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-grain 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

Random notes:

- Use ***defn*** blocks to define terms, such as ***hardware transactional memory***. However, ***defn*** blocks should always have an accompanying definition. Their use is a good mnemonic for remembering that one should define important terms.
- In text, *italics* or *emphasis* blocks are used to draw attention to an otherwise well-defined term or phrase.

As Moore's law plateaus [**?**] 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. Recently, Intel [**?**] and IBM [**?**] have both introduced mainstream multicores supporting restricted ***hardware transactional memory*** (HTM)[1] [**?**] and a new ingredient in the solution to the synchronization problem is on the horizon. Hardware transactions are faster than traditional coarse-grained locks [**?**] and software transactions [**?**], yet they have similar performance to well-engineered software using fine-grained locks and atomic instructions (e.g. COMPARE-AND-SWAP [**?**]) [**?**]. 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 HTM is justified by the expected performance. Characterizing these conditions is the goal of this paper.

Implementing transactions involves 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. When transactions execute, local reads and writes to memory are tracked and recorded

[1] The Intel and IBM systems are both *restricted* in that they are a *best effort* hardware transactional memory system [**?**]. This means that there is no guarantee that an attempted transaction will complete successfully, even in the absence of contention from other cores, due to hardware restrictions arising from the imperfect implementation.

in corresponding read sets 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.

We show in this paper the results of experiments we ran to determine where the read sets and write sets are implemented in the Intel x86 and IBM PowerPC architectures. 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. These contributions will provide important insights to the limits of hardware transactional memory and better enable its effective utilization in future multicore programs.

2 Experimental Setup

[[WCH: Every section and subsection and subsubsection etc. gets its own topic paragraph.]]

2.1 Hardware Specifications

The results from our experiments should only be fully accepted with respect to the microprocessors we specify in this section, although the conclusions will generally apply to different generations of the hardware.

The Intel experimental machine contains a Haswell i7-4770 processor with

- 4 3.40GHz cores; 4 total hardware threads
- 64 byte wide cache lines
- 8mB shared 16-way *L3* cache
- 32kB per-core 8-way *L1* cache

The IBM experimental machine contains a Power8 processor with

- 10 3.425GHz cores; 80 total hardware threads
- 128 byte wide cache lines
- 80mB shared 8-way *L3* cache (roughly 8mB per-core)
- 64kB per-core 8-way *L1* cache

2.2 Hardware Transactional Memory Interface

All experiments are written in C and compiled with GCC, optimization level *-O0*. Our experiments use the GCC hardware transactional memory intrinsics interface.

3 Capacity Constraints

There are physical limitations to the size of hardware transactions that are governed by how they are implemented in hardware. These capacity constraints determine when a transaction will inevitably abort, even in the case of zero contention. It is thus critical to shed light on the hardware implementation of transactions in order to know when they are not a feasible synchronization solution at all.

Hardware transactional memory leverages existing caching protocols to maintain read sets and write sets. As such, we devised an 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 see how different stride amounts affect maximum possible transaction sizes. 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 sets and write sets are implemented. We report these results for both the Intel and IBM machines.

Intel

Figure 1 shows 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 75000 cache lines and still commit..

Figure 2 shows the result of a strided read-only access experiment. The stride amount is the number of cache lines stepped over per access. 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 $2^2 = 4$ cache lines stepped over per access, the transaction can reliably read $2^{14} = 16384$ 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 keeps hitting 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.

Figure 1 and Figure 2 show experimental results that indicate that the read set is very closely related to, if not directly stored inside, 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 $2^{16} = 65536$ cache lines because it is impossible for the whole read set to fit perfectly into the *L3*. 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 set associativity of the *L3*, which is 16 cache lines.

Figure 3 illustrates the result of an identical array access experiment, except now the transactions are write-only instead of read-only. A single write-only transaction can reliably commit about 400 cache lines.

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, but 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.

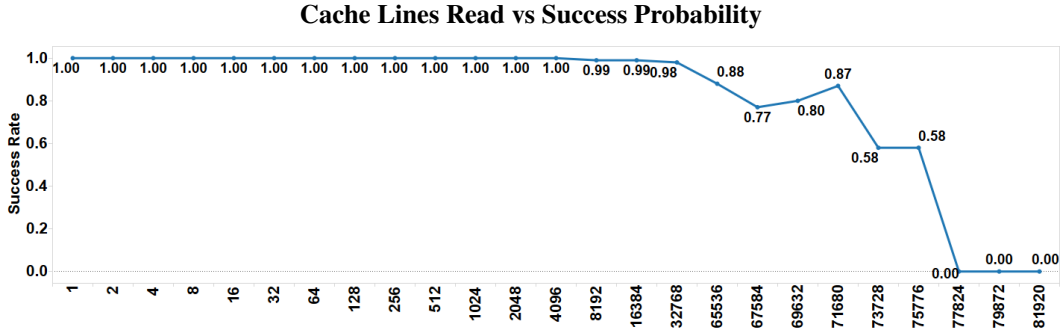

Figure 1: The read set maximum sequential access capacity on the Intel machine is around 75,000 cache lines

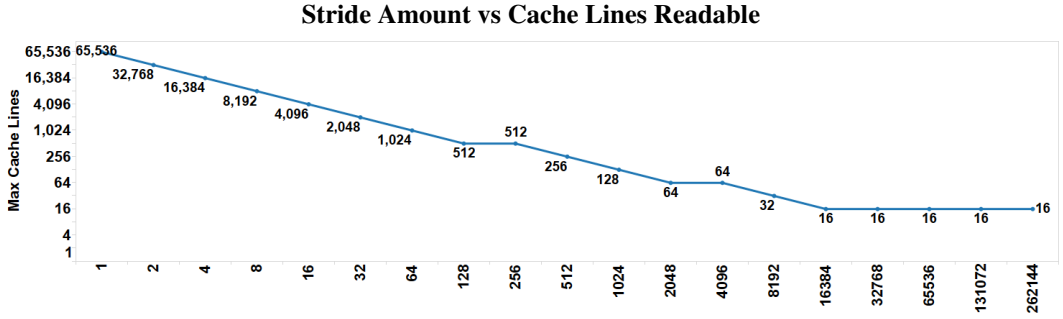

Figure 2: Doubling the stride amount halves the size of successful read-only transactions to a minimum of 16 on the Intel machine

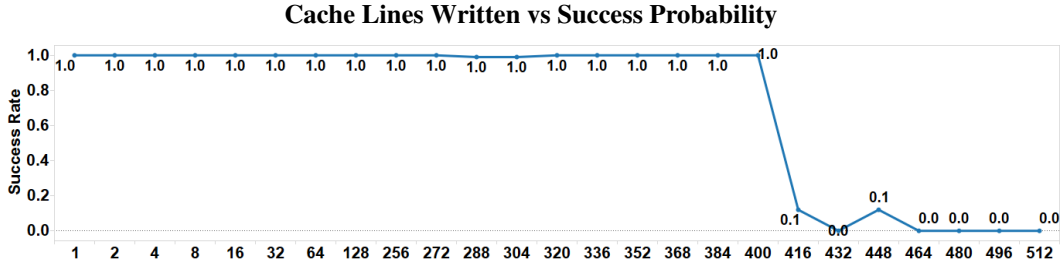

Figure 3: The write set maximum sequential access capacity on the Intel machine is around 400 cache lines

The evidence from Figure 3 and Figure 4 suggests that the write set is very closely related to, if not directly stored inside, the *L1* cache. The size of the *L1* cache is 512 cache lines, and it makes sense that the write set can only reach about 400 cache lines because it cannot possibly fit exactly into the *L1*. The associativity of the *L1* is 8, and this number coincides with the number of cache lines that can still be written to for large strided access patterns that hit the same cache set.

Judging from the results of our capacity constraint experiments, it is likely the case that the read set is implemented through the *L3* cache and the write set is implemented through the *L1* cache on the experimental Intel machine, and even more generally on other Intel x86 microprocessors that support hardware transactional memory.

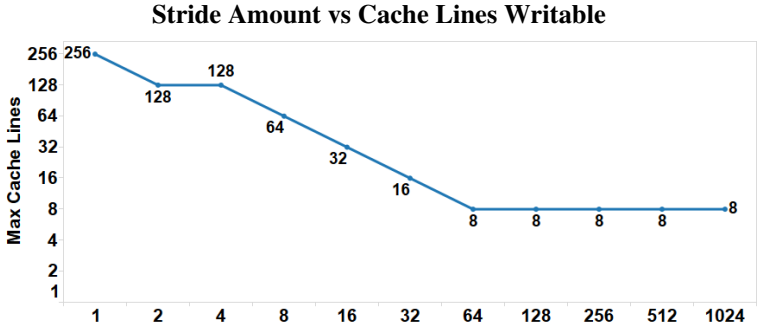

Figure 4: Doubling the stride amount halves the size of successful write-only transactions to a minimum of 8 on the Intel machine

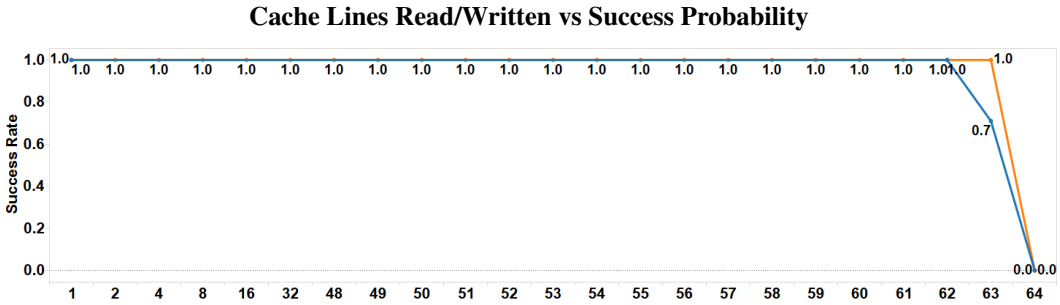

Figure 5: The read set and write set maximum sequential access capacity on the IBM machine is 63 cache lines

IBM

Figure 5 shows that the maximum number of cache lines that can be read or written in a single hardware transaction on the IBM machine is very clearly 63 cache lines.

The results of our strided access experiment for both read-only and write-only transactions appear to be identical in Figure 6; the doubling-stride-amount $\rightarrow$ halving-transaction-size effect is still observed and a minimum of 16 cache lines can be read or written in a single transaction.

The maximum observed hardware transaction size of 63 on this machine is far too small to be attributed to even the *L1* cache, which holds 512 cache lines. The identical measurements between the read set and write set found in Figures 5 and 6 suggest that there is little distinction between the handling of reads and writes in a transaction on the IBM machine. From these results, we conclude that there are dedicated caches for transactions on the IBM machine and, more generally, for the PowerPC architecture. The dedicated caches likely each have 4 cache sets and a set associativity of 16, for a total of 64 cache lines.

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.

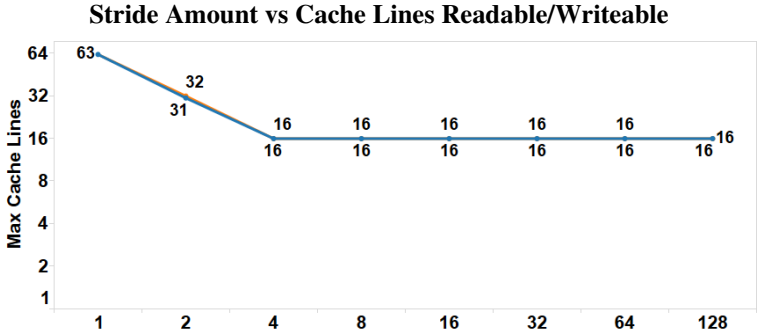

Figure 6: Doubling the stride amount halves the size of successful read-only and write-only transactions to a minimum of 16 on the IBM machine

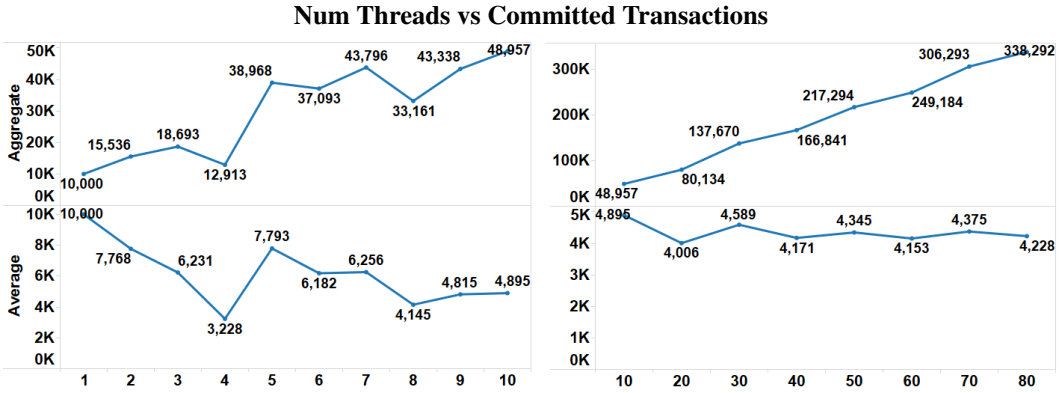

Figure 7: Aggregate committed transactions increases with the number of threads while average committed transactions remains constant, suggesting that there is a dedicated cache per core on the IBM machine

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.

4 Discussion/Implications

Now that the results are presented, I feel the need to discuss their implications, like my anecdotal encounter with the failed program on the IBM machine. Does this warrant its own section, or should we cram it into the conclusion?

5 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-core, 4-set, 16-way dedicated cache that maintains the read and write sets on the IBM machine. *INSERT SOME DISCUSSION/IMPLICATION/REVELATION, like how IBM is not useful for sufficiently large transactions*. 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.

6 References