Helper Transactions: Enabling Thread-Level Speculation via A Transactional Memory System

Richard M. Yoo
yoo@ece.gatech.edu

Hsien-Hsin S. Lee
leehs@ece.gatech.edu

School of Electrical and Computer Engineering
Georgia Institute of Technology
Atlanta, GA 30332

ABSTRACT

As multi-core processors become readily available in the market, how to exploit parallelization opportunities to unleash the performance potential has become the utmost concern. Thread-Level Speculation (TLS) has been studied as one such enabling technique to automatically extracting possibly non-conflicting threads for execution in a program. On the other hand, transactional memory (TM) systems have received much attention recently as a promising alternative for parallel programming due to its simplicity. However, its current scope of use is largely limited to the elimination of traditional locks. In this paper, we discuss how TM systems can be extended to enable TLS, thereby significantly extending the usage model of TM systems. Among the many TLS techniques, we discuss a design in-depth to effectively exploit out-of-order procedure fall-through speculation. In this design, speculatively executed transactions gracefully degenerate to helper threads in the worst case scenario.

1. INTRODUCTION

Homogeneous multi-core processors are aimed at improving throughput with multiple hardware contexts. Nonetheless, finding enough tasks to exercise these contexts can be a challenge. Applications such as web servers, database systems, scientific applications, etc., can be easily benefited. Whereas, most of the common applications used by desktop users will unlikely obtain performance advantage of a multi-core due to their sequential nature. For these applications, the make or break of multi-core architecture will highly depend on how many independent multiple threads can be extracted and executed in parallel.

Thread-Level Speculation (TLS) was proposed as a performance technique to address this problem [22, 5, 20, 1, 24]. Using TLS, a program can be divided into a number of possibly non-conflicting tasks. The spawn points of these tasks are usually determined by high level programming language constructs, *e.g.*, loops, if-then-else statements, procedure fall-throughs, etc. The hardware then speculatively executes these tasks in parallel, performing squashing and rollbacks for those tasks that violate the inter-task dependency. Ideally, TLS can spawn a sufficient number of threads to exercise these multiple hardware contexts on multi-core processors. When correctly speculated, performance of an application will be improved.

Transactional Memory (TM) [7, 6, 2, 19, 13] is another approach to increase concurrency on a multi-core processor. Unlike TLS, TM does not spawn a new task to exploit more parallelism. Rather, TM systems help the existing threads perform better by relaxing the strict exclusion of thread execution imposed by traditional locks. A transaction essentially contains a sequence of instructions that must be executed in an atomic fashion; these instructions either commit or abort as a single large operation. Transactions are allowed to execute speculatively to obtain more parallelism. It is the responsibility of the underlying TM system to detect and abort transactions that violate memory dependency.

Nonetheless, the use of a TM system has been seemingly limited to the liberation of traditional lock-based programming. Compared to the complexity of implementing TM in the hardware, the realizable benefits are rather small. In fact, TLS and TM share many functionalities, such as dependency violation detection, result buffering, checkpointing, and replay. Therefore, some recent work did attempt to utilize TM in the context of TLS [8, 24]. However, their usage limits the TM only to its dependency violation detection and result buffering capabilities.

In this paper, we show that by extending TM with proper task spawning mechanism, context passing mechanism (Section 4.1), and sequential ordering scheme (Section 4.2), TM can be readily transformed into a full-blown TLS platform. This approach significantly extends the use cases of TM systems, while lowering the implementation costs for TLS on a TM-enabled system. In particular, we discuss a design in-depth to effectively exploit out-of-order procedure fall-through speculation. In this design, speculatively executed transactions gracefully degenerate to helper threads in the worst case scenario.

Table 1 positions our design in the context of various parallelism-enabling techniques. As can be seen from the table, helper transactions technique positions itself as a unique merger of TM system and out-of-order TLS; although it natively supports out-of-order procedure fall-through speculation, it can still be applied to parallelize traditional locks.

2. THREAD-LEVEL SPECULATION

In TLS, a program is first divided into multiple tasks. In a compiler-supported TLS implementation, the boundary of these tasks are determined by high level programming language constructs: *e.g.*, loops, if-then-else statements, and procedure fall-throughs.

Procedure fall-through speculation, in particular, exploits the functional level parallelism between the called procedure and the continuing code that comes right after the function

Implementation	Category	Architecture	Operand Passing Network	Out-of-order Spawn	Lock Parallelization
Multiscalar [22]	TLS	Dedicated	Y	-	-
SVC [3]	TLS	SMP	Y	-	-
Hydra [5]	TLS	CMP	-	Y	-
PolyFlow [1]	TLS	SMT	-	Y	-
TLS4OutOrder [20, 9]	TLS	CMP	-	Y	-
Voltron [24, 8]	TLS	CMP	Y	Y	?
Helper Transactions	TM + TLS	CMP	-	Y	Y
TCC [6]	TM	CMP	-	-	Y
UTM [2]	TM	CMP	-	-	Y
LogTM [13, 14]	TM	CMP	-	-	Y

Table 1: Comparison of Various Parallelism-Enabling Techniques

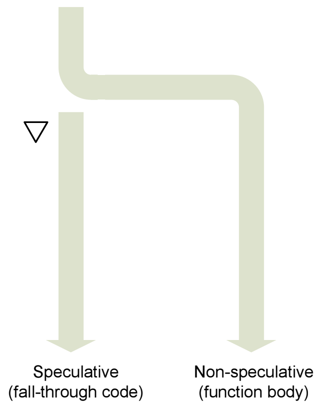
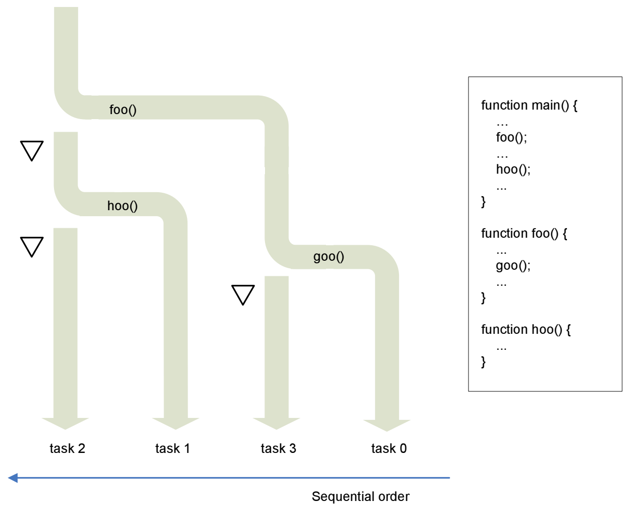


Figure 1: Procedure Fall-Through Speculation



```
function main() {
    ...
    foo();
    ...
    hoo();
    ...
}

function foo() {
    ...
    goo();
    ...
}

function hoo() {
    ...
}
```

Figure 2: Out-of-Order Procedure Fall-Through Speculation

call. Figure 1 shows a typical task tree for procedure fall-through speculation.[1] In this figure, upon a function call, the main task speculatively spawns a new task with the fall-through code, while continuing itself into the callee function. Instructions for the function body and the fall-through code are then executed in parallel.

When a task can spawn more than one speculative task throughout its lifetime, the task spawning scheme is called out-of-order spawn [20]. Figure 2 shows a typical task tree for the case of procedure fall-through when out-of-order spawns are enabled. In this diagram, the non-speculative task that continued into foo() encounters another function call goo(), spawning another speculative task (task 3). The speculative task executing the fall-through code of main() (task 1) also encounters another call hoo() and spawns a speculative task itself (task 2). This task spawning scheme is called out-of-order because the spawn order of tasks (0-1-2-3) does not follow the sequential order (0-3-1-2) of tasks.

Maintaining sequential order in in-order spawning is relatively simple; the order of processors imply the sequential order of the tasks executing on those processors [3, 11]. For out-of-order spawning, maintaining the sequential order among tasks becomes much more complex. However, there is an urge for the need for out-of-order spawning to extract additional parallelism [9, 11, 20]. Our design includes a mechanism to maintain sequential order for out-of-order spawning.

[1]In this diagram and throughout the paper, the ▽ symbol denotes spawning a new task or thread.

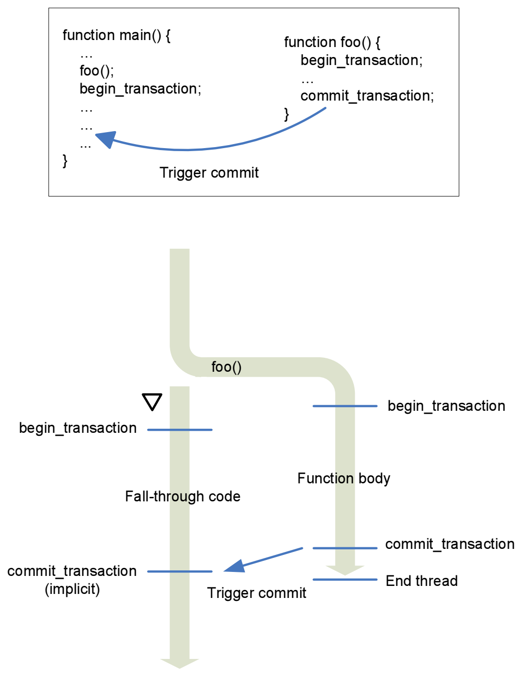
3. MAPPING TLS ONTO A TM SYSTEM

This section overviews how the procedure fall-through speculation can be mapped onto the TM framework. Detailed implementations will be described in Section 4. Section 3.1 describes the basic approach. Section 3.2 then describes the changes needed to support out-of-order spawning on TM systems.

3.1 The Basics

From the TM perspective, each task in TLS basically amounts to a transaction. Figure 3 shows the result when procedure fall-through speculation of Figure 1 is mapped onto TM.

When encountering a function call, the main thread spawns a new thread with the fall-through code. The main thread then continues on to the function body. However, the function body is guarded with begin_transaction and commit_transaction instructions. At the same time, the spawned thread starts a transaction itself upon starting execution. The TM then detects any memory dependency violation between the function body transaction and the fall-through code transaction.



```
function main() {
    ...
    foo();
    begin_transaction;
    ...
    ...
    ...
}
```

```
function foo() {
    begin_transaction;
    ...
    commit_transaction;
}
```

Figure 3: Procedure Fall-Through Speculation on TM

The most difference in this approach over a conventional TM is that each transaction will execute a different code region. Therefore, there must be a sequential order among transactions. In Figure 3, the transaction executing the function body sequentially precedes the transaction executing the fall-through code. Hence, the function body transaction must commit before the other transaction.

This can be achieved with 2 mechanisms. First, for TM systems with eager conflict detection [2, 13], the TM should abort the fall-through transaction in favor of function body transaction upon detecting conflicts. The aborted transaction will rollback to the point right before the fall-through code, and resume execution. Notice that on a multi-core system with a shared L2 cache, the aborted transaction, in fact, may improve the overall execution performance by warming up the instruction cache with the fall-through code [18].

Second, upon reaching the commit_transaction instruction, the fall-through transaction should be stalled until the function body transaction commits. This can also be achieved by not inserting any explicit commit_transaction instruction in the fall-through transaction; the pairwise order is recorded, and the commit of the function body transaction implicitly triggers the commit of the fall-through transaction. The implicit commit scheme can optimize performance since it allows the fall-through transaction to continue execution, instead of stalling and waiting for the function body transaction to commit. Figure 3 also depicts such optimization.

3.2 Supporting Out-of-Order Spawn

The greatest challenge in supporting out-of-order spawn on a TM system lies in how to define the semantics of a transaction spawning new transactions. This can be easily achieved by mapping out-of-order spawns to nested transactions [17, 15, 14, 12, 16]. In a TM system supporting nested transactions [17, 12, 14], a transaction may have multiple concurrent children transactions.

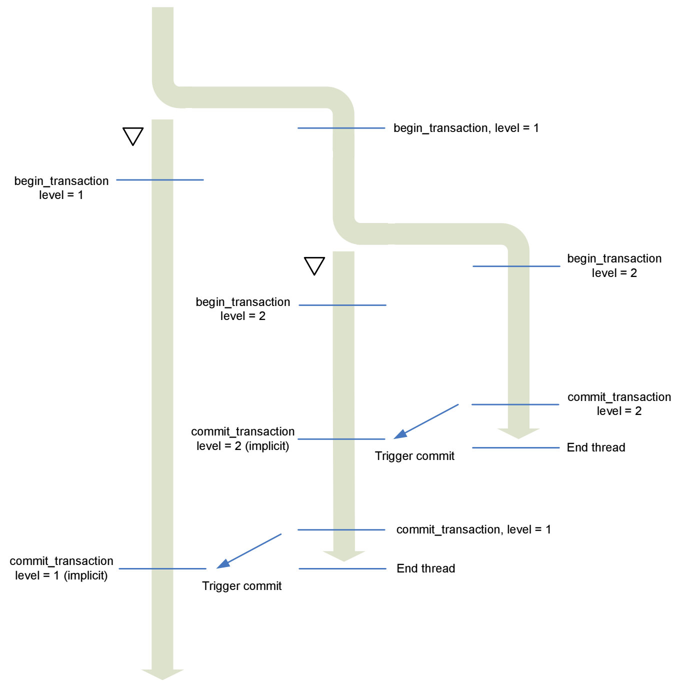


Figure 4: Out-of-Order Procedure Fall-Through Speculation on TM

Specifically, there are two distinct schemes in supporting nested transactions: closed nested transaction [14, 17] and open nested transaction [12, 16]. Due to the space limitation, we only focus on the case of closed nested scheme. However, our approach can be implemented on the open nested scheme as well, since we commit a transaction only when it is guaranteed that its outer transaction will commit.

Figure 4 shows the task tree when an out-of-order spawn is mapped to nested transactions. In this diagram, all begin_transaction and commit_transaction are explicit instructions except where denoted as implicit. Basically, at each spawn point, the spawning thread increases its nesting level by 1. At the same time, the spawned thread starts a transaction at the same nesting level as the spawning transaction. For example, in Figure 4, the rightmost transaction (the non-speculative one) increases its transaction level from 1 to 2 at the second spawn point; the spawned thread then starts a transaction at the same level (=2).

In the same way as in Section 3.1, sequential ordering is maintained by 1) aborting more speculative transaction in favor of less speculative transaction upon conflict, and by 2) stalling the commit of the more speculative transaction until the less speculative transaction commits. Again, in Figure 4, all speculative transactions should stall on explicit commit_transaction instruction. The figure also shows the case where the commit of the less speculative transaction triggers the commit of the more speculative transaction. In this figure, the rightmost transaction triggers the commit of the middle transaction, and so forth.

In this particular example, the middle transaction encounters commit_transaction instruction after the rightmost transaction encounters commit_transaction instruction. Figure 5, on the contrary, shows the case where the middle transaction (more speculative) encounters commit_transaction

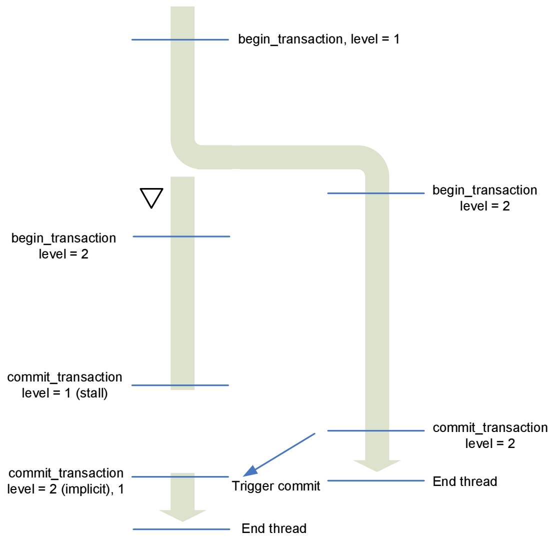

begin_transaction, level = 1
begin_transaction level = 2
begin_transaction level = 2
commit_transaction level = 1 (stall)
commit_transaction level = 2
commit_transaction level = 2 (implicit), 1
Trigger commit
End thread
End thread

Figure 5: Speculative Transaction Stalling for Less Speculative Transaction

instruction before the rightmost transaction (less speculative) commits.

As described in the figure, the more speculative transaction stalls for the less speculative transaction to commit. Notice that there is a strict ordering among the commits — a transaction can commit only when it is least speculative (non-speculative). Otherwise, a transaction should stall on an explicit commit_transaction instruction.

The mechanism described so far would be sufficient to support TLS on a TM system. However, without the dedicated operand passing network, spawning the fall-through code as a new transaction on a remote core can be costly when all the register dependencies between the code prior to the function call and the fall-through code should be converted into memory dependency. Figure 6 shows the modified approach to alleviate this problem.

In this modified approach, upon encountering a function call, the main thread spawns a new thread with the function body rather than the fall-through code as shown in Figure 4 or in a generic TLS implementation. The main thread then increases its transaction level, and continues onto the fall-through code. This way, the memory traffic for context passing can be radically reduced. Moreover, this approach significantly simplifies the compiler's support (Section 4.1).

Nonetheless, this scheme requires the *partial abort* support [12, 14] from the TM system. As in Figure 6, if the middle transaction conflicts with the rightmost transaction, the middle transaction should be aborted due to the sequential ordering. Without the partial abort support, this will end up aborting the middle transaction up to and including transaction level 1. With partial abort support, the middle transaction only needs to rollback up to the point where it started transaction level 2. This way, the transaction information pertaining to transaction level 1 will be kept intact.

In summary, with closed, nested TM supporting partial abort, the out-of-order spawn can be implemented on TM. However, there is one caveat — by a nested TM we refer to a 'true' nested TM which can support multiple concurrent child transactions. Notice that not many hardware TM implementations faithfully implement nested transactions [2, 6, 13, 14]; what they implement is actually 'linear nesting,' which only supports one child transaction at a time. Following the specification and the discussion in [17], a true nested TM should support multiple child transactions. Hardware architects have long ignored this point since they assumed such an execution model does not exist. Our design shows that a true nested TM is crucial to implement out-of-order TLS on TM.

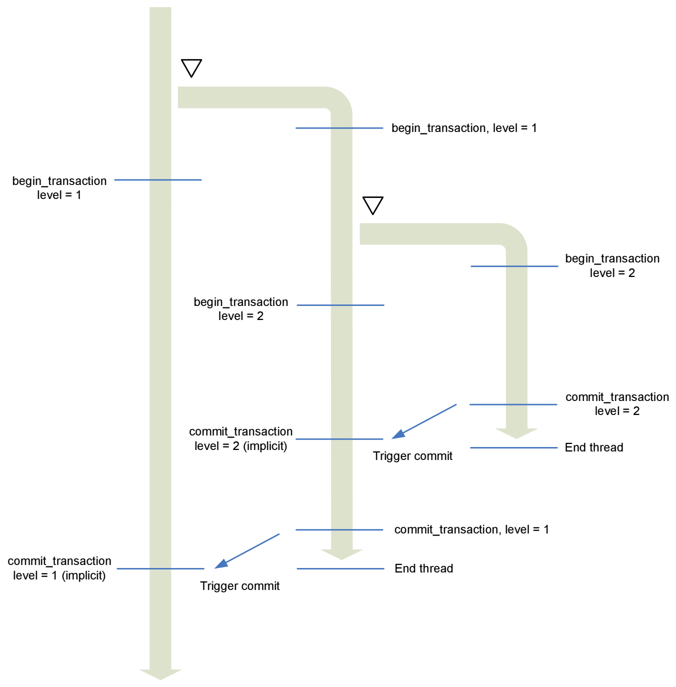

begin_transaction, level = 1
begin_transaction level = 1
begin_transaction level = 2
begin_transaction level = 2
commit_transaction level = 2
commit_transaction level = 2 (implicit)
Trigger commit
End thread
commit_transaction, level = 1
commit_transaction level = 1 (implicit)
Trigger commit
End thread

Figure 6: Out-of-Order Procedure Fall-Through Speculation on TM (Revised)

4. EXTENDING TM SYSTEM

This section discusses how to actually implement the design presented in Section 3.2. Since TM does not include task spawning mechanism and context passing mechanism, these functions should be implemented in the compiler (Section 4.1). Moreover, the sequential ordering mechanism should be built into the TM; this is then discussed in Section 4.2.

4.1 Compiler Support

To support task spawning, the compiler should insert codes for thread creation. As discussed in Section 3.2, the newly created thread should execute the function body guarded with transactions. Moreover, the continuation thread should start a transaction itself. Listings 1 and 2 show the necessary transformations.

Listing 1: Sample Code Before TLS Transformation

```
int main ( int argc, char* argv[])
{
    int a, b, c;

    ...
    foo( a, b, c);
    ...
}
```

```
int foo ( int arg0, int arg1, int arg2)
{
    ... // function foo body
}
```

Listing 2: Sample Code After TLS Transformation

```
int in_memory_a, in_memory_b, in_memory_c;

int main ( int argc, char* argv[])
{
    int a, b, c;

    ...
    in_memory_a = a;
    in_memory_b = b;
    in_memory_c = c;

    create_thread( _tls_foo);

    begin_transaction;
    ...
}

void* _tls_foo ( void* arg)
{
    int arg0, arg1, arg2;

    arg0 = in_memory_a;
    arg1 = in_memory_b;
    arg2 = in_memory_c;

    begin_transaction;
    foo( arg0, arg1, arg2);
    commit_transaction;
}

int foo ( int arg0, int arg1, int arg2)
{
    ... // function foo body
}
```

Listing 1 shows the original code, while the listing 2 shows the code after the necessary transformations were applied. Upon encountering the function call foo(a, b, c), the compiler should generate a new function _tls_foo() whose body is actually the function call itself. Also, the body should be guarded with begin_transaction instruction and commit_transaction instruction. Then, the function call is replaced with the thread spawning directive, to spawn a new thread with the newly created function _tls_foo(). Finally, the begin_transaction instruction is inserted right before the fall-through code so that the spawning thread will start a transaction itself.

Note that the compiler does not insert any explicit commit_transaction instruction to the fall-through code. This facilitates the use of implicit commit. In this implementation, the begin_transaction instruction and the commit_transaction instruction should be modified to denote whose child transaction the instruction is beginning (committing). This can be accomplished by assigning a unique ID to each of the transactions, and then passing this ID as an argument [17].

Moreover, assuming the new thread will be executing on a remote core, all register dependencies should be converted into memory dependencies, so that the correct register values are passed on to the spawned thread. However, since we are spawning a new thread for the callee, not the fall-through code, the register dependency only exists through the arguments of the callee function. Therefore, only the arguments for the called function need to be translated into memory dependencies.

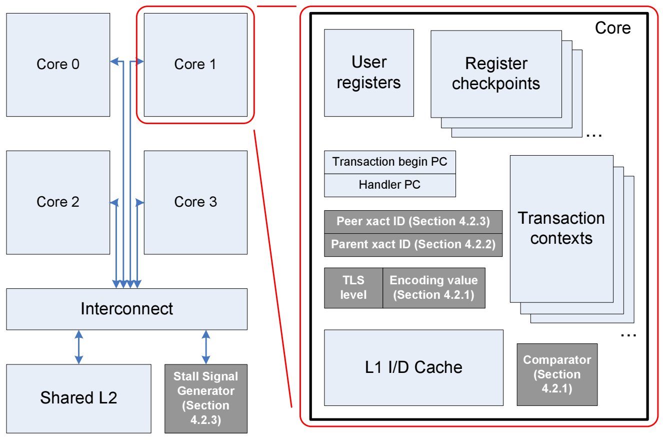


Figure 7: Helper Transactions Hardware Overview

This can be achieved by assigning a 'conduit' for each of the arguments. A new variable which has the same type as the original argument is declared to reside in memory, and the argument is passed through this variable. The compiler should insert store instructions to copy the arguments into the memory, and load instructions to retrieve them back.

Notice that global (static scope) variables will reside in memory by default. However, on platforms which could pass global variables through registers, the compiler should force these variables to reside in memory. This is typically done by adding a volatile attribute to the variable [9].

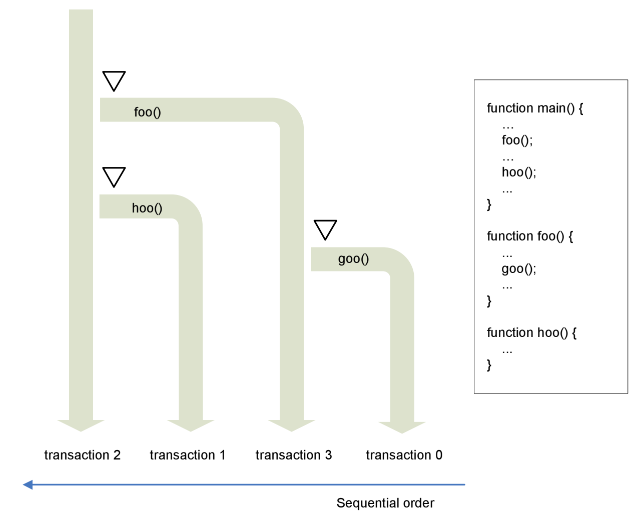
4.2 Hardware Extensions

Hardware extensions focus on implementing the sequential ordering scheme. Figure 7 shows the modifications required to support TLS on TM systems. Darker blocks indicate the added modules. Section 4.2.1 first discusses on how to encode the ordering information in hardware. Based on this encoding scheme, Section 4.2.2 discusses how to abort transactions. Lastly, in Section 4.2.3 we present the hardware module to guarantee sequential ordering for transaction commits.

4.2.1 Encoding the Sequential Ordering

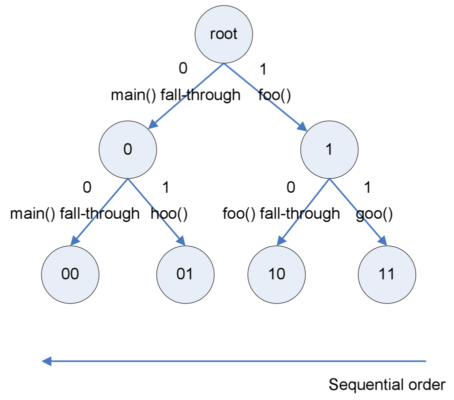
As pointed out in Section 3.1, now that the transactions are executing different code regions, there must be a sequential order among the transactions. This ordering should be directly implemented in the TM since it determines 1) which transaction to abort when there is a dependency violation, and 2) which transaction should stall on commit. To better deliver the idea, we introduce the notion of binary tree to sequential ordering.

Figure 8(a) shows an example task tree, which is the TM view of the task tree in Figure 2. Figure 8(b) shows this example task tree encoded in a binary tree. In this binary tree, each node represents a single nested transaction — two child nodes represent the two child transactions of a transaction. The root node represents the state where there are no active transactions. Notice that according to the definition of nested transactions [16, 17], only the leaf nodes represent the currently executing transactions. In this diagram, the



```
function main() {
    ...
    foo();
    ...
    hoo();
    ...
}

function foo() {
    ...
    goo();
    ...
}

function hoo() {
    ...
}
```

(a) Example Task Tree



(b) Task Tree Encoded in a Binary Tree

Figure 8: Encoding the Sequential Order in a Binary Tree

value in each node represents the encoded sequential ordering.

The value is maintained as follows. When a transaction reaches a function call site, it starts two child transactions: one with the function body, and the other with the fall-through code. Notice that the child transaction executing the function body sequentially precedes the child transaction executing the fall-through code. This information is passed down to child transactions as bits 1 and 0; the child transaction executing the function body receives 1, while the other child transaction executing the fall-through code receives 0. Each child transaction then appends this bit to its parent's encoding value as its own encoding value.

Overall, the node value encodes the 'lineage' of the current transaction. For example, when the value is 10, we know that the transaction has been reached by executing a function body at the first spawn point, followed by the execution of the fall-through code at the second spawn point.

By comparing these encoding values, we can determine which transaction precedes the others in sequential ordering. For example, in Figure 8(a), when the transaction continues executing the fall-through code of foo() (transaction 3) and the transaction executing function hoo() (transaction 1) conflict over a memory block, by comparing their encoding values 10 and 01, we know that transaction 3 precedes transaction 1 in sequential order. Therefore, transaction 3 should be continued in this case.

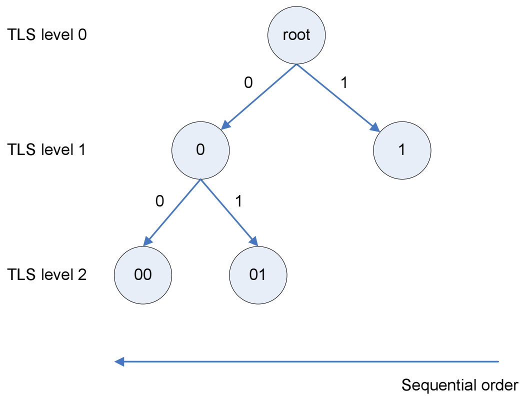


Figure 9: An Example Requiring Valid Digits

Nonetheless, to compare bit patterns, we need the notion of valid digits. Figure 9, for example, represents the situation after the two least speculative child transactions have committed.

To guarantee correct sequential ordering between the transaction encoded as 1 and the transaction encoded as 01, we need to know how many digits to compare. TLS levels can be used for this purpose. TLS levels denote the nesting depth of the current transaction. In our encoding scheme, TLS level actually denotes the number of valid digits to compare. The pseudo-code to compare encoding values with TLS level information is given in Listing 3.

Listing 3: Pseudo Code to Compare Sequential Encoding Values

```
bool remoteIsOlder (Message inMsg)
{
    int local_encoding    = m_encoding;
    int local_tls_level   = m_tls_level;
    int remote_encoding   = inMsg.encoding;
    int remote_tls_level  = inMsg.tls_level;

    int diff = abs( remote_tls_level -
                    local_tls_level);

    if (local_tls_level < remote_tls_level) {
        remote_encoding >>= diff;
    } else {
        local_encoding >>= diff;
    }

    return (remote_encoding > local_encoding);
}
```

The code in Listing 3 returns true if the remote transaction precedes the local transaction in sequential order. Applying this scheme to compare (encoding = 01, TLS level = 2) against (encoding = 1, TLS level = 1) yields true, correctly representing that 1 precedes 01 in sequential order.

By including the encoding and the TLS level information in cache coherence messages, and by performing the comparison in each memory controller, each transaction can determine the sequential ordering between its peers and itself without storing the binary tree in a central structure.

In the rest of the paper we assume that the valid digits are implicit: *e.g.*, encoding value 01 implies that it has two valid digits.

4.2.2 *Aborting a Subtree of Transactions*

In Section 4.2.1 we showed how to utilize encoding values and TLS levels to determine which transaction precedes the other in sequential ordering. This information can be used to abort the offending transaction. However, when aborting a speculative transaction, transactions that are more speculative than the aborting transaction should also be aborted to maintain the proper sequential order. In our implementation, we conservatively abort a subtree of transactions. Listing 4 describes the algorithm to determine the root of such aborting subtree.

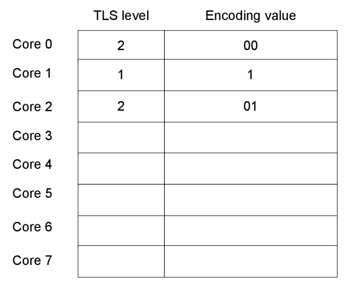
Listing 4: Pseudocode to Determine the Root of the Aborting Subtree

```
Node getSubtree (Node victim, Node offender)
{
    while (victim.TLS_level != offender.TLS_level)
    {
        if (victim.TLS_level > offender.TLS_level) {
            victim = victim.parent;
        } else {
            offender = offender.parent;
        }
    }

    while (victim.parent != offender.parent)
    {
        victim = victim.parent;
        offender = offender.parent;
    }

    return victim;
}
```

In Listing 4, the first while loop level the victim and the offender by replacing a node with its parent until they are at the same TLS level. The second while loop then percolates the information up until the victim and the offender reach the same parent. In hardware, victim.parent operation amounts to passing the offender information up to its outer transaction, and offender.parent operation amounts to shifting the encoding bit of the offender.

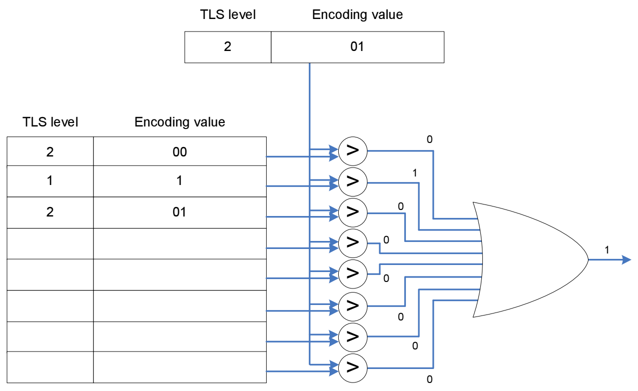
For example, in Figure 9, when transaction 01 and transaction 1 conflicts, from Section 4.2.1 we know that the transaction 01 should be aborted. In the first while loop of Listing 4, transaction 01 percolates the information up to its outer transaction, transaction 0. The second while loop will return right away since transaction 0 and transaction 1 share the same parent. So in this example, transaction 0 becomes the root of the aborting subtree.

The root then initiates the aborting process by recursively aborting transactions in postorder: its left child first, then its right child, then itself. Each aborting transaction is responsible for aborting its two child transactions. This aborting process guarantees that the transactions are aborted in reverse sequential order: from the most speculative transaction to the least speculative transaction. Note that only the root node transaction should resume after the abort; it is semantically incorrect to resume the nested transaction when its outer transaction has been aborted.



	TLS level	Encoding value
Core 0	2	00
Core 1	1	1
Core 2	2	01
Core 3		
Core 4		
Core 5		
Core 6		
Core 7		

(a) Example Status of the Module



(b) Module Generating Stall Signal

Figure 10: Hardware Mechanism to Order Transaction Commits

In this context, when to resume the aborted transaction could be an interesting optimization choice. Instead of restarting the aborted transaction right away, the TM could determine to stall [13, 21] resuming the transaction until the conflicting less speculative transaction commits. In this case the aborted transaction effectively becomes the helper thread. In TM systems supporting contention manager [21, 10], this can be easily implemented by integrating the sequential order with the contention management scheme. On those systems without contention manager, a sequential order aware transaction scheduling could be applied [23].

4.2.3 *Ordering the Commits*

Due to the sequential ordering, commits of transactions should occur in a specific order. There should be a central module to serialize the commits, similar to the way a reorder buffer serializes the commits of out-of-order instructions. Figure 10 shows the organization and the operation of this module.

In this module, each entry in the table maintains the TLS level and the encoding value of the current transaction. Assuming that the TM can support only one transaction per core, there should be as many entries as there are cores. When a transaction begins, it updates the pertaining entry with its TLS level and the encoding value, as calculated in Section 4.2.1. When a transaction commits, if the transaction was the left child of an outer transaction, it updates the entry with outer transaction's TLS level and the encoding value. When it is the right child transaction or it does not have an outer transaction, it simply clears the entry. Figure

10(a) shows the status of the module when executing the task tree of Figure 9.

Upon reaching the explicit commit_transaction instruction, a transaction should consult this module to determine whether to stall or not. Figure 10(b) shows the example of transaction 01 consulting the module in the context of Figure 9.

In this diagram, each comparator next to the entry performs the encoding comparison as in Listing 3. This time, the local_encoding variable in the pseudo-code notation pertains to the transaction querying the module. Each comparison is performed in parallel, then the output is ORed to generate the stall signal. Notice that the pseudo-code correctly generates value 0 for the comparison with itself and with an empty entry. Stall signal will be 1 when a sequentially preceding transaction is executing — the querying transaction should stall. By polling this module periodically, the transaction can determine when it is the least speculative transaction: when it can commit. To guarantee atomicity, the table should be single-ported.

Moreover, to facilitate implicit commit (Section 3.1), function body transaction should trigger the commit of its sibling fall-through transaction. This can be achieved by assigning a unique ID to each of the transactions, and then storing the sibling information in the transaction context. This trigger message should be buffered on the receiving side, because it might receive the trigger message while it has been aborted but has not resumed yet, or it could be executing a different transaction.

5. FUTURE WORK

Procedure fall-through speculation has been pointed out as a significant source of speculation. Similar to out-of-order execution that exploits ILP, TLP can be exposed by enabling out-of-order spawning, in particular, for a future many-core system which may find in-order spawning scheme too restricted in exploiting TLP. Many prior research thrusts [22, 9, 11] have already demonstrated that out-of-order spawning can significantly improve performance over its in-order spawning counterpart.

More quantitatively, Marcuello *et al.* [11] pointed out that about 76% of the SPEC INT95 application code can potentially benefit from the procedure fall-through speculation. It was also shown in [1] that procedure fall-through tasks comprise more than 30% of the entire TLS tasks. In more realistic implementations supporting out-of-order spawning, techniques in [20, 1] showed that the procedure fall-through speculation alone realizes 10% to 20% speedup for SPEC INT2000, over the single-threaded execution. We expect our implementation to show similar speedup results.

There are previous researches utilizing TM in TLS context [24, 8, 4]. However, in these works, the use of TM was significantly limited either 1) to statistically parallelizing the loop [8] or 2) to support in-order spawning only [4]. To the best of our knowledge, our paper is the first to exploit out-of-order spawning on TM framework. It will be interesting to observe to what extent the parallelism can be exploited when procedure fall-through speculation and lock parallelization are applied at the same time.

6. CONCLUSION

In this paper, we proposed and discussed a design in-depth to show how to extend a TM system to support TLS. In particular, we studied the procedure fall-through speculation. Compared to TLS systems, TM systems lack task spawning mechanism, context passing mechanism, and sequential ordering mechanism. By implementing task spawning and context passing mechanism in a compiler, and by implementing sequential ordering mechanism in hardware, we have successfully demonstrated a TM framework that supports TLS. Specifically, by mapping the out-of-order task spawning scheme onto the closed nested transaction semantics, we proposed a holistic way to support out-of-order spawning directly on TM systems.

7. ACKNOWLEDGMENTS

This work was supported in part by an NSF CAREER Award (CNS-0644096).

8. REFERENCES

[1] M. Agarwal, K. Malik, K. M. Woley, S. S. Stone, and M. I. Frank. Exploiting postdominance for speculative parallelization. In *Proceedings of the 2007 International Symposium on High Performance Computer Architecture*, February 2007.

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

[3] S. Gopal, T. N. Vijaykumar, J. E. Smith, and G. S. Sohi. Speculative versioning cache. In *Proceedings of the 4th International Symposium on High-Performance Computer Architecture*, pages 195–205, 1998.

[4] L. Hammond, B. D. Carlstrom, V. Wong, B. Hertzberg, M. Chen, C. Kozyrakis, and K. Olukotun. Programming with transactional coherence and consistency (TCC). In *Proceedings of the 11th Intl. Conference on Architectural Support for Programming Languages and Operating Systems*, October 2004.

[5] L. Hammond, M. Willey, and K. Olukotun. Data speculation support for a chip multiprocessor. In *ASPLOS-VIII: Proceedings of the 8th International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 58–69, 1998.

[6] 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*, pages 102 – 113, June 2004.

[7] 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*, pages 289 – 300. May 1993.

[8] S. A. Lieberman, H. Zhong, and S. Mahlke. *Extracting Statistical Loop-Level Parallelism using Hardware-Assisted Recovery.* Technical Report CSE-TR-528-07, University of Michigan, February 2007.

[9] W. Liu, J. Tuck, L. Ceze, W. Ahn, K. Strauss, J. Renau, and J. Torrellas. POSH: A TLS compiler that exploits program structure. In *Proceedings of the 11th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, pages 158–167, 2006.

[10] V. Marathe, W. Scherer III, and M. Scott. Adaptive software transactional memory. In *Proceedings of the 19th International Symposium on Distributed Computing*, 2005.

[11] P. Marcuello and A. González. A quantitative assessment of thread-level speculation techniques. In *14th International Conference on Parallel and Distributed Processing Symposium*, pages 595–604, May 2000.

[12] A. McDonald, J. Chung, B. D. Carlstrom, C. C. Minh, H. Chafi, C. Kozyrakis, and K. Olukotun. Architectural semantics for practical transactional memory. In *ISCA '06: Proceedings of the 33rd Annual International Symposium on Computer Architecture*, pages 53–65, 2006.

[13] K. E. Moore, J. Bobba, M. J. Moravan, M. D. Hill, and D. A. Wood. LogTM: Log-based transactional memory. In *Proceedings of the 12th International Conference on High Performance Computer Architecture*, pages 254 – 265, February 2006.

[14] M. J. Moravan, J. Bobba, K. E. Moore, L. Yen, M. D. Hill, B. Liblit, M. M. Swift, and D. A. Wood. Supporting nested transactional memory in LogTM. In *ASPLOS-XII: Proceedings of the 12th International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 359–370, 2006.

[15] J. E. B. Moss. *Nested transactions: An approach to reliable distributed computing*. PhD thesis, Massachusetts Institute of Technology, 1981.

[16] J. E. B. Moss. Open nested transactions: Semantics and support. In *Workshop on Memory Performance Issues*, 2006.

[17] J. E. B. Moss and A. L. Hosking. Nested transactional memory: Model and preliminary architecture sketches. In *The 2005 Workshop on Synchronization and Concurrency in Object Oriented Languages, held at OOPSLA*, October 2005.

[18] J. Pierce and T. Mudge. Wrong-path Instruction Prefetching. In *Proceedings of the 29th Annual ACM/IEEE International Symposium on Microarchitecture*, pages 165–175, 1996.

[19] R. Rajwar, M. Herlihy, and K. Lai. Virtualizing transactional memory. In *Proceedings of the 32nd Annual International Symposium on Computer Architecture*, pages 494 – 505, June 2005.

[20] J. Renau, J. Tuck, W. Liu, L. Ceze, K. Strauss, and J. Torrellas. Tasking with out-of-order spawn in TLS chip multiprocessors: Microarchitecture and compilation. In *ICS '05: Proceedings of the 19th Annual International Conference on Supercomputing*, pages 179–188, 2005.

[21] W. N. Scherer, III and M. L. Scott. Advanced contention management for dynamic software transactional memory. In *Proceedings of the 24th Annual ACM Symposium on Principles of Distributed Computing*, pages 240 – 248, 2005.

[22] G. S. Sohi, S. E. Breach, and T. N. Vijaykumar. Multiscalar processors. In *Proceedings of the 22nd Annual International Symposium on Computer Architecture*, pages 414–425, 1995.

[23] R. M. Yoo and H.-H. S. Lee. Adaptive transaction scheduling for transactional memory systems. In *Proceedings of the 20th ACM Symposium on Parallelism in Algorithms and Architectures, Special Track on Hardware and Software Techniques to Improve the Programmability of Multicore Machines*, June 2008.

[24] H. Zhong, S. Lieberman, and S. Mahlke. Extending multicore architectures to exploit hybrid parallelism in single-thread applications. In *Proceedings of the 2007 International Symposium on High Performance Computer Architecture*, February 2007.