed to perform useful work. In such situations, even if a transaction does not do any useful work, the commit operation still takes place. This commit involves writing back the modifications made to the local variables and to the local grid in the *expand* phase. While these values do not have any use in the application the interconnect is used to communicate the speculatively modified cached entries to the next level memory. We are aware that optimisations to the algorithm can address some of these issues in the algorithm presented, but the effect of these optimisations can be negated by actions beyond the programmers control such as nesting Lee within another transaction.

2.2 Red-Black Tree

A Red-Black tree is a data structure used in computer science. The major operations associated with it are search, insert and delete. Though we used this data structure for the discussion, the situation can be applied to any application that interacts with data stores. The insert and the delete operations incorporate some form of search facility because, before inserting an item, we need to find whether another item with the same key already exists in the tree. Similarly, to perform a delete operation we need to find the item with the corresponding key. We could define the search operation to return True if it finds an item with the given key, then the insert and the delete operations can be performed accordingly. In a TM version of the Red-Black tree, a transaction is comprised of either, the search and the insert or the search and the delete, and we cannot break the atomicity between the two methods. The pseudocode of the algorithm is shown in Code 2.

Code 2 Red-Black Tree TM pseudocode

```
insert_item(key){
    Begin_Transaction;
    bool isFound=search(key);
    if(!isFound){
        insert(key);
    }
    End_Transaction;
}
delete_item(key){
    Begin_Transaction;
    bool isFound=search(key);
    if(isFound){
        delete(key);
    }
    End_Transaction;
}
```

An insert operation can be considered as useful only if it tries to insert an item that does not exist in the tree. In case of the delete, it becomes useful only if the item exists in the tree. This issue applies to any application that uses a data stores. For example, a customer trying to book a room in a hotel will first search the room prices and availability, if a suitable one is found they will reserve it. In a TM version, both *search* and *book* have to be within a single transaction and it becomes useful only if the customer completes the booking stage. However, with the current approaches, regardless of the usefulness of a transaction, a commit operation takes place.

2.3 Exceptions

If we consider a Java program that has a critical section that can produce exceptions, we can have the *Begin_Transaction* instruction after the *try* keyword. In a hardware TM, any operation performed after this instruction will be performed speculatively until the *End_Transaction* is executed. Therefore if we place the *End_Transaction* within the *try-catch* block, it will not get executed if an exception is thrown before it. Since we need to execute *End_Transaction* regardless of whether an exception is thrown or not, we can place the instruction within the *finally* block as shown in Code 3. In this scenario it maybe that if a transaction is to be considered useful it should not produce any exceptions. However, regardless of the usefulness of the commit the application asks the underlying TM system to perform it.

2.4 Performance Impact

To get an intuition of the usefulness of commits we ex-

Code 3 Java TM code with exceptions

```
try{
    Begin_Transaction;
    ...
    ...
}catch(Exception e){}
finally{
    End_Transaction;
}
```

ecuted Lee-TM [23] and a TM version of Red-Black tree with a lazy-lazy HTM system similar to TCC [7] using 2-8 cores. We instrumented the code to check the usefulness of the transaction at commit time. For Lee-TM, if the *backtrack* phase is not executed we consider it as a non-useful commit. In Red-Black tree, inserting an already existing entry or deleting a non-existing entry are considered as non-useful commits. For Lee-TM we used a 75X75 grid with 320 routes to explore and for Red-Black tree, we used a tree with 20000 entries and transactionally inserting/deleting 16000 items with 50% probability for each action. Table 1 shows non-useful commits as a percentage of the total commits. From Table 1 it is clear that a large number of commits are

Processors	Lee-TM	Red-Black Tree
2	35%	49%
4	35%	49%
8	34%	49%

Table 1: Non-useful Commits

not useful for the overall program completion. However, the lack of usefulness does not reduce the number of commits that are done by the underlying TM system. Therefore we believe that a new functionality should be added to the existing hardware TM specification to allow the programmer to tell the underlying TM *"Discard this transactions computation and proceed"*.

3. Related Work

In this section we review and summarise proposed TM semantics, and discuss how they can be used to achieve the above objective. We group them into software and hardware approaches.

3.1 Software Approaches

Early release [10, 22] has been proposed as a way of reducing contention in both hardware and software TM systems. The objective is that a programmer can remove an entry from the read set of its current transaction. Thereafter any write operation to this location by other transactions will not cause conflicts with the current transaction. If we are to use this feature to achieve our objective, then we have to modify our code to remove all the memory locations that are read and written during current transaction from the read and write sets, if the current transaction is found to be non-useful. This may be possible with STMs and also with some HTMs, but most recent HTMs use signatures to keep track of read and write sets. Signatures are implemented as a fixed width bit representation, in which certain bits are set according to the address being considered. One of the features of signatures is that an element can be added to it, but cannot be removed. Therefore we cannot use early release to achieve our objective.

The TM construct *orElse* [9] is used to execute a second transaction if the first one fails. If we used this approach for our Lee-TM example, we would need to add a *dummy* transaction as the "retry" condition. Then according to the definition of *orElse*, in the case of a non-useful transaction the dummy transaction should commit. While this achieves our aim it does so with the loss of clear and concise syntax.

Crowl *et al.* proposed to integrate TM semantics in to C++ [12]. There, the authors discuss different ways to exit from a transaction. According to the authors, the "normal" way to end a transaction is to commit it. They also suggest to commit a transaction even if it is exited with typical C language keywords like *return, break, continue* and *goto*. In their specification another way to end a transaction is to exit with *longjmp*. The idea is to abandon the speculative operation without finishing it. Thereafter the environment is restored with the one saved by *setjmp*. This is similar to an abort operation, but still requires the transaction to be restarted, this will lead to a live lock in our Lee-TM example. Consider the case where *expand* cannot reach its destination because all the possible paths have been blocked, hence returns false. The same transaction will continue to retry until it succeeds, but as no route exists this will never happen.

TM constructs *__tm_abort* [16] and *user-level aborts* [21] have the same objective as our *Exit_Transaction*. The proposed behaviour is, once executed within a critical section, to discard all the speculative modifications and to transfer the control to the statement immediately following the critical section. However in these semantics, the programmer looses the ability to explicitly *abort* or *restart* transactions, they can only abort.

The xfork [18] framework allows programmers to define logical relationships between sibling transactions. The basic idea is to define nested transactions as AND, OR, or X-OR. When declared as AND all the sibling transactions should be completed in order to commit a transaction, when declared as X-OR only one successful transaction should be committed, and for transactions declared as OR each sibling transaction can fail or succeed independently. If we use this approach in the Lee-TM example, we can define both *expand* and *backtrack* methods as AND sibling transactions ensuring that we delay the execution of the latter until the former completes due to data dependencies. Xfork API for AND guarantees that, if any of the siblings returns false, no transaction will commit and the transactions will retry. Once again in the case of Lee-TM this can cause a live lock.

Finally, programming language extensions like *abox* [8] have been proposed in order to handle exceptions raised within critical sections. However no direct hardware support is provided for this and our proposal fits the required purposes well.

3.2 Hardware Approaches

McDonald *et al.* [14] propose the first Instruction Set Architecture (ISA) for HTM. Alongside with the functionalities expressed by previously proposed HTMs, authors suggest three major operations to manage transactions: *xbegin, xcommit, xabort*. As per with the *abort_transaction* function in Log-TM [15], *xabort* instruction executes a code that is registered with the abort handler. The purpose is to allow

the programmer to explicitly abort a transaction. If we use this feature to achieve our objective, we need to construct a *dummy* function which can explicitly transfer the control to the end of the atomic block. Such a function would just contain a *goto* statement to move the execution to the end of the block. We then need to register this function with the abort handler. Thereafter by invoking *xabort*, control can be transferred to the end of the atomic block. However, we still need to inform the hardware not to restart the transaction once the abort function completes because the default behaviour of *xabort* is to do so. Since such a facility is not provided, our objective cannot be achieved with the proposed ISA.

Notary [24] is a TM system which proposes to separate private and shared data and to exclude private data from the read and write sets of a transaction. In their approach authors propose using separate virtual pages for shared and private memory locations. If we take their approach compiler and or programming language support is needed to allocate all the private data, including stack, in those private pages. Then for the Lee-TM application we can make the write set size to zero when the *backtrack* phase is not executed. However, excluding local variables from the write set may pose consistency violations in TM. Sanyal *et al.* [20] propose an undo buffer to overcome this problem, but this approach requires extra hardware whose size cannot be determined in advance and also requires modifications to the memory management and to the run time systems.

Hardware support for TM has already been incorporated within Azul chips [5]. Their API also provides an *Abort* instruction which marks all the speculatively modified cache lines as invalid. However, it is not well defined whether the control is transferred to the beginning of the critical section [14] or to the end [21]. The Sun's Rock processor [4, 6] also provides TM support. Their design comes with only two extra instructions: *chkpt* and *commit*. When a transaction is started by a *chkpt* instruction, a pc-relative fail address can be registered with the transaction itself, control is then transferred to this address in case of an abort. They also provide an unconditional trap instruction which provides the facility to cause explicit aborts from software. Unfortunately this fail address feature is not able to achieve our objective because we have to register it at the beginning of the transaction and at that time we don't know whether the transaction is going to be useful or not. If we were to use this fail-address feature modifications are required. These modifications include when registering the pc-relative fail address, two addresses need to be registered: one pointing to the beginning of the transaction and the other pointing to the end of it. Later when invoked from the use code, we need to pass a value which indicates whether to retry or to exit. Then depending on this value, the abort mechanism will decide which operation to perform.

The Advanced Synchronization Facility(ASF) [1] is a proposal for extending hardware support for lock free data structures. They introduce 7 new instructions including *Abort*. Like with the ISA proposed by McDonald *et al.* [14], the *Abort* instruction rolls back the speculative region and the state is restored using the snapshot taken when the *Speculate* instruction was executed. The control is then transferred to the instruction proceeding the *Speculate* instruction. This instruction is a *JNZ* instruction which as the *Abort* has set the zero flag will jump to a handler. This handler can then decide based on the flags set by the abort what its should do next. Jumping to the end of the transaction is an option, but becuase there are no flags to signal that the user initiated the abort this cannot be done as the direct result of the programmer including a *Exit_Transaction* instruction. As a result while the required changes are small it currently cannot be used to achieve our objectives.

IBM recently presented the speculation support in their Blue Gene/Q chip [17]. However neither the ISA additions nor the API is available to discuss how we can achieve our objective in their system.

3.3 Summary

From the survey presented in this section, we can see that there is no direct hardware proposal to avoid committing non-useful transactions, despite such a semantic being proposed at the programming language level [16, 21]. This is ironic given that such a feature is more valuable to HTM than to STM because STM systems can produce a write set of zero size for Lee-TM when the *backtrack* phase is not executed. This is not the case with most of the hardware TM systems.

4. Exit_Transaction

At the level of the algorithm the programmer knows when a transaction is useful and when it is not. If provided with API calls similar to *Begin_Transaction* and *End_Transaction* the programmer can invoke these to notify the underlying TM system that the current transaction is not useful, hence discarding all the speculative operations and progressing to the end of the atomic section. We call this functionality as *Exit_Transaction*. The intended behaviour of the function is similar to an abort operation except that it does not restart the same transaction. It is also similar to a commit operation in the sense that control is transferred to the end of the atomic block, but speculative modifications are not communicated. Adding this functionality to an existing program does not add any complexity because the information to make this decision is already available in the program itself. In Code 4 we show how to modify the examples used in Section 2 to enable them to use this proposed feature.

Exit_Transaction can also used to increase the programmability of certain problem statements. For example consider a situation where a linked-list needs to be reversed if it found to be greater than a given threshold. A conventional way of achieving this is to count the elements of the list in a first pass and if it found to have a length longer than the given threshold use a second pass to reverse. This is shown in the top portion of Code 5. With the introduction of *Exit_Transaction*, we can rewrite the reverse method to speculatively reverse the list while counting the elements so requiring just one pass. Thereafter if the number of elements is greater than the given threshold, the transaction is committed, but if the condition is not met, the speculative modifications are abandoned using *Exit_Transaction* so the list remains unchanged. This is shown in the lower part of Code 5.

One could argue that we have not taken any explicit measures to transfer the control to the instruction following the atomic block as in *__tm_abort* [16] or *user-level aborts* [21] proposals. However, this is not required in our approach because when the *Exit_Transaction* is invoked the program counter register is already pointing to the end of the atomic

Code 4 Adding Exit_Transaction to existing code

```
Begin_Transaction;
bool isFound=expand();
if(isFound){
    backtrack();
    End_Transaction;
}else{
    Exit_Transaction;
}
-----------------------------------------------------
insert_item(key){
    Begin_Transaction;
    bool isFound=search(key);
    if(!isFound){
        insert(key);
        End_Transaction;
    }else{
        Exit_Transaction;
    }
}
-----------------------------------------------------
try{
    Begin_Transaction;
    ...
    ...
    End_Transaction;
}catch(Exception e){
    Exit_Transaction;
}
```

Code 5 Improving Programmability with *Exit_Transaction*.

```
length = count(list);
if(length > THRESHOLD) {
    reverse(list)
}
-----------------------------------------------------
modified_reverse(list){
    Begin_Transaction;
    while(more elements) {
        count
        reverse
    }
    if(count > THRESHOLD){
        End_Transaction;
    }else{
        Exit_Transaction;
    }
}
```

block. This can be seen in all the modified codes shown in Code 4. It is not possible to construct a case where such control transfer is necessary because any code after the *Exit_Transaction* instruction is simply not transactional. If there is a situation where code after the test needs to be transactional this means that the particular condition in the atomic block cannot be used to measure the usefulness of the commit. For example consider the top portion of the code shown in Code 6. Here one might consider the condition in line 3 as the condition to measure the usefulness and may modify the code to use the *Exit_Transaction* feature. The modified code is shown in the lower part of Code 6. This is clearly a wrong usage of the *Exit_Transaction* function because regardless of the condition in line 3, the statement in line 8 in the original code gets executed within the same transaction. Therefore we cannot use *Exit_Transaction* feature in this situation.

Code 6 Incorrect usage of Exit_Transaction

```
0: Begin_Transaction;
1: AAA
2: BBB
3: if(condition){
4:  CCC
5:  DDD
7: }
8: EEE
9: End_Transaction;
-----------------------------------------------------
0: Begin_Transaction;
1: AAA
2: BBB
3: if(condition){
4:  CCC
5:  DDD
6:  End_Transaction;
7: }else{
8:  Exit_Transaction;
9: }
10: EEE
```

5. Architectural support for Exit_Transaction

In this section we discuss how to extend two hardware TM systems to support *Exit_Transaction*. For this we used an improved version of Transactional Memory Coherence and Consistency (TCC) [7] as the baseline architecture. The transactional memory implementation in the baseline is similar to any other lazy-lazy hardware TM system. In order to provide an unbounded amount of transactional data, the baseline uses hardware signatures [19] to maintain the read and write sets, using parallel bloom filters to increase accuracy. Since our baseline architecture is based on TCC which does not implement any coherence protocols, transactions are used to maintain coherence and consistency as well. Therefore at the end of a transaction the next level memory copies are updated and local copies which are read or written to are flushed. This is necessary to avoid local caches ending up using stale data due to the lack of conventional coherence protocols.

When a processor needs to commit a transaction, it first requests commit permission from a centralised commit arbiter. Commit permission is granted based on a least recently granted policy. Once the commit permission is granted, the committing processor broadcasts its write-signature to all the other processors. Upon receiving this write-signature, each processor performs a bitwise AND operation on their read-signature. If all the hashes in the resulting signature are non-zero, then it is considered as a conflict and the processor aborts. After sending the write-signature to all the other processors, the committing processor updates the next level memory (either level 2 cache or main memory) with all the speculatively modified values. During this commit phase the communication arbiter denies any request to use the interconnect. Once the next level memory is updated with all the speculatively modified cache entries, all these entries are flushed and both read and write signatures are cleared.

5.1 Baseline-1: TM-S

Commits are serialised in the first baseline (TM-S) when addressing cache overflows within a transaction. That is, when a cache entry modified in the transaction needs to be ejected while a processor is inside a transaction, permission

is sought from the overflow arbiter. Like commit permission overflow permission is also granted based on a least recently granted policy. Once the overflow permission is granted, the processor flushes the cache line from its L1 cache and updates the corresponding entry either in L2 cache or main memory. An extra 'W' bit is used to mark all the speculatively modified entries. A dirty bit is not sufficient for this purpose because the entry could have been dirty due to a write operation performed outside a transaction. If the 'W' bit is not set, there is no need to seek overflow permission and the processor can flush it to its original location. If an overflow request is denied, the processor stalls until the request is granted. Earlier in this section, we said that the commit arbiter operates on a *least-recently-granted* policy. There is an exception to this in TM-S version of the baseline. That is, once the overflow permission is granted to a processor, all the commit requests from other processors are denied, until the overflowing processor commits. This is because once a speculatively modified entry is written back to the next level memory the original value is lost making this transaction non-reversible. Allowing cache overflows to speculatively modified entries, can be considered as violating atomicity and isolation properties of the lazy-lazy transactional memory. This is because the overflowed cache entries can now be read by other processors before their current transaction commits. However we can maintain the consistency property by making the current transaction of the overflowing processor an unabortable one. As described earlier this is achieved by denying all the commit requests until the overflowing processor commits.

In TM-S unabortable transactions raise issues for the implementation of *Exit_Transaction*. For example in a situation where a processor has been given overflow permission and then invoked *Exit_Transaction*. According to the specification of *Exit_Transaction*, now it should discard all the speculative changes and move to the end of the atomic block. However, we cannot discard all the changes because the overflowed memory operations have modified the original memory locations and their old values have been lost. If we did not have this *Exit_Transaction* feature we would commit this transaction even if it did not do any useful work. Such a commit would write back the remaining speculative values in the cache to their memory locations in the next level memory. We will now examine what would happen if this transaction is not committed. In the case of Lee-TM the modifications made to the local grid would not be visible to other processors, but none of the other processors can access these values, this is also true for the Red-Black tree. When deciding the required functionality for the *Exit_Transaction* for an unabortable processor in the TM-S system, we have the flexibility to either commit or discard the remaining speculative values. In this experiment we used the latter, that is to discard the remaining speculative values in the level 1 cache. However, if as in the linked list example in Code 5 we allow transactions to speculatively modify global state and rely on *Exit_Transaction* to revert the changes further work, would be required to guarantee consistency is not undermined.

5.2 Baseline-2: TM-U

In order to support an unbounded amount of transactional data the second baseline implementation (TM-U) overflows into a separate uncached area of memory. This is the same as in Unbounded Transactional Memory (UTM) [2]. The design and the protocol are similar to those of UTM, except that as we use signatures TM-U does not stall to check for potential conflicts that might arise from overflowed locations. When a cache line with the dirty bit set is to be overflowed the entire cache line including the tag, valid, dirty and data bits are preserved in this uncached area. Each entry is indexed by a modulo of the hash value of the overflowed memory location. Each processor also has an extra register (*Overflow_Address*) which points to the starting location of this separate area. If more than one memory location produced the same index, a linked list is formed. Finding an entry involves first retrieving the index and then retrieving the corresponding cache entry or list of cache entries stored under that index. A linear search is then performed by comparing the tag and index of each element in the list. TM-U has an extra bit called O per cache line to indicate the overflow status. This is set when a cache line is overflowed and is cleared only when a transaction commits or aborts. Even if an existing cache line is replaced with new data, this bit does not get changed.

In the TM-U approach, overflowing transactions are not serialised and the original memory locations are not modified. Therefore when *Exit_Transaction* is invoked no extra effort is required to undo the speculative operations. Discarding transactional operations can be done by invalidating speculatively modified cache entries and clearing the uncached area of the memory. Adding support for *Exit_Transaction* in this baseline does not incur any consistency violations.

6. Evaluation

The evaluation of the *Exit_Transaction* feature is presented in this section. After discussing the evaluation setup in section 6.1, in section 6.2 we show that using the *Exit_Transaction* function a TM system can outperform the lazy-lazy HTM systems that do not support such a feature. We also characterise the results in the same section.

6.1 Evaluation Setup

As our proposal relies on transactions we modelled a lazy-lazy hardware transactional memory system in Simics [13], a full system simulator running Linux (version 2.6.16). The TM system is configured with the components shown in Table 2. In addition to these, the baseline-2 (TM-U) uses a perfect hash function to index its overflowed memory locations. The Lee's routing algorithm [23] and a TM version of the Red-Black tree are used to evaluate the *Exit_Transaction* feature. Both applications were modified to exit from a transaction if it is found not to be useful, as shown in Code 4. Unmodified versions of both applications were executed for comparison purposes. Lee-TM uses 75X75 grid and 320 routes as the input. A tree with 20,000(0↔200,000) nodes and 16000(0↔200,000) insertions/deletions with 50% probability for each was used for the Red-Black tree experiment.

6.2 Performance

Figure 1 shows the speedup measured using the *Exit_Transaction* function over baseline architectures that do not support such functionality. With TM-S architecture using *Exit_Transaction* functionality we achieved a speedup of upto 1.35X and with TM-U the speedup is 1.23X.

Component	Feature
Processors	1-8
L1 Data Cache	2 way assoc, 64 B line, 32 KB size, 2 cycle latency, private per core
Signature	2048 Bits, 4 Parallel H3 [3] Hash functions
L2 Data Cache	8 way assoc, 64 B line, 4 MB size, 20 cycle latency, shared
Interconnect	Split-transaction bus, 4 cycle latency, 64 B data width
Main Memory	100 cycle latency

Table 2: Components and Features.

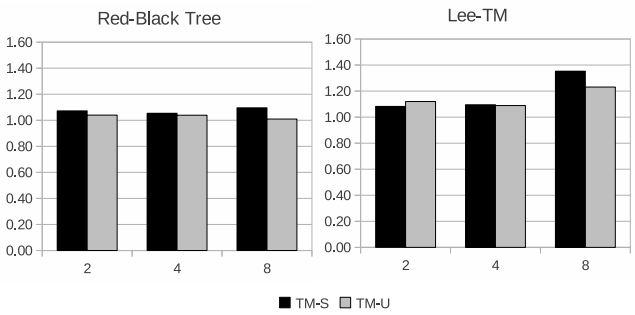

Figure 1: Speedup of using *Exit_Transaction* over baseline implementations.

We characterise our results with several parameters in order to measure the effect of the *Exit_Transaction* feature and what makes the speedup to vary between both systems and applications. First we analysed the percentage of transactions that used the *Exit_Transaction* feature. Table 3 shows how many times *Exit_Transaction* feature has been used as a percentage of total commits. There we can see for the Red-Black tree, the percentage is around 50% in both systems, and 10%-12% for Lee-TM. Since the Red-Black tree uses the *Exit_Transaction* feature more than Lee-TM, one might expect it to show bigger speedup in Figure 1, however this is not the case.

Processors	TM-S		TM-U	
	Lee-TM	RB Tree	Lee-TM	RB Tree
2	11.40%	50.11%	12.02%	49.97%
4	11.48%	49.84%	12.00%	49.77%
8	11.33%	50.24%	11.84%	49.73%

Table 3: Usage of Exit_Transaction as a Percentage of Total Commits

Next we measured the size of the write set. If the size of the write set is small the amount of time spent committing may not make a significant difference to the overall execution time and the risk of overflow and false conflicts will be smaller. Therefore we analysed the amount of speculative data committed in both applications. Table 4 shows the number of bytes committed per transaction in both applications for both systems. From Table 4 we can see that Lee-TM has a significantly bigger write set size in comparison to that of the Red-Black tree. Therefore exiting from a transaction without committing it gives a bigger advantage to Lee-TM than it does to the Red-Black tree. This results in Lee-TM showing better speedups than the Red-Black tree example.

Processors	TM-S		TM-U	
	Lee-TM	RB Tree	Lee-TM	RB Tree
2	9519.15	518.38	9742.59	580.00
4	9941.44	529.47	9704.24	588.78
8	11864.72	534.09	9929.95	592.80

Table 4: Bytes Committed per Transaction

Not using the interconnect for non-useful commits reduces the contention for it, in other words, one of the overheads incurred by unnecessary commits is the bus contention. In both TM systems the communication arbiter is designed to give the highest precedence to commit requests. Since a commit phase takes time to complete all bus requests are denied during this time, this further increases the bus contention. Figure 2 shows the number of requests denied as a result of bus contention in both TM systems. TM-S-Baseline refers to the TM-S architecture(section 5.1) without support for *Exit_Transaction*, similarly TM-S-Exit_Transaction refers to the TM-S architecture (section 5.1) with support for *Exit_Transaction*. The same applies for TM-U-Baseline and TM-U-Exit_Transaction.

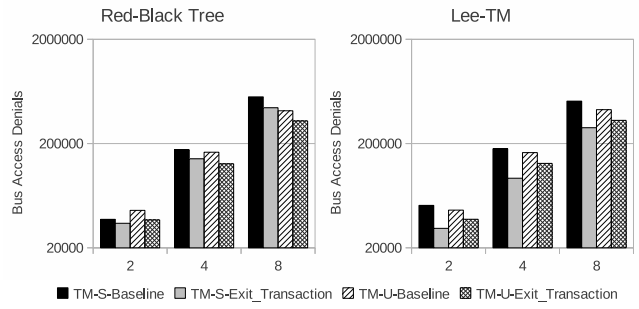

Figure 2: Bus Contention

From Figure 2 we can see that as expected that both applications show less bus contention when the *Exit_Transaction* functions is being used. This is mainly because of the removal of commits that are not useful to the completion of the program.

We also measured the number of false positives when testing for conflicting transactions in both applications on both systems when the *Exit_Transaction* function is used. This is shown in Figure 3. For the Red-Black tree there is no significant difference between two TM systems, but in the case of Lee-TM we can see that TM-U produces more false positives than TM-S. The reason for this is that the transactions in Lee-TM often cannot be held in the level 1 cache and overflow during the execution of the atomic block. Because TM-S serializes commits in this situation it only has one large transaction at any given time. In the case of TM-U, there can be any number of large transactions running at a given time, this increases the probability of false positives. In addition when a transaction becomes longer, more addresses are inserted to the signature. When more addresses are inserted to the signature it increases the probability of producing false positives.

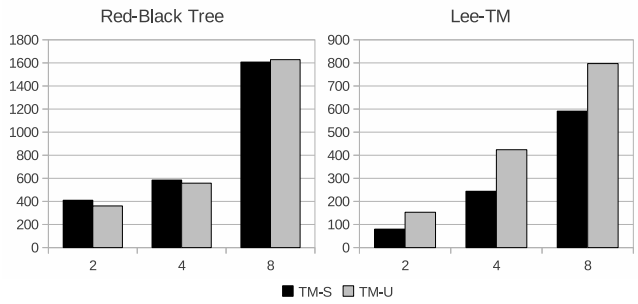

Figure 3: False Positives

7. Applicability of Exit_Transaction on other TM Systems

So far we have only demonstrated the advantages of *Exit_Transaction* with respect to lazy versioning HTM systems, we will now consider the applicability of our proposal to eager versioning HTM systems. In an eager system all modifications are made in-place and this reduces the commit overhead. In such a system when *Exit_Transaction* is invoked it has to discard all the speculatively written entries as in a lazy versioning system. If the transaction fits in the L1 caches the cost of this process is same for both eager versioning and lazy versioning HTMs. If the transaction has overflowed the cache, for eager versioning HTMs this involves reading the original value from a separate log and replacing the modified entry with this value. For lazy versioning HTMs this depends on how the overflows are handled. For examples in TM-S baseline it is not possible to restore such memory locations as the original value is not recorded. In the case of TM-U baseline, this involves clearing the overflow area of the memory. This means that lazy versioning HTMs have an advantage over the eager ones when a transaction does not fit in the L1 cache. However, this costly step is is only required if we are supporting speculative modifications to data structures as in the linked list example. For our benchmarks this would not be required. Avoiding the commit phase for non-useful transactions reduces the bus contention which counts for a certain fraction of the reported speedups. In the case of eager versioning HTMs this will not result in as direct an advantage as it does for lazy ones.

We do believe even eager versioning HTMs will also benefit from *Exit_Transaction* functionality being provided. For example, consider the situation where a transaction fits in the L1 cache of a eager HTM system. Even though the transaction is not useful, a commit operation is performed. Since the transaction fits in the cache, no communication is done at the commit phase. For simplicity, lets assume the cache is filled with transactionally modified entries. Later when a cache miss is encountered space has to be allocated in the current cache by writing back an existing entry. Even though this entry is modified, the value has no use as it belongs to a non-useful transaction. If *Exit_Transaction* functionality is being provided, it could have cleared all these entries thereby avoiding this communication. A similar situation where *Exit_Transaction* can be useful to eager HTMs is when a context switch happens after the commit phase of a non-useful transaction. In such situations all the dirty cache entries need to be saved before allocating space for cache requests of the new process. This saving of state requires communication, if *Exit_Transaction* functionality is provided this communication can be avoided by clearing the dirty cache entries of non-useful transactions.

8. Conclusion

This paper has presented a case for the need of the *Exit_Transaction* function to be added to hardware TM systems. In addition, we also showed how an existing HTM system can be extended to support such a feature. We evaluated the *Exit_Transaction* feature using the Lee-TM and a transactional version of the Red-Black tree with two hardware TM systems. Our results showed that with hardware support for *Exit_Transaction*, we can achieve a speedup of up to 1.35X for the applications tested. This speedup is gained from a combination of lower false positives, lower bus contention and less wasted processor time.

9. Acknowledgements

We would like to thank anonymous reviewers for their advice on the paper. Isuru Herath is supported by an Overseas Research Studentship and a School of Computer Science studentship from the University of Manchester. Demian Rosas is supported by National Council of Science and Technology of Mexico. Dr. Goodman is supported by the European Community's Seventh Framework Programme (FP7/2007-2013) under grant agreement no 249013 (TERAFLUX-project). Dr. Luján is supported by a Royal Society University Research Fellowship.

10. REFERENCES

[1] Advanced Micro Devices. Amd advanced synchronization facility proposal. `http://developer.amd.com/tools/ASF/Pages/default.aspx`, 2009.

[2] C. S. Ananian, K. Asanovic, B. C. Kuszmaul, C. E. Leiserson, and S. Lie. Unbounded transactional memory. *IEEE Micro*, 26:59–69, 2006.

[3] J. L. Carter and M. N. Wegman. Universal classes of hash functions (extended abstract). In *Proceedings of the ninth annual ACM symposium on Theory of computing*, STOC '77, pages 106–112, New York, NY, USA, 1977. ACM.

[4] S. Chaudhry, R. Cypher, M. Ekman, M. Karlsson, A. Landin, S. Yip, H. Zeffer, and M. Tremblay. Rock: A high-performance sparc cmt processor. *Micro, IEEE*, 29(2):6 –16, march-april 2009.

[5] C. Click. Htm will not save the world. Presentation at TMW10 workshop, May 2010.

[6] D. Dice, Y. Lev, M. Moir, and D. Nussbaum. Early experience with a commercial hardware transactional memory implementation. In *Proceeding of the 14th international conference on Architectural support for programming languages and operating systems*, ASPLOS '09, pages 157–168, New York, NY, USA, 2009. ACM.

[7] L. Hammond, V. Wong, M. Chen, B. D. Carlstrom, J. D. Davis, B. Hertzberg, M. K. Prabhu, H. Wijaya, C. Kozyrakis, and K. Olukotun. Transactional memory coherence and consistency. In *Proceedings of the 31st annual international symposium on Computer architecture*, ISCA '04, pages 102–, Washington, DC, USA, 2004. IEEE Computer Society.

[8] D. Harmanci, V. Gramoli, and P. Felber. Atomic boxes: coordinated exception handling with transactional memory. In *Proceedings of the 25th European conference on Object-oriented programming*, ECOOP'11, pages 634–657, Berlin, Heidelberg, 2011. Springer-Verlag.

[9] T. Harris, S. Marlow, S. Peyton-Jones, and M. Herlihy. Composable memory transactions. In *Proceedings of the tenth ACM SIGPLAN symposium on Principles and practice of parallel programming*, PPoPP '05, pages 48–60, New York, NY, USA, 2005. ACM.

[10] M. Herlihy, V. Luchangco, M. Moir, and W. N. Scherer, III. Software transactional memory for dynamic-sized data structures. In *Proceedings of the twenty-second annual symposium on Principles of distributed computing*, PODC '03, pages 92–101, New York, NY, USA, 2003. ACM.

[11] M. Herlihy and J. E. 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, New York, NY, USA, 1993. ACM.

[12] V. L. M. M. Lawrence Crowl, Yossi Lev and D. Nussbaum. Integrating transactional memory into c++. In *TRANSACT '07: 2nd ACM SIGPLAN Workshop on Transactional Computing*, August 2007.

[13] P. Magnusson, M. Christensson, J. Eskilson, D. Forsgren, G. Hallberg, J. Hogberg, F. Larsson, A. Moestedt, and B. Werner. Simics: A full system simulation platform. *Computer*, 35:50–58, 2002.

[14] A. McDonald, J. Chung, B. D. Carlstrom, C. C. Minh, H. Chafi, C. Kozyrakis, and K. Olukotun. Architectural semantics for practical transactional memory. In *Proceedings of the 33rd annual international symposium on Computer Architecture*, ISCA '06, pages 53–65, Washington, DC, USA, 2006. IEEE Computer Society.

[15] K. Moore, J. Bobba, M. Moravan, M. Hill, and D. Wood. Logtm: log-based transactional memory. In *High-Performance Computer Architecture, 2006. The Twelfth International Symposium on*, pages 254 – 265, 2006.

[16] Y. Ni, A. Welc, A.-R. Adl-Tabatabai, M. Bach, S. Berkowits, J. Cownie, R. Geva, S. Kozhukow, R. Narayanaswamy, J. Olivier, S. Preis, B. Saha, A. Tal, and X. Tian. Design and implementation of transactional constructs for c/c++. In *Proceedings of the 23rd ACM SIGPLAN conference on Object-oriented programming systems languages and applications*, OOPSLA '08, pages 195–212, New York, NY, USA, 2008. ACM.

[17] M. Ohmacht. Hardware support for transactional memory and thread-level speculation in the ibm blue gene/q system. Presentation at Wild and Sane Ideas in Speculation and Transactions workshop, October 2011.

[18] H. Ramadan and E. Witchel. The Xfork in the road to coordinated sibling transactions. In *TRANSACT '09: 4th ACM SIGPLAN Workshop on Transactional Computing*, February 2009.

[19] D. Sanchez, L. Yen, M. D. Hill, and K. Sankaralingam. Implementing signatures for transactional memory. In *Proceedings of the 40th Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO 40, pages 123–133, Washington, DC, USA, 2007. IEEE Computer Society.

[20] S. Sanyal, S. Roy, A. Cristal, O. S. Unsal, and M. Valero. Dynamically filtering thread-local variables in lazy-lazy hardware transactional memory. In *HPCC '09: Proceedings of the 2009 11th IEEE International Conference on High Performance Computing and Communications*, pages 171–179, Washington, DC, USA, 2009. IEEE Computer Society.

[21] T. Shpeisman, A.-R. Adl-Tabatabai, R. Geva, Y. Ni, and A. Welc. Towards transactional memory semantics for c++. In *Proceedings of the twenty-first annual symposium on Parallelism in algorithms and architectures*, SPAA '09, pages 49–58, New York, NY, USA, 2009. ACM.

[22] T. Skare and C. Kozyrakis. Early release: Friend or foe? In *TRANSACT '06: 1st ACM SIGPLAN Workshop on Languages, Compilers, and Hardware Support for Transactional Computing*, June 2006.

[23] I. Watson, C. Kirkham, and M. Lujan. A study of a transactional parallel routing algorithm. In *Proceedings of the 16th International Conference on Parallel Architecture and Compilation Techniques*, PACT '07, pages 388–398, Washington, DC, USA, 2007. IEEE Computer Society.

[24] L. Yen, S. C. Draper, and M. D. Hill. Notary: Hardware techniques to enhance signatures. In *MICRO '08: Proceedings of the 2008 41st IEEE/ACM International Symposium on Microarchitecture*, pages 234–245, Washington, DC, USA, 2008. IEEE Computer Society.