# Delegation and Nesting in Best-effort Hardware Transactional Memory*

Yujie Liu
Lehigh University
yul510@cse.lehigh.edu

Stephan Diestelhorst
Advanced Micro Devices, Inc. and Dresden University of Technology
stephan.diestelhorst@amd.com

Michael Spear
Lehigh University
spear@cse.lehigh.edu

## ABSTRACT

The guiding design principle behind best-effort hardware transactional memory (BEHTM) is simplicity of implementation and verification. Only minimal modifications to the base processor architecture are allowed, thereby reducing the burden of verification and long-term support. In exchange, the hardware can support only relatively simple multiword atomic operations, and must fall back to a software run-time for any operation that exceeds the abilities of the hardware.

This paper demonstrates that BEHTM simplicity does not prohibit advanced and complex transactional behaviors. We exploit support for immediate non-transactional stores in the AMD Advanced Synchronization Facility to build a mechanism for communication among transactions. While our system allows arbitrary communication patterns, we focus on a design point where each transaction communicates with a system-wide manager thread. The API for the manager thread allows BEHTM transactions to delegate unsafe operations (such as system calls) to helper threads, and also enables the creation of nested parallel transactions. This paper also explores which forms of nesting are possible, and identifies constraints on nesting that are a consequence of how BEHTM is designed.

## Categories and Subject Descriptors

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

## General Terms

Algorithms, Design

*At Lehigh University, this work was supported by the US National Science Foundation under grant CNS-1016828; and by financial support from Google.


*SPAA'12,* June 25–27, 2012, Pittsburgh, Pennsylvania, USA.


## Keywords

Transactional Memory, Nesting, Synchronization, Allocation

## 1 Introduction

The promise of transactional memory (TM) [24] lies in its simplicity: rather than reason about mutual exclusion, lock granularity, and deadlocks, with TM programmers simply annotate code regions that must execute atomically; a run-time system ensures correctness while finding as much parallelism as possible among these atomic blocks. Unfortunately, pure software TM (STM) incurs significant overhead due to per-access conflict detection overheads. Hardware TM (HTM) requires significant modification to processor implementations, possibly including the cache coherence protocol. The resulting verification requirements are considered too onerous for a technology that has not yet proven its merit.

An emerging approach is to minimally extend the hardware to provide limited support for TM. The resulting best-effort HTM (BEHTM) executes some transactions fully in hardware, and requires a fallback to a software system in all other cases. Example BEHTM systems include the Sun Rock prototype processor [13, 18], Intel's Transactional Synchronization Extensions (TSX) [27], and the AMD Advanced Synchronization Facility proposal (ASF)[1] [15]. BEHTM systems do not modify the cache coherence protocol, and they place hard constraints on the number of locations that can be read or written by a BEHTM transaction. They also forbid certain operations (e.g., system calls) within BEHTM transactions, and permit transactions to abort and restart due to low-level hardware events (such as TLB misses or page faults).

Recent efforts have shown that it is possible to exploit BEHTM to simplify algorithm design [17], make a locking STM non-blocking with minimal overhead [20], and even accelerate a general-purpose TM system by running small transactions entirely in hardware [16, 39]. In this last category, a key challenge is to minimize the number of transactions that must fall back to the software system to complete, because the existence of such transactions causes hardware transactions to incur greater overhead.

Surprisingly, many simple operations, such as memory allocation, require BEHTM transactions to fall back to software mode. In HTM publications, true allocation often is avoided artificially by having each thread pre-allocate object pools that are large enough to accommodate all possible allocations needed by a simulation. In STM, allocation is typically managed via *ad-hoc* open nesting [26]. While the details of why allocation within a hardware transaction

[1]ASF is an experimental feature and has not been announced for any future product.

is problematic are deferred to Appendix A, it suffices to note that any allocation might perform a system call to enlarge the heap, and that non-transactionally calling a lock-based allocator is dangerous because the parent transaction risks aborting while holding a lock.

In the AMD ASF proposal, a BEHTM transaction can perform *immediate* non-transactional loads and stores. These accesses are not intended for general use: if used incorrectly, they can violate failure atomicity and cause memory corruption. However, Riegel et al. demonstrated that careful use of non-transactional accesses can significantly enhance the usefulness of BEHTM [39].

In a similar vein, we show that the existing support for non-transactional accesses in ASF can be used to enable communication among transactions. In our system, transactions communicate with nontransactional helper threads, which can perform allocation and system calls on the transaction's behalf. In addition to supporting delegation, we show that helper threads can serve as nested transaction coordinators, and thus that it is possible to implement some forms of nested parallel transactions in ASF. We also identify nesting patterns that cannot be supported by BEHTM. These include unattainable patterns of sharing between a parent and its child, as well as limits on the number of closed-nested child BEHTM transactions that can exist in any parent/children relationship.

The remainder of this paper is organized as follows: We provide more background on BEHTM operation in Section 2. In Section 3, we discuss the abstract communication channel that we implement using non-transactional accesses. Sections 4 and 5 discuss how this communication channel can be used for delegation and nesting, respectively. Section 6 discusses relationships with prior work. In Section 7, we summarize our findings and suggest directions for future research.

# 2 BEHTM Background

Early HTM proposals aimed to provide strong performance without requiring a supporting software run-time system. To this end, they modified the processor cache [23,33] and included mechanisms for supporting "unbounded" transactions [2, 8, 32, 37]. Such transactions could speculatively access (for reading and writing) more data than could fit in their L1 cache, and could survive context switches.

While appealing, these proposals required significant changes to hard-to-verify components of the processor. A more pragmatic approach, embodied by best-effort TM designs, is to minimally change the hardware such that only a limited class of transactions are supported, and then delegate the execution of all other transactions to a software run-time.

The most prominent examples of BEHTM are the Sun Rock prototype processor [13,18] and the AMD Advanced Synchronization Facility proposal (ASF) [15].[2] These systems share many characteristics. They both use a traditional invalidation cache protocol, resulting in an "attacker wins" conflict resolution policy [10], with conflicts detected at the granularity of cache lines. While this protocol allows for obstruction-free transactions, it is prone to livelock and false conflicts [10], and thus a software contention manager is recommended even for hardware transactions.

Both proposals place limits on the behavior of transactional code in terms of the absolute number of locations that can be read or written. In Rock, the L1 cache buffers transactional loads, with L1 capacity and dynamic associativity conflicts determining the total number of loads a transaction can perform. Transactional stores stall in the store buffer until the transaction commits. Multiple stores to the same location are not likely to be coalesced. As an unimplemented proposal, ASF is less concrete. It allows for the L1 cache to bound the number of loads and stores, for a dedicated processor structure (the locked line buffer, or LLB) to manage all transactional accesses, or for a combination of the two (e.g., an LLB for stores and the L1 cache for loads) [14].

The most notable differences between Rock and ASF relate to non-transactional accesses. Both support non-transactional loads that happen "immediately." In ASF, immediate non-transactional stores are also allowed. These are presumably intended to prevent capacity from being wasted on stack writes in the register-limited x86 architecture. In Rock, non-transactional stores stall in the store buffer. Thus while they still "happen," they do so only when the transaction commits or aborts.

For completeness, we also note that Rock checkpoints the entire architectural state of the CPU when starting a transaction, whereas ASF checkpoints only the PC and stack pointer. Rock and ASF also forbid certain operations, possibly including floating-point operations, compare and swap, or TLB miss handlers. These points are immaterial to this paper.

In the absence of non-transactional stores, the correctness of code that uses BEHTM is easy to verify, due to a simple guarantee by the hardware: any violation of the BEHTM rules (e.g., capacity, TLB misses, unsupported instructions, etc) causes the transaction to abort and return a status code that can be used to resolve the problem (e.g., by manually filling the TLB or switching to software mode). Conflicting memory accesses (e.g., transactional or non-transactional operations by another thread) also cause aborts. Because BEHTM does not support escape actions [45], all aborts should be thought of as occurring "immediately;" put another way, an abort cannot be temporarily suppressed.

Non-transactional stores create many complications. First, because conflict detection occurs at cache-line granularity, the programmer must ensure that a line is not accessed both transactionally and non-transactionally; these accesses can cause immediate program termination. Second, if non-transactional stores are used to acquire and release locks in non-transactional code called from a transactional context, then the immediacy of aborts makes possible that a transaction will acquire a lock and then abort, rendering the lock permanently unavailable. Last, correct ordering of non-transactional stores relative to transactional and non-transactional loads may require the use of memory fences in some circumstances.

# 3 Extending BEHTM for Verifiable Communication

We now define methods and structures that can be used for communication involving BEHTM transactions, and then discuss an implementation using ASF's non-transactional stores. Note that while arbitrarily complex mechanisms can be built atop ASF's non-transactional load/store abilities, we limit our presentation to a simple design (a single bidirectional one-word communication channel per CPU core) that we believe could easily be added to Rock or TSX. We have not yet identified a situation in which this simple design sacrifices generality, though high performance implementations may wish to make more aggressive use of ASF's rich support for nontransactional accesses.

We define `TxChannel` as an unbuffered, bidirectional communication medium with exactly two endpoints. Communication is asynchronous and polling-based. The contents of `TxChannel`

[2] Details about the implementation of Intel's TSX hardware were not available at the time this paper was written, but available information suggests that apart from its lack of support for nontransactional loads and stores within transactions, TSX should behave like Rock and ASF.

consist of a fixed-size message (e.g., one memory word), which can be atomically read from either endpoint via the `LdTxChannel` method. Similarly, either endpoint can atomically overwrite the contents of `TxChannel` via the `StTxChannel` method.[3]

The `ConfigTxChannel` method binds a thread to a channel and returns a handle to the channel. We further constrain the `TxChannel` mechanism as follows: when thread $T$ is executing a BEHTM transaction, it may not have more than one channel assigned to it, and it can neither bind to a channel, nor release its binding to a channel. Channel assignments are immutable for the duration of a transaction. Outside of a BEHTM transaction, a thread can change its channel configurations, and can communicate on multiple channels.

In the ASF specification, non-transactional loads and stores performed by a BEHTM transaction occur immediately, and do not incur transactional bookkeeping. The simplest implementation of `TxChannel` in ASF requires no hardware modifications: each `TxChannel` is simply a reserved memory region, `LdTxChannel` is a non-transactional load, and `StTxChannel` is a non-transactional store.For channels aligned on 8-byte boundaries, these loads and stores are naturally atomic using existing instructions in the x86 ISA. At application start, a pool of channels is created by allocating sufficient memory. `ConfigTxChannel` assigns an endpoint from this pool to its caller.

In ASF, no extra hardware is required for this `TxChannel` implementation. However, the overall correctness requires run-time software support. There are two main challenges. First, we must guarantee that endpoints are used correctly. Software must enforce the binding of a single thread to each channel endpoint (one approach can be found in the Singularity OS [19]). Second, the runtime system must ensure that a BEHTM transaction does not transactionally access the line on which its `TxChannel` is stored. This can be guaranteed by padding each channel to a cache line, and aligning the channel on a cache line boundary. To prevent false sharing, padding should take into consideration the size of L2 (and L3) cache lines, and whether the L2 prefetch unit always requests adjacent lines. In the worst case, modern machines might require each `TxChannel` to pad to four lines (256 bytes).

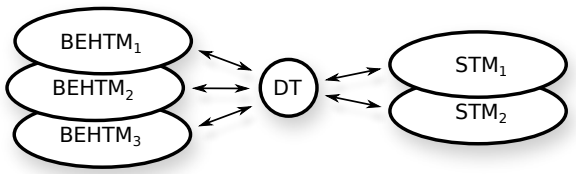
For well-structured communications, the atomicity of `LdTxChannel` and `StTxChannel` can be relaxed to enable larger channel sizes. Consider a 64-byte (8-word) channel. If one word is reserved as a status field, then a message can be sent by using non-transactional stores to set the other seven words of the channel. Then the message can be marked ready by setting the status field via an eighth non-transactional store. For its part, `LdTxChannel` would begin by (non-transactionally) spinning on the status word, and would not read the remainder of the channel until the status word is set appropriately. As long as one endpoint cannot perform consecutive `StTxChannel` instructions without intervening acknowledgments, the illusion of atomicity is preserved.[4]

# 4 Delegation

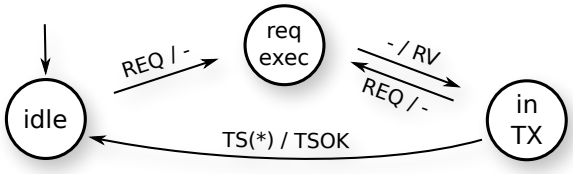
By convention, a BEHTM transaction is connected to at most one `TxChannel`. While transactions could communicate directly, we propose that the most hardware-agnostic mechanism involves the use of a distinguished software thread ($DT$). In our system, $DT$ serves as the remote endpoint for *every* transaction. The resulting topology is depicted in Figure 1. In multi-chip environments, the number of $DT$ threads could be dynamically adjusted (e.g., to ensure one per chip) without requiring the hardware to support multiple `TxChannel` interfaces, though the $DT$ threads might need to synchronize with each other.

[3]Given this definition, `TxChannel` can be thought of as an atomic register shared between exactly two threads.

[4]On architectures with relaxed memory consistency, one fence instruction would be required per channel access.




1: Multiple BEHTM and STM transactions using a single service thread $DT$.




2: `TxChannel` state transitions, with messages sent to / by $DT$. If a transaction sends REQ to DT, it must send a TS on commit / abort. TS cannot be sent while DT is processing a request (reqexec).

## 4.1 Communication Between Transactions and $DT$

Transactions communicate with $DT$ via their unique `TxChannels`, using the `LdTxChannel` and `StTxChannel` commands. Because remote accesses to cache lines also accessed by a BEHTM transaction will cause it to abort, we suggest pass-by-value parameters whenever possible.

Assuming `TxChannel` is large enough to hold entire messages, the protocol for communication between a BEHTM thread and $DT$ is as follows. There are two message types sent to $DT$: an operation request (REQ), containing an ID of the operation to be performed and the required operands; and transaction status (TS), signaling either abort (A) or commit (C). Communication is always initiated by the BEHTM thread through a REQ message. $DT$ can send two message types: a return value packet (RV), also serving as ack of receipt of REQ; and ack on receipt of TS(*) (TSOK). The state machine governing the protocol is depicted in Figure 2. In addition, the following constraints apply: (1) TS(*) cannot be sent until after the transaction commits or aborts. (2) Before sending RV, $DT$ must register onAbort and onCommit handlers for the requested operation. (3) Upon receipt of TS(A), $DT$ calls all onAbort handlers registered on the `TxChannel`. (4) On receipt of a TS(C), $DT$ calls all onCommit handlers registered on the `TxChannel`.

Lifting the size constraint on the exchanged messages can be accomplished easily by breaking send operations of larger messages into multiple short messages (REQ/RV) with the respective acknowledgment messages (REQOK/RVOK). We omit this for clarity and brevity of the presentation. More detail can be found in Appendix B.

In our description, a single $DT$ listens to requests by all BEHTM transactions. If all requested actions are thread-safe, it is straightforward to use multiple service threads within $DT$, or to place a $DT$ thread on each core. We leave this as future work.

A BEHTM thread can request execution of any instruction that is forbidden within its context. This includes requests for $DT$ to perform floating-point operations, allocations, I/O and other system calls, and arbitrary library calls. The BEHTM transaction is free to continue executing in parallel with the service of its request by $DT$.

## 4.2 Handling Aborts During Delegation

We must expect that a BEHTM transaction $B$ is vulnerable to abort at any time, including during communication with $DT$. To prevent races, $B$ cannot notify $DT$ of its abort unless it is sure that $DT$ is not currently writing a message to $B$. Thus $B$ must always be able to ascertain the channel state (e.g., by executing `LdTxChannel`).

With this guarantee, $B$ can invoke arbitrary library code through $DT$. If $B$ aborts while the library is running OS code or holds a lock, $DT$ will simply complete, release locks, and send an RV. Only then will $B$ send TS(A), at which time the delegated operation will be undone by $DT$. As an example, consider allocation: an allocated region must be returned to the heap if the transaction fails, and a deallocation must be deferred until the transaction commits [21, Ch. 5]. Bookkeeping this information in $DT$, and then finalizing or undoing operations in response to a commit/abort message, is straightforward.

## 4.3 Constraints

While our mechanism does not require general support for non-transactional stores within a BEHTM transaction, it must handle the possibility that the delegate and the BEHTM transaction share memory. ASF and Rock both provide strong atomicity [9], thus dangerous accesses will result in the BEHTM transaction aborting. If the correctness of the delegate depends on it reading a value written by the transaction, that value will not be available and arbitrary faults can ensue. Our prohibition on passing parameters by reference resolves this problem, and should not be a burden for the types of syscalls and library calls we aim to support.

A second concern is that polling for acks can lead to code that is difficult to analyze statically. While an adversary could easily create such code, we expect well-written code to encapsulate such control flow in objects that generate a few easily-recognizable instruction sequence patterns.

## 4.4 Overhead

When our mechanism is not used, a transaction must perform a single `LdTxChannel` on commit or abort. The call will return TSOK, indicating an idle channel (i.e., the transaction did not use the channel and need not send TS(*)). When delegation is used, overhead can be approximated by the number of round-trip cache communications that occur, with all but the last occurring before the transaction commits. This overhead can be reduced in two ways: First, when the delegate has no return value, polling for RV can be delayed until after the transaction commits. Second, in ASF it is technically possible to create parameter and return value packets for large messages in memory, and to transfer their ownership by passing pointers. While this technique obviates REQOK and RVOK messages, we do not recommend it: it is difficult to verify, requires strong store-store ordering, and would require a long-term commitment to broad hardware support for immediate non-transactional stores.

In practice, other sources of latency will arise. One example is the time to service the request and the time taken for $DT$ to receive the request. If this delay is too great, it is possible to partition the transactions and employ multiple delegation threads.[5] Another is the overhead of commit and abort handlers. This cost is likely to be application-specific, though existing experience with allocation suggests that commit handlers should be inexpensive (e.g., resetting a log) in the common case.

# 5 Nesting

Our delegation mechanism is essentially an open-nested transaction, in which the parent uses BEHTM, and the child may or may not. It is further constrained in that the parent and child are assumed to share no memory. We now show how $DT$ can serve as a transaction manager, enabling many interesting patterns of parallel nested child transactions.

Our goal is for a parent transaction to request that $DT$ launches many child transactions on its behalf, and that children run in parallel. We assume an underlying TM similar to Hybrid NOrec [16,39]: STM and BEHTM transactions run concurrently, using the values stored in memory to detect conflicts. A single lock ensures atomicity when STM transactions commit, and BEHTM transactions incur extra overhead to commit when there are active STM transactions. STM transactions do not update memory until commit time. This contrasts with most nested STM proposals, which rely on in-place update [1, 5, 36, 43] to avoid overhead. However, in-place update appears incompatible with strong language-level memory consistency models [30].

We analyze both both open and closed nesting, to see how BEHTM can be used in the parent and/or child transactions. We restrict our study to the case when a parent does not make progress while any children are running [35]. This assumption is common in prior explorations of nested parallel children, and removing this prohibition does not substantially affect our findings. Similarly, we do not consider deep nesting: the capacity limits of BEHTM make it unlikely that deep nesting would be practical.

We extend $DT$ from the prior section to support nesting in a manner similar to the closed-nested transaction coordinator used by Ramadan et al. [38]. In that work, a transaction communicates with the manager to request that child transactions be assigned to idle threads, and then the children coordinate through the manager to commit "into" the parent. In Ramadan's work, children often would commit together, and thus $DT$ would coordinate commit or unwind of children. We extend this system with support for both open nesting and BEHTM. We also add support for sandboxing child operations (the need for which is discussed below), and expose multiple mechanisms for merging child results into the parent. These mechanisms balance capacity constraints of a BEHTM transaction with run-time overhead and generality imposed by sharing constraints.

## 5.1 Sharing Data Between a Parent and its Nested Child

Most nested STM systems employ Agrawal et al.'s definition of conflicting transactions [1]. The definition permits maximal sharing among a parent and its children: a child transaction never conflicts with its parent, even if it reads or writes locations that its parent has accessed. This definition even holds when an open-nested transaction modifies a location its parent has also modified. Implementations that allow maximal sharing are necessarily complex [5].

At a high level, our system works by having parent transactions request that $DT$ launch child transactions, and then relying on $DT$

[5] Note that concentrating certain operations on a single thread (e.g., system calls [12] or allocation [42]) may offer greater benefit than multiple $DT$ threads.

| ASA | RSO | SFF |
|---|---|---|
| $x \in C_{read} \Rightarrow x \notin P_{read} \bigcup P_{write}$<br>$x \in C_{write} \Rightarrow x \notin P_{read} \bigcup P_{write}$ | $x \in C_{read} \Rightarrow x \notin P_{write}$<br>$x \in C_{write} \Rightarrow x \notin P_{read} \bigcup P_{write}$ | $x \in C_{read} \Rightarrow x \notin P_{write}$ |

3: Rules governing overlap between child ($C$) and parent ($P$) memory accesses.

to mediate the commit of those children. The parent-child relationship is invisible to the BEHTM hardware. Because BEHTM interprets any cache-level conflict as a cause for abort, we are unable to support the maximal model.

In more detail, suppose BEHTM transaction $P$ has read location $X$ and then requested that $DT$ execute a nested child $C$. The cache line containing $X$ will be tracked by the processor executing $P$, and thus if $C$ fetches the cache line containing $X$ into a writable state, the line will be evicted from $P$'s processor's cache and $P$ will abort. A similar condition occurs if $P$ writes $X$ and $C$ attempts to read $X$. Consequently, we consider three weaker communication patterns for parents and children, shown in Figure 3:

- **Access set augmenting (ASA):** $C$ is an ASA child of $P$ if it does not access any cache line also accessed by $P$.
- **Read share only (RSO):** $C$ is an RSO child of $P$ if all overlapping accesses are reads by $P$ and $C$.
- **Store forward forbidding (SFF):** $C$ is an SFF child of $P$ if it does not read a location written by $P$. $C$ may write to locations read or written by $P$, and may read locations read by $P$.

## 5.2 Sandboxing

If $C$ reads a location $X$ written by its BEHTM parent $P$, $P$ will abort and $C$ will continue running, using the value of $X$ that existed prior to $P$'s write. This can lead to erroneous behavior: Suppose $P$ writes $X = |X|$, and $C$ computes $\sqrt{X}$. The programmer is likely to assume that $X \geq 0$ holds when $C$ begins, and thus if the underlying TM run-time executing $C$ is opaque [22], $C$ should never fault. If $P$ uses BEHTM, $C$'s read will abort $P$ and return the value of $X$ from before $P$ began; thus, $C$ can execute with an inconsistent view of memory in which $X < 0$.

To address this problem, the TM runtime must ensure that $C$'s inconsistent view of memory does not cause it to produce a visible fault, enter an infinite loop, execute a computed jump to non-transactional code, or corrupt the heap. If it cannot be statically proven that $C$ does not read $P$'s writes, the child must coordinate with $DT$ to achieve a degree of run-time sandboxing: before any potentially unsafe operation, the child must query $DT$ to check if $P$ is still live. To prove $P$ live, $DT$ must communicate with $P$; it cannot simply read $P$'s status, because $P$ may be in its abort handler.

Our solution uses two new message types, CHKSTATE and LIVE. After a parent $P$ sends REQ to invoke a set of child transactions, it executes a spin loop in which it reads from its channel, awaiting an RV or CHKSTATE message. As before, RV indicates that the child operations are complete, and $P$ can resume. When CHKSTATE is received, $P$ immediately sends LIVE. If $P$ aborts, it must wait for a CHKSTATE or RV message, and then send TS(A). Failing to wait could result in its message racing with $DT$'s next CHKSTATE or RV message.

The details of when sandboxing is needed have been explored elsewhere [40, 41]. Our contribution is discovering that sandboxing is needed even for an opaque TM if a child $C$ of a BEHTM parent $P$ can cause $P$ to abort. Beyond the concerns raised in past work, we must ensure that if $C$ computes an address and writes to it, the address computation was not based on inconsistent reads. This requirement is outside of the sandboxing that BEHTM guarantees (i.e., that faults within a transaction cause it to abort), so a software child using in-place update must ensure its parent is live before it writes to a location that might not be protected by the TM. Otherwise, a subsequent undo action might participate in a data race. For the same reason, open-nested children (BEHTM and STM) must ensure their parents are live before committing. Sandboxing is also needed for some non-transactional stores performed by a BEHTM child (e.g., those that are not to the current stack frame) and by STM parents of closed-nested BEHTM children.

## 5.3 Findings

We now present our findings for all combinations of open and closed nesting, and BEHTM and STM transactions, in which at least one transaction uses BEHTM. A summary appears in Figure 4.

**Open-nested BEHTM Child of BEHTM Parent:** In our model, a BEHTM parent can have an unlimited number of open-nested BEHTM children. However, because the BEHTM parent cannot forward its speculative stores, these children must obey the ASA or RSO models.[6]

The algorithm is simple: $P$ sends a REQ message to $DT$ to request that the children run, and then $DT$ assigns work to threads in its private pool. These threads' transactions communicate with $DT$ via the `TxChannel` mechanism. Because $P$ uses BEHTM, if child $C$ attempts to read any location $P$ has written, or to write any location $P$ has read, $P$ will abort. $C$ will continue to run, but because it uses BEHTM, its effects are guarded until the commit point. Because an open-nested child $C$ commits before its parent $P$, rather than atomically with $P$, child commit is straightforward: $DT$ uses CHKSTATE to ensure that $P$ is live, and then allows $C$ to commit according to the Hybrid NOrec protocol. This is sufficient for correctness: because $C$ is at its commit point, if $P$ has not aborted, $C$'s accesses must have been ASA or RSO. Should $C$ abort at any time, $DT$ restarts it only if $P$ is still live. When $C$ commits, it sends an RV to $DT$. The RV message includes the commit and abort handlers that must run when $P$ completes, as well as any values to return to $P$. Because each open-nested child commits or aborts independently, no coordination is needed among children.

**Open-nested STM Child of BEHTM Parent:** An open-nested software child of a BEHTM parent runs in almost the same manner as just described. However, because BEHTM is not used in the child, it must sandbox more operations. Again, any number of children are possible, and these children must be ASA or RSO.

There is one additional restriction: because STM transactions in Hybrid NOrec use buffered writes, writes by $C$ to locations read by $P$ do not cause $P$ to abort during $C$'s execution. Instead, they cause $P$ to abort during $C$'s "redo" phase, after $C$ has committed, which can lead to erroneous output. To prevent this, whenever $C$ adds a location $L$ to its write set, we require it to execute a write prefetch with $L$ as the operand. If $P$ has read $L$, the prefetch will cause $P$ to abort before $C$'s pre-commit check.

[6] In this and subsequent discussion, the parent can simulate a limited number of known child reads of parent writes by sending those speculatively written values as parameters to the child in the initial REQ message.

| **Parent** | BEHTM | BEHTM | STM | BEHTM | STM | BEHTM |
|---|---|---|---|---|---|---|
| **Child** | BEHTM | STM | BEHTM | BEHTM | BEHTM | STM |
| **Nesting** | Open | Open | Open | Closed | Closed | Closed |
| **Valid Access Patterns** | ASA/RSO | ASA/RSO | ASA/RSO | None | ASA/RSO | ASA/RSO/SFF |

4: Nesting and memory access patterns that can take advantage of BEHTM.

**Open-nested BEHTM Child of STM Parent:** In previous sections, we saw that the ability of a BEHTM parent to immediately detect conflicts with its children due to non-ASA/RSO sharing meant that little work was required by the parent when deciding whether a child could commit. The only cost was for the communication to ensure that the parent had not already aborted. When the parent is an STM transaction, more work is needed at this point: the parent must validate its read set and write-prefetch every location in its write set before sending LIVE.

Read-set validation serves two ends: it ensures $P$ is still active (otherwise, $C$'s commit could be erroneous) and causes $C$ to abort if it has speculative writes to locations $P$ has read. Note that validation does not cause $C$ to abort due to read-read sharing in BEHTM. Similarly, write prefetching causes $C$ to abort if it read from or stored to anything $P$ had written. The net effect is that $C$ can only commit if it is ASA or RSO.

If the language memory model allowed for in-place update, then child reads of parent writes could be forwarded to the BEHTM child. We leave explorations of this topic as future work.

**Closed-nested BEHTM Child of BEHTM Parent:** We are aware of two mechanisms for committing a closed-nested child. In STM, the more common approach is for $C$ to merge its read and write sets into $P$. The other, which resembles distributed transactions, is for $C$ to wait for $P$ to reach its commit point, and then employ some protocol by which the two commit or abort together.

When a closed-nested child uses BEHTM, there is no software-accessible structure through which the child can communicate its accesses to its parent. Thus the first mechanism cannot be employed. In current BEHTM systems, the second approach cannot be achieved either. Thus a BEHTM parent cannot have a closed-nested BEHTM child.

To understand why the second approach cannot work, we show that it is impossible in current BEHTM systems to achieve a coordinated commit of two transactions. Recall that in BEHTM, a transaction executes a `commit` instruction to make all its effects visible (i.e., a one-phase commit protocol). Prior to issuing this command, the transaction is vulnerable to aborts (e.g., due to memory conflicts or context switches). After issuing the command, the transaction's updates are visible to other threads. Suppose we wish for BEHTM transactions $A$ and $B$ to commit as a single atomic event. By definition, $A$ commits exactly when it executes its `commit` instruction. The same is true of $B$. If while $A$ is issuing `commit`, $B$ experiences a context switch, then $A$ will commit while $B$ aborts. A symmetric case covers $B$ issuing `commit` before $A$. Even in the absence of additional threads, atomic group commit of two (or more) BEHTM transactions is not possible.

**Closed-nested BEHTM Child of STM Parent:** There is a slight difference between this and the prior pattern. The child BEHTM transaction still cannot communicate its read and write sets to the parent. However, a single closed-nested BEHTM child can commit atomically with its software parent, and thus this pattern can be supported in a limited sense.

Suppose that STM parent $P$ invokes the closed-nested child $C$ at the end of $P$'s execution. In this case, once $C$ finishes its computation, both it and $P$ can commit. Our solution is for $C$ to send its RV to $DT$, who then instructs $P$ to "pre-commit." Once $P$ completes this step, $DT$ informs $C$ to commit. If $C$ succeeds, then $P$ is instructed to finalize its commit. Otherwise, $P$ aborts. Note that closed-nested STM children could also be involved in the commit; they would simply "commit into" $P$ prior to the coordination between $P$ and its BEHTM child.

Using Hybrid NOrec as an example, $P$'s pre-commit consists of (a) acquiring the global commit lock, (b) validating its read set, and (c) write-prefetching all elements of the write set. These steps ensure that no other transaction will invalidate $P$, that $P$ is valid, and that $C$ will abort if it performed non-ASA/RSO accesses. Note, too, that $C$ need not be the last step in $P$'s computation. In that case, the ultimate commit follows the same protocol as already described. When $C$ returns control to $P$, $C$ has not committed and $P$ must perform the write prefetch and validate steps (but not acquire the global lock), and then determine (through $DT$) that $C$ is still live before continuing. Furthermore, $P$ must be sandboxed from this point forward, so that any attempt to use a location written by $C$ will not result in a fault. Sandboxing in this case entails $P$ querying $C$ to ensure $C$ has not aborted because an abort by $C$ could signify that $P$ attempted to read one of $C$'s writes.

**Closed-nested STM Child of BEHTM Parent:** The final pattern we consider is when $P$ uses BEHTM but $C$ does not. Naturally, $C$ cannot read $P$'s writes. All other sharing patterns are allowed, and thus ASA, RSO, and SFF children are possible.

Because the child is an STM transaction, it has precise logs of the address/value pairs it read and the address/value pairs it intends to write. Note that $C$ does not perform write-prefetching. To commit, $C$ simply passes all of this information to $P$ in the RV message. Before $P$ continues, it validates $C$'s reads by checking that each address it received still holds the value that was sent. These checks use *transactional* reads, and thus merge $C$'s reads into $P$'s read set. $P$ then re-executes $C$'s writes using *transactional* stores, to add them to its write set.

There are three caveats. First, $C$ must be sandboxed so attempts to read $P$'s writes do not lead to erroneous behavior. Second, while writes to locations read or written by $P$ are possible, overwrites of $P$'s writes are not likely to succeed in practice because most writes are preceded by a read to the same location. Third, and most important, the size of $C$'s read and write sets cannot be so large as to overflow the BEHTM capacity constraints imposed on $P$. Otherwise $P$ will ultimately abort. While it is possible to communicate only $C$'s writes in RV, and then require $DT$ to validate $C$'s reads before $P$ commits, the resulting constraints on the behavior of $P$ limit the technique's usefulness.

## 5.4 Relationship to Delegation

Delegation and nesting serve fundamentally different roles. Delegation addresses the use of orthogonal components that are forbidden within a BEHTM context, whereas nesting is a more general approach to achieving parallelism. Thus with delegation we assume that each delegate effectively operates on a private region of the heap, and there is no sharing between a delegate and the invoking transaction (though in reality there will be false conflicts, such

as those arising when the allocator traverses metadata stored in the header of allocated objects). Because a parent abort cannot affect the delegate, delegation does not introduce the need for sandboxing, whereas nesting does.

The models also differ with respect to parallelism. With nesting, the parent does not execute in parallel with its children, because doing so can increase the likelihood of conflicts (particularly given BEHTM "attacker wins" conflict resolution). In contrast, delegation is naturally parallel, because delegate and transaction accesses are assumed to be disjoint.

In addition, we require nesting to employ the same transactional runtime as the parent, whereas delegate actions need not, and so a delegate could use locks, STM, or lock-free programming (perhaps assisted by ASF). The cost of this flexibility is that interactions with a delegate are pass-by-value.

# 6 Related Work

In the earliest (non-parallel) nested STM [36], several of the cases that BEHTM cannot handle were also problematic. The authors claimed that a child overwriting a parent's read should be rare in practice, which mitigates our inability to support the pattern in BEHTM. They also identified a child overwriting a parent's write as challenging. We can only support this pattern with closed-nested STM children, and our solution resembles theirs. Lastly, that work identified programming pitfalls such as deadlock in abort handlers; we expect their proposed solutions to apply to our work.

NePaLTM [43] used a mutex to run child transactions serially. Later, Barreto et al. implemented Agrawal's algorithm [5]. Since they used in-place update, their system was not compatible with strong language-level memory models. However, it supported more patterns of sharing than we can achieve when using BEHTM. They found that the time required to propagate updates to a parent increased the likelihood of false conflicts. This is likely to affect our closed-nested STM child transactions. Baek et al. implemented the first closed-nested STM (NeSTM) with buffered update [3]. Their implementation would experience "zombie" transactions in some cases. Our discovery of the need for sandboxing explains and prevents zombies.

Coordinated sibling transactions (CST) [38] are a closed-nested system supporting complex patterns of group commit for child transactions. Our use of a $DT$ thread resembles CST's use of a master transaction coordinator to manage thread pools. We believe that our system could be used to implement the CST coordination patterns, with the caveats that a BEHTM parent can have only software CST children, a software parent can have at most one BEHTM child, and children must obey the access rules in Section 5.

**Hardware Implementations of Nesting:** Escape actions are a means of deferring aborts in HTM, which enable simple delegation operations such as time-related syscalls [45]. Moravan et al. [34] extended LogTM with (non-parallel) open-nested transactions, and FaNTM [4] accelerated NeSTM via hardware signatures. These techniques all required modifications to the pipeline or coherence protocol. FaNTM is also prone to complex deadlocks and livelocks as a result of partial rollback. Our use of livelock-free Hybrid NOrec avoids such problems.

**Beyond Nesting:** Our system can support many other patterns of communication, such as safe futures [44] and CST. Another promising direction is to implement communicators [28, 29], though the requirement that all participants in a communicator commit together necessitates that the communicator dynamically limits to one the number of BEHTM transactions bound to it.

# 7 Conclusions

This paper presented a system that supports delegation and nesting in best-effort hardware transactional memory. Our design is in keeping with the spirit of BEHTM, in that it requires minimal changes to existing hardware; in fact, the AMD ASF proposal requires no further extensions to implement our abstract communication channel, and we believe that a minimal implementation of our communication abstraction in the Rock prototype processor or forthcoming TSX platform would be straightforward.

Our system allows BEHTM transactions to invoke delegates to perform non-transactional actions on their behalf, such as system and library calls. As necessary, the transaction can register handlers to unwind or finalize the behavior of delegates on transaction commit or abort. We employ a dedicated helper thread to manage the relationship between a transaction and its delegates; ultimately, we showed that the mechanism can generalize to support open and closed-nested parallel transactions. While delegation prevents fallback to software mode for simple operations, such as time-related system calls and allocation, nesting allows for enhanced parallelism in workloads in which top-level transactions conflict with high frequency, but each possesses a significant amount of internal parallelism. Unfortunately, the promise of nested parallelism is tempered by the limitations we discovered, which appear to be intrinsic to BEHTM. When these limitations are encountered, the system will need to fall back to a pure software mode of execution.

We leave as future work a thorough performance evaluation of nesting in BEHTM because it requires more precise simulation support than is currently available. We are also investigating whether minimal hardware modifications could enable group commit, decrease the cost of sandboxing, or enable simple data forwarding among transactions. A further area of exploration relates to the performance tradeoff of supporting multiple `TxChannels` per transaction, or multiple $DT$ threads.

# 8 References


[1] Kunal Agrawal, Jeremy Fineman, and Jim Sukha. Nested Parallelism in Transactional Memory. In *Proceedings of the 13th ACM Symposium on Principles and Practice of Parallel Programming*, Salt Lake City, Utah, February 2008.

[2] C. Scott Ananian, Krste Asanovic, Bradley C. Kuszmaul, Charles E. Leiserson, and Sean Lie. Unbounded Transactional Memory. In *Proceedings of the 11th International Symposium on High-Performance Computer Architecture*, February 2005.

[3] Woongki Baek, Nathan Bronson, Christos Kozyrakis, and Kunle Olukotun. Implementing and Evaluating Nested Parallel Transactions in Software Transactional Memory. In *Proceedings of the 22nd ACM Symposium on Parallelism in Algorithms and Architectures*, Santorini, Greece, June 2010.

[4] Woongki Baek, Nathan Bronson, Christos Kozyrakis, and Kunle Olukotun. Making Nested Parallel Transactions Practical using Lightweight Hardware Support . In *Proceedings of the 24th ACM International Conference on Supercomputing*, Tsukuba, Japan, June 2010.

[5] Joao Barreto, Aleksandar Dragojevic, Paulo Ferreira, Rachid Guerraoui, and Michal Kapalka. Leveraging Parallel Nesting in Transactional Memory. In *Proceedings of the 15th ACM Symposium on Principles and Practice of Parallel Programming*, Bangalore, India, January 2010.

[6] Andrew Baumann, Paul Barham, Pierre-Evariste Dagand, Tim Harris, Rebecca Isaacs, Simon Peter, Timothy Roscoe, Adrian Schupbach, and Akhilesh Singhania. The Multikernel: A New OS Architecture for Scalable Multicore Systems. In *Proceedings of the 22nd ACM Symposium on Operating Systems Principles*, Big Sky, Mont., October 2009.

[7] Emery Berger, Kathryn McKinley, Robert Blumofe, and Paul Wilson. Hoard: A Scalable Memory Allocator for Multithreaded Applications. In *Proceedings of the 9th International Conference on Architectural Support for Programming Languages and Operating Systems*, Cambridge, Mass., November 2000.

[8] Colin Blundell, Joe Devietti, E Christopher Lewis, and Milo Martin. Making the Fast Case Common and the Uncommon Case Simple in Unbounded Transactional Memory. In *Proceedings of the 34th International Symposium on Computer Architecture*, San Diego, Calif., June 2007.

[9] Colin Blundell, E Christopher Lewis, and Milo M. K. Martin. Subtleties of Transactional Memory Atomicity Semantics. *Computer Architecture Letters*, 5(2), November 2006.

[10] Jayaram Bobba, Kevin Moore, Haris Volos, Luke Yen, Mark Hill, Michael Swift, and David Wood. Performance Pathologies in Hardware Transactional Memory. In *Proceedings of the 34th International Symposium on Computer Architecture*, San Diego, Calif., June 2007.

[11] Silas Boyd-Wickizer, Haibo Chen, Rong Chen, Yandong Mao, M. Frans Kaashoek, Robert Morris, Aleksey Pesterev, Lex Stein, Ming Wu, Yue hua Dai, Yang Zhang, and Zheng Zhang. Corey: An operating system for many cores. In *Proceedings of the 8th USENIX Symposium on Operating Systems Design and Implementation*, San Diego, Calif., December 2008.

[12] Silas Boyd-Wickizer, Austin Clements, Yandong Mao, Aleksey Pesterev, M. Frans Kaashoek, Robert Morris, and Nickolai Zeldovich. An Analysis of Linux Scalability to Many Cores. In *Proceedings of the 9th USENIX Symposium on Operating Systems Design and Implementation*, Vancouver, B.C., Canada, October 2010.

[13] Shailender Chaudhry, Robert Cypher, Magnus Ekman, Martin Karlsson, Anders Landin, and Sherman Yip. Rock: A High-Performance Sparc CMT Processor. *IEEE Micro*, 29(2):6–16, March–April 2009.

[14] Dave Christie, Jae-Woong Chung, Stephan Diestelhorst, Michael Hohmuth, Martin Pohlack, Christof Fetzer, Martin Nowack, Torvald Riegel, Pascal Felber, Patrick Marlier, and Etienne Riviere. Evaluation of AMD's Advanced Synchronization Facility within a Complete Transactional Memory Stack. In *Proceedings of the EuroSys2010 Conference*, Paris, France, April 2010.

[15] Jaewoong Chung, Luke Yen, Stephan Diestelhorst, Martin Pohlack, Michael Hohmuth, Dan Grossman, and David Christie. ASF: AMD64 Extension for Lock-free Data Structures and Transactional Memory. In *Proceedings of the 43rd IEEE/ACM International Symposium on Microarchitecture*, Atlanta, Ga., December 2010.

[16] Luke Dalessandro, Francois Carouge, Sean White, Yossi Lev, Mark Moir, Michael Scott, and Michael Spear. Hybrid NOrec: A Case Study in the Effectiveness of Best Effort Hardware Transactional Memory. In *Proceedings of the 16th International Conference on Architectural Support for Programming Languages and Operating Systems*, Newport Beach, Calif., March 2011.

[17] Dave Dice, Yossi Lev, Virendra Marathe, Mark Moir, Marek Olszewski, and Dan Nussbaum. Simplifying Concurrent Algorithms by Exploiting Hardware TM. In *Proceedings of the 22nd ACM Symposium on Parallelism in Algorithms and Architectures*, Santorini, Greece, June 2010.

[18] David Dice, Yossi Lev, Mark Moir, and Daniel Nussbaum. Early Experience with a Commercial Hardware Transactional Memory Implementation. In *Proceedings of the 14th International Conference on Architectural Support for Programming Languages and Operating Systems*, Washington, D.C., March 2009.

[19] Manuel Fahndrich, Mark Aiken, Chris Hawblitzel, Orion Hodson, Galen Hunt, James Larus, and Steven Levi. Language Support for Fast and Reliable Message-based Communication in Singularity OS. In *Proceedings of the EuroSys2006 Conference*, Leuven, Belgium, April 2006.

[20] Francois Carouge and Michael Spear. A Scalable Lock-Free Universal Construction with Best Effort Transactional Hardware. In *Proceedings of the 24th International Symposium on Distributed Computing*, Cambridge, Mass., September 2010.

[21] Keir Fraser. *Practical Lock-Freedom*. PhD thesis, King's College, University of Cambridge, September 2003.

[22] Rachid Guerraoui and Michal Kapalka. On the Correctness of Transactional Memory. In *Proceedings of the 13th ACM Symposium on Principles and Practice of Parallel Programming*, Salt Lake City, Utah, February 2008.

[23] Lance Hammond, Vicky Wong, Mike Chen, Brian D. Carlstrom, John D. Davis, Ben Hertzberg, Manohar K. Prabju, Honggo Wijaya, Christos Kozyrakis, and Kunle Olukotun. Transactional Memory Coherence and Consistency. In *Proceedings of the 31st International Symposium on Computer Architecture*, Munich, Germany, June 2004.

[24] Tim Harris, James Larus, and Ravi Rajwar. *Transactional Memory, 2nd edition*. Synthesis Lectures on Computer Architecture. Morgan & Claypool, 2010.

[25] Danny Hendler, Itai Incze, Nir Shavit, and Moran Tzafrir. Flat Combining and the Synchronization-Parallelism Tradeoff. In *Proceedings of the 22nd ACM Symposium on Parallelism in Algorithms and Architectures*, Santorini, Greece, June 2010.

[26] Richard L. Hudson, Bratin Saha, Ali-Reza Adl-Tabatabai, and Benjamin Hertzberg. A Scalable Transactional Memory Allocator. In *Proceedings of the International Symposium on Memory Management*, Ottawa, Ont., Canada, June 2006.

[27] Intel Corp. *Intel Architecture Instruction Set Extensions Programming Reference*, 319433-012a edition, February 2012.

[28] Mohsen Lesani and Jens Palsberg. Communicating Memory Transactions. In *Proceedings of the 16th ACM Symposium on Principles and Practice of Parallel Programming*, San Antonio, Tex., February 2011.

[29] Victor Luchangco and Virendra Marathe. Transaction Communicators: Enabling Cooperation Among Concurrent Transactions. In *Proceedings of the 16th ACM Symposium on Principles and Practice of Parallel Programming*, San Antonio, Tex., February 2011.

[30] Vijay Menon, Steven Balensiefer, Tatiana Shpeisman, Ali-Reza Adl-Tabatabai, Richard Hudson, Bratin Saha, and Adam Welc. Practical Weak-Atomicity Semantics for Java STM. In *Proceedings of the 20th ACM Symposium on Parallelism in Algorithms and Architectures*, Munich, Germany, June 2008.

[31] Maged M. Michael. Scalable Lock-Free Dynamic Memory Allocation. In *Proceedings of the 25th ACM Conference on Programming Language Design and Implementation*, Washington, D.C., June 2004.

[32] Chi Cao Minh, Martin Trautmann, JaeWoong Chung, Austen McDonald, Nathan Bronson, Jared Casper, Christos Kozyrakis, and Kunle Olukotun. An Effective Hybrid Transactional Memory System with Strong Isolation Guarantees. In *Proceedings of the 34th International Symposium on Computer Architecture*, San Diego, Calif., June 2007.

[33] Kevin E. Moore, Jayaram Bobba, Michelle J. Moravan, Mark D. Hill, and David A. Wood. LogTM: Log-based Transactional Memory. In *Proceedings of the 12th International Symposium on High-Performance Computer Architecture*, February 2006.

[34] Michelle Moravan, Jayaram Bobba, Kevin Moore, Luke Yen, Mark Hill, Ben Liblit, Michael Swift, and David Wood. Supporting Nested Transactional Memory in LogTM. In *Proceedings of the 12th International Conference on Architectural Support for Programming Languages and Operating Systems*, San Jose, Calif., October 2006.

[35] Eliot Moss and Antony L. Hosking. Nested Transactional Memory: Model and Preliminary Architecture Sketches. In *Proceedings of the Workshop on Synchronization and Concurrency in Object-Oriented Languages*, San Diego, Calif., October 2005.

[36] Yang Ni, Vijay Menon, Ali-Reza Adl-Tabatabai, Antony Hosking, Rick Hudson, Eliot Moss, Bratin Saha, and Tatiana Shpeisman. Open Nesting in Software Transactional Memory. In *Proceedings of the 12th ACM Symposium on Principles and Practice of Parallel Programming*, San Jose, Calif., March 2007.

[37] Ravi Rajwar, Maurice Herlihy, and Konrad Lai. Virtualizing Transactional Memory. In *Proceedings of the 32nd International Symposium on Computer Architecture*, Madison, Wis., June 2005.

[38] Hany Ramadan and Emmett Witchel. The Xfork in the Road to Coordinated Sibling Transactions. In *Proceedings of the 4th ACM SIGPLAN Workshop on Transactional Computing*, Raleigh, N.C., February 2009.

[39] Torvald Riegel, Patrick Marlier, Martin Nowack, Pascal Felber, and Christof Fetzer. Optimizing Hybrid Transactional Memory: The Importance of Nonspeculative Operations. In *Proceedings of the 23rd ACM Symposium on Parallelism in Algorithms and Architectures*, June 2011.

[40] Bratin Saha, Ali-Reza Adl-Tabatabai, Richard L. Hudson, Chi Cao Minh, and Benjamin Hertzberg. McRT-STM: A High Performance Software Transactional Memory System For A Multi-Core Runtime. In *Proceedings of the 11th ACM Symposium on Principles and Practice of Parallel Programming*, New York, N.Y., March 2006.

[41] Michael F. Spear, Maged M. Michael, Michael L. Scott, and Peng Wu. Reducing Memory Ordering Overheads in Software Transactional Memory. In *Proceedings of the 2009 International Symposium on Code Generation and Optimization*, Seattle, Wash., March 2009.

[42] Devesh Tiwari, Sanghoon Lee, James Tuck, and Yan Solihin. MMT: Exploiting Fine-Grained Parallelism in Dynamic Memory Management. In *Proceedings of the 24th International Parallel and Distributed Processing Symposium*, Atlanta, Ga., April 2010.

[43] Haris Volos, Adam Welc, Ali-Reza Adl-Tabatabai, Tatiana Shpeisman, Xinmin Tian, and Ravi Narayanaswamy. NePalTM: Design and Implementation of Nested Parallelism for Transactional Memory Systems. In *23rd European Conference on Object-Oriented Programming*, Genova, Italy, July 2009.

[44] Adam Welc, Suresh Jagannathan, and Antony Hosking. Safe Futures for Java. In *Proceedings of the 20th ACM Conference on Object-Oriented Programming, Systems, Languages, and Applications*, San Diego, Calif., October 2005.

[45] Craig Zilles and Lee Baugh. Extending Hardware Transactional Memory to Support Non-Busy Waiting and Non-Transactional Actions. In *Proceedings of the 1st ACM SIGPLAN Workshop on Languages, Compilers, and Hardware Support for Transactional Computing*, Ottawa, Ont., Canada, June 2006.

# A Use Case: Memory Allocation

TM makes it easy to express concurrent algorithms that operate on dynamic data structures. However, it is crucial that the allocation and reclamation of memory used to implement the data structure is handled correctly. While data structure microbenchmarks avoid the need to allocate from *within* a transaction by pre-allocating a single node before performing an insertion transaction, and by buffering a single deallocation in a removal transaction until the transaction commits, real code is not afforded this luxury. Once transactions are composed, it becomes impossible to predict the number and size of allocations, or the number of deallocations.

Most STM run-times provide a safe allocator interface [26] in which a transaction suspends when an allocation is requested, the allocator is invoked, and the transaction logs the result so the corresponding memory can be freed if the transaction aborts. These allocators also defer all deallocation until the transaction commits. This approach is not possible for BEHTM for several reasons. A rather trivial point is that the logging operations consume limited resources that should be used to write program data. More significantly, most allocators have some form of synchronization (e.g., locks). If the BEHTM transaction accesses this lock transactionally, it inherits a contention hotspot with all other allocations; if it accesses the lock non-transactionally, it risks aborting before releasing the lock, causing a deadlock. In addition, memory leaks are possible if the transaction aborts after allocating memory but before logging it. Finally, modern allocators [7, 31] typically maintain a local heap that, if depleted, must be expanded by invoking the OS. Such an instruction would cause the BEHTM transaction to immediately abort, because context switches are unsupported.[7]

[7]This is a context switch in the broader sense: a change of the scope and privilege level of execution–not necessarily a switch to a different user-space application.

Even code that *might* perform such operations is not safe to call, and thus such simple operations as allocating or freeing memory usually require transactions to transition into software mode. Even if these operations were transaction-safe without locks, they would likely introduce aborts due to meta-data contention. A variety of approaches to reducing contention, varying from one-off solutions for allocation [42] and syscalls [11] to general lock-aggregating techniques like flat combining [25], all support the notion that aggregating certain computations on a single thread can effectively reduce latency by replacing meta-data contention and syscall overhead with the cost of a single round-trip communication.

## A.1 Delegated Memory Allocation

These constraints on allocation within BEHTM transactions make it an ideal candidate for the delegation mechanism outlined in Section 4. The simplest implementation will begin with BEHTM transaction $T$ sending an allocation request to $DT$, consisting of an identifier for the `malloc` operation and a size parameter. $DT$ then performs the allocation, registers a corresponding undo action, and returns the obtained pointer through the channel. After making the request, $T$ simply polls for a return value, using non-transactional loads. When $T$ commits, it sends a message to $DT$ indicating that the undo action can be unregistered; when $T$ aborts, it informs $DT$ that the undo action must be performed.

Deallocation can be achieved similarly: $T$ sends a message consisting of an identifier for the `free` operation and the pointer to free. $DT$ then logs the deallocation for replay at commit time, and sends an acknowledgment. $T$ can run in parallel with $DT$, but cannot re-use the channel until it receives $DT$'s (empty) response.

## A.2 Performance Estimate

Delegated memory allocation incurs a communication overhead but prevents the cost associated with abort and restart of the hardware transaction. In addition, reduced synchronization through centralized execution can reduce the latency of the allocation operation.

If we ignore the gains that come from a simpler allocator design, we can estimate the worst-case overhead. First, we consider the raw cost of sending a message via shared memory, when the receiver is actively polling for a message. We conducted an experiment on a quad-core AMD Phenom™ II X4 945 Processor, running at 3.0 GHz, to measure round-trip time for communication between two cores. Using a single cache line for transfer and polling, the round-trip time was around 600 CPU cycles. Through careful placement of buffers, prefetching and polite polling, we could reduce the effective round-trip time to 300 cycles. This is in line with results from Baumann et al., where the authors report a round-trip time of 450 cycles for on-die communication for a similar AMD system [6]. The use of shallower cache hierarchies or write-through / inclusive caches could reduce this latency significantly.

This evaluation ignored the possibility that a transaction might wait while $DT$ satisfies other requests. In general, this delay can be ignored: Tiwari et al. showed that under high load, centralized allocators can actually perform better, due to decreased meta-data contention [42]; under low load, the delay is already negligible.

# B Channel/Delegation Pseudocode

Listings 1 and 2 show an implementation of simple send and receive primitives for small and large messages, as discussed in Section 3. If used from within a BEHTM transaction, these functions use ASF non-transactional loads and stores; if used from outside, they use plain, atomic memory accesses.

Delegating the execution of unsafe operations from within a BEHTM transaction to a service thread $DT$ (Section 4) can be accomplished through the code in Listing 3 – while the corresponding handler in $DT$ is shown in Listing 4. We present only code that handles a single, dedicated `TxChannel` in $DT$; however, the code remains similar if $DT$ handles multiple `TxChannels`: the blocking receive operation would need to be replaced by a select-style operation, and abort and commit handlers would need to be per-channel. The BEHTM transaction would use the respective SIGNAL functions from Listing 3 when it aborts / commits.

If only a single-word channel were allowed, the mechanism of Listing 2 could suffice, with the REQ and RV messages sent 63 bits at a time (31 bits at a time in 32-bit mode). While this seems onerous, reserving a bit to identify the sender of the previous message in the `TxChannel` is necessary, so that a TS(A) message is not sent by a transaction's abort handler while $DT$ is sending RV. We also note that in existing 64-bit systems, several high bits of pointers are unused. Thus, even with single-word channels, allocation and deallocation could be performed without introducing the need for REQOK/RVOK messages.

**Listing 1** Sending and receiving small messages with ASF, using non-transactional accesses and relying on well-formed ping-pong communication.

```
procedure SENDMSGASF(src, dst, type, d[])
    TxChannel[1...7] ← d[0...6]                ▷ store the payload
    TxChannel[0] ← (src, type)
end procedure

procedure RCVMSGASF(src, dst, type, d[])
    repeat
        (s, type) ← TxChannel[0]                                ▷ spin
    until s = src
    d[0...6] ← TxChannel[1...7]
end procedure
```

**Listing 2** Sending long messages by chunking and acknowledging each chunk.

```
procedure SENDLONGMSGASF(src, dst, type, d[])
    rem ← len(d)
    while rem > 0 do                ▷ store operands in temporary
        n ← max(rem, 7)
        tmp[0...(n−1)] ← d[i...(i+n−1)]
        (rem, i) ← (rem − n, i + n)
        sendMsgASF(src, (type, n, rem), tmp))
        rcvMsgASF(dst, t, ign)                          ▷ wait for ack
        assert(t = (type, OK))
    end while
end procedure

procedure RCVLONGMSGASF(src, dst, type, results[])
    i ← 0
    repeat                                  ▷ receive message in chunks
        rcvMsgASF(src, t, d)
        (type, n, rem) ← t
        results[i...(i+n−1)] ← d[0...(n−1)]
        sendMsgASF(dst, (type, OK), ∅))                 ▷ ack
        i ← i + n
    until rem = 0
end procedure
```

**Listing 3** Delegating an operation to $DT$, using the primitives from either Listing 1 or Listing 2.

```
procedure DELEGATESYNCTODT(id, operands[], res[])
    sendMsg(TX, DT, REQ, (id, operands))
    rcvMsg(DT, TX, type, res)
    assert(type = RV)
end procedure

procedure SIGNALABORTTODT
    sendMsg(TX, DT, TS, A)
    rcvMsg(DT, TX, type, ign)
    assert(type = TSOK)
end procedure

procedure SIGNALCOMMITTODT
    sendMsg(TX, DT, TS, C)
    rcvMsg(DT, TX, type, ign)
    assert(type = TSOK)
end procedure
```

**Listing 4** Handling delegation requests inside $DT$.

```
procedure HANDLEDELEGATION
    abortHnd ← ∅
    commitHnd ← ∅
    repeat
        rcvMsg(TX, DT, t, d)
        if t = REQ then
            (id, op) ← d
            if abortHandler(id) then
                abortHnd ←
                abortHnd ∪ (abortHandler(id), op)
            end if
            if commitHandler(id) then
                commitHnd ←
                commitHnd ∪ (commitHandler(id), op)
            end if
            res[] ← handle(id, op)
            sendMsg(DT, TX, RV, res)
        end if
    until t = TS
    if d = C then
        executeAll(commitHnd)
    else if d = A then
        executeAll(abortHnd)
    end if
    sendMsg(DT, TX, TSOK, ∅)
end procedure
```