# Physical Limitations of Hardware Transactional Memory

**Abstract**

Hardware transactional memory (HTM) is a promising solution to the synchronization problem in multicore software due to its simple semantics and superior performance relative to coarse-grained locks. Hardware transactions offer a performance advantage over software implementations by harnessing the power of existing cache-coherence mechanisms which are already fast, automatic, and parallel. The source of superior performance, however, is also the root of their weakness: existing implementations of hardware transactions will abort when the working set exceeds the capacity of the underlying hardware. Before we can incorporate this nascent technology into high-performing concurrent data structures, it is necessary to first investigate the physical capacity constraints of hardware transactions to inform the design of such data structures. We were able to divine details of the HTM implementation of Intel's Haswell and IBM's PowerPC architectures from our investigation of the capacity constraints and these characterizations provide much needed understanding of the systems for which practitioners design algorithms and data structures.

## 1 Introduction

As Moore's law plateaus [13] the transition to multicore microprocessors is in full swing. High-performing concurrent programs require the effective utilization of these multicore chips, but synchronization overhead and complexity has been a major roadblock to building fast concurrent programs. Transactional memory [6] was originally proposed as a programming abstraction that could achieve high performance while maintaining the simplicity of coarse-grained locks [14]. Recently, Intel [9, 12] and IBM [1, 7, 11] have both introduced mainstream multicores supporting restricted ***hardware transactional memory*** (HTM) and a new ingredient in the solution to the synchronization problem is on the horizon. The Intel and IBM systems are both *restricted* in that they are a *best effort* hardware transactional memory system [1, 4, 8, 9]. This means that there is no guarantee that an attempted transaction will complete successfully, even in the absence of memory contention from other cores, due to hardware restrictions arising from the imperfect implementation. Hardware transactions are faster than traditional coarse-grained locks and software transactions [14] [2], yet they have similar performance to well-engineered software using fine-grained locks and atomic instructions (e.g. COMPARE-AND-SWAP [5]) [14]. However, *restricted* HTM introduces a wrinkle into this otherwise panacean technology: sometimes transactions will fail even when executed serially due to limitations of the underlying hardware implementation. The conditions under which such a failure may occur dramatically impacts whether the complexity of designing a software system using restricted HTM is justified by the expected performance. Characterizing these conditions is the goal of this paper.

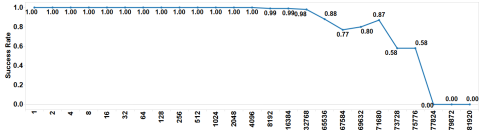


**Figure 1:** Lines Read vs Success Probability.

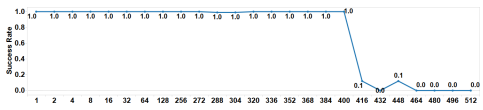


**Figure 2:** Lines Written vs Success Probability.

Transactions require the logical maintenance of ***read sets***, the set of memory locations that are read within a transaction, and ***write sets***, the set of memory locations that are written within a transaction [6]. When transactions execute, local reads and writes to memory are tracked and recorded

in corresponding read and write sets. Upon completion of a transaction, the memory state is validated for consistency before the transaction ***commits***, meaning that the modifications to memory are visible to other threads.

Transactions may ***abort*** due to a conflict with another concurrently executing transaction when an inconsistent memory state is detected, such as when one or many memory locations in one thread's write set intersects one or many memory locations in another thread's read set or write set. We call this a ***conflict abort***. In addition to ***conflict aborts***, hardware transactions specifically suffer from *capacity aborts* when the underlying hardware lacks sufficient resources to maintain the read or write set.

**Experimental Setup** In this paper we summarize results of experiments that determine how the read and write sets are maintained in hardware for recent Intel x86 and IBM PowerPC architectures. In particular, the Intel machine contains a Haswell i7-4770 processor with 4 cores running at 3.4GHz, 8 hardware threads, 64B cache lines, an 8MB 16-way shared L3 cache, 256KB per-core 8-way L2 caches, and 32KB per-core 8-way L1 caches. We also tested an IBM Power8 processor with 10 cores running at 3.4GHz, 80 hardware threads, 128B cache lines, an 80MB 8-way shared L3 cache, and 64KB per-core 8-way L1 caches. All experiments are written in C and compiled with GCC, optimization level `-O0`.[1] Our experiments use the GCC hardware transactional memory intrinsics interface.[2]

The focus of our experiments is to explore scenarios where these capacity aborts are inevitable so that we know when hardware transactions are not a feasible synchronization solution at all, even in the case of no contention. For example, Figures 1 and 2 illustrate the maximum read and write capacity, respectively, of a hardware transaction on an Intel processor. In Section 2 we will explore other types of constraints imposed on HTM operation by the Intel and IBM implementations. This characterization should inform software development attempting to use the newly available HTM support and hybrid hardware / software transactional memory systems [3, 10].

# 2 Capacity Constraints

Physical limitations to the size of hardware transactions are governed by how they are implemented in hardware. Such capacity constraints determine when a transaction will inevitably abort, even in the case of zero contention. We devised a parameterizable array access experiment to measure the maximum cache line capacity of sequential read-only and write-only hardware transactions. We also experimented with strided memory access patterns to detect whether the read and write sets are maintained on a per-cache line basis or a per-read / per-write basis. With knowledge of the maximum sequential access capacity and also the maximum strided access capacity, we can draw conclusions about where in the caching architecture the read and write sets are maintained.

### Intel

In this section, we experimentally support the hypothesis that the Intel HTM implementation uses the L1 cache to store write sets and the L3 cache to store read sets.

Figure 1 summarizes the result of a sequential read-only access experiment. The data points in the graphs represent the success probability of the transaction with respect to the number of cache lines read. The results indicate that a single transaction can reliably read around 75,000 contiguous cache lines.

[1] We compiled with `-O0` because we found that higher optimization levels sometimes caused spurious transaction aborts, thus confounding our results.

[2] The x86 intrinsics are `_xbegin` and `_xend`; the PowerPC intrinsics are `__builtin_tbegin` and `__builtin_tend`.

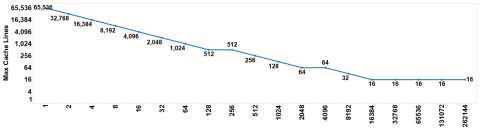

**Figure 3:** Stride Amount vs Cache Lines Readable.

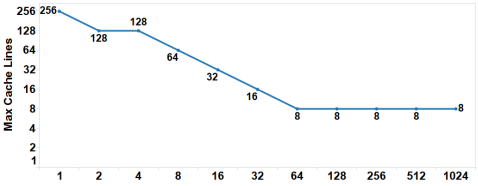


**Figure 4:** Stride Amount vs Cache Lines Writable.

Figure 3 shows the result of a strided read-only access experiment. The stride amount is the number of cache lines per iteration (e.g. reading cache lines 1, 5, 9, 13, 17 etc. indicates a stride of 4). Each data point in the graph represents the maximum number of cache lines that can be reliably read with respect to the stride amount. For example, the third data point in the graph indicates that when the stride amount is $2^2$ ($= 4$) (e.g. accessing every fourth cache line), the transaction can reliably read $2^{14}$ ($= 16,384$) cache lines and commit. The significance of this graph is that the number of cache lines that can be read in a single transaction is generally halved as we double the stride amount, presumably because the access pattern accesses the same few cache sets while completely skipping over other sets. It is important to note that the plot plateaus at $2^4$ ($= 16$) cache lines, which is not-coincidentally the associativity of the L3 cache.

Figures 1 and 3 show experimental results that indicate that the read set is maintained in the L3 cache. The L3 cache of this Intel machine has a maximum capacity of $2^{17}$ cache lines, which explains why read-only transactions cannot fit much more than $2^{16}$ ($= 65,536$) cache lines because it is unlikely for the whole read set to fit perfectly into the L3 due to the hash function mapping physical address to L3 cache bank. The minimum of 16 cache lines readable when the stride amounts are large enough to consecutively hit the same cache set further supports the notion that the read set is maintained through the L3 because this value exactly coincides with the L3 associativity.

We also conducted similar experiments for write-only accesses patterns. Figure 2 illustrates the result of an identical array access experiment, except that the transactions are write-only instead of read-only. A single write-only transaction can reliably commit about 400 contiguous cache lines. The size of the L1 cache is 512 cache lines and a transaction must also have sufficient space to store other program metadata (e.g. the head of the program stack).

Figure 4 illustrates that the number of cache lines that can be written in a single transaction is also generally halved as we double the stride amount. However, even as we increase the stride amount significantly, the number of cache lines that a transaction can reliably write to does not fall below 8, corresponding to the associativity of the L1 cache. This suggests that, at worst, one is limited to storing all writes in a single, but entire, set of the L1 cache.

**IBM**

In this section, we experimentally support the hypothesis that the IBM HTM implementation uses a dedicated structure to maintain read and write sets, choosing not to extend the functionality of the existing cache structures as with the Intel implementation. In addition, we observe that the dedicated structures used for read and write set maintenance is not shared among the 8 threads per core, but rather each thread is allocated its own.

The results of our strided access experiment for both read-only and write-only transactions appear to be identical in Figure 6, where the maximum number of reads or writes in a transaction is 64 and that the maximum transaction size halves as we double the stride amount with a minimum of 16. The maximum observed hardware transaction size is far too small to be attributable to even the L1 cache, which holds 512 cache lines. Thus, we conclude that there are dedicated caches for transactions in the IBM implementation independent of the standard core caches, and that these caches likely each have 4 sets and an associativity of 16.

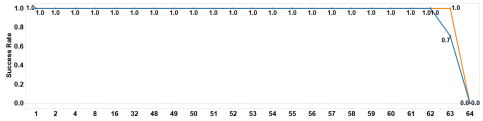


**Figure 5:** Cache Lines Read/Written vs Success Probability.

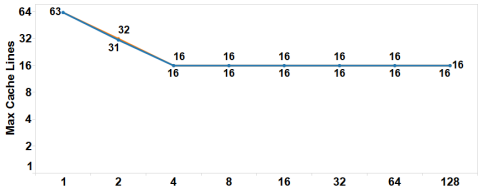


**Figure 6:** Stride Amount vs Cache Lines Readable/Writeable.

A natural next question is whether this IBM machine has 10 dedicated caches that are spread across each core, or if there are 80 dedicated caches that are spread across each hardware thread. To determine the difference, we experimented and measured the number of successful write-only transactions that concurrently running threads were able to complete. Each thread makes 10000 transaction attempts to write 40 thread-local padded cache lines and then commit. The transaction size of 40 cache lines is designed to sufficiently fill up the dedicated caches per transaction to induce capacity aborts in the case of shared caches.

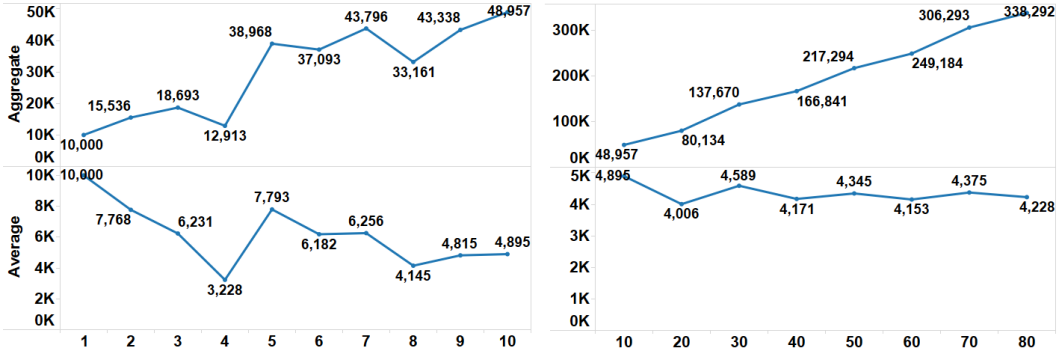


**Figure 7:** Number of Threads vs Committed Transactions.

We see in Figure 7 compelling reason to believe that there are 80 dedicated caches, one for each hardware thread. Each spawned software thread is pinned to a unique hardware thread in round robin fashion such that the distribution is even across the 10 cores. If all 8 of the hardware threads on a single core share a single dedicated cache, we would expect to see sublinear speedup, perhaps even degraded performance, as we spawn more running threads and assign them to the same core. Instead, we observe a linear increase in the aggregate number of successfully committed transactions, while the average per-thread number of successful transactions is constant. Although the general 45% success rate suggests some level of contention between the running threads, it is most likely not due to per-core sharing of a dedicated cache.

# 3 Conclusion

Hardware transactional memory is an exciting new solution to the synchronization problem in multicore software. Before we can effectively use this new technology, we must first understand the physical limits of hardware transactions that are inherent to the Intel and IBM microprocessors on which they are implemented. Our capacity constraint benchmarks revealed that the read sets are implemented through the L3 and the write sets are implemented through the L1 on the Intel machine;

on the other hand, there is a per-hardware-thread, 4-set, 16-way dedicated cache that maintains the read and write sets on the IBM machine.

Developers using HTM on Intel's Haswell microprocessors have a lot of flexibility with hardware transaction size, but they should be wary of how the caching behavior of non-transactional code might affect HTM performance. Practitioners on IBM's PowerPC microprocessors should be cautious of the tight restriction on transaction size, but fortunately they only need to reason about HTM performance within the scope of a single hardware thread.

We anticipate that these findings will move us in the right direction to better understanding hardware transactional memory, ultimately enabling its proliferation into future concurrent programs.

# 4 References


[1] H. W. Cain, M. M. Michael, B. Frey, C. May, D. Williams, and H. Le. Robust architectural support for transactional memory in the power architecture. ISCA '13, 2013.

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

[3] P. Damron, A. Fedorova, Y. Lev, V. Luchangco, M. Moir, and D. Nussbaum. Hybrid transactional memory. *SIGPLAN Not.*, 2006.

[4] R. Dementiev. Exploring intel transactional synchronization extensions with intel software development emulator. `https://software.intel.com/en-us/blogs/2012/11/06/exploring-intel-transactional-synchronization-extensions-with-intel-software`, 2012.

[5] M. Herlihy. Wait-free synchronization. *ACM Trans. Program. Lang. Syst.*, 1991.

[6] M. Herlihy and J. E. B. Moss. Transactional memory: Architectural support for lock-free data structures. In *ISCA*, 1993.

[7] IBM. Ibm power systems s814 and s824 technical overview and introduction, 2014.

[8] IBM. Performance optimization and tuning techniques for ibm processors, including ibm power8, 2014.

[9] Intel. Intel architecture instruction set extensions programming reference, 2012.

[10] A. Matveev and N. Shavit. Reduced hardware transactions: A new approach to hybrid transactional memory. SPAA '13, 2013.

[11] R. Merritt. Ibm plants transactional memory in cpu. `http://www.eetimes.com/document.asp?doc_id=1260096`, 2011.

[12] J. Reinders. Transactional synchronization in haswell. `https://software.intel.com/en-us/blogs/2012/02/07/transactional-synchronization-in-haswell`, 2012.

[13] M. Y. Vardi. Moore's law and the sand-heap paradox. *Commun. ACM*, 2014.

[14] R. M. Yoo, C. J. Hughes, K. Lai, and R. Rajwar. Performance evaluation of intel; transactional synchronization extensions for high-performance computing. In *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*, 2013.