Between All and Nothing–Versatile Aborts in Hardware Transactional Memory

Stephan Diestelhorst
ARM Ltd., Cambridge, UK
TU Dresden, Germany
stephan.diestelhorst@gmail.com

Martin Nowack
Christof Fetzer
TU Dresden, Germany
martin,christof@se.inf.tu-dresden.de

Michael Spear
Lehigh University, USA
spear@cse.lehigh.edu

Abstract

Hardware Transactional Memory (HTM) implementations are becoming available in commercial, off-the-shelf components. While generally comparable, some implementations deviate from the strict all-or-nothing property of pure Transactional Memory. Instead of trying to hide them, we lift these deviations to a simple *transactional resurrection* mechanism that can be used to accelerate and simplify both transactional and non-transactional programming constructs. We implement our modifications both architecturally and micro-architecturally in a detailed HTM proposal, without changes to system software and only light modifications to the existing HTM microarchitecture. We then show application of transactional resurrection in both transactional and non-transactional parallel programming: hybrid transactional memory; transactional escape actions; alert-on-update; and transactional suspend / resume.

Categories and Subject Descriptors C.1.2 [*Computer Systems Organization*]: Processor Architectures—Multiprocessors; D.1.3 [*Software*]: Programming Techniques—Parallel Programming

Keywords computer architecture, synchronisation, transactional memory, cross thread communication

1. Introduction

Originally proposed in 1993, Hardware Transactional Memory[11] (HTM) has at last gained traction with industry, and leading microprocessors have incorporated HTM support [13–15]. However, these products provide a much less exotic flavor of HTM than those proposed by researchers [26, 27]. They generally offer a comparable best-effort HTM with strong isolation, but very loose capacity specifications, as capacity is usually determined by size and organisation of the cache used to track the transactional working set.

Clearly, there is a gap between what the hardware provides now and in the near future,[1] and the compelling features suggested in academia. We show how to extend a basic HTM proposal to bridge this gap and bring features proposed in academia to product-grade HTM proposals.

Even though the various HTM proposals and forthcoming products have many similarities in their core feature set, on the periphery the proposals differ, for example how they treat the register state, and in the availability and design of mechanisms that allow code to escape through the transactional layer. Comparing, for example, Intel's Transactional Synchronization Extension (TSX) and AMD's Advanced Synchronization Facility (ASF), both provide best-effort transactional memory (Restricted Transactional Memory (RTM) in Intel's proposal), but differ in (1) the way they treat the snapshot / rollback of a transaction's register state, (2) non-transactional accesses from within a transaction, and (3) the availability of a minimum capacity guarantee.

TSX snapshots all registers on transaction start, and restores them automatically on abort; it also does not provide instructions to bypass the transactional mechanisms (e.g., loads within a transaction that are not tracked, or stores within a transaction that do not roll back). ASF provides the opposite: registers are not automatically saved and restored, but instead software needs to manually save live registers on transaction start and restore them on abort. Additionally, ASF allows programs to bypass the transactional mechanisms through the application of an existing instruction prefix to mark memory operations as *non-transactional*; these operations will appear to take effect immediately, rather than at the end of the transaction. A similar feature was present in the cancelled Rock processor [4].

We explore the different policies for register snapshotting and propose *transactional resurrection* as a lightweight mechanism which we use to synthesise features such as alert-on-update[34], escape actions[22] and transactional suspend / resume; thus achieving a rich transactional programming environment.

We focus on extending the HTM interface, but are careful not to increase hardware verification costs or require changes to existing system software. In particular, we do not extend the architected state of applications, and thus the operating system and hypervisor can remain oblivious of the extensions, e.g., when performing context switches. Our hardware modifications are non-invasive in nature and do not require any additional associative tracking structures, or other deep changes to the processor pipeline or the cache coherence / memory subsystem.

Our contributions in this paper are: we propose the mechanism of transactional resurrection that allows aborted hardware transactions to resume; we implement these mechanisms as four new instructions in a detailed architectural and micro-architectural HTM prototype (ASF); on top of these, we build transaction suspend / resume, escape actions, and multi-location alert-on-update. Finally,

[1] IBM POWER8 2014 [17], IBM zEC12 [2], Intel Haswell [39]

TRANSACT '15, June 15–16, 2015, Portland, OR, USA.

ACM X-XXXXX-XX-X/XX/XX.
http://dx.doi.org/10.1145/nnnnnn.nnnnnn

we evaluate performance overheads and demonstrate the functionality of our implementation in a full-system, cycle-level simulator.

The paper is structured as follows: first, we give a brief introduction to ASF in (Section 2) and detail our proposed hardware extensions (Section 3). We present the different higher level use cases for the extensions (Section 4,5,6) and conclude with an evaluation (Section 7) and related work (Section 8).

2. Background: ASF

For our design we extend AMD's Advanced Synchronization Facility (ASF) [6]. In this section we briefly review how ASF exposes core HTM functionality, as well as the unique aspects of ASF that we use to build a more robust programming environment.

ASF transactions are started with the `SPECULATE` instruction which creates a partial checkpoint of the thread state. `SPECULATE` also serves as the entry to an abort handler if a transaction fails to commit. The `COMMIT` instruction ends a transaction, making all transactional updates immediately and atomically visible to memory. Within a transaction, regular x86 `MOV` instructions and prefixed `LOCK MOV` instructions (which can be either loads or stores) are used to distinguish between immediate, irrevocable accesses that escape the transaction and transactional accesses (i.e., stores are buffered until commit, and loads are tracked in the cache). The polarity of the `LOCK` prefix is determined by selecting either `SPECULATE` or `SPECULATE_INV` to start the transaction. The former executes undecorated accesses non-transactionally and uses prefixes to mark transactional accesses, while the latter inverts the scheme and is more similar to TSX, Rock and other HTM proposals. In the remainder of this paper we will explicitly state the transactional property of accesses.

ASF provides strong isolation: transactions will detect conflicts with concurrent accesses, even when those concurrent accesses occur outside of a transaction. Conflicts are resolved through a simple requester-wins abort policy which always aborts the transaction that added the conflicting item to its working set first. When per-core private data caches are used to detect conflicts, this policy can be supported without any change to the underlying cache coherence protocol, thereby reducing the verification cost of transactional extensions to the ISA.

In case of an abort, ASF will undo any speculative memory writes, but will keep the processor registers and all other memory updates visible. The CPU redirects execution to the instruction following `SPECULATE` and provides an error code (in register `rax`) with information about the abort reason. The application should check the error code and branch to an abort handler that will take appropriate measures (e.g., back-off and restart the transaction). Aborts in ASF happen synchronously with the condition for the abort, and may occur between any two instructions in the transaction. As a "best effort" HTM implementation, additional causes of aborts include, but are not limited to, system calls, exceptions and interrupts (to include timer interrupts), and capacity/conflict cache evictions (i.e., due to the transaction's working set exceeding the size of the cache).

3. Resurrection–Aborts with Continuation

When an ASF transaction aborts, almost the entire register state is available to the abort handler. The only exceptions are the registers used to: convey the abort cause (`rax, rflags`); restore the stack pointer (`rsp`); and change the control flow to the instruction after the `SPECULATE` instruction (`rip`). If the values of these registers were made available, the abort handler could resume execution inside the transaction (ignoring for now that the abort would clear the transaction's working set).

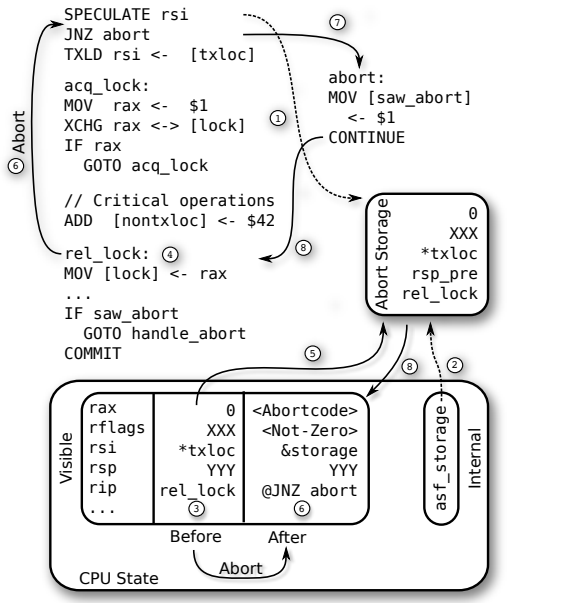

Figure 1. Basic Functionality of Abort with Continuation

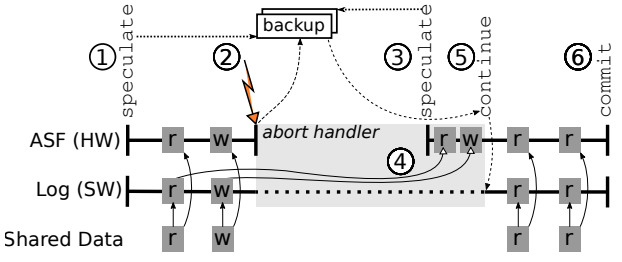

Figure 2. Suspend/Resume mechanism: (1) A hardware transaction is started (`SPECULATE` with "backup" as argument), but the transaction body is instrumented so that accesses will also be logged; (2) In case of an abort, ASF records the instruction pointer `rax` and executes the abort handler; (3) If the transaction can be recovered, the handler starts a new hardware transaction; (4) The working set is replayed; (5) The transaction resumes the normal execution (using `CONTINUE`); (6) The transaction commits its hardware transaction and resets its logs.

One option to place these register values is to use additional, new registers for either storing the old content of the overwritten registers, or conveying the necessary additional abort information and not use existing registers for that purpose. Both variants do increase the footprint of the architectural register state of applications. Therefore, operating systems and hypervisors would then have to be aware of these registers and save / restore them on context switches. To avoid affecting systems software, the register state must go elsewhere: we let the programmer allocate a buffer to hold the old values of the overwritten registers and provide the location of the buffer as a parameter to an extended `SPECULATE` instruction.

In Figure 1 we present the main interaction: `SPECULATE` is extended so that it accepts a memory buffer location parameter (1), looks up the virtual address and translates it to a physical address, and also checks write permissions to the location. Any page faults that could occur when accessing the buffer are thus already resolved before the transaction starts. In the event of nested transactions, the `SPECULATE` instruction ignores this parameter: Since ASF only

supports flat/subsumption nesting, there is no meaning or benefit to saving multiple register checkpoints.

The CPU keeps the resulting physical address in an internal register (2) and starts the transaction. The transaction executes, mutating the CPU's register state (3). In case of an abort (4), the processor first copies `rax`, `rip`, `rsi`, and `rflags` into the application-provided buffer (5), and then updates these registers and control flow to reflect the abort condition (6), with `rsi` additionally holding the buffer address. Furthermore, `rsp` will no longer be restored:[2] this prevents stack smashing due to signals or interrupt handlers running within the abort handler. The application code checks for aborts and branches to an abort handler (7). The abort handler can simply restore `rsi` and `rsp` from the the buffer pointed to by `rsi` and reproduce the original ASF abort functionality. However, it can also resume the code in the transaction by restoring all overwritten registers from the buffer (8). Since existing assembly primitives cannot restore all registers without overwriting an additional temporary register, we provide a new `CONTINUE` instruction that performs a simple micro-code sequence to restore the registers.

To simplify the ASF interface, we also provide `RDINVMODE` and `WRINVMODE` instructions. These allow the programmer to detect and change the behaviour of `MOV` and `LOCK MOV` within transactions.

3.1 Handling synchronous aborts asynchronously

ASF transactions are aborted immediately in case there is a reason for it (e.g. contention). This provides strong isolation guarantees for transactions but puts the burden on a programmer to reason about correctness, as a transaction might be aborted at every machine level instruction. Especially, if non-transactional modifications are made inside a transaction (e.g. memory allocation), a correct state has to be preserved. With the proposed extensions, it is easy to translate HTM's synchronous aborts into asynchronous aborts. The abort handler will simply set a thread local variable (`saw_abort` in the example) signalling an abort, and will then execute `CONTINUE`. The code inside the transaction will query the variable at suitable intervals and then can handle the earlier abort asynchronously.

3.2 Cost

From a hardware perspective, the required changes atop ASF are minimal: memorising an additional pointer during the execution of the transaction is easily achieved in either an internal register or in scratchpad memory. The changes to `SPECULATE`, aborts and the new `CONTINUE` instruction can be effectively coded in microcode. Pre-checking the allocated buffer location for a proper virtual to physical address mapping when executing `SPECULATE` ensures that the processor will always be able to store the continuation information and no abort page fault deadlock can occur.[3]

Implementing our changes on top of other industry HTM proposals, such as Intel TSX, requires slightly more effort. Our additions to the abort handler and new instructions remain lightweight, but non-transactional accesses are usually absent from the currently specified proposals. Discussion with hardware designers showed, however, that supporting non-transactional accesses is neither overly complex nor requires a lot of silicon, but instead has been postponed due to lack of demand and semantic corner cases. We show how non-transactional accesses can be used in a beneficial, controlled, and semantically clear manner.

Best-effort TMs, such as ASF, usually do not virtualise the transactional resources on context switches; instead, they abort on-

[2] The old value of `rsp` from `SPECULATE` is stored in the buffer, instead.

[3] Since page faults cause aborts, we would otherwise risk deadlock from the following cycle: abort transaction → store to buffer to keep overwritten register values → page fault due to buffer access → abort transaction because of a page fault

```
1:  procedure SAHTM_START                ▷ Start a software-assisted HTM transaction
2:      buf_1, buf_2 ← malloc()
3:      (buf_1.other, buf_2.other) ← (buf_2, buf_1)   ▷ Cross-link buf_1 and buf_2
4:      log ← ()
5:      SPECULATE buf_1                  ▷ Start the hardware transaction
6:      error ← rax
7:      if error ≠ 0 then
8:          return SAHTM_ABORT(rsi, error)   ▷ Handle errors and resume if possible
9:      return IN_TX

10: procedure SAHTM_ABORT(s, error)      ▷ Handle aborts, s holds the resume state
11:     push regs
12:     retry:
13:     if error.type = CONTENTION then
14:         rsp ← s.abort_rsp                ▷ Do a full abort
15:         return ABORTED
16:     else if error.type = FAR then    ▷ Resurrect after interrupts, page-faults
17:         (error, s) ← SAHTM_RESURRECT(s)  ▷ Successful resurrection does not return
18:         goto retry
19:     else if error.type = ILLEGAL then ▷ Emulate syscalls etc.
20:         s ← EMULATE(s)
21:         (error, s) ← SAHTM_RESURRECT(s)  ▷ Resurrect after successful emulation
22:         goto retry
23:     else
24:         ...                              ▷ Handle other abort reasons

25: procedure SAHTM_COMMIT
26:     COMMIT                           ▷ Just commit the HTM transaction
27:     free(buf_1, buf_2)
```

Figure 3. Handling the life-cycle of HTM transactions with suspend / resume extensions. Applicable to both software-assisted and hardware-extended suspend / resume HTM.

going transactions. In our design we do the same, thereby avoiding resource-hungry virtualisation. Note that the full architectural CPU state can be reconstructed by the abort handler even after a context switch: saving the registers clobbered by the hardware abort in virtual memory lets them survive the context switch, and the OS's existing context save / restore mechanism naturally takes care of all other registers.

4. OS-transparent Transaction Suspend / Resume

Suspend / resume appears in the new IBM POWER8 HTM proposal [13], but relies on additional registers and special handling in the OS when dealing with suspended transactions. Suspending a transaction is useful for tolerating short execution of other code, for example dealing with hardware interrupts, syscalls or exceptions. We now show how to enable full transaction suspension and resume in ASF without changes to the OS, by building upon the simple extensions we proposed in Section 3.

Our general approach is to let the transaction abort instead of suspending it, and then offer a mechanism to *resurrect* the aborted transaction when resumption is necessary. From an architectural perspective, this means that suspended transactions are the same as aborted transactions, and thus do not require special treatment by the OS. Transaction-aware OS can use transactions without concerns for the application's usage of the transactional memory resources and legacy OS can handle applications using HTM. Our extended abort mechanism allows full access to all registers in the

```
28: procedure SAHTM_RESURRECT(s)                ▷ s holds the resume state
29:     if s.resume_ip ∈ lines(31 – 36, 41 – 48) then ▷ Aborted while resuming
30:         s ← s.other                          ▷ Squash abort recursion
31:     SPECULATE s.other                        ▷ Start HTM container transaction
32:     error ← rax
33:     if error = 0 then                        ▷ Successful start of HTM container
34:         SAHTM_REPLAY()                       ▷ Replay transactional working set from SW log
35:         pop regs
36:         CONTINUE s                           ▷ Restore full state and return to resurrected transaction.
37:     else                                     ▷ Abort in the resurrected transaction
38:         s ← rsi                              ▷ Update resumed state from new abort site
39:         s.abort_rsp ← s.other.abort_rsp
40:         return (error, s)                    ▷ Outer logic handles abort condition and retries

41: procedure SAHTM_REPLAY                       ▷ Replay and validate transactional accesses
42:     for (addr, val, rw) ∈ log do             ▷ from the SW log
43:         if rw = READ then
44:             txload tmp ← [addr]              ▷ Add to read set
45:             if val ≠ tmp then                ▷ and validate value
46:                 ABORT CONTENTION             ▷ Use HTM abort to unravel validation failure
47:         else
48:             txstore [addr] ← val             ▷ Redo stores

49: procedure SAHTM_TXLOAD(addr)                 ▷ Software-assisted read barrier
50:     log ← (log, (addr, ∅, READ))             ▷ Append a sentinel protecting against an abort
51:                                              ▷ between line 52 and 53 missing replay of addr
52:     txload val ← [addr]                      ▷ Add to HTM read set
53:     (log, (addr, ∅, READ)) ← (log, (addr, val, READ))   ▷ Update with proper read value
54:     return val

55: procedure SAHTM_TXSTORE(addr, val)           ▷ Software-assisted write barrier
56:     log ← (log, (addr, val, WRITE))          ▷ Append to log
57:     txstore [addr] ← val                     ▷ Add to HTM write set
```

Figure 4. Resurrection and replay of aborted transactions, logging read / write barriers. For simplicity, we omit handling different sizes in SAHTM_TXSTORE and SAHTM_TXLOAD.

abort handler, so that it can restore the transaction's register state exactly as it was at the time of the suspension (abort).

In Figure 2, we depict the time line of a suspend / resume cycle. Resume / resurrection is initiated in the abort handler when the suspend condition has been handled, for example when control has passed back to the application after invocation of an hardware interrupt handler. A new transaction is started with `SPECULATE` with a new buffer for storing the abort state and `CONTINUE` then restores the suspended transaction's register state and resurrects it. Figure 3 shows the general behaviour in pseudo-code.

Clearly, making available all register state of the transaction to the abort handler is not enough to resurrect the aborted transaction, because the transactional working set in memory is rolled back in ASF upon any abort. We present two options to deal with resurrecting the transactional working set: (1) tentatively keeping transactional state across aborts in hardware, or (2) adding minimal logging instrumentation in software.

Inspecting the different abort reasons, we find that not all of them require immediate roll-back of the working set. In particular, for aborts not caused by violations of the integrity of the working set (i.e., aborts other than (certain types of) contention

```
1: procedure MESSYHTM_RESURRECT                  ▷ Resurrect with additional HW support
2:     if s.ip ∈ lines(4 – 8) then
3:         s ← s.other                           ▷ Squash abort recursion.
4:     SPEC_RESURRECT s.other                    ▷ Start HTM transaction using working set still in cache
5:     error ← rax
6:     if error = 0 then                         ▷ Successful resurrection
7:         pop regs                              ▷ No need to restore the read / write set
8:         CONTINUE s                            ▷ Restore full state and return to resurrected transaction.
9:     else                                      ▷ Abort in the resurrected transaction
10:        s ← rsi                               ▷ Update resumed state from new abort site
11:        s.abort_rsp ← s.other.abort_rsp
12:        return (error, s)                     ▷ Outer logic handles abort condition and retries

13: procedure HTM_TXLOAD(addr)                   ▷ Read barrier for resurrection with HW support
14:     txload val ← [addr]
15:     return val

16: procedure HTM_TXSTORE(addr, val)             ▷ Write barrier for resurrection with HW support
17:     txstore [addr] ← val
```

Figure 5. Hardware support in the caches to tentatively keep the aborted transaction's working set significantly simplifies the resurrection logic.

or capacity evictions), it may be possible to keep the transactional state tentatively in the cache and make it available to the resurrecting `SPECULATE` / `CONTINUE` pair. We propose *Messy-HTM* that extends the `SPECULATE` instruction so that the cache can distinguish between a request to start with a clean transactional state (`SPECULATE`) and attempts to reactivate the old transactional state and aborts when this fails (`SPEC_RESURRECT`). To keep the OS unmodified, the speculative state is cleared when the processor sees an event that causes a TLB flush, usually indicating a context switch. Nevertheless, the proposed suspension / resurrection mechanism can tolerate brief kernel invocations, for example due to interrupts or system calls from within the transaction.

If lazy clearing of transactional state proves too complex for an HTM implementation, or if support for surviving full context switches is desired, we can employ a lightweight hybrid TM approach of *Software-assisted HTM* (SAHTM): transactional accesses can be manually logged in a thread local buffer, which is used to validate and replay the hardware transaction upon resurrection.[4] We store in the buffer transactionally read and written values, the associated addresses, access sizes and access types (read / write). The buffer is updated during transactional execution by using *non-transactional* stores. Since the HTM is used to provide proper conflict detection and versioning, the log is append-only and never read during normal operation: buffered updates that are stored in the log are also performed as part of the transaction, and loads that are tracked in the log need not be validated, since these loads are also part of the transaction's working set. To reduce the overhead of logging, we combine the address and meta-data into a single 64 bit word (note that virtual addresses are only 48 bit wide in the current x86-64 specification). We contrast the details for resurrection and the respective read and write barriers in Figure 4 with SAHTM and in Figure 5 with Messy-HTM that keeps the cache content available. In combination with full hybrid TM systems em-

[4] Note that software and hybrid TM systems already require this instrumentation, and that compiler support for automatically adding this instrumentation is available in several production-grade compiler frameworks.

ploying STM for large transactions, SAHTM's logging mechanism allows seamless transition from SAHTM to STM execution without requiring an abort: the SAHTM's access log is replayed into the STM's tracking data structures. In total, we add points on the performance vs capacity / functionality spectrum. In Section 7 we will quantify the performance characteristics of each of these options.

4.1 Cost

There is no additional hardware cost associated with the logging-based SAHTM variant of resurrecting transactions, and even keeping the transactional working set in the caches with Messy-HTM requires only minimal changes. The cache remains unchanged from an existing cache-based HTM design, especially if that HTM already offers non-transactional accesses, as is the case with ASF. We change the handling of aborts: the core will not clear the transactional state in the caches when encountering an interrupt, exception, or instruction that calls into the OS. Furthermore, the core needs to memorise the fact that there is a suspended transaction and detect TLB flushes. The cost for these modifications in terms of silicon real-estate is small, but they incur a design and verification cost. We acknowledge the cost and offer an intermediate update step with our logging-based SAHTM approach.

The OS does not need to manage the transactional state of the application with our extensions, because all detection of context switches happens in hardware conservatively and is handled by fully aborting the transaction. Because the register state of each application reflects the aborted state already, no TM-specific update of the architectural state has to occur during / for a context switch. Accessing transactionally written data when the transaction is suspended is dangerous in all suspend / resume proposals, because hardware may need to drop the transactional updates to ensure the consistency of the working set. Our SAHTM approach side-steps the issue by hiding the transactional updates from the invoked OS routines. If data needs to be reliably transferred into the OS handler through memory, ASF's non-transactional stores provide a safe way to protect against spontaneously disappearing working sets. It is also possible to instrument library codes so that they perform a lookup in the log. This is reminiscent of techniques for achieving open nesting in BEHTM [19]. We will show a way to safely handle state transferal into the kernel in Section 5.

5. Escape Actions from Hardware Transactions

ASF already allows individual loads and stores to escape from a transaction. Composing longer code blocks escaping these mechanisms (as in [40]) is complicated due to the synchronous nature of aborts in ASF; whenever a condition for abort is detected (to include concurrent memory conflicts and timer interrupts), control flow can transfer from the middle of a basic block into the abort handler. This is usually not an issue with transactional code since all side-effects are tracked and rolled back. However, interrupting an escape action while it has not finished executing can leave escaped data in an inconsistent state.

Given our mechanism for suspend/abort from Section 4, it is straightforward to provide support for escape actions as well. For simplicity, we adhere to the principles set forth for delegated escape actions [19], namely that an escape action's accesses are disjoint with respect to the calling transaction's read and write sets.

Without loss of generality, we assume that escape actions consist entirely of non-transactional code (i.e., they use `RD/WRINVMODE` at their entry and exit to set and restore this status). To support escape actions, we update a thread-local field F prior to beginning the escape action. Should an abort occur during the escape action, the abort handler first checks F: if it is set, the handler memorises information about the abort in another field H, and then uses a `CONTINUE` to immediately resume the escape action. In this manner, the (non-transactional) escape action code will not be aborted while holding locks, or while at some point where invariants may not hold. Upon completion of the escape action, the code registers any undo actions related to the escape action, clears F, and checks H. If H indicates that an abort occurred during the escape action, the program uses the additional information saved by the handler to complete the abort, closely resembling an explicit abort in a software TM implementation.

In the event that the escape action requires a context switch or system call, we seamlessly transition to a more heavyweight suspend/resume operation. This may require the transaction to abort and restart in SAHTM mode, if accesses have not been logged. However, such a transition is only necessary if the system call cannot be emulated; otherwise, the `CONTINUE` would immediately return to the OS trap instruction, which would abort the newly started transaction and return to the abort handler. By switching on the fly, we can suppress aborts for lightweight escape actions, while still supporting escape actions that must be executed from a non-transactional context.

5.1 Hardware Cost and OS Interaction

There is no additional hardware cost to provide support for escape actions. With respect to OS interaction, the common case again requires no support. However, run-time libraries that ought to run as escape actions will require wrapper code to manage the F and H flags. If an escape action must access state modified by the transaction, then the action must be rewritten to check the access logs managed by the transaction, and the transaction will require SAHTM instrumentation. More complex software-based techniques are possible, wherein concurrent transactions are blocked and the transaction executing the escape action temporarily becomes irrevocable. In the absence of workloads requiring such functionality, the cost of this approach is likely too high.

Note, too, that it is not necessary to execute every escape action as a non-transactional operation. With the new instructions to control whether `LOCK` prefixes indicate transactional or non-transactional accesses, binaries (such as libc functions) may be called in either an escaping fashion or through making all their memory references transactional. By storing the current mode of the transaction (inverted / non-inverted, attempt continue / abort) in a thread-local variable, we can ensure that the right strategy is employed in the abort handler, and that the best approach is taken for each library function.

6. Multi-Location Alert-on-Update

Alert-on-update (AOU) is a mechanism that uses transactional read set tracking to generate user-level signals upon certain cache evictions [34]. To synthesise alert-on-update on top of our extended abort behaviour, we begin an ASF transaction via the `SPECULATE` command, use transactional (`LOCK`-prefixed) loads in place of AOU loads, and keep all other memory accesses of the program non-transactional. We also non-transactionally manage a record of all AOU-loaded locations (Figure 6). Whenever a monitored (AOU) location is written to by another core and evicted from the cache, the ASF transaction aborts and jumps to its abort handler, which serves as (or chains to) the alert handler (replacing the abort handling in Figure 3). Note that changes to program state will not roll back on transaction abort, since we have chosen in this case for the default behaviour of loads and stores to be non-transactional.

After the handler finishes resolving the alert, it starts a new transaction, re-adds the monitored location(s) to the working set, and continues execution at the previously aborted location through a `CONTINUE` instruction. Due to the overlapping nature of starting a transaction before executing `CONTINUE` to restore the state of the preceding transaction, we must take care to use alternating buffers

```
1: function ALOAD(address)    ▷ Adds an alert on update of location address
2:     repeat
3:         val_1 ← [address]
4:         locs ← locs \ (address, *) ∪ (address, val_1)    ▷ Add value early to prevent data race
5:         txload val_2 ← [address]
6:     until val_1 = val_2    ▷ Ensure that this was a race-free ALOAD
7:     return val_1
```

Figure 6. Implementing alert-on-update with ASF.

for the storage of the clobbered registers. To prevent unbounded abort recursion, we flatten aborts in the overlap region. The abort handler will also be invoked for other reasons than changes in the monitored location(s), such as syscalls and timer interrupts, but those cases can be discerned through the abort condition codes presented to the abort handler by ASF. Continuing the transaction is usually enough to continue execution, but some cases require simple emulation of instructions illegal within transactions (such as I/O-related system calls). Often, the mechanisms discussed in previous sections suffice for this emulation. In other cases, lightweight instrumentation is required to (a) `COMMIT` the `SPECULATE` operation, then perform the operation, and then begin a new `SPECULATE` and restore all AOU loads. Note that this is simpler than suspend/resume, because there is no transactional state that must be protected during the (escaped) syscall.

6.1 Privatisation-safe STM with AOU

To demonstrate the utility of AOU, we consider its use to strengthen the correctness guarantees of an STM algorithm without adding overhead. In general, language-level implementations of transactional memory (TM) require the TM implementation to be privatisation safe [1]. Roughly, this means that execution with transactions appears equivalent to an execution in which all critical sections are protected by a single global lock [20]. This, in turn, boils down to two problems [35]: when committing a transaction T that logically transitions some region of memory R to a state in which other threads can no longer access R transactionally, T must be sure that (a) any transaction that committed or aborted before T committed must not still be cleaning up its changes to R[5], and (b) any transaction still running will not continue to use data in region R. These two problems are sometimes called the "delayed cleanup" and "doomed transaction" problems.

The most general solution to both problems is a heavyweight quiescence mechanism, in which every committing writer transaction must wait for all concurrent transactions to commit or abort *and clean up* before it departs from its commit function. Decoupled solutions to the problem tend to scale better, but these solutions rely on polling to solve the doomed transaction problem: when T commits, concurrent doomed transaction D may be in the midst of accessing R, and will not determine that it should abort until its *next* access of shared memory. The problem with this approach is that when T wishes to deallocate R, it cannot prevent concurrent accesses by D. The only known solution in this case is to change the allocator, so that T's deallocation is deferred until D completes [12]. This mechanism, also popular in RCU synchronisation, can result in an unbounded delay between when T commits and when R is finally reclaimed. Consequently, production STM implementations choose the quiescence approach.

[5] We use the term "clean up" to refer to write-back in STM implementations that use commit-time locking, and also to refer to undo operations at abort time in STM implementations that lock and modify memory before reaching their commit point.

Our implementation of AOU enables the use of decoupled validation without incurring the risk of unbounded delay during reclamation. The key observation is that if transactions use AOU to monitor the notification location that they formerly polled, then they will be notified *immediately* when it changes, due to transactional abort. The notified thread can then ensure its validity with respect to the newly committed transaction. If it remains valid, it can resume; otherwise it will abort. Crucially, the interruption, validation, and abort of a transaction will occur between when T commits and when D might next access R. That is, if T deallocates R, it cannot affect D so long as D's validation does not access R. In practical terms, this prohibits algorithms that use value-based validation [8, 24], but otherwise carries no cost.

In prior work on AOU [34], the AOU hardware monitored some subset of read locations and metadata to avoid validation. In contrast, we use the AOU mechanism as a polling-free, low-latency, immediate cross-core (cross-thread) communication mechanism that invokes the STM's validation handler upon the commit of every writing transaction; this requires AOU tracking of only a *single* location, and is thereby beneficial for even the most capacity-constrained HTM implementations. Since our AOU implementation is built atop our transparent suspend/resume mechanism, it is practical: system calls and quantum expirations will cause a transaction to resume from inside its validation handler, and aborts during calls to lock-based libraries (such as `malloc`) can be delayed safely. Furthermore, the use of AOU-based notification simplifies the STM algorithm, eliminating some comparisons for corner-case behaviours.[6]

6.2 Hardware Dependence

In contrast to the transactional suspend and resume mechanism from Section 4, with AOU we employ the transactions as an auxiliary wrapper to non-transactional code. Therefore, we expect to see non-transactional accesses as the norm, which is well reflected by ASF's (non-inverted) `SPECULATE` instruction. For hardware implementations with small HTM implementations, programming with AOU will side-step the capacity limitations of the HTM, while still gaining some benefits relative to a pure STM library.

As before, the OS / hypervisor can remain oblivious of usage of AOU in application code. This is an improvement over the original alert-on-update proposal [34]. However, in some cases it may be necessary to wrap system calls as escape actions that run outside of a transactional context, in order to prevent infinite aborts at the trap instruction. As discussed above, these escape actions are simpler than those in Section 5: since we aren't using the HTM for data versioning, we can simply commit the `SPECULATE` region, run the escape action, and then start a new `SPECULATE` region.

7. Evaluation

In this section, we evaluate the performance of our extensions. All experiments were performed on Marss86, a cycle-accurate x86 simulator [25]. We extended Marss86 with ASF support based on PTLsim-ASF [5],[7] and then added the features discussed in this paper. Our evaluation of AOU-enhanced STM uses benchmarks from the RSTM [33] open-source library, and we evaluate suspend/resume performance in the TinySTM [5] toolchain. Experiments ran for 10,000 successful transaction commits per thread, and all re-

[6] These simplifications, while useful, are lengthy and limited in novelty. We intend to distribute them via open-source channels, but due to limited space cannot include code listings in this submission.

[7] We ported and heavily extended code from PTLsim-ASF (`https://github.com/stephand/ptlsim/tree/ptlsim_asf`) to work with the new memory hierarchy in Marssx86 and made changes available at `https://bitbucket.org/stephand/marss86-asf`.

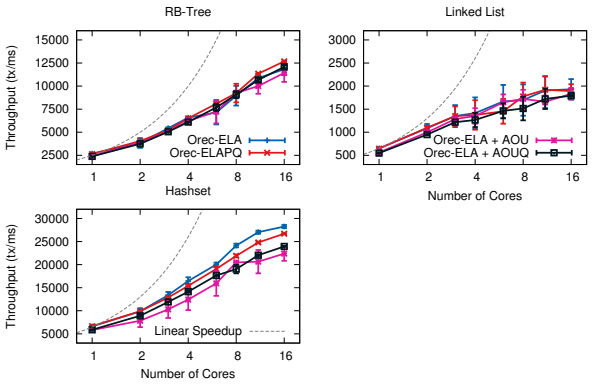

Figure 7. Throughput for RSTM OrecELA with AOU-enhanced privatization safety.

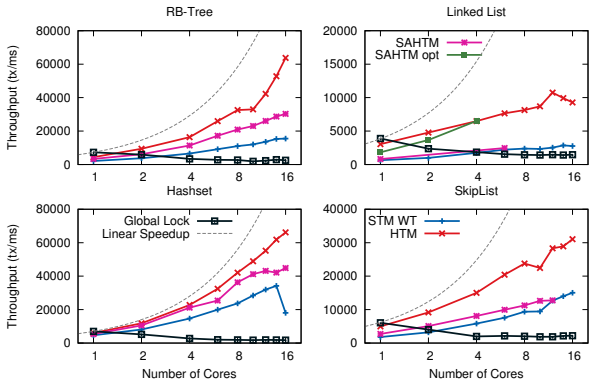

Figure 8. Throughput for Intset benchmark using different TM implementations

sults are the average of three (RSTM RB-Tree and Hashset) or six (RSTM Linked List) trials. Our simulated machine features a modern processor with 16 out-of-order cores, with per-core L1 and L2 caches and a shared L3 cache, and realistic default sizes for them and DRAM. All experiments run in full-system simulation, with Ubuntu Linux 12.04 LTS. All RSTM code was compiled with GCC 4.6.3 and the "-O3 -flto" optimisation options. The TinySTM-based experiments use the Clang version of DTMC compiler [5] using LLVM 3.2 with optimisation "-O3 -flto".

7.1 Alert on Update for Privatisation-Safety

We enhance two variants of RSTM's OrecELA STM implementation with our AOU implementation outlined in Section 6: *OrecELA* and *OrecELA + AOU* use ordered commit departure, *OrecELAPQ* and *OrecELA + AOUQ* perform quiescence to achieve privatization-safety. The AOU-enhanced variants do not require delayed reclamation and therefore are not prone to unbounded delays between memory free and actual availability of the memory. Instead, they can immediately reclaim memory at transaction commit without inducing segmentation violations in doomed readers.

We use three data-structures implementing the integer set interface that are provided with the RSTM source code: ListBench, a singly linked list; HashBench, a hash table; and TreeBench, using a Red-Black tree to store the set. We vary the thread count, fix the total number of transactions and keep other parameters at their defaults (set size of 256 elements, pre-filled to 128 elements, value range 0 - 255, update rate 66%).

Comparing the performance for the three data structures in Figure 7, we find that, overall, performance of the four solutions is comparable. The linked list produces large levels of contention that cause significant jitter in our results at higher thread counts. The AOU-based STMs change the timing of validation due to their synchronous signalling. In the linked list, abort characteristics change for long transactions, because a short committing writer causes an immediate validation and abort of a long, uncommitted transaction that has traversed the list. In the non-AOU STM variants, that long transaction would also be doomed but detect the conflict only when it tries to commit.

In the hash set experiment, all transactions are small and conflict rarely. For the AOU-enabled cases, the frequent writers force many AOU-induced revalidations causing overhead for the short transactions. The net effect is an increase of the effective length of the transactions. The present performance delta reflects this, but we argue that the additional application-level guarantees outweigh the performance impact of this application worst-case.

In summary, we show that our multi-location AOU prototype (based on the extensions to our best-effort HTM ASF from Section 3) works with small overhead and can be used to enhance an STM library. We found that some work in the STM logic was necessary to adapt it to handle the AOU-induced notifications arriving at arbitrary code locations. To mitigate, we selectively switched between synchronous and asynchronous handling (see Section 5).

7.2 Transaction Suspend / Resume

For testing the transaction suspend / resume with SAHTM, we base our work on the HyTM implementation in TinySTM which uses plain ASF HTM with minimal additional software support to implement serial irrevocable execution and memory management [5]. We extend read and write accesses to additionally log accessed addresses and values as described in Section 4. To reduce memory pressure and to keep the log overhead small, we encode additional meta data (size of the accessed data, read/write operation) in the upper 8 bits of the address part of the log entries.

TinySTM contains four benchmarks implementing the integer set interface: linked list, skip list, hash set, and red-black tree. We evaluate each with an initial capacity of 256 elements and an update rate of 20%. Figure 8 shows the throughput achieved for an increasing number of cores for those benchmarks. We compare four different transactional memory implementations: *Global Lock* which uses a single global lock for synchronisation - no accesses inside a transaction are instrumented; *STM WT* a state-of-the-art write through implementation for Software TM [28]; *HTM* the hardware TM implementation using ASF as described above; and our new implementation, *SAHTM*, that adds the logging read/write barriers. *SAHTM opt* is a proof-of-concept *hand-optimised* binary that bypasses shortcomings in LLVM's optimisation passes, in particular hoisting loop invariants across barriers and handling inline assembly with memory operands. We manually performed these optimisations on the SAHTM binary and briefly show first results for linked list.

Looking at the throughput and transactional statistics (not shown due to space limitations), we find that the RB-Tree and Hash Set scale well, and HTM and SAHTM behave very similarly for all core counts. The difference in throughput is due to the additional overhead of the read/write barrier implementation. For higher core counts, (> 8) SAHTM's scalability shows a more similar behaviour to STM. The reason for that is the additional

overhead of the barriers and the need to replay the logs in case of contention.

Both Linked and Skip List are a high-intensity workload for all of our TM implementations: the linear scanning loop exposes every bit of added overhead, due to very little other logic and predictable access patterns. SAHTM has a slight performance advantage over the STM implementation but the logging overheads are much more exposed than in RB-Tree and Hash Set. HTM scales better, because its list traversal loop is extremely small and the additional logging in SAHTM consumes memory and core execution bandwidth–causing significant slowdown. The major reason for SAHTM's overhead is missed optimisation opportunities by the compiler (appropriate hoisting and reuse of the pointer to the log structure instead of fetching them for each iteration in the loop) adding superfluous instructions and extra memory traffic. Our manually optimised prototype (SAHTM opt) for Linked List shows the potential benefit from these optimisation which should be performed by the compiler. This implementation is almost on par with the HTM implementation for the thread counts we managed to test.

In summary, our suspend/resume experiments show that it is indeed possible to implement transaction suspend/resume through resurrection of aborted transactions in an OS-transparent way. For data-structures with complex access patterns, the additional logging instructions and memory traffic are apparently effectively hidden in branch mis-predictions, cache misses and available instruction-level parallelism. We see strong hints that carefully controlling the optimisation in our framework can significantly drive down the execution resource demand of our logging code and thus can have low overheads also in simple data access patterns.

8. Related Work

For nearly a decade, researchers have been exploring mechanisms for exploiting bounded HTM resources in more robust and programmable ways. One of the earliest proposals, from Zilles and Baugh [40], introduced suspend/resume as a mechanism for allowing hardware transactions to avoid the size constraints of HTM when executing operations that either (a) are known to never cause conflicts, or (b) are best served with other concurrency control mechanisms (e.g., memory allocation). In that proposal, the hardware still controlled the execution of the transaction, with their extensions serving only as a means of temporarily suspending the transaction. Furthermore, without ASF-style HTM resources, this work required significant changes to the underlying HTM, whereas our implementation can leverage existing ASF support to require only a minimal amount of additional hardware and software extension. Transaction escape actions [22] provided a similar feature in LogTM, though again there was a noticeable hardware cost.

A more aggressive approach to exploiting bounded HTM is exhibited by the many hardware accelerated software TM (HASTM) systems. These proposals typically extended a traditional ISA with features resembling TM hardware, most notably mechanisms for tracking locations accessed within a region [30, 34, 36]. While these features closely resembled those of more complete HTM proposals, the control of transactions was fully delegated to a software library. Additional proposals offered programmable control of data versioning and buffering [21, 31, 32]. This offered low enough overhead to be competitive with full HTM implementations, but typically with less hardware complexity. At the extremes, Casper et al. showed that an out-of-core FPGA-based prototype could deliver strong transactional performance [3], and Carouge et al. showed that HTM resources could make an existing STM algorithm lock-free without affecting performance [10].

Our work on hybrid TM is inspired by a wide variety of algorithms proposed in the literature. These algorithms attempt to build a runtime system in which some transactions are fully controlled by software, and others accelerated by hardware. The benefit of such a system is that there is a graceful fallback for those transactions whose memory accesses or running times extend beyond the limits of the HTM subsystem. However, small, short transactions must, in turn, sacrifice some performance in order to be compatible with these software transactions. Initial hybrid systems focused on correctness and non-blocking progress [16], after which the focus turned to systems in which transactions operated in distinct modes (i.e., software-only, hardware-only, and serial) [5, 18]. Later works showed that true concurrency between hardware-controlled and software-controlled transactions was possible, but that specific characteristics of the hardware (most notably the availability of non-transactional loads and stores within the hardware transaction) was critical to achieving good performance [9, 29]. This paper builds on prior work by showing that minor extensions to the HTM can simplify the implementation of such systems without affecting performance.

Several groups have also explored the use of HTM resources for purposes orthogonal to scalable concurrent execution of language-level transactions. The original AOU paper [34] proposed several uses of AOU outside of TM implementation, such as for reducing the cost of polling in event-based systems and implementing a limited form of active messages [37]. Neelakantam et al. [23] showed that hardware TM extended with a self-abort instruction could be used by compilers for speculative unsafe optimisation. Most recently, Unsal et al. have shown that HTM resources could be used both to detect and prevent transient faults during sequential execution [38], and as a mechanism for lowering power consumption by running a processor at an extremely low voltage, and then using transactional rollback and recovery to compensate for faults that occur during execution [7].

9. Conclusion

In this paper, we presented small modifications to the ASF HTM proposal that change abort handling to allow transaction resurrection, and added two instructions for explicitly managing the polarity of transactional / non-transactional accesses. With only these two minor extensions, neither of which requires extensive hardware verification or changes to cache structures and protocols, we showed that Alert-on-Update, Escape Actions, and Suspend / Resume can all be supported in an otherwise relatively simple hardware TM. We believe that our modifications lie in the same complexity realm as the differences between the various HTM industry proposals and thus can be implemented in actual hardware, for example as an extension to first generation HTM support. This is a promising direction that can turn these *synchronisation* extensions into *synchronisation and speculation* extensions that support a rich transactional programming environment.

Acknowledgments

Stephan received funding from the European Community's Seventh Framework Programme [FP7/2007-2013] under the ParaDIME Project, grant agreement No. 318693, and developed related, initial ideas while employed at Advanced Micro Devices, Inc. At Lehigh University, this work was supported by the US National Science Foundation under grant CNS-1016828 and CCF-1218530. Martin received funding by Deutsche Forschungsgemeinschaft (grant agreement No. FE 1035/1-2.)

References

[1] A.-R. Adl-Tabatabai and T. Shpeisman (Eds.). Draft Specification of Transactional Language Constructs for C++, Aug. 2009. `http://software.intel.com/file/21569`.

[2] J. Brewer. IBM Unveils zEnterprise EC12, a Highly Secure System for Cloud Computing and Enterprise Data. `http://www-03.ibm.com/press/us/en/pressrelease/38653.wss`, Aug. 2012.

[3] J. Casper, T. Oguntebi, S. Hong, N. Bronson, C. Kozyrakis, and K. Olukotun. Hardware Acceleration of Transactional Memory on Commodity Systems. In *Proceedings of the 16th International Conference on Architectural Support for Programming Languages and Operating Systems*, Newport Beach, Calif., Mar. 2011.

[4] S. Chaudhry, R. Cypher, M. Ekman, M. Karlsson, A. Landin, and S. Yip. Rock: A High-Performance Sparc CMT Processor. *IEEE Micro*, 29(2):6–16, March–April 2009.

[5] D. Christie, J.-W. Chung, S. Diestelhorst, M. Hohmuth, M. Pohlack, C. Fetzer, M. Nowack, T. Riegel, P. Felber, P. Marlier, and E. Riviere. Evaluation of AMD's Advanced Synchronization Facility within a Complete Transactional Memory Stack. In *Proceedings of the EuroSys2010 Conference*, Paris, France, Apr. 2010.

[6] J. Chung, L. Yen, S. Diestelhorst, M. Pohlack, M. Hohmuth, D. Grossman, and D. 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., Dec. 2010.

[7] A. Cristal, O. Unsal, G. Yalcin, C. Fetzer, J.-T. Wamhoff, P. Felber, D. Harmanci, and A. Sobe. Leveraging Transactional Memory for Energy-efficient Computing below Safe Operation Margins. In *Proceedings of the 8th ACM SIGPLAN Workshop on Transactional Computing*, Houston, TX, Mar. 2013.

[8] L. Dalessandro, M. F. Spear, and M. L. Scott. NOrec: Streamlining STM by Abolishing Ownership Records. In *Proceedings of the 15th ACM Symposium on Principles and Practice of Parallel Programming*, Bangalore, India, Jan. 2010.

[9] L. Dalessandro, F. Carouge, S. White, Y. Lev, M. Moir, M. Scott, and M. 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., Mar. 2011.

[10] 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., Sept. 2010.

[11] M. P. Herlihy and J. E. B. Moss. Transactional Memory: Architectural Support for Lock-Free Data Structures. In *Proceedings of the 20th International Symposium on Computer Architecture*, San Diego, Calif., May 1993.

[12] R. L. Hudson, B. Saha, A.-R. Adl-Tabatabai, and B. Hertzberg. A Scalable Transactional Memory Allocator. In *Proceedings of the International Symposium on Memory Management*, Ottawa, Ont., Canada, June 2006.

[13] *Power ISA(tm) Transactional Memory*. IBM(R), 2.07 edition, Dec. 2012.

[14] *Intel(R) Architecture Instruction Set Extensions Programming Reference*. Intel Corp., 319433-012a edition, Feb. 2012.

[15] C. Jacobi, T. Slegel, and D. Greiner. Transactional Memory Architecture and Implementation for IBM System z. In *45th Int. Symp. On Microarchitecture*, 2012.

[16] S. Kumar, M. Chu, C. J. Hughes, P. Kundu, and A. Nguyen. Hybrid Transactional Memory. In *Proceedings of the 11th ACM Symposium on Principles and Practice of Parallel Programming*, New York, N.Y., Mar. 2006.

[17] H. Le, G. Guthrie, D. Williams, M. Michael, B. Frey, W. Starke, C. May, R. Odaira, and T. Nakaike. Transactional memory support in the IBM POWER8 processor. *IBM Journal of Research and Development*, 59(1):8–1, 2015.

[18] Y. Lev, M. Moir, and D. Nussbaum. PhTM: Phased Transactional Memory. In *Proceedings of the 2nd ACM SIGPLAN Workshop on Transactional Computing*, Portland, OR, Aug. 2007.

[19] Y. Liu, S. Diestelhorst, and M. Spear. Delegation and Nesting in Best Effort Hardware Transactional Memory. In *Proceedings of the 24th ACM Symposium on Parallelism in Algorithms and Architectures*, Pittsburgh, PA, June 2012.

[20] V. Menon, S. Balensiefer, T. Shpeisman, A.-R. Adl-Tabatabai, R. Hudson, B. Saha, and A. 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.

[21] C. C. 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 *Proceedings of the 34th International Symposium on Computer Architecture*, San Diego, Calif., June 2007.

[22] M. Moravan, J. Bobba, K. Moore, L. Yen, M. Hill, B. Liblit, M. Swift, and D. 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., Oct. 2006.

[23] N. Neelakantam, R. Rajwar, S. Srinivas, U. Srinivasan, and C. Zilles. Hardware Atomicity for Reliable Software Speculation. In *Proceedings of the 34th International Symposium on Computer Architecture*, San Diego, Calif., June 2007.

[24] M. Olszewski, J. Cutler, and J. G. Steffan. JudoSTM: A Dynamic Binary-Rewriting Approach to Software Transactional Memory. In *Proceedings of the 16th International Conference on Parallel Architecture and Compilation Techniques*, Brasov, Romania, Sept. 2007.

[25] A. Patel, F. Afram, S. Chen, and K. Ghose. MARSSx86: A Full System Simulator for x86 CPUs. In *Design Automation Conference 2011 (DAC'11)*, 2011.

[26] R. Rajwar, M. Herlihy, and K. Lai. Virtualizing Transactional Memory. In *Proceedings of the 32nd International Symposium on Computer Architecture*, Madison, Wis., June 2005.

[27] H. E. Ramadan, C. J. Rossbach, and E. Witchel. Dependence-aware Transactional Memory for Increased Concurrency. In *Proceedings of the 41st annual IEEE/ACM International Symposium on Microarchitecture*, pages 246–257. IEEE Computer Society, 2008.

[28] T. Riegel, C. Fetzer, and P. Felber. Time-Based Transactional Memory with Scalable Time Bases. In *Proceedings of the 19th ACM Symposium on Parallelism in Algorithms and Architectures*, San Diego, California, June 2007.

[29] T. Riegel, P. Marlier, M. Nowack, P. Felber, and C. 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.

[30] B. Saha, A.-R. Adl-Tabatabai, and Q. Jacobson. Architectural Support for Software Transactional Memory. In *Proceedings of the 39th IEEE/ACM International Symposium on Microarchitecture*, Orlando, FL, Dec. 2006.

[31] A. Shriraman, M. F. Spear, H. Hossain, S. Dwarkadas, and M. L. Scott. An Integrated Hardware-Software Approach to Flexible Transactional Memory. In *Proceedings of the 34th International Symposium on Computer Architecture*, San Diego, Calif., June 2007.

[32] A. Shriraman, S. Dwarkadas, and M. L. Scott. Flexible Decoupled Transactional Memory Support. In *Proceedings of the 35th International Symposium on Computer Architecture*, Beijing, China, June 2008.

[33] M. Spear. Lightweight, Robust Adaptivity for Software Transactional Memory. In *Proceedings of the 22nd ACM Symposium on Parallelism in Algorithms and Architectures*, Santorini, Greece, June 2010.

[34] M. F. Spear, A. Shriraman, L. Dalessandro, S. Dwarkadas, and M. L. Scott. Nonblocking Transactions Without Indirection Using Alert-on-Update. In *Proceedings of the 19th ACM Symposium on Parallelism in Algorithms and Architectures*, San Diego, Calif., June 2007.

[35] M. F. Spear, L. Dalessandro, V. J. Marathe, and M. L. Scott. Ordering-Based Semantics for Software Transactional Memory. In *Proceedings of the 12th International Conference On Principles Of DIstributed Systems*, Luxor, Egypt, Dec. 2008.

[36] S. Stipic, S. Tomic, F. Zyulkyarov, A. Cristal, O. Unsal, and M. Valero. TagTM - Accelerating STMs with Hardware Tags for Fast Meta-data

Access. In *Proceedings of the 2012 Design, Automation & Test in Europe Conference*, Dresden, Germany, Mar. 2012.

[37] T. von Eicken, D. Culler, S. Goldstein, and K. E. Schauser. Active Messages: A Mechanism for Integrated Communication and Computation. In *Proceedings of the 19th International Symposium on Computer Architecture*, Gold Coast, Australia, May 1992.

[38] G. Yalcin, O. Unsal, and A. Cristal. FaulTM: Error Detection and Recovery Using Hardware Transactional Memory. In *Proceedings of the 2013 Design, Automation & Test in Europe Conference*, Grenoble, France, Mar. 2013.

[39] R. M. Yoo, C. J. Hughes, K. Lai, and R. Rajwar. Performance evaluation of intel transactional synchronization extensions for high-performance computing. In W. Gropp and S. Matsuoka, editors, *International Conference for High Performance Computing, Networking, Storage and Analysis, SC'13, Denver, CO, USA - November 17 - 21, 2013*, pages 19:1–19:11. ACM, 2013. ISBN 978-1-4503-2378-9. . URL `http://doi.acm.org/10.1145/2503210.2503232`.

[40] C. Zilles and L. 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.