

# On Reconciling Hardware Atomicity, Memory Models, and `__tm_waiver`

Sean White and Michael Spear

Department of Computer Science and Engineering, Lehigh University  
{saw207, spear}@cse.lehigh.edu

A key feature of modern TM language proposals is the `__tm_waiver` construct [7], which provides a weak form of open nesting where the “waivered” code may not access data that is also accessed within the calling transaction. The code also cannot use transactions, occurs immediately, and its effects are not rolled back. `__tm_waiver` can be used to spawn threads, make library and system calls, and communicate with other threads. The programmer is responsible for preventing races between “waivered” blocks and other code.

Unfortunately, `__tm_waiver` support in hardware TM can break the processor memory consistency model. The problem stems from the fact that the “waivered” code must happen immediately. The two prominent Best-Effort Hardware Transactional Memory (BEHTM) systems, Sun’s Rock processor [6] and AMD’s Advanced Synchronization Facility (ASF) proposal [1], illuminate the subtleties of this problem. Both permit nontransactional loads and stores within a hardware transaction (stores may not be to locations that also have been accessed transactionally). Thus it would seem that these BEHTM systems provide support for `__tm_waiver`: code within a `__tm_waiver` block would simply use nontransactional loads and stores (but note that it still cannot perform syscalls). However, Rock requires that nontransactional stores idle in the processor store buffer until the transaction commits or aborts. Thus while the stores occur regardless of the transaction’s ultimate success or failure, they do not happen immediately. This preserves the Total Store Order memory model of the SPARC processor [2], but limits the usefulness of these nontransactional stores. In contrast, ASF allows nontransactional stores to happen immediately. This provides greater flexibility, but changes the processor memory model [5].

Clearly, there are arguments for both options. If nontransactional stores occur immediately, they can be used to communicate with other threads; spawning nested transactions or interacting with the operating system

through helper threads would not require more custom hardware (recall that BEHTM strives to have a *minimal* impact on the overall CPU area and verification process). We expect that BEHTM will be well-served by a program-wide helper thread that performs simple syscalls on behalf of hardware transactions: many of the problem cases identified by Baugh and Zilles [3] could be overcome in this manner, via simple synchronous communication, rather than via costly fallback to software mode.

On the other hand, weakening the memory model seems dangerous. Considerations include whether the memory model must also be weakened outside of transactional code, and how to verify that the program remains correct under this memory model, particularly if some data can be accessed transactionally, nontransactionally, and in `__tm_waiver` blocks. Furthermore, weakening the memory model may also decrease the usefulness of BEHTM, e.g., by preventing lock elision [8].

BEHTM store reordering (as in ASF) appears different than compiler reordering, in that there is no analogy to “compiler fences” or `volatile` keywords for preventing it. The processor may always reorder nontransactional stores before transactional ones. Furthermore, when nontransactional stores are used for synchronous communication, they will require language annotations to specify how they can be reordered with respect to other nontransactional accesses. Should existing language constructs suffice, or will new ones be needed?

We take the position that immediately performing some transactional stores is necessary, and propose an extension that provides a more predictable and controlled mechanism for weakening strict memory consistency models. Specifically, we propose three families of store instructions within transactions:

- **Transactional** – These only occur if the transaction commits, and not if the transaction aborts.
- **Nontransactional** – These occur regardless of whether the transaction commits or aborts, but do not

occur until the commit or abort event happens, and occur in order with respect to transactional stores.

- **Immediate** – These occur regardless of whether the transaction commits or aborts, and should not idle in a store buffer or otherwise delay until a commit/abort event before becoming visible to other processors.

At this early stage, we see no reason to support three classes of loads, as nontransactional loads may occur immediately regardless of whether they are being used for inter-thread communication; transactional and nontransactional loads should suffice. Note that our operation classes can apply to the intermediate form of transactional code during compilation, and may enable more powerful static analysis. They are not merely a hardware feature.

Rock and ASF also differ on their attitude toward modality: in Rock transactions, all stores are transactional unless marked explicitly as nontransactional. In ASF, all stores are nontransactional unless marked explicitly as transactional. In the current Rock proposal, “poison” instructions such as `compare-and-swap` (`cas`) are not permitted within a transaction. Although it is appealing to overload such an instruction as the “Immediate” store instruction, we believe this would not be wise for two reasons. First, these instructions currently enforce memory ordering. We would prefer not to introduce unnecessary ordering, but we would also like to avoid predicating whether an instruction causes ordering on its calling context. Secondly, making these instructions valid within a transactional context would break lock elision on Rock. Suppose the code being executed with lock elision uses a `cas` to acquire a lock. If `cas` is an Immediate store, the acquire will happen immediately. The lock release is most-likely a regular (and by modality nontransactional) store. This asymmetry could render locks unacquirable for transactions that acquire with `cas` and then abort, even after releasing the lock.

On architectures with relaxed memory consistency, such as POWER, there does not seem to be a need for our new class of store operations, as the processor memory model already allows store reordering. However, our classification should still improve static analysis. Furthermore, we suspect that the meaning of memory fence instructions within a hardware transaction may change: intuitively, they should mean nothing for an atomic hardware block if the hardware TM is strongly atomic [4] and does *not* allow nontransactional stores. However, if nontransactional stores are allowed, then it is not clear whether fences should enforce ordering only among nontransactional accesses, or among all accesses. A new class of fences, rather than a new class of store instructions, may be required.

Our position is motivated by practicality, and a desire to preserve the usefulness of TM for lock elision while still allowing synchronous communication from within transactions. Clearly, verifying program correctness under this scheme will be even more challenging than under existing memory consistency models. Our hope is that by proposing a way to combine desirable features present in Rock and ASF, we can achieve the benefits of each while incurring limited additional cost, particularly for the yet-to-be-determined common case.

## References

- [1] Advanced Micro Devices, Inc. Advanced Synchronization Facility: Proposed Architectural Specification. Technical Report Publication #45432, rev. 2.1, Advanced Micro Devices, Inc, Mar. 2009. Available as [developer.amd.com/assets/45432-ASF\\_Spec.2.1.pdf](http://developer.amd.com/assets/45432-ASF_Spec.2.1.pdf).
- [2] S. V. Adve and K. Gharachorloo. Shared memory consistency models: A tutorial. *Computer*, 29(12):66–76, 1996.
- [3] L. Baugh and C. Zilles. An Analysis of I/O and Syscalls in Critical Sections and Their Implications for Transactional Memory. In *Proceedings of the 2nd ACM SIGPLAN Workshop on Transactional Computing*, Portland, OR, Aug. 2007.
- [4] C. Blundell, E. C. Lewis, and M. M. K. Martin. Subtleties of Transactional Memory Atomicity Semantics. *Computer Architecture Letters*, 5(2), Nov. 2006.
- [5] J. Chung, D. Christie, M. Pohlack, S. Diestelhorst, M. Hohmuth, and L. Yen. Compilation of Thoughts about AMD Advanced Synchronization Facility and First-Generation Hardware Transactional Memory Support. In *Proceedings of the 5th ACM SIGPLAN Workshop on Transactional Computing*, Paris, France, Apr. 2010.
- [6] D. Dice, Y. Lev, M. Moir, and D. 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, DC, Mar. 2009.
- [7] Y. Ni, A. Welc, A.-R. Adl-Tabatabai, M. Bach, S. Berkowitz, J. Cownie, R. Geva, S. Kozhukow, R. Narayanaswamy, J. Olivier, S. Preis, B. Saha, A. Tal, and X. Tian. Design and Implementation of Transactional Constructs for C/C++. In *Proceedings of the 23rd ACM SIGPLAN Conference on Object Oriented Programming Systems Languages and Applications*, Nashville, TN, USA, Oct. 2008.
- [8] R. Rajwar and J. R. Goodman. Speculative Lock Elision: Enabling Highly Concurrent Multithreaded Execution. In *Proceedings of the 34th IEEE/ACM International Symposium on Microarchitecture*, Austin, TX, Dec. 2001.