Quantifying the Capacity Limitations of Hardware Transactional Memory

William Hasenplaugh
whasenpl@mit.edu

Andrew Nguyen
tanguyen@mit.edu

Nir Shavit
shanir@mit.edu

Abstract

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 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 investigate these capacity constraints in order to better inform programmers of their abilities and limitations.

This paper provides the first comprehensive empirical study of the "capacity envelope" of HTM in Intel's Haswell and IBM's Power8 architectures, providing what we believe is a much needed understanding of the extent to which one can use these systems to replace locks.

1 Introduction

As Moore's law has plateaued [25] over the last several years, the number of researchers investigating technologies for fast concurrent programs has doubled approximately every two years. [1] High performance concurrent programs require the effective utilization of ever-increasing core counts and perhaps no technology has been more anticipated toward this end than Hardware Transactional Memory (HTM). Transactional memory [12] was originally proposed as a programming abstraction that could achieve high performance while maintaining the simplicity of coarse-grained locks [29] and recently Intel [15,22] and IBM [2,13,19] have both introduced mainstream multicore processors supporting ***restricted*** HTM. Hardware transactions are faster than traditional coarse-grained locks and software transactions [4,29], yet they have similar performance to well-engineered software using fine-grained locks and atomic instructions (e.g. COMPARE-AND-SWAP [11]). The Intel and IBM systems are both *restricted* in that they are a *best effort* hardware transactional memory implementation [2,7,14,15]: transactions can fail due to limitations of the underlying hardware implementation even when executed serially. 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.

Related Work Recently, several researchers have considered variations of hybrid transactional memory (HyTM) systems [5, 6, 18] which exploit the performance potential of recent HTM implementations, while preserving the semantics and progress guarantees of software transactional memory (STM) systems [24]. Underlying all of this work is the assumption that hardware constraints on the size of transactions are sufficiently unforgiving, supported by recent sequential access evaluations of Haswell [8, 9, 21, 23], that elaborate workarounds are justified. For instance, Xiang et al. [27, 28] propose the decomposition of a transaction into a nontransactional read-only planning phase and a transactional write-mostly completion phase in order to reduce the size of the actual

[1] We can observe this exponential increase in research activity related to concurrent programming via a search of the ACM Digital Library [1] for papers with titles including relevant phrases (eg. transactional memory, concurrent data structures, parallel runtimes etc.) for each year over the last few decades.

transaction. Wang et al. [26] use a similar nontransactional execution phase and a transactional commit phase in the context of an in-memory database in order to limit the actual transaction to the database meta-data and excluding the payload data. Likewise, Leis et al. [16] use *timestamp ordering* [3] to glue together smaller transactions in order to compose one large transaction in an in-memory database.

Background 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 [12]. Upon completion of a transaction, the memory state is validated for consistency before the transaction ***commits***, making modifications to memory visible to other threads. In addition to ***conflict aborts*** that occur due to concurrent transactional accesses to the same memory address[2], hardware transactions suffer from ***capacity aborts*** when the underlying hardware lacks sufficient resources to maintain the read or write set of an attempted transaction.

Read and write sets are often maintained in hardware using an extension to an existing cache hierarchy. Caches in modern processors are organized in ***sets*** and ***ways***, where a surjection from memory address to set number is used in hardware to restrict the number of locations that must be checked on a cache access. The number of ways per set is the ***associativity*** of the cache and an address mapping to a particular set is eligible to be stored in any one of the associated ways. To maintain the read and write sets of a transaction, one can “lock” each accessed memory address into the cache until the transaction commits. The logic of the cache coherence protocol can also be extended to ensure atomicity of transactions by noting whether or not a cache-to-cache transfer of data involves an element of a transaction’s read or write set. These extensions to the caches and the cache coherence protocol are very natural and lead to high performance, however the nature of the design reveals an inherent weakness: caches are finite in size and associativity, thus such an architecture could never guarantee forward progress for arbitrarily large transactions.

Contributions In this paper we summarize results of the first comprehensive empirical study of the “capacity envelope” for recent Intel Haswell and IBM Power8 restricted HTM implementations using experiments that determine how the read and write sets are maintained in hardware. We conclude that the read and write sets are maintained in the L3 and L1 cache, respectively, for the Intel Haswell. In addition, the IBM Power8 dedicates a small 64-entry cache per hardware thread. This characterization should inform software development attempting to use the newly available HTM support and HyTM systems [6, 17, 18].

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.

Experimental Setup The Intel machine we experimented on 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`.[3] Our code uses the GCC hardware transactional

[2]Specifically, a conflict abort occurs when one thread’s write set intersects at least one memory location in the read or write set of another thread

[3]We compiled with `-O0` because it generates precisely the assembly that we wish to test (i.e. a simple sequence of memory accesses with very few supporting arithmetic instructions), whereas higher optimization levels sometimes optimize away the

memory intrinsics interface and is available online [20].

Intel We experimentally support the hypothesis that the Intel HTM implementation uses the L3 cache to store read sets and the L1 cache to store write sets.

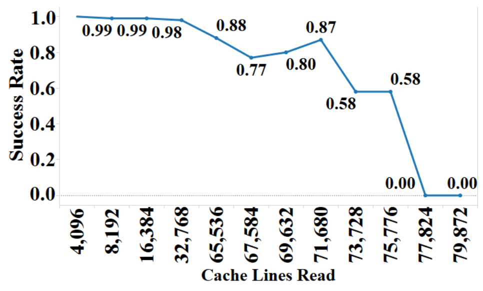

Figure 1: Lines Read vs Success Rate.

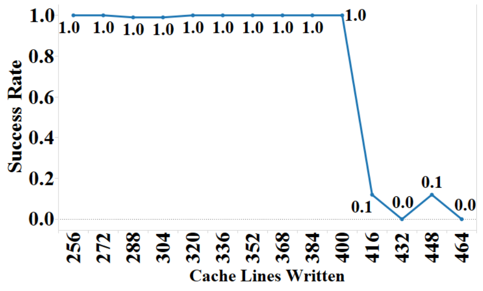

Figure 2: Lines Written vs Success Rate.

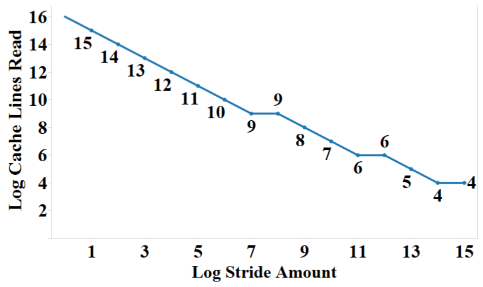

Figure 3: $\log_2$ Stride vs $\log_2$ Lines Read.

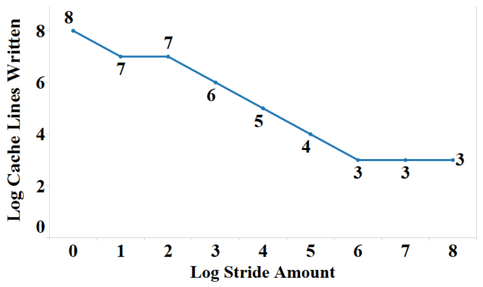

Figure 4: $\log_2$ Stride vs $\log_2$ Lines Written.

Figure 1 summarizes the result of a sequential read-only access experiment where data points represent the success probability of the transaction with respect to the number of cache lines read. We see that a single transaction can reliably read around 75,000 contiguous cache lines. The L3 cache of the Intel machine has a maximum capacity of 2^{17} $(= 131,072)$ cache lines and it is unlikely for much more than half of the total capacity to fit perfectly into the L3 due to the hash function mapping physical address to L3 cache bank. This hash function gives rise to a 'balls in bins' behavior for the load on each cache set. That is, n cache lines can be accessed such that the most loaded set contains $\Theta(\lg n/\lg\lg n) = A$ cache lines [10], where A is the associativity of the cache. We observe empirically in this setting (i.e. $A = 16$) that the maximum capacity is roughly half of the L3.

Figure 3 shows the result of a strided read-only access experiment. The stride amount indicates the number of cache lines stepped over per iteration (e.g. reading cache lines 1, 5, 9, 13, 17 etc. indicates a stride of 4) and each data point 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. We can see 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 pattern accesses progressively fewer cache sets while completely skipping over the other sets. It is important to note that the plot plateaus at 2^4 $(= 16)$ cache lines. When the stride amounts are large enough to consecutively hit the same cache set we see support for the hypothesis that the read set is maintained in the L3 cache because the minimum number of readable values never drops below 16, 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

entire loop (e.g. read-only tests).

other program metadata (e.g. the head of the program stack), thus we would not expect to fill all 512 lines perfectly.

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 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 copy.

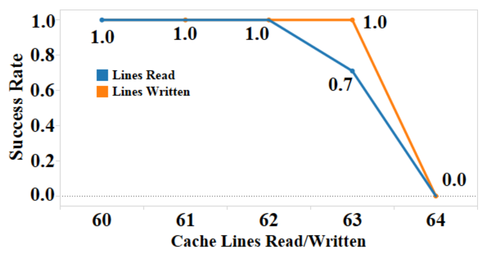

Figure 5: Lines Read/Written vs Success Rate.

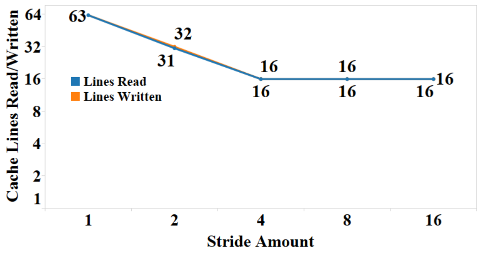

Figure 6: Stride vs Lines Read/Written.

The results of our sequential and strided access experiments for both read-only and write-only transactions appear to be identical in Figure 5 and 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.

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 10,000 transaction attempts to write 40 thread-local 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.

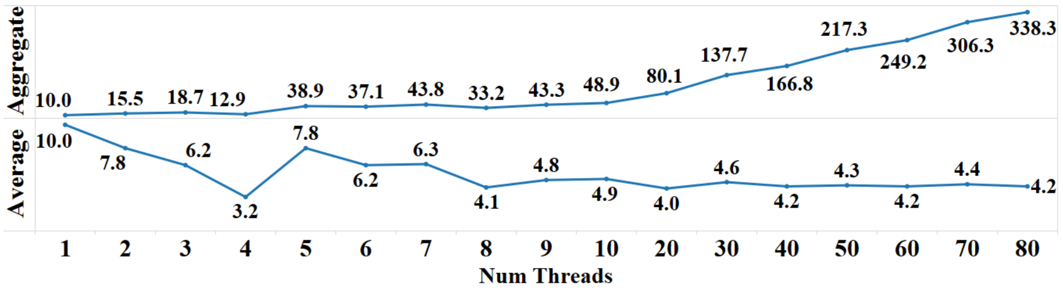

Figure 7: Number of Threads vs Aggregate/Average Committed Transactions (Thousands).

We see in Figure 7 evidence that there are dedicated caches for each hardware thread and that they are not shared among threads within a core. 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 (or even no) speedup 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 because the addition of other threads does not decrease the aggregate throughput.

3 References

[1] ACM. ACM digital library. `http://dl.acm.org/`.

[2] 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. In *Proceedings of the 40th Annual International Symposium on Computer Architecture*, ISCA '13, pages 225–236, New York, NY, USA, 2013.

[3] M. J. Carey. *Modeling and Evaluation of Database Concurrency Control Algorithms*. PhD thesis, 1983. AAI8413325.

[4] C. Cascaval, C. Blundell, M. Michael, H. W. Cain, P. Wu, S. Chiras, and S. Chatterjee. Software transactional memory: Why is it only a research toy? *Queue*, 6(5):40:46–40:58, 2008.

[5] L. Dalessandro, M. F. Spear, and M. L. Scott. NOrec: Streamlining STM by abolishing ownership records. In *Proceedings of the 15th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPoPP '10, pages 67–78, New York, NY, USA, 2010.

[6] P. Damron, A. Fedorova, Y. Lev, V. Luchangco, M. Moir, and D. Nussbaum. Hybrid transactional memory. In *Proceedings of the 12th International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '06, pages 336–346, New York, NY, USA, 2006.

[7] R. Dementiev. Exploring intel transactional synchronization extensions with intel software development emulator. Intel Developer Zone, November 2012.

[8] N. Diegues, P. Romano, and L. Rodrigues. Virtues and limitations of commodity hardware transactional memory. In *Proceedings of the 23rd International Conference on Parallel Architectures and Compilation*, PACT '14, pages 3–14, New York, NY, USA, 2014.

[9] B. Goel, R. Titos-Gil, A. Negi, S. A. McKee, and P. Stenström. Performance and energy analysis of the restricted transactional memory implementation on Haswell. In *Proceedings of the 2014 IEEE 28th International Parallel and Distributed Processing Symposium*, IPDPS '14, pages 615–624, Washington, DC, USA, 2014.

[10] G. H. Gonnet. Expected length of the longest probe sequence in hash code searching. *Journal of the Association for Computing Machinery*, 28(2):289–304, 1981.

[11] M. Herlihy. Wait-free synchronization. *ACM Transactions on Programming Languages and Systems*, 13(1):124–149, 1991.

[12] M. Herlihy and J. E. B. Moss. Transactional memory: Architectural support for lock-free data structures. In *Proceedings of the 20th Annual International Symposium on Computer Architecture*, ISCA '93, pages 289–300, New York, NY, USA, 1993.

[13] IBM. IBM power systems S814 and S824 technical overview and introduction. Redpaper REDP-5097-00, IBM Corporation, 2014.

[14] IBM. Performance optimization and tuning techniques for IBM processors, including IBM POWER8. Redbooks SG24-8171-00, IBM Corporation, July 2014.

[15] Intel. Intel architecture instruction set extensions programming reference. Developer Manual 319433-012A, Intel Corporation, 2012.

[16] V. Leis, A. Kemper, and T. Neumann. Exploiting hardware transactional memory in main-memory databases. In *ICDE*, 2014.

[17] A. Matveev and N. Shavit. Reduced hardware transactions: A new approach to hybrid transactional memory. In *Proceedings of the Twenty-fifth Annual ACM Symposium on Parallelism in Algorithms and Architectures*, SPAA '13, pages 11–22, New York, NY, USA, 2013.

[18] A. Matveev and N. Shavit. Reduced hardware NOrec: A safe and scalable hybrid transactional memory. In *Proceedings of the Twentieth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '15, pages 59–71, New York, NY, USA, 2015.

[19] R. Merritt. IBM plants transactional memory in CPU, August 2011.

[20] A. Nguyen. Transactional memory capacity test. https://github.com/nauttuan/wttm_capacity_test.

[21] M. M. Pereira, M. Gaudet, J. N. Amaral, and G. Araújo. Multi-dimensional evaluation of Haswell's transactional memory performance. In *Proceedings of the 2014 IEEE 26th International Symposium on Computer Architecture and High Performance Computing*, SBAC-PAD '14, pages 144–151, Washington, DC, USA, 2014.

[22] J. Reinders. Transactional synchronization in Haswell. Intel Developer Zone, February 2012.

[23] C. G. Ritson and F. R. M. Barnes. An evaluation of Intel's restricted transactional memory for CPAs. In *Communicating Process Architectures 2013*, pages 271–292, November 2013.

[24] N. Shavit and D. Touitou. Software transactional memory. In *Proceedings of the 14th Annual ACM Symposium on Principles of Distributed*, pages 204–213, Ottawa, Ontario, Canada, 1995.

[25] M. Y. Vardi. Moore's law and the sand-heap paradox. *Communications of the ACM*, 57(5):5–5, May 2014.

[26] Z. Wang, H. Qian, J. Li, and H. Chen. Using restricted transactional memory to build a scalable in-memory database. In *Proceedings of the Ninth European Conference on Computer Systems*, EuroSys '14, pages 26:1–26:15, New York, NY, USA, 2014.

[27] L. Xiang and M. L. Scott. Composable partitioned transactions. In *Workshop on the Theory of Transactional Memory*, 2013.

[28] L. Xiang and M. L. Scott. Software partitioning of hardware transactions. In *Proceedings of the 20th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPoPP '15, pages 76–86, New York, NY, USA, 2015.

[29] 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*, SC '13, pages 19:1–19:11, New York, NY, USA, 2013.