# Hardware Acceleration of Transactional Memory on Commodity Systems

Jared Casper Tayo Oguntebi Sungpack Hong
Nathan G. Bronson Christos Kozyrakis Kunle Olukotun

Pervasive Parallelism Laboratory
Stanford University
{jaredc, tayo, hongsup, nbronson, kozyraki, kunle}@stanford.edu

## Abstract

The adoption of transactional memory is hindered by the high overhead of software transactional memory and the intrusive design changes required by previously proposed TM hardware. We propose that hardware to accelerate software transactional memory (STM) can reside outside an unmodified commodity processor core, thereby substantially reducing implementation costs. This paper introduces Transactional Memory Acceleration using Commodity Cores (TMACC), a hardware-accelerated TM system that does not modify the processor, caches, or coherence protocol.

We present a complete hardware implementation of TMACC using a rapid prototyping platform. Using this hardware, we implement two unique conflict detection schemes which are accelerated using Bloom filters on an FPGA. These schemes employ novel techniques for tolerating the latency of fine-grained asynchronous communication with an out-of-core accelerator. We then conduct experiments to explore the feasibility of accelerating TM without modifying existing system hardware. We show that, for all but short transactions, it is not necessary to modify the processor to obtain substantial improvement in TM performance. In these cases, TMACC outperforms an STM by an average of 69% in applications using moderate-length transactions, showing maximum speedup within 8% of an upper bound on TM acceleration. Overall, we demonstrate that hardware can substantially accelerate the performance of an STM on unmodified commodity processors.

***Categories and Subject Descriptors*** C.1.4 [*Processor Architectures*]: Parallel—Distributed Architectures

***General Terms*** Algorithms, Design, Performance

***Keywords*** Transactional Memory, FPGA, Hardware Acceleration

## 1. Introduction

Transactional memory (TM) [19, 23] is a potential way to simplify parallel programming. Ideally, TM would allow programmers to make frequent use of large transactions and have them perform as well as highly optimized fine-grain locks. However, this ideal cannot be realized until there are real systems capable of executing large transactions with low overhead. Our aim in this paper is to describe a TM system that strikes a reasonable balance between performance, cost and system implementation complexity.

Researchers have proposed a wide variety of TM systems. There are systems implemented completely in hardware (HTMs), completely in software (STMs), and more recently, systems with both hardware and software components (*hybrid* TMs). To put our contributions in context, we now briefly review the strengths and weaknesses of the various TM design alternatives.

### 1.1 TM Design Alternatives and Related Work

#### STM

Software transactional memory (STM) systems [15, 18, 25, 31, 32, 36] replace the normal loads and stores of a program with short functions ("barriers") that provide versioning and conflict detection. These transactional read and write barriers must themselves be implemented using the low-level synchronization operations provided by commodity processors. The barriers can be inserted automatically by a transaction-aware compiler [1, 2, 39] or managed runtime [31], added by a dynamic instrumentation system [28], or invoked manually by the programmer. STMs increase the number of executed instructions, perform extra loads and stores, and require metadata that takes up cache space and needs to be synchronized. The resulting inherent performance penalty means that despite providing good scalability, most STMs fall far short of the performance offered by hardware-based approaches to TM. There have been proposals that reduce the overhead required [39], but they do so by giving up on the promise of TM–they require small transactions that are used rarely. Hence, using these STMs is as difficult as using fine-grain locks. As a result of these limitations, STMs have been largely constrained to the domain of research [9].

#### HTM

At the opposite end of the spectrum from STM is hardware transactional memory (HTM) [4, 5, 11, 17, 24, 29]. HTM systems eliminate the need for software barriers by extending the processor or memory system to natively perform version management and conflict detection entirely in hardware, allowing them to demonstrate impressive performance. Version management in an HTM is performed by either buffering speculative state (typically in the cache or store buffer) or by maintaining an undo log. Metadata that allows conflict detection is typically stored in Bloom filters (signatures) or in bits added to each line of the cache. The close synergy of the hardware with the processor core and cache allow these systems to provide very high levels of performance; however this tight relationship causes the system to be inflexible and more costly. Recent advances in HTM design address both of these problems



*ASPLOS'11,* March 5–11, 2011, Newport Beach, California, USA.

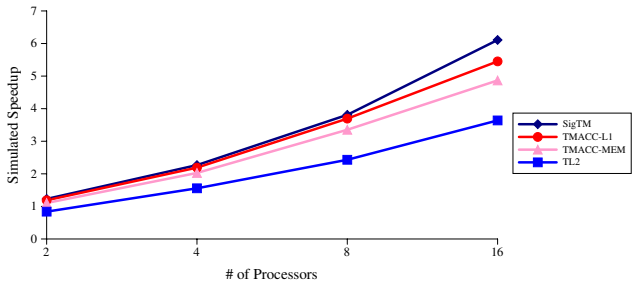


**Figure 1.** Average (mean) performance on the STAMP suite of two simulated TMACC systems, one two cycles away from the core (L1) and one two hundred cycles away (MEM). These are compared to TL2, a pure STM, and an in-core hybrid TM system much like SigTM.

by minimizing the coupling between the TM and the processor core [33, 40], but even decoupled HTM designs introduce non-trivial design complexity and disturb the delicate control and data paths in the processor core. The first-level cache has the effect of hiding loads and stores from the outside world, making it impossible to construct an out-of-core pure HTM system. Previous studies have not explored the possibility of adding transactional acceleration hardware without modifying a commodity processor core.

#### HybridTM

One way of limiting the complexity required by an HTM is to provide a limited best-effort HTM that falls back to an STM if it is unable to proceed [12, 14, 20, 22, 38]. These systems are particularly well-suited for supporting lock-elision and small transactions. However, applications that use large transactions (or cannot tune their transactions to avoid capacity and associativity overflows) will find that they derive no benefit. This approach is especially problematic as the research community explores transactional memory as a programming model, since it prescribes a limit on how transactions may be used efficiently.

#### Hardware Accelerated STM

Hardware accelerated STMs are a type of hybrid TM that use dedicated hardware to improve the performance of an STM. This hardware typically consists of bits in the cache or signatures that accelerate read set tracking and conflict detection. Existing proposals extend the instruction set to control new data paths to the TM hardware. Explicit read and write barriers then use the TM hardware to accelerate conflict detection and version management [7, 30, 34].

### 1.2 TMACC Motivation

We observe that hardware acceleration of an STM's barriers only requires that the runtime be able to communicate with the hardware; the TM hardware need not be part of the core or connected to the processor with a dedicated data path. Commodity processors are already equipped with a network that provides high bandwidth, low latency, and dedicated instructions for communication: the coherence fabric. This leads to the unexplored design space of hardware accelerated TM systems that do not modify the core, or Transactional Memory Acceleration using Commodity Cores (TMACC). Early simulation results, presented in Figure 1, show the promising potential of TMACC systems to perform within five to ten percent of an in-core hybrid TM system. These results also suggest that much of that performance can be realized despite a relatively large latency between the processing cores and the TMACC hardware.

Keeping the hardware outside of the core maintains modularity, allowing architects to design and verify the TM hardware and processor core independently. This significantly reduces the cost and risk of implementing TM hardware and allows designers to migrate a core design from one generation to the next while continuing to provide transactional memory acceleration.

There is therefore great benefit in exploring TM systems that can be feasibly constructed using commodity processors. Such systems will allow researchers to:

1. better understand and fine-tune TM semantics using real hardware and large applications
2. explore the extent of speedup and hardware acceleration possible without modifying the processor core
3. better understand the issues associated with tolerating the latency of out-of-core hardware support for TM

To derive these benefits in this paper, we describe the design and implementation of a hardware accelerated TM system, implemented with commodity processor cores. Like the accelerators presented in systems like FlexTM [33], BulkSC [10], LogTM-SE [40], and SigTM [7], we use Bloom filters as signatures of a transaction's read and write sets. Unlike these previous proposals, our Bloom filters are located outside of the processor and require no modifications to the core, caches, or coherence protocol. In this paper we also address the non-trivial challenges encountered when the acceleration hardware is moved out of the core.

The major contributions of this paper are:

- We present a system (both software and hardware) for Transactional Memory Acceleration using Commodity Cores (TMACC). We detail two novel algorithms for transactional conflict detection, both of which employ general purpose out-of-core Bloom filters. (Section 3).
- We describe two techniques for tolerating the latency of fine-grained asynchronous communication with an out-of-core accelerator (Section 2).
- We construct a complete hardware implementation of TMACC using FARM [27], a rapid prototyping platform. We also shed light on practical intuition gained regarding issues one must consider when adding TM acceleration hardware (Section 2.3).
- We demonstrate the potential of TMACC by evaluating our implementation using a custom microbenchmark and the STAMP benchmark suite. We show that, for all but short transactions, it is not necessary to modify the processor to obtain substantial improvement in TM performance. TMACC outperforms an STM by an average of 69%, showing maximum speedup within 8% of an upper bound on TM acceleration (Section 4).

The rest of the paper is structured as follows: Section 2 introduces our overall approach, details our hardware platform and TM acceleration modules on the FPGA, and describes techniques to mitigate the high communication latencies between the cores and the out-of-core hardware. Section 3 describes the TM schemes we propose and explains how the hardware is utilized in these schemes. Section 4 presents the results of experiments using our microbenchmark and the STAMP suite and projects the performance of an ASIC TMACC implementation. Finally, Section 5 concludes.

## 2. Accelerating TM

In this section we present our system for Transactional Memory Acceleration using Commodity Cores, or TMACC. We first give a high level overview of our design decisions and describe our general use of Bloom filters. We then briefly introduce our experimentation vehicle, the FARM prototyping platform. We follow with a more detailed description of our Bloom filter hardware, which is general and flexible enough to be placed anywhere in the system. We describe how we implement this hardware using FARM. Finally we present techniques to tolerate the command reordering encountered while communicating with off-chip acceleration hardware.

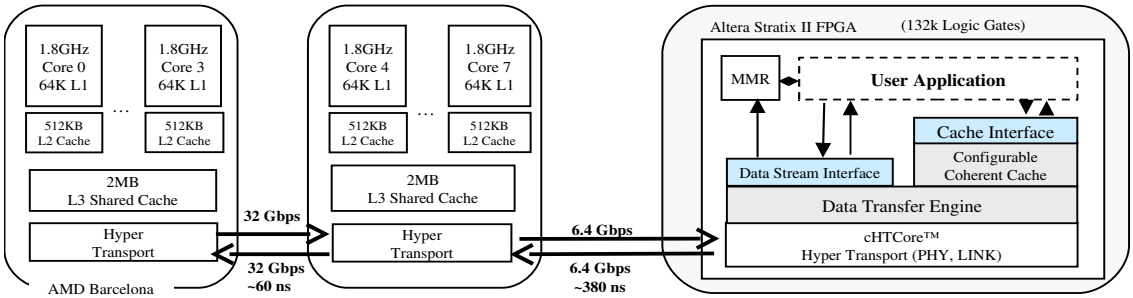


**Figure 2.** Diagram of system topology on the FARM platform.

In any TM system, the processor must have very low latency access to transactionally written data while hiding that data from other executing threads. Performing this version management in hardware and being able to promptly return speculatively written data would almost certainly require modification of the L1 data cache or the data path to that cache. Previously proposed HTM systems use buffers next to the L1, or the L1 itself, to store this speculative data until the transaction commits. Imposing out-of-core latencies on these accesses would significantly degrade performance. We therefore conclude that performing hardware-based or hardware-assisted version management in a TMACC system is impractical.

To address this issue of version management, our software runtime uses a heavily optimized chaining hash table as a write buffer. A transactional write simply adds an address/data pair to this hash table. Each transactional read must first check for inclusion of the address in the write buffer. If it is present the associated data is used; otherwise, a normal load is performed. The hash table is optimized to return quickly in the common case where the key (the address) is not in the table. Once the transaction has been validated and ordered (i.e. given permission to commit), the write buffer is walked and each entry applied directly to memory. The details of our implementation are out of the scope of this paper as write buffer data structures are more thoroughly explored elsewhere [13, 15, 25, 31].

Application of the write buffer could potentially be performed by the TMACC hardware, freeing the processor up to continue on to the next transaction. However, initial experiments showed that any benefit is outweighed by the impact of reloading the data into processor's cache after application of the write buffer. This is an area of potential future work.

Like version management, checkpointing the architectural state at the beginning of a transaction and restoring that state upon rollback would require significant modification to the processor core in order to be effectively and efficiently handled in hardware. We thus perform this entirely within the software runtime using the standard `sigsetjmp()` and `longjmp()` routines.

This leaves conflict detection as the best target for out-of-core hardware acceleration. After all, the speculative nature of an optimistic TM system means that the latency of the actual detection of conflicts is not on the critical path. Conflict detection is a primary contributor to execution overhead in STM systems, and many STM proposals have attempted to improve it.

In this work, we present two novel methods for performing conflict detection, both of which use Bloom filters as signatures of a transaction's read and write set. Bloom filters [3] have been shown to be an effective data structure for holding sets of keys with very low overhead and have been used for multiple applications, including the acceleration of transactional memory [7, 10, 36, 40]. Like several other TM proposals, TMACC uses Bloom filters to encode the read and write sets of running transactions. When a transaction commits, each address that is written can be quickly checked against the read and write sets of other concurrent transactions in order to discover conflicts. Details of the TM algorithm can be found in Section 3. The TMACC system presented in this work assumes a lazy optimistic STM. There are no fundamental reasons, however, why TMACC could not be used to accelerate an eager pessimistic system. We have a working draft of such a scheme, but do not present it in this paper.

### 2.1 FARM Prototyping Platform

In order to fully qualify a TMACC design, we needed a platform which would allow for easy experimentation with real applications. Due to its performance and flexibility, the FARM system [27] represents such a platform. FARM features an FPGA coherently connected to two commodity CPUs and physically interconnected through a backplane via HyperTransport. Figure 2 shows a diagram of its topology. As shown in the figure, FARM provides two logical interfaces for communication with the CPU: a) a *coherent interface* which uses cache lines managed by the coherence protocol and b) a *data stream interface* which provides streaming (or "fire-and-forget") non-coherent communication. For brevity, we omit the implementation and performance details of the system and refer the reader to the paper on FARM [27] for this information.

### 2.2 Bloom filters

Figure 3 presents a block diagram of a collection of Bloom filters. Note that while logic symbols are used, Figure 3 does not represent a physical implementation, but a logical diagram of the functionality provided. In addition to the normal add, clear, and query operations, each individual Bloom filter provides functionality to copy bits in from another filter or broadcast out its bits to other filters. Each Bloom filter also has a tag associated with it, which can be used, for example, to associate a Bloom filter with a particular thread of execution. Programmability of the module is achieved in the control block, which can be programmed to translate high level application-specific operations to the low level operations (add, query, clear, copy in, and copy out) sent to each individual Bloom filter. These operations can potentially be predicated by the $tag\_hit$ and $tag\_gt$ signals.

On FARM, the Bloom filters are placed in the placeholder marked "User Application" in Figure 2. We use four randomly selected hash functions from the $H_3$ class [8]. We considered using PBX hashing [41], which is optimized for space efficiency, but we were not constrained by logic resources on the FPGA. We perform copying by stepping through the block ram word by word. In order to reduce the number of cycles needed to copy, filters requiring copy support use additional RAM blocks to widen the

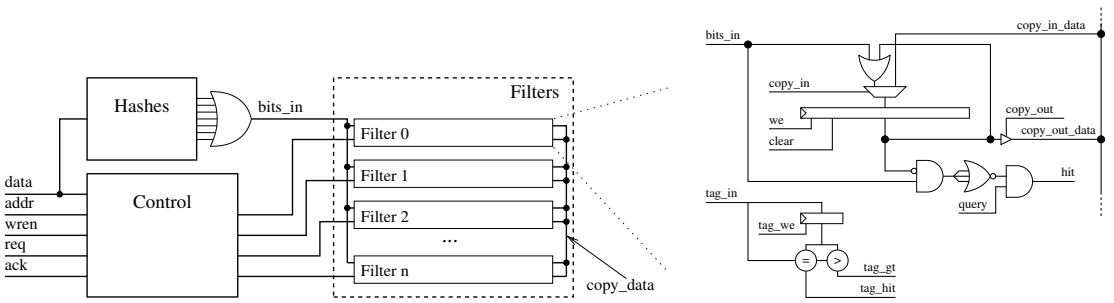


**Figure 3.** Logical block diagram of Bloom filters.

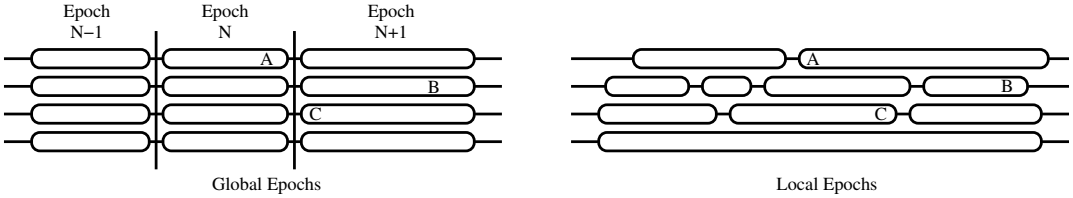


**Figure 4.** To determine the ordering of events, time is divided into epochs, either globally or locally. In the global epochs example, it is known that A comes before B and C, but not the relative ordering of B and C. In the local case, it is known that C comes before B, but not the ordering of A and B or A and C because their epochs overlap.

interface, resulting in more logic cells for the datapaths. All filters are logically 4 Kbits in size.

Software communicates with the Bloom filters using the memory subsystem, which is the fastest (both highest bandwidth and lowest latency) I/O path to and from a commodity processor core. Uncached "fire-and-forget" stores can be used to send asynchronous commands to the filters, such as a request to add an address to a transaction's read set. FARM's data stream interface provides similar functionality; however, its Barcelona processors are not able to perform true fire-and-forget stores. Instead, "write-combining" memory is used to provide a way to stream data to the FPGA with minimal impact on the running processor [27]. The Bloom filter hardware performs commands serially in the order they are received by the FPGA. The implementation is pipelined, allowing the filters to easily process all incoming commands even when the link is fully saturated.

For asynchronous responses, such as a filter match notification indicating a conflict between transactions, the filters use FARM's coherent interface to store a message in a previously agreed upon memory location, or *mailbox*[26]. The application receives notification of Bloom filter matches (i.e. conflicts) by periodically reading this mailbox. In the common case of no conflicts, this check is very cheap as it consists of a read that hits the processor's L1 cache.

Using out-of-core Bloom filters that communicate using the memory system allows us to easily perform virtualization. The software runtime maintains the pool of Bloom filters, explicitly managing the binding between software threads and hardware filters. Issues such as interrupt handling, context switching, and thread migration are thus transparent to the acceleration hardware. If the hardware were added to the processor core, these issues would become much more complex and expensive, as the core would be physically tied to a specific Bloom filter.

### 2.3 Tolerating command reordering

For many applications, like TM, that require fine-grained (frequent) communication between the processor and an accelerator, asynchronous communication is essential for performance. When using fully asynchronous communication to out-of-core devices, however, it is incorrect to assume that commands are received by the accelerator in the same order they were dispatched from the processors. Consider the following example: One processor sends a command to add an address to a transaction's read set; this command stalls in the processor's write-combining buffer. Later, a committing transaction on another processor sends notification that it is writing to that same address. This notification arrives immediately (before the preceding add to read set by the first processor) and thus the conflict is missed because the FPGA sees the commit notification and the add to the read set command in reverse order. To avoid the performance penalty of a more synchronous communication scheme (e.g. an mfence after each command), accelerators such as those in TMACC must therefore reason about possible command reorderings.

To address this serious issue, we present epoch-based reasoning and apply the technique to our Bloom filter accelerators. In this scheme, we split time into variable sized epochs, either locally determined (local epochs) or globally agreed upon (global epochs). Global epochs can be implemented using a single shared counter variable that is atomically incremented when a thread wants to move the system into a new epoch. To inform the accelerator of the epoch in which a command is executed, the epoch counter, which will usually be in the L1 cache, is read and included in the command. The accelerator then compares the epochs of commands to determine a coarse ordering, with the atomic increment providing the necessary happens-before relationship between threads. The accelerator cannot determine the ordering of commands with the same epoch number, since it may only assume the commands were fired at some point during the epoch (see Figure 4). Thus, the granularity of epoch changes determines the granularity at which the accelerator is able to determine ordering.

The potentially high overhead of maintaining a single global counter can be eliminated by using epochs local to each thread. When a thread wants to move into a new local epoch, it sends a command to the accelerator to inform it of an epoch change and performs a memory fence to ensure any command tagged with the new epoch number happens after the accelerator sees the epoch change. The epoch change command can often be included in an

| Function | Description |
|---|---|
| `HW_AddToReadSet(tid, reference, epoch)` | Asynchronously adds `reference` to `tid`'s read set and enables notification for any write that could possibly make this read inconsistent. Queries each write set that has an epoch number less than or equal to `epoch` for `reference`, triggering a conflict in `tid` if a match is found or if `epoch` is less than the epoch of the oldest write set. |
| `HW_WriteNotification(tid, reference, epoch)` | Asynchronously queries all reads sets, except `tid`'s, and triggers a conflict in any transaction whose read set includes `reference`. Adds `reference` to the write set for epoch `epoch`, clearing and replacing an old epoch's write set if necessary. |
| `HW_AskToCommit(tid)` | Synchronously processes all outstanding commands and returns the conflict status of `tid`. |

**Table 1.** TMACC hardware functions used by TMACC-GE.

**Algorithm 1** Pseudocode for the TMACC-GE runtime.

```
procedure WriteBarrier(tid, ptr, val)
    AddToWritebuffer(tid.wb, val)
procedure ReadBarrier(tid, ptr)
    HW_AddToReadSet(tid, ptr, global_epoch)
    if WritebufferContains(tid.wb, ptr) then
        return WritebufferLookup(tid.wb, ptr)
    WaitForFreeLock(ptr)
    Return *ptr
procedure Commit(wb)
    AcquireLocksForWB(wb)
    epoch = global_epoch
    if (violation_mailbox[wb.tid] == true) then return failure
    for entry in wb do
        HW_WriteNotification(wb.tid, entry.address, epoch)
    violated = HW_AskToCommit(wb.tid)                ▷ Synchronous
    if violated then ReleaseLocks(); return failure
    for entry in wb do *(entry.address) = entry.specData
    AtomicIncrement(global_epoch)
    ReleaseLocks()
    return success
```

existing synchronous command with low cost. While this scheme has less overhead, it leaves the accelerator with less information about the ordering of events. Like the global scheme, the accelerator may only assume the command was fired at some point during the epoch; therefore the relative ordering of commands from different threads can only be determined if their epochs do not overlap, as illustrated in Figure 4.

## 3. Algorithm Details

We propose two different transactional memory algorithms in this paper: one using global epochs (TMACC-GE) and one using local epochs (TMACC-LE). In both of these schemes, a filter match represents a conflict that requires a transaction to abort, and a pre-set mailbox is used to notify the STM runtime. Both schemes provide privatization safety. Publication safety could be provided by constraining the commit order as in an STM; we don't expect TMACC to make this either easier or harder.

When using Bloom filters to perform conflict detection, an important decision is what logical keys are put into the Bloom filter to designate a shared variable. This decision determines the granularity at which conflicts are detected. In our systems, we simply use the virtual address of the shared variable as the key (later referred to as a reference). For structures and arrays, each unique word is a separate shared variable. An object identifier or something similar could be also be used as a reference.

### 3.1 Global Epochs

In the global epoch scheme, the Bloom filters are split into two banks. One bank maintains the read set for each active transaction in the system. Each read set holds the references read during the execution of the associated transaction. The other bank contains filters which hold the write set for a given epoch; the write set is composed of writes that were performed by any transaction during that epoch. The Bloom filter tags are used to determine which Bloom filter in this bank corresponds to what epoch. When the filters receive a $HW\_AddToReadSet$, the reference is added to the transaction's read set and checked against the write set for the given and all previous epochs. A conflict is signalled on any match, thus ensuring a match against any write that could have occurred prior to the read. When the filters receive a $HW\_WriteNotification$, the reference is added to the given epoch's write set and checked against each transaction's read set, ensuring that any read that could possibly come after, or has come after, the associated write will signal a conflict. In the case that there is not a filter currently associated with the epoch of a $HW\_WriteNotification$, and the epoch is greater than the oldest epoch for which a filter exists (i.e. this is a new epoch), the write set filter of the oldest epoch is cleared and replaced with a new write set containing the address to be added (and tagged with the new epoch number). If no filter exists for the epoch in either a $HW\_WriteNotification$ or a $HW\_AddToReadSet$, and the epoch is older than the oldest epoch for which a write set exists, then the command comes from an epoch that is too old to have a filter and conservatively triggers a conflict. Since the ordering of reads and writes within the same epoch cannot be determined, this scheme has the effect of logically moving all reads to the end of the epoch in which they are performed and all writes to the beginning. These operations are summarized in Table 1.

Algorithm 1 gives high level pseudo-code for the algorithm used by the TMACC-GE software runtime. Each read is instrumented to inform the Bloom filters of the reference being read. Since the command is asynchronous, the only per read barrier cost of doing conflict detection is the cost of firing off the command to the FPGA. To commit the transaction, the runtime first acquires locks for each address in its write buffer, using a similar low-overhead striped locking technique as TL2 [15]. To ensure that all of its writes are assigned to the same epoch, a local copy of the global epoch counter is stored and used to inform the hardware of all the references that are about to be committed. Locks are necessary to ensure that any readers of partially committed state perform the read in the same epoch as the commit. Without them, the epoch could be incremented and a read of a partial commit performed in the following epoch. This read would (incorrectly) not be flagged as a conflict. Once all of the locks are obtained, the running transaction must synchronize with the filters to ensure that it has not been violated up until the point the filters perform the $HW\_AskToCommit$ operation. If the transaction read a value that had been committed in the current or any previous epoch, either the $HW\_WriteNotification$ would have matched on the read set and triggered a conflict, or the $HW\_AddToReadSet$ would have matched against one of the epoch's write sets. Therefore, when the $HW\_AskToCommit$ is performed on the FPGA, the transaction's

| Function | Description |
|---|---|
| `HW_AddToReadSet(tid, reference)` | Asynchronously adds `reference` to `tid`'s read set, and enables notification for any write that could possibly make this read inconsistent. Queries `tid`'s missed set and the write set for every other transaction for `reference`, triggering a conflict in `tid` on a match. |
| `HW_WriteNotification(tid, reference, epoch)` | Asynchronously queries all reads sets except `tid`'s, triggering a conflict in transactions whose read set includes `reference`. Adds `reference` to `tid`'s read set and to `epoch`'s write set. |
| `HW_ClearMissedSet(tid)` | Asynchronously clears `tid`'s missed set, moving this transaction to a new local epoch. |
| `HW_ClearWriteSet(tid)` | Asynchronously copies the content of `tid`'s write set into every other transaction's missed set, then clears the write set. |
| `HW_AskToCommit(tid)` | Synchronously processes all outstanding commands and returns the conflict status of `tid`. Clears `tid`'s read and missed set in preparation for a new transaction. |

**Table 2.** TMACC hardware functions used by TMACC-LE.

**Algorithm 2** Pseudocode for the TMACC-LE runtime.

```
procedure WRITEBARRIER(tid, ptr, val)
    AddToWritebuffer(tid.wb, val)
procedure READBARRIER(tid, ptr)
    HW_AddToReadSet(tid, ptr)
    if WritebufferContains(tid.wb, ptr) then
        return WritebufferLookup(tid.wb, ptr)
    if TimeForNewLocalEpoch() then
        HW_ClearMissedSet(tid); mfence
    Return *ptr
procedure COMMIT(wb)
    for entry in wb do
        HW_WriteNotification(wb.tid, entry.address)
    violated = HW_AskToCommit(wb.tid)                ▷ Synchronous
    if violated then return failure
    for entry in wb do *(entry.address) = entry.specData
    HW_ClearWriteSet(wb.tid)
    return success
```

read set is coherent and consistent if no conflict has been seen by the FPGA. The transaction is then placed in the global ordering of transactions on the system and allowed to apply its write buffer to memory. Once the write buffer has been applied, the transaction atomically increments the global epoch counter so that any thread that reads the newly committed value will read it in the new epoch and not be violated. It then releases the locks and returns.

It is important to note that the locks used in TMACC-GE are simple mutex locks only used to ensure the atomicity of a commit, not the versioned locks used for conflict detection in TL2. TMACC-GE can thus use coarser grain locking than TL2. We found that $2^{16}$ locks is idle for TMACC-GE, while TL2 performs best with $2^{20}$.

### 3.2 Local Epochs

To perform conflict detection using local epochs, each transaction is assigned three filters: a read set, a write set, and a missed set. As before, the read set maintains the references read during the transaction. The write set holds references that are currently being committed by a transaction, and the missed set holds references committed by any other transaction during the local epoch. When a filter receives a $HW\_AddToReadSet$, the reference is checked against all other transactions' write sets and the reading transaction's missed set, ensuring that any write that could have occurred before the associated read (i.e. in the current local epoch) will trigger a conflict. A $HW\_WriteNotification$ causes the reference to be added to the transaction's write set and checked against all other transactions' read sets, ensuring a conflict will be triggered for any read that could have potentially seen the result of the corresponding write. The written reference is also added to the transaction's read set, preventing write-write conflicts which cause a race during write buffer application. Finally, $HW\_ClearWriteSet$ first copies (merges) the write set into all other missed sets and then clears the write set. This allows each transaction to independently decide when it no longer needs to consider missed writes as potentially conflicting. The transaction does this with $HW\_ClearMissedSet$ which clears its own missed set, effectively moving it into a new local epoch. $HW\_WriteNotification$ could add references directly to the other transaction's missed sets, but having the intermediate step of using the local write set allows the transaction to abort a commit without polluting the other missed sets.

Algorithm 2 gives high level pseudo-code for the algorithm used by the TMACC-LE software runtime. The main difference in this software runtime, as compared to TMACC-GE, is the absence of locks during commit. Locks are not needed when using local epochs because the missed sets cause all of the writes performed during a commit to be logically moved to the beginning of an epoch defined locally for each transaction, not globally. Therefore, each transaction individually ensures that any of its own reads of a partial commit will signal a conflict, an effort which won't be frustrated by the update of a global epoch outside of the transaction's control.

In the local epoch scheme, an epoch is implicitly defined by what writes are contained in the transaction's missed set filter; thus no explicit local epoch counter is needed. In addition to firing a $HW\_AddToReadSet$ and locating the correct version of the datum, read barriers may choose to begin a new local epoch by sending a $HW\_ClearMissedSet$ command. A memory fence is then used to ensure that any subsequent read (and its corresponding $HW\_AddToReadSet$) must wait until the $HW\_ClearMissedSet$ is complete and a new missed set has begun to collect writes performed in the new epoch. This eliminates the possibility that a conflicting read is performed during a local epoch update and the conflict lost. Periodically incrementing the local epoch is not necessary for correct operation but reduces the number of false conflicts and is especially important in applications using long-running transactions.

## 4. TMACC Performance Evaluation

In this section, we present the performance and analysis of the TMACC-GE and -LE architectures implemented on FARM. We present the performance results in two parts. First, we present results from a microbenchmark that is used to explore the full range of TM application parameters. Second, we present results of full applications from the STAMP benchmark suite [6]. We show where the STAMP applications fit into the design space as characterized by the microbenchmark parameters and how these parameters explain the performance results. Finally, we project the performance of an ASIC TMACC implementation.

**Algorithm 3** Pseudocode for microbenchmark.

```
static int gArray1[A1];
static int gArray2[A2];
procedure uBENCH(A1, A2, R, W, T, C, N, tid)
    prob_rd = R/(R+W);
    for t = 1 to T do
        TM_BEGIN();
        for j = 1 to (R + W) do
            do_read = random(0,1) ≤ prob_rd ? true : false;
            addr1 = random(0,A1/N) + tid*A1/N;
                                    ▷ addr1 does not conflict with others
            if do_read then
                TM_READ(gArray1[addr1]);
            else
                TM_WRITE(gArray1[addr1]);
            if C == true then
                addr2 = random(0,A2);
                                    ▷ addr2 possibly conflicts with others
                if do_read then
                    TM_READ(gArray2[addr2]);
                else
                    TM_WRITE(gArray2[addr2]);
        TM_END();
```

### 4.1 Microbenchmark Analysis

In order to characterize the performance of TMACC-LE and TMACC-GE, we used an early version of EigenBench [21] which is a simple synthetic microbenchmark specially devised for TM system evaluation. This microbenchmark has two major advantages over a benchmark suite composed of complex applications. First, transactional memory is a complex system whose performance is affected by several application parameters. The microbenchmark makes it simple to isolate the impact of each parameter, independently from the others. Second, a microbenchmark allows us to get a theoretical upper bound on the best possible performance given a set of parameters. We arrive at this bound by simply executing a multi-threaded trial run without the protection of transactional memory or locking. Doing this with a real application would almost certainly produce incorrect results. We call this unattainably good performance the "unprotected" version.

Algorithm 3 shows the pseudocode for the microbenchmark. The algorithm, at the core, is nothing more than multiple threads executing a random set of array accesses. Several parameters are necessary: $A1$ and $A2$ are the sizes of two arrays, the first a partitioned array for non-conflicting accesses, the second a smaller shared array for conflicting accesses; $R$ and $W$ are, respectively, the average number of reads and writes, per transaction; $T$ is the number of transactions executed per thread; $N$ is the number of threads; and $C$ is a flag determining whether or not conflicting accesses should be performed. Note that if $C$ is unset, there should be no violations since every thread only accesses its partition of the array. If $C$ is set, then the shared $A2$ array is accessed in addition to the normal accesses to $A1$, decoupling the working set size and the read/write ratio from the probability of violation.

We now use the microbenchmark to evaluate the performance of our two TMACC systems across several different variables. Table 3 shows the parameter sets used in the study, and the performance results are displayed in Figure 5. All graphs in this section show both speedup relative to sequential execution with no locking or transactional overhead (solid lines) and the percentage of started transactions that were violated (dotted lines). In all graphs except for (e), speedup is shown for 8 threads.

Throughout our analysis, the baseline STM for comparison is TL2 [15], which is generally regarded as a high-performing, modern STM implementation that is largely immune to performance pathologies. We use the basic GV4 versioned locks in TL2, the default in the STAMP distribution [37]. We use TL2 because its algorithms for version management and conflict detection are the closest match to the TMACC algorithms, allowing for the best indication of the speedup achieved using the hardware. SwissTM [16] is the highest performing STM of which the authors are aware and provides 1.1 to 1.3 times the performance of TL2 on the STAMP applications presented here. We also present the best possible performance using the aforementioned "unprotected" method as an upper bound. Note that this is truly an upper bound and usually unattainable because it will produce incorrect results in the face of any conflicts. Throughout the analyses of results, TMACC-GE and TMACC-LE represent the schemes described in Section 3.

Graph (a) shows the impact of working set size on TM systems. The prominent knee in the performance of each system corresponds to the working set size outgrowing the on-chip cache. Below the knee, where all user data and TM metadata fit on-chip, TL2 is spared from off-chip accesses and outperforms the TMACC systems which must still pay the costly round trip communication with the FPGA. This effect would be heavily mitigated with faster (or closer) hardware, and it is certainly rare for the working set of real parallel workloads to fit in the on-chip cache.

Above the knee, we observe that both TMACC-GE and TMACC-LE significantly outperform TL2, around 1.35x and 1.75x respectively, approaching the upper bound of 1.95x. In this region, TL2's performance suffers because its extra metadata causes significant cache pressure. Specifically, TL2 relies on its metadata for conflict detection, so its metadata grows proportionally to a transaction's read set. TMACC-GE, on the other hand, uses metadata only for commit, so its metadata grows with a transaction's write set, which is almost always smaller than its read set.

Graph (b) explores the impact of transaction size on speedup and violation rate. In this graph, we see a well-defined difference in speedup among the systems. In the flat region in the middle, the speedup of each system is nearly identical to the speedup of large working sets in graph (a). For small transactions, TMACC-GE's speedup diminishes because the relative cost of the FPGA round trip latency and global epoch management grows as transaction size decreases. We will take a closer look at short transactions in graph (e). For large transactions, the performance of TMACC-LE drops because the lack of ordering information in local epochs causes the missed sets to become polluted and emit more false positives. This is one case where global epochs are preferred over local epochs.

Graph (c) depicts the impact of varying the probability of violations by turning on $C$ and varying the size of $A2$ in our microbenchmark. Note that the graph uses semi-log axes. With a small $A2$, there are many violations and transactional retries dominate performance, making the conflict detection overhead less important. As $A2$ grows, contention decreases and the conflict detection overhead becomes more important, explaining the expanding performance gap between TMACC-LE, with its low-overhead conflict detection, and the others.

Graph (d) explores the impact of write set size, and again it is not surprising that the false positive rate of TMACC-LE becomes non-trivial due to the inherent pessimism in the local epoch scheme. However, these false positives are not enough to outweigh the performance advantage of low-overhead conflict detection.

Interestingly, TMACC-GE also shows diminishing speedup as write-set size increases. On closer inspection, we found that this degradation is due to the cache line migration of locks between the two CPU sockets during commit. As explained in Section 3.1, TL2 uses more locks than TMACC-GE so it is not as sensitive to this issue. Increasing the number of locks used by TMACC-GE diminishes the effect, but reduces overall performance. Having the FPGA participate in the coherence fabric significantly increases the

| | parameter set label | A1*sizeof(int) | A2 | R | W | C | N |
|---|---|---|---|---|---|---|---|
| (a) | working-set size | 0.5 $\sim$ 64 (MB) | - | 80 | 4 | $false$ | 8 |
| (b) | transaction size | 64 (MB) | - | 10 $\sim$ 400 | $max(1, R * 0.05)$ | $false$ | 8 |
| (c) | true conflicts | 64 (MB) | 256 $\sim$ 16,384 | 40 | 2 | $true$ | 8 |
| (d) | write-set sizes | 64 (MB) | - | 80 | 1 $\sim$ 128 | $false$ | 8 |
| (e) | # of threads (med-sized TX) | 64 (MB) | - | 80 | 4 | $false$ | 1 $\sim$ 8 |
| | # of threads (small-sized TX) | 64 (MB) | - | 4 | 1 | $false$ | 1 $\sim$ 8 |

**Table 3.** Parameter sets used in the microbenchmark evaluation. The labels here match those used in Figure 5.

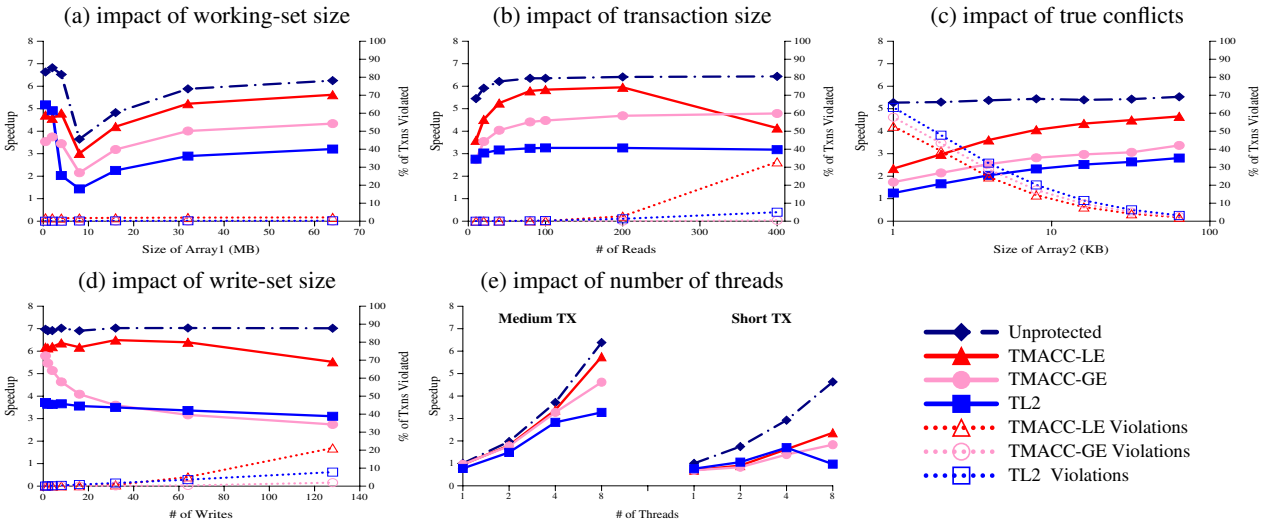


**Figure 5.** Microbenchmark performance for various parameter sets. Speedup is shown for 8 threads (except in (e))

last level cache miss penalty for all processors. This is a prominent factor in the TMACC-GE results, and experiments in Section 4.3 show that moving to an ASIC implementation would largely eliminate the performance degradation of TMACC-GE seen here.

Graph (e) examines the impact of number of threads using both medium-sized transactions and small-sized transactions. Overall, the systems show worse performance for small-sized transactions because they all pay a constant overhead per transaction, which is not easily amortized by short transactions. With the long communication delay to the FPGA, TMACC-GE and TMACC-LE are unable to achieve better performance than TL2 for short transactions running on 2 or 4 threads. While the FARM system limits us to 8 threads, scalability to many more threads can be achieved using multiple FPGAs. This scheme would require communication between the FPGAs and is left for future work.

The dramatic drop in TL2 performance for short transactions at 8 threads is the result of moving from a single chip to two chips and the large miss penalty described above. Taking the FPGA out of the system eliminates this drop in performance as shown in Section 4.3. We note that this poor TL2 performance on FARM is only present when transactions are very short.

To summarize, we see that TMACC provides significant acceleration of transactional memory except when transactions are too short to amortize the extra overhead imposed by communicating with the Bloom filters. We also find that in the case of TM acceleration, global epochs only perform better than local epochs when a large number of shared reads and writes are performed in a relatively short running transaction. In this case, the lack of ordering information is a larger factor in system performance.

### 4.2 Performance Evaluation using STAMP

In this section, we evaluate the performance of TMACC on FARM using STAMP[6], a transactional memory benchmark suite composed of several applications which vary in data set size, memory access patterns, and size of transactions. Intruder, bayes, and yada from the STAMP suite did not execute correctly in the 64-bit environment of FARM (even using TL2) due to bugs in the STAMP code and have been omitted from the study. Bayes's and yada's long transactions with a high violation rate are similar to those in labyrinth, and intruder's short transactions are similar to those in kmeans-high. Thus, the absence of these apps does not significantly reduce the coverage of the suite. Table 4 summarizes the input parameters and the key characteristics of each application. Cycles per transaction were measured during single-threaded execution with no read and write barriers. We can roughly group the applications into two sets by transaction size: vacation, genome, and labyrinth have larger transactions while ssca2 and kmeans use smaller transactions. Kmeans has large amounts of spatial locality in its data access and thus uses fewer cycles per transaction despite having more shared reads and writes.

For this analysis, we include RingSTM [36]. This STM system uses a similar approach to accelerating transactional barriers as TMACC, but the Bloom filters are implemented in software rather than hardware. Like TMACC but unlike TL2, RingSTM provides privatization safety. Our RingSTM implementation is based on the latest open-source version [35] and uses the single-writer algorithm. To provide a better comparison to TL2 and our TMACC variants, this implementation uses the write buffer implementation from TL2 instead of the hash table typically used in RingSTM. In our experiments, the ring is configured to have 1024 entries, where each entry is a 1024-bit filter.

| Name | Input parameters | RD/tx | WR/tx | CPU cycles/tx | Memory usage (MB) | Conflicts |
|---|---|---|---|---|---|---|
| vacation-low | n2 q90 u98 r1048576 t4194304 | 220.9 | 5.5 | 37740 | 573 | very low |
| vacation-high | n4 q60 u90 r1048576 t4194304 | 302.14 | 8.5 | 37642 | 573 | low |
| genome | g16384 s64 n16777216 | 55.8 | 1.9 | 48836 | 1932 | low |
| kmeans-low | m256 n256 65536-d32-c16.txt | 25 | 25 | 690 | 16 | high |
| kmeans-high | m40 n40 65536-d32-c16.txt | 25 | 25 | 680 | 16 | low |
| ssca2 | s20 i1.0 u1.0 l3 p3 | 1 | 2 | 2360 | 1320 | very low |
| labyrinth | x512-y512-z7-n512.txt | 180 | 177 | $6.1 * 10^9$ | 32 | high |

**Table 4.** STAMP benchmark input parameters and application characteristics.

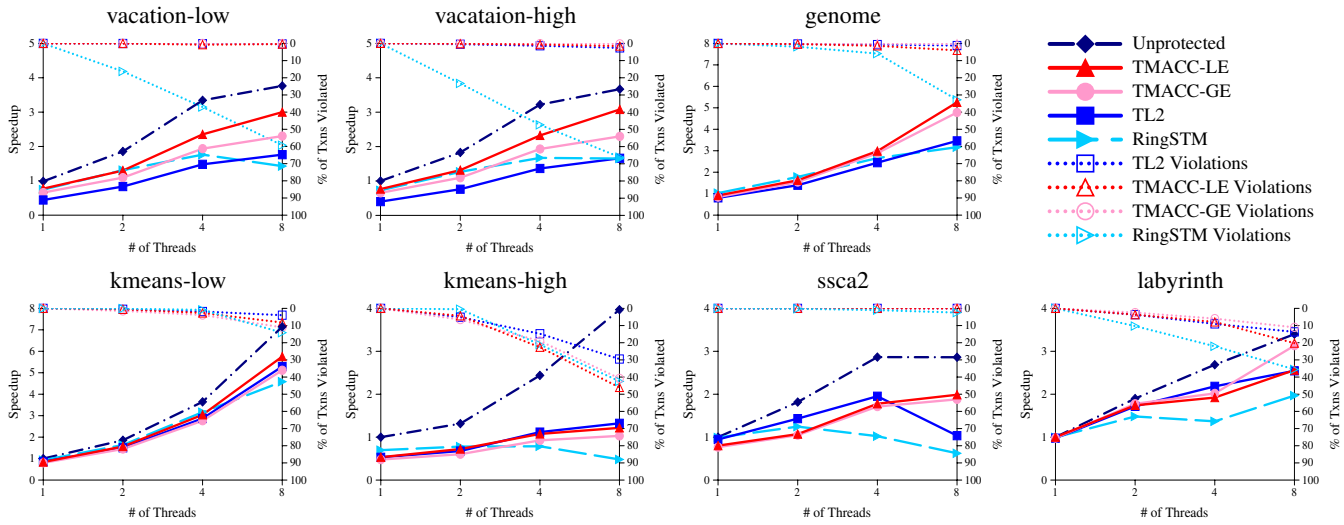


**Figure 6.** STAMP performance on the FARM prototype.

Figure 6 shows performance results from executing the STAMP applications on the FARM prototype. In this graph, we present speedups of 1, 2, 4 and 8 cores and the percentage of started transactions that were violated. At first glance, we see that the general trends we saw in the microbenchmark are present in the STAMP applications; TMACC performs well with large transactions but is unable to provide acceleration to small transactions. We also provide the unprotected execution time, using the same method we used in Section 4.1. As before, the result of such execution is incorrect and serves as a strict upper bound. As expected, not all applications were able to run unprotected; some would crash or fall into infinite loops.

For vacation-high, vacation-low, and genome, the common characteristics are a relatively large number of reads per transaction, small number of writes per transaction, and small number of conflicts. See Table 4 for exact values. Commit overhead is low due to the small write set and minimal time wasted retrying transactions because of the small number of conflicts. Also, constant overheads such as register checkpointing are amortized over the long running length. Thus, in these large-transaction applications, the numerous reads make the barrier overhead the dominant factor influencing performance of the TM system. We saw this effect in Figure 5.(b). This graph uses a microbenchmark parameter set which corresponds to the characteristics of these applications, and we see a very similar spread in performance results for the large-transaction STAMP applications. Performance gain with respect to TL2 for these applications averages 1.36x for TMACC-GE and 1.69x for TMACC-LE. Unprotected execution provides an average speedup of 2.18x. Note that for vacation-high running on TMACC-LE, while the number of reads is about 300, the drop shown in Figure 5.(b) does not happen because vacation-high does not have as many writes as the microbenchmark used in that graph.

The TMACC systems perform similar to RingSTM for low thread counts but do not suffer from the drop in performance at higher thread counts like RingSTM. The drop in performance at higher thread counts seen in RingSTM arises because it is unable to quickly check individual reads against write set filters like TMACC is able to do. It instead checks read set filters against write set filters, and this filter to filter comparison has a much higher probability of false positives, leading to very high false conflict rates and significantly degrading performance.

Kmeans-low features a relatively small number of reads, large number of writes, and small number of conflicts. From Figure 5.(b), we can expect that a small number of reads will diminish the performance gap between TL2 and TMACC. We also see in Figure 5.(d) that the large number of writes will further diminish TMACC-GE's performance. The combined effect explains what we see for kmeans-low in Figure 6 where for 8 threads TMACC-LE shows a 9% acceleration over TL2 but TMACC-GE is 5% slower.

Even though kmeans-high has very similar characteristics to kmeans-low except for the number of conflicts, the large number of violations in kmeans-high overshadows any other effects and limits the speedup of all three systems to a mere 1.3x with 8 threads. This situation is captured in Figure 5.(c) where the performance of the three systems converges as the rate of violation increases. As in kmeans-low, the small transactions make it difficult to amortize the communication overheads of TMACC and it is not able to achieve any speedup over TL2. Both TMACC systems were addi-

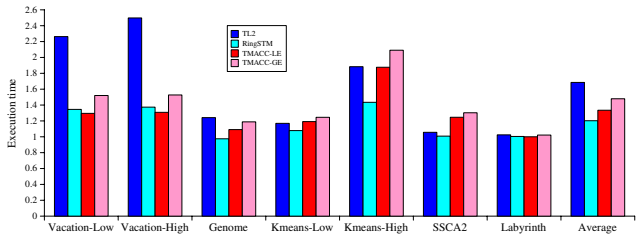


**Figure 7.** Single threaded execution time relative to sequential execution.

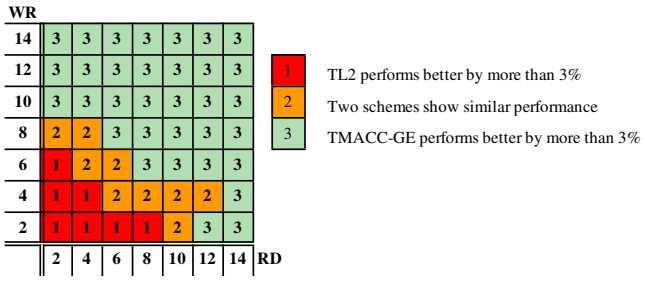


**Figure 8.** Performance comparison of TMACC-GE (ASIC) and TL2 for short transactions.

tionally undermined by an even larger number of violations than TL2, which is interesting because Figure 5.(c) shows the TMACC systems having fewer violations in the face of true conflicts. We suspect this is a result of TL2's versioned locks giving more importance to the lower bits of the address in performing conflict detection. This causes TL2 to have fewer false positives when addresses are close together, as they are in kmeans-high. The single-writer variant of RingSTM we use is not able to scale because of the large number of writes in both kmeans-low and kmeans-high, even though its violation rate is comparable to the other systems.

Like kmeans-low and kmeans-high, TMACC performance on ssca2 is bound by communication latency. The characteristics of ssca2 are well captured by the microbenchmark parameter set used to produce the short transactions graph in Figure 5.(e) which mirrors the ssca2 speedup graph in Figure 6. Refer to the discussion of graph (e) in Section 4.1 for an explanation of the results. RingSTM violates 2.5% of transactions when running 8 threads while the others violate less than 0.01%. ssca2 has such a large number of transactions that even a 2.5% violation rate adds significant overhead.

Labyrinth is a special case. As seen in Table 4, this application has a very large number of computational cycles inside each transaction. The execution time is therefore decided by non-deterministic execution paths and the number of violated transactions rather than TM overhead. In Figure 5.(c) we saw that, in general, TMACC-GE has fewer false positives than the other systems. So in labyrinth with 8 threads, the TMACC-GE system minimized the number of violations and performed well. For labyrinth's long-running transactions, the periodic intra-transaction increment of the TMACC-LE local epoch was especially important.

Finally, Figure 7 highlights the single thread overhead of the systems using the single threaded execution time relative to sequential execution time. We see that TMACC and RingSTM have less overhead than TL2 running vacation because of the frequent barriers. As transactions get smaller in applications like kmeans and ssca2, commit time becomes more important and the TMACC systems suffer, while RingSTM continues to do well. Note that TMACC-GE consistently has more overhead than TMACC-LE because of the extra time required to (unnecessarily) obtain the locks during commit. With few barriers and very long transactions, labyrinth has almost no overhead in any of the systems.

### 4.3 Performance Projection for TMACC ASIC

In the previous sections, we have observed a few artificial effects caused by the large cache miss penalty in the FARM system. Since both TMACC and TL2 witness performance degradation due to these issues, an interesting question is whether the conclusions drawn thus far would still be valid in a system absent of these latency anomalies, such as an off-chip ASIC or part of the *uncore* on a chip. The acceleration hardware as presented does not require a high clock frequency and would occupy a small silicon footprint in modern processes. Thus in this section, we modify our system to project the performance of TMACC onto the design point of an off-chip ASIC. This could be either a stand-alone chip, or part of the system's north bridge or memory controller, for example. The performance of an on-chip TM accelerator would be even better, since it has a shorter round-trip latency.

To simulate the performance of an ASIC TMACC implementation, we first detach the FPGA from the system, eliminating the FPGA-induced snoop latency witnessed by all coherent nodes on every cache miss. Then, we replace FPGA-communication software routines with idle loops in which we control the number of iterations to simulate different desired communication latencies. In addition, we change the conflict detection to report a conflict randomly with a given probability. We keep all the STM overheads but simulate hardware latency. This modified system is a performance simulator; like the unprotected version it does not provide serializable execution, but can serve as good indicator of real performance.

For the projection study, we repeated the microbenchmark experiments performed in Section 4.1 using these techniques. We used the measured off-chip cache miss latency as the communication latency in our simulation, the rationale being that the ASIC is about as "far" away from the processor as DRAM. In general, we found the trends and conclusions are the same as those presented in Section 4.1 expect where we explicitly mentioned otherwise in the discussions of graphs (d) and (e) of Figure 5. We omit graphs of the results due to space constraints.

A common trend seen in all the experiments is that the performance of TMACC-GE now comes closer to the unprotected, since the ASIC design point significantly reduces the cache migration latency, and thus the overhead of global epoch management. As noted in the discussion of graph (d) in Section 4.1, the dramatic performance degradation of TMACC-GE as the write set grows disappears with the reduced cache miss penalty of an ASIC implementation. Also, the performance of TL2 with small transactions no longer drops dramatically when moving to a dual socket configuration. Both TMACC systems also performed better than before for short transactions; TMACC-LE outperforms TL2 on 8 threads by 9% now, but TMACC-GE still falls 5% short of TL2 performance.

To determine the point where TMACC-GE begins to outperform TL2, we repeated the short transaction experiment from Figure 5.(e), sweeping the number of reads and writes from 2 to 14, the result is presented as a schmoo plot in Figure 8. When there are more than 8 reads or writes, TMACC-GE is able to match the performance of TL2. When there are more than 12, there are enough accelerated barriers to compensate for the extra cost of communication, and TMACC-GE outperforms TL2. TMACC-LE outperformed TL2 for all of these points. The inability of TMACC to accelerate very small transactions suggests that TMACC would compliment a system that targets small transactions, such as a best-effort HTM that uses a processor's write buffer to store speculative data and falls back to using TMACC for larger transactions.

### 4.4 Comparison with Simulation

We now briefly contrast our experiences and results with hardware to our early exploratory work done using software simulation. Considerable effort went in to making our simulations "cycle accurate", and our performance predictions for SigTM and TL2, presented in Figure 1, roughly matched the results presented in the corresponding papers. Initial results from the actual hardware, however, were quite different from those the simulator had predicted. One main reason for the discrepancy was the difference between the simulated and actual CPUs. The simplistic CPU model used in simulation (in-order with one non-memory instruction per cycle) drastically overstated the importance of reducing the instruction count in the transactional read and write barriers. Modern processors, such as those in FARM, are much more tolerant of extra instructions in barriers, reducing the benefit of eliminating those instructions.

Another primary source of inaccuracy arose from the fact that our simulated interconnect did not model variable latency and command reordering. The presence of these in a real system led us to develop the global and local epoch schemes presented in this paper and thus significantly impacted the performance of the system. In addition, our simulator assumed the processors were capable of performing true "fire-and-forget" stores with weak consistency without affecting the execution of the core. We therefore did not model the write combining buffer and its effect on system performance. In addition, smaller data sets used to run simulation in a reasonable time frame affected the system performance very differently than a real workload, in terms of bandwidth consumption, caching effects and TLB pressure.

Even though we could have performed a more accurate simulation and we eventually approached our desired performance using a modified design, we believe our experiences provide a strong example of the importance of building actual hardware prototypes. Although developing and verifying hardware requires increased time and effort when compared with using a simulator, hardware is essential to accurately gauge the performance of proposed architectural improvements and to bring out the many issues one might encounter in actually implementing the idea. Having a hardware implementation is also a strong evidence of the correctness and validity of a system.

## 5. Conclusion

In conclusion, we have presented an architecture, TMACC, for accelerating STM without modifying the processor cores. We constructed a complete hardware implementation of TMACC using a commodity SMP system and FPGA logic. In addition, two novel algorithms which use the TMACC hardware for conflict detection were presented and analyzed. Using the STAMP benchmark suite and a microbenchmark to quantify and analyze the performance of a TMACC accelerated STM, we showed that TMACC provides significant performance benefits. TMACC outperforms a plain STM (TL2) by an average of 69% in applications using moderate-length transactions, showing maximum speedup within 8% of an upper bound on TM acceleration. TMACC provides this performance improvement even in the face of the high communication latency between TMACC and the CPU cores. Overall, this paper demonstrates that it is possible to accelerate TM with an out-of-core accelerator and mitigate the impact of fine-grained communication with careful design.

## Acknowledgments

This work was funded by DARPA contract, Oracle order 630003198; DOE contract, Sandia order 942017; Army contract AHPCRC W911NF-07-2-0027-1; and the Stanford PPL affiliates program, Pervasive Parallelism Lab: NVIDIA, Oracle/Sun, AMD, Intel, and NEC. We also thank A&D Technology Inc. for their help with the Procyon system and Michael Spear for providing a custom RingSTM implementation for our use.

## References

[1] A.-R. Adl-Tabatabai, B. Lewis, V. Menon, B. R. Murphy, B. Saha, and T. Shpeisman. Compiler and runtime support for efficient software transactional memory. In *PLDI '06: ACM SIGPLAN Conference on Programming Language Design and Implementation*, 2006.

[2] W. Baek, C. Cao Minh, M. Trautmann, C. Kozyrakis, and K. Olukotun. The OpenTM transactional application programming interface. In *PACT '07: 16th Internation Conference on Parallel Architecture and Compilation Techniques*, 2007.

[3] B. Bloom. Space/time trade-offs in hash coding with allowable errors. *Communications of ACM*, 1970.

[4] C. Blundell, J. Devietti, E. C. Lewis, and M. M. K. Martin. Making the fast case common and the uncommon case simple in unbounded transactional memory. In *ISCA '07: 34th International Symposium on Computer Architecture*, 2007.

[5] J. Bobba, N. Goyal, M. Hill, M. Swift, and D. Wood. Tokentm: Efficient execution of large transactions with hardware transactional memory. In *ISCA '08: 35th International Symposium on Computer Architecture*, 2008.

[6] C. Cao Minh, J. Chung, C. Kozyrakis, and K. Olukotun. STAMP: Stanford transactional applications for multi-processing. In *IISWC '08: Proc. The IEEE International Symposium on Workload Characterization*, 2008.

[7] C. Cao Minh, M. Trautmann, J. Chung, A. McDonald, N. Bronson, J. Casper, C. Kozyrakis, and K. Olukotun. An effective hybrid transactional memory system with strong isolation guarantees. In *ISCA '07: 34th International Symposium on Computer Architecture*, 2007.

[8] J. L. Carter and M. N. Wegman. Universal classes of hash functions. *Journal of Computer and System Sciences*, 18(2), 1979.

[9] C. Cascaval, C. Blundell, M. Michael, H. W. Cain, P. Wu, S. Chiras, and S. Chatterjee. Software transactional memory: Why is it only a research toy? *Queue*, 6(5), 2008.

[10] L. Ceze, J. Tuck, P. Montesinos, and J. Torrellas. BulkSC: bulk enforcement of sequential consistency. In *ISCA '07: 34th International Symposium on Computer architecture*, 2007.

[11] H. Chafi, J. Casper, B. D. Carlstrom, A. McDonald, C. Cao Minh, W. Baek, C. Kozyrakis, and K. Olukotun. A scalable, non-blocking approach to transactional memory. In *HPCA '07: 13th International Symposium on High Performance Computer Architecture*, 2007.

[12] S. Chaudhry, R. Cypher, M. Ekman, M. Karlsson, A. Landin, S. Yip, H. Zeffer, and M. Tremblay. Simultaneous speculative threading: a novel pipeline architecture implemented in sun's rock processor. In *ISCA '09: 36th Intl. Symposium on Computer Architecture*, 2009.

[13] L. Dalessandro, M. F. Spear, and M. L. Scott. NOrec: streamlining STM by abolishing ownership records. In *PPoPP '10: 15th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPoPP '10, 2010.

[14] P. Damron, A. Fedorova, Y. Lev, V. Luchangco, M. Moir, and D. Nussbaum. Hybrid transactional memory. In *ASPLOS '06: 12th Internation Conference on Architectural Support for Programming Languages and Operating Systems*, October 2006.

[15] D. Dice, O. Shalev, and N. Shavit. Transactional locking II. In *DISC '06: 20th Internation Symposium on Distributed Computing*, 2006.

[16] A. Dragojević, R. Guerraoui, and M. Kapalka. Stretching transactional memory. In *PLDI '09: ACM SIGPLAN Conference on Programming Language Design and Implementation*, 2009.

[17] 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 *ISCA '04: 31st International Symposium on Computer Architecture*, 2004.

[18] T. Harris and K. Fraser. Language support for lightweight transactions. In *OOPSLA '03: 18th ACM SIGPLAN Conference on Object-oriented Programing, Systems, Languages, and Applications*, 2003.

[19] M. Herlihy and J. E. B. Moss. Transactional memory: Architectural support for lock-free data structures. In *ISCA '93: 20th International Symposium on Computer Architecture*, 1993.

[20] O. S. Hofmann, C. J. Rossbach, and E. Witchel. Maximum benefit from a minimal HTM. In *ASPLOS '09: 14th International Conference on Architectural Support for Programming Languages and Operating Systems*, 2009.

[21] S. Hong, T. Oguntebi, J. Casper, N. Bronson, C. Kozyrakis, and K. Olukotun. Eigenbench: A simple exploration tool for orthogonal tm characteristics. In *IISWC '10: International Symposium on Workload Characterization*, 2010.

[22] S. Kumar, M. Chu, C. J. Hughes, P. Kundu, and A. Nguyen. Hybrid transactional memory. In *PPoPP '06: 11th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, 2006.

[23] J. Larus and R. Rajwar. *Transactional Memory*. Morgan Claypool Synthesis Series, 2006.

[24] M. Lupon, G. Magklis, and A. González. FASTM: A log-based hardware transactional memory with fast abort recovery. In *PACT '09: 18th International Conference on Parallel Architecture and Compilation Techniques*, 2009.

[25] V. J. Marathe, W. N. Scherer III, and M. L. Scott. Adaptive Software Transactional Memory. In *DISC '05: 19th International Symposium on Distributed Computing*, 2005.

[26] S. S. Mukherjee, B. Falsafi, M. D. Hill, and D. A. Wood. Coherent network interfaces for fine-grain communication. In *ISCA '96: 23rd International Symposium on Computer Architecture*, 1996.

[27] T. Oguntebi, S. Hong, J. Casper, N. Bronson, C. Kozyrakis, and K. Olukotun. FARM: A prototyping environment for tightly-coupled, heterogeneous architectures. In *FCCM '10: 18th Symposium on Field-Programmable Custom Computing Machines*, 2010.

[28] M. Olszewski, J. Cutler, and J. G. Steffan. JudoSTM: A dynamic binary-rewriting approach to software transactional memory. In *PACT '07: 16th International Conference on Parallel Architecture and Compilation Techniques*.

[29] H. E. Ramadan, C. J. Rossbach, D. E. Porter, O. S. Hofmann, A. Bhandari, and E. Witchel. Metatm/txlinux: transactional memory for an operating system. *SIGARCH Computer Architecture News*, 35(2), 2007.

[30] B. Saha, A. Adl-Tabatabai, and Q. Jacobson. Architectural support for software transactional memory. In *MICRO '06: International Symposium on Microarchitecture*, 2006.

[31] B. Saha, A.-R. Adl-Tabatabai, R. L. Hudson, C. Cao Minh, and B. Hertzberg. McRT–STM: A high performance software transactional memory system for a multi-core runtime. In *PPoPP '06: 11th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, 2006.

[32] T. Shpeisman, V. Menon, A.-R. Adl-Tabatabai, S. Balensiefer, D. Grossman, R. L. Hudson, K. Moore, and B. Saha. Enforcing isolation and ordering in stm. In *PLDI '07: Conference on Programming Language Design and Implementation*, 2007.

[33] A. Shriraman, S. Dwarkadas, and M. L. Scott. Flexible decoupled transactional memory support. In *ISCA '08: 35th International Symposium on Computer Architecture*, 2008.

[34] A. Shriraman, M. F. Spear, H. Hossain, V. J. Marathe, S. Dwarkadas, and M. L. Scott. An integrated hardware-software approach to flexible transactional memory. *SIGARCH Computer Architecture News*, 35, June 2007.

[35] M. F. Spear. Lightweight, robust adaptivity for software transactional memory. In *SPAA '10: 22nd ACM Symposium on Parallelism in Algorithms and Architectures*, 2010.

[36] M. F. Spear, M. M. Michael, and C. von Praun. RingSTM: scalable transactions with a single atomic instruction. In *SPAA '08: 20th Symposium on Parallelism in Algorithms and Architectures*, 2008.

[37] STAMP: Stanford transactional applications for multi-processing. `http://stamp.stanford.edu`.

[38] F. Tabba, M. Moir, J. R. Goodman, A. Hay, and C. Wang. NZTM: Nonblocking zero-indirection transactional memory. In *SPAA '09: 21st Symposium on Parallelism in Algorithms and Architectures*, 2009.

[39] C. Wang, W.-Y. Chen, Y. Wu, B. Saha, and A.-R. Adl-Tabatabai. Code generation and optimization for transactional memory constructs in an unmanaged language. In *CGO '07: International Symposium on Code Generation and Optimization*, 2007.

[40] L. Yen, J. Bobba, M. R. Marty, K. E. Moore, H. Volos, M. D. Hill, M. M. Swift, and D. A. Wood. LogTM-SE: Decoupling Hardware Transactional Memory from Caches. In *HPCA '07: 13th International Symposium on High Performance Computer Architecture*, 2007.

[41] L. Yen, S. Draper, and M. Hill. Notary: Hardware techniques to enhance signatures. In *MICRO '08: 41st International Symposium on Microarchitecture*, 2008.