Investigation of Hardware Transactional Memory

by

Andrew T. Nguyen

Submitted to the
Department of Electrical Engineering and Computer Science
in Partial Fulfillment of the Requirements for the Degree of

Master of Engineering in Electrical Engineering and Computer Science

at the

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

June 2015

Author . . . .
Department of Electrical Engineering and Computer Science
May 22, 2015

Certified by . . . .
Nir Shavit
Professor of Computer Science and Engineering
Thesis Supervisor

Accepted by . . . .
Albert R. Meyer
Chairman, Masters of Engineering Thesis Committee

Investigation of Hardware Transactional Memory

by

Andrew T. Nguyen

Submitted to the Department of Electrical Engineering and Computer Science
on May 22, 2015, in Partial Fulfillment of the
Requirements for the Degree of
Master of Engineering in Electrical Engineering and Computer Science

Abstract

Hardware transactional memory is a new method of optimistic concurrency control that can be used to solve the synchronization problem in multicore software. It is a promising solution due to its simple semantics and good performance relative to traditional approaches. Before we can incorporate this nascent technology into high-performing concurrent programs, it is necessary to investigate the physical capacity constraints and performance characteristics of hardware transactions in order to better inform programmers of their abilities and limitations.

Our investigation involves the first empirical study of the "capacity envelope" of HTM in Intel's Haswell and IBM's Power8 architectures. We additionally survey how contention parameters, such as transaction size or write ratio, affect HTM performance and we capture these trends in a regression model for predicting the throughput of HTM-enabled concurrent programs. Through our investigation, we aim to provide what we believe is a much needed understanding of the extent to which one can use HTM to replace locks.

Thesis Supervisor: Nir Shavit
Title: Professor of Computer Science and Engineering

Acknowledgments

Nir Shavit, my advisor, provided invaluable technical insight. His guidance helped direct my focus toward relevant research problems, which ultimately led to our most significant findings. I am grateful that he made himself available, even though he was thousands of miles away on sabbatical!

My assistant advisor, William Hasenplaugh, was my pillar of support. Whenever I encountered a confounding problem, he both encouraged me to keep pressing forward and also guided me through solving the problem. Our frequent technical discussions led to the key revelations in this thesis. I could not have completed this thesis without his patience and support, and for that I am deeply appreciative.

I thank William Leiserson for facilitating my access to an HTM-enabled Intel Haswell machine to conduct my experiments, and also for helping me sort out machine issues that left me perplexed.

I further thank Pascal Felber and Patrick Marlier for allowing me to experiment on their IBM Power8. Without access to this machine, I would not have been able to collect the IBM experimental results.

Thanks! It has been a remarkable ride.

Contents

Chapter 1

Opening

1.1 Introduction

As Moore's law has plateaued [20] 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 [9] was originally proposed as a programming abstraction with simple semantics that could also achieve good performance, and Intel [13, 17] and IBM [3, 11, 15] have both recently introduced mainstream multicore processors supporting *restricted* HTM.

Hardware transactions offer a performance advantage over software implementations [4, 19] by harnessing the power of existing cache coherence mechanisms which are already fast, automatic, and parallel. HTM has been shown to achieve the high performance of well-engineered software using fine-grained locks and atomic instructions (e.g. *compare-and-swap* [8]) [25] while maintaining the simplicity of coarse-grained locks [25]. The source of their superior performance, however, is also the root of their weakness: the Intel and IBM systems are both *best effort* hardware transactional

[1]This estimation was determined by searching the ACM Digital Library within years to find out how many unique researchers were publishing papers with 'transactional memory' in the title.

memory implementations [3, 7, 12, 13] because transactions can fail when the working set exceeds the capacity of the underlying hardware. The capacity constraints that dictate the conditions under which these failures inevitably occur dramatically influence whether the complexity of designing a software system using restricted HTM is justified by the expected performance.

The feasibility tradeoff imposed by the capacity constraints is just one consideration in the design of software systems using restricted HTM. If the ultimate goal is to build fast concurrent programs, then we must also focus our attention on finding the optimal use cases for HTM. This motivation leads us to explore the design space of multicore programs to understand how hardware transactions perform in different cases of contention.

Our goal in this paper is to characterize the capacity constraints of HTM and to discover their performance characteristics with respect to contention parameters like transaction size or write ratio. These are the steps we have taken to move closer to this end:

- Empirically study the capacity constraints of hardware transactions to expose the hardware implementations that dictate these limits
- Articulate a set of contention parameters, like transaction size or write ratio, that sufficiently span the multicore design space and can be used to synthetically generate different cases of contention for benchmarking
- Discover performance trends of hardware transactions with respect to different contention parameters
- Capture these trends in a multivariate linear regression model that can be used to predict HTM performance in real multicore programs

We anticipate these contributions will provide a much needed understanding of hardware transactional memory to better enable its effective utilization in future multicore programs.

1.2 Related Work

Recently, several researchers have considered variations of hybrid transactional memory (HyTM) systems [5, 6, 14] which exploit the performance potential of recent HTM implementations, while preserving the semantics and progress guarantees of software transactional memory (STM) systems [19]. Underlying all of this work is the assumption that hardware constraints on the size of transactions are sufficiently unforgiving that elaborate workarounds are justified. For instance, Xiang et al. [23, 24] 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 transaction. Similarly, Wang et al. [22] use a 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. These related works validate the need for an understanding of the HTM capacity constraints.

Wang et al. [21] studied the performance sensitivity of HTM to a variety of application patterns. Our investigation takes this idea further by exploring HTM performance in a broader expanse of the multicore design space. For example, we also experiment with padding memory locations, varying the level of contention between threads, and varying the amount of work done between transaction attempts.

1.3 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 [9]. Upon completion of a transaction, the memory state is validated for consistency before the transaction ***commits***, making modifications to memory visible to other threads. Transactions may ***conflict abort*** when one thread's write set intersects at least one memory location in the read or write set of another thread, as illustrated in Table 1.1. In addition to conflict aborts, hardware transactions suffer from ***capacity aborts*** when the underlying hardware lacks sufficient resources to maintain the read or write set of an attempted transaction.

	$read_A(X)$	$write_A(X)$
$read_B(X)$	commit	abort
$write_B(X)$	abort	abort

Table 1.1: Read and Write Conflicts to Memory Location X Between Threads A and B

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.

1.4 Experimental Setup

The performance characteristics of hardware transactions are naturally dependent on the underlying hardware. The results from our experiments should only be fully accepted with respect to the microprocessors we specify in this section, although the conclusions will still generally apply to different generations of the hardware. 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
- 8MB 16-way shared L3 cache
- 32KB per-core 8-way L1 caches

We also tested an IBM Power8 processor with

- 10 cores running at 3.425GHz
- 80 hardware threads
- 128B cache lines
- 80MB 8-way shared L3 cache
- 64KB per-core 8-way L1 caches

All experiments are written in C and compiled with GCC, optimization level `-O0`.[2] Our experiments use the GCC hardware transactional memory intrinsics interface.

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

Chapter 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.

2.1 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 2-1 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

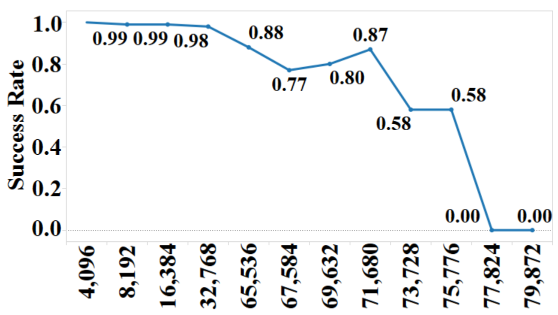


Figure 2-1: Lines Read vs Success Rate

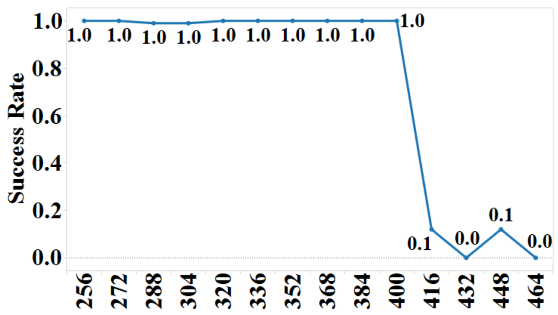


Figure 2-2: Lines Written vs Success Rate

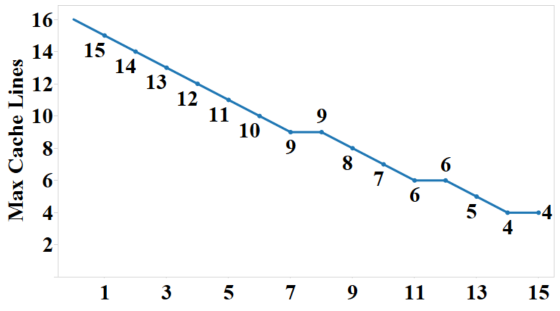


Figure 2-3: $\lg_2$ Stride vs $\lg_2$ Lines Readable

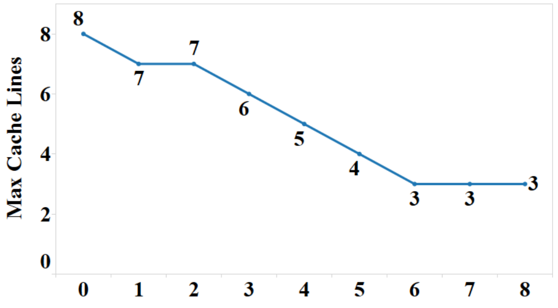


Figure 2-4: $\lg_2$ Stride vs $\lg_2$ Lines Writable

of the total capacity to fit perfectly into the L3 due to the hash function mapping physical address to L3 cache bank.

Figure 2-3 shows the result of a strided read-only access experiment. The stride amount indicates the number of cache lines 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 ($= 4$) (i.e. 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 access 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 2-2 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), thus we would not expect to fill all 512 lines perfectly.

Figure 2-4 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.

2.2 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.

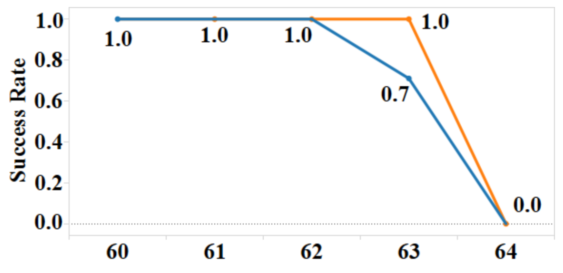


Figure 2-5: Lines Read / Written vs Success Rate

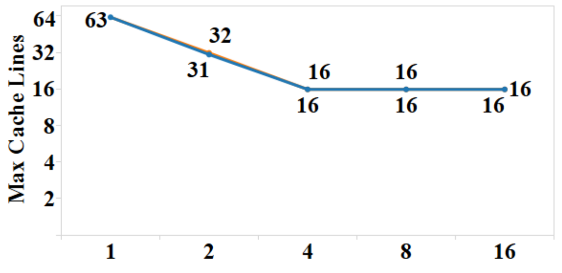


Figure 2-6: Stride vs Lines Readable / Writeable

The results of our sequential and strided access experiments for both read-only and write-only transactions appear to be identical in Figure 2-5 and Figure 2-6, 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.

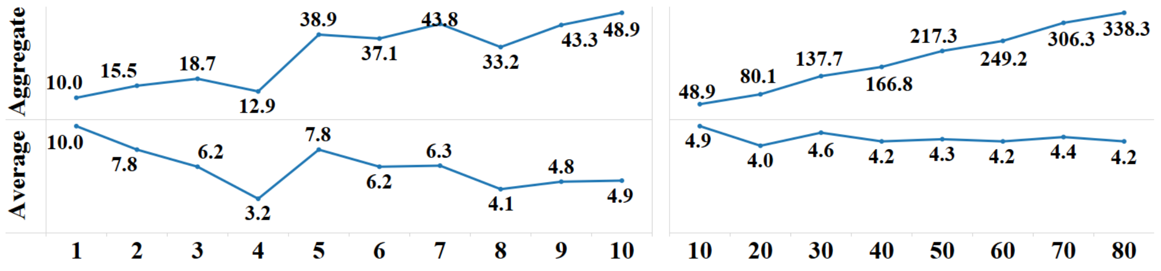


Figure 2-7: Number of Threads vs Committed Transactions (Thousands)

We see in Figure 2-7 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.

2.3 Implications

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 behavior of nontransactional code sharing a cache with transactional code might affect HTM performance, as well as how the access pattern of transactional code can limit transaction size. IBM's Power8 developers 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.

Chapter 3

Performance Characteristics

A program can be described by some combination of contention parameters, and it is distinguishable from another program if even a single parameter setting is different. For instance, a program with threads that only ever access 10 different memory locations is inherently different from one with 100 different memory locations, and that program is even further distinguishable from one with 1000 different memory locations. These programs illustrate a few of the many different cases of contention that exist in the multicore programming space.

To explore the behavior of HTM under these different cases of contention, we model the use of hardware transactions by a parameterizable array access experiment. A single run of the experiment involves measuring the aggregate throughput, given a specific setting of the contention parameters, of concurrent threads transactionally reading and / or incrementing counters of a shared array.

Even for a simple experiment like this, the space of all such multicore programs is infinite because of the unbounded variability of contention parameters. Thus, we constrained our parameter set and measured the performance characteristics of hardware transactions in this controlled space.

3.1 Multicore Basis Set

The parameter space of multicore programs consists of many variables such as the transaction size, the memory access pattern, or the number of concurrent threads. For both the Intel and IBM machines, we experimented using the following contention parameter set with corresponding values that sufficiently span the multicore program space; we term this the ***multicore basis set***:

- ***random*** $\in$ ***(0 1)***
  denotes sequential array access or random array access.

- ***padded*** $\in$ ***(0 1)***
  denotes accesses to a simple array of 32 bit counters or to one where individual counters are padded to cache lines.

- ***counters*** $\in$ ***(1 2 4 8 16 32 64 128 256 512 1024 2048 4096)***
  is the number of counters in the shared array; this simulates the level of contention in a program–fewer counters result in higher contention for those fewer memory locations, and vice versa.

- ***workBetween*** $\in$ ***(0 5 10 15 20)***
  represents the amount of nontransactional work done between each transaction. More specifically, threads execute a naive recursive fibonacci, *fib(workBetween)*, between transactions.

- ***workWithin*** $\in$ ***(1 5 10 15 20)***
  is the number of memory locations accessed within each transaction; it is the size of the critical section in a program.

- ***writeRatio*** $\in$ ***(0 1 10 25 50 75 100)***
  is the percentage of write accesses in each transaction. To elaborate, *writeRatio = 25* means that 25% of the array accesses are writes (increments) and 75% are reads.

rand	pad	counters	between	within	write %	threads	success	throughput
⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮
1	0	1024	5	5	25%	3	0.52	12.1M tx/s
1	0	1024	5	5	25%	4	0.36	10.6M tx/s
1	0	1024	5	5	50%	1	0.99	8.8M tx/s
⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮

Table 3.1: Intel x86 Example Experimental Results

- ***threads* $\in$ *(1 2 3 4 8 16 32 64)***
  is the number of threads that are concurrently trying to atomically read and/or increment the counters in the shared array. Note that our Intel machine experiments omit the *(8 16 32 64)* values because the machine only has 4 non-hyperthreaded cores.

For each cross product of our basis set, we run our experiment on that synthetically generated case of contention and we record throughput and transaction success rate:

- throughput $:= \frac{tx\_successes}{runtime}$
- success rate $:= \frac{tx\_successes}{tx\_successes + tx\_restarts}$

The result is 36400 performance measurements for the Intel machine, and 63700 measurements for the IBM machine. Table 3.1 illustrates exemplary measurements from the Intel results; the IBM results are identical in form.

3.2 Trends

With the 36400 Intel measurements and 63700 IBM measurements, we can plot throughput and success rate while modulating individual contention parameters in order to observe how those modulated parameters affect HTM performance. In all subsequent plots, the unmodulated parameters are marginalized by averaging the performance values.

Figure 3-1 illustrates the HTM performance difference between sequential and random memory accesses while varying the number of counters on the Intel machine.

With fewer than 8 counters, it hardly matters whether the access pattern is sequential or random because the contention for those few counters is so high that conflict aborts are rampant. However, as we increase the number of counters in the array and reduce the contention for those shared counters, we see much higher throughput when accessing memory sequentially compared to accessing memory randomly. This makes sense because when there are fewer conflict aborts, as is the case when there are more counters that the concurrently executing threads can operate on, then optimizations like data prefetching during sequential access begin to make observable differences in performance. The results for the IBM machine in Figure 3-2 are very similar to those of the Intel machine. When the number of counters is small, we see that there is little difference between random access and sequential access. With more counters, we again observe that sequential memory access results in higher transaction throughput than random memory access. The implication of these results is that when programming with hardware transactions, accessing memory sequentially will generally result in higher performance than accessing memory randomly.

A common optimization in concurrent programming is to pad memory accesses to reduce false sharing.[1] We now do a comparison between unpadded versus padded memory accesses that is similar to the previous analysis of sequential versus random memory accesses. In Figure 3-3 we observe slightly mixed results for the Intel machine, with some evidence of higher transaction throughput when padding memory accesses. The reason for the slightly mixed results is because there are two conflicting effects of padding. First, padding memory accesses reduces false sharing and reduces conflict aborts, thus improving performance. Second, padding a single 4 byte counter to the full 64 byte Intel machine cache line results in less batch accessing, which can actually reduce performance because many more (up to 16x) cache lines may need to be fetched in the padded case than in the unpadded case when accessing the same number of counters.

[1]False sharing occurs when two logically independent memory locations reside on the same cache line and one or both of those memory locations are accessed by different threads, resulting in an invalidation of the whole cache line. Padding memory locations to reside entirely on different cache lines eliminates this false sharing problem.

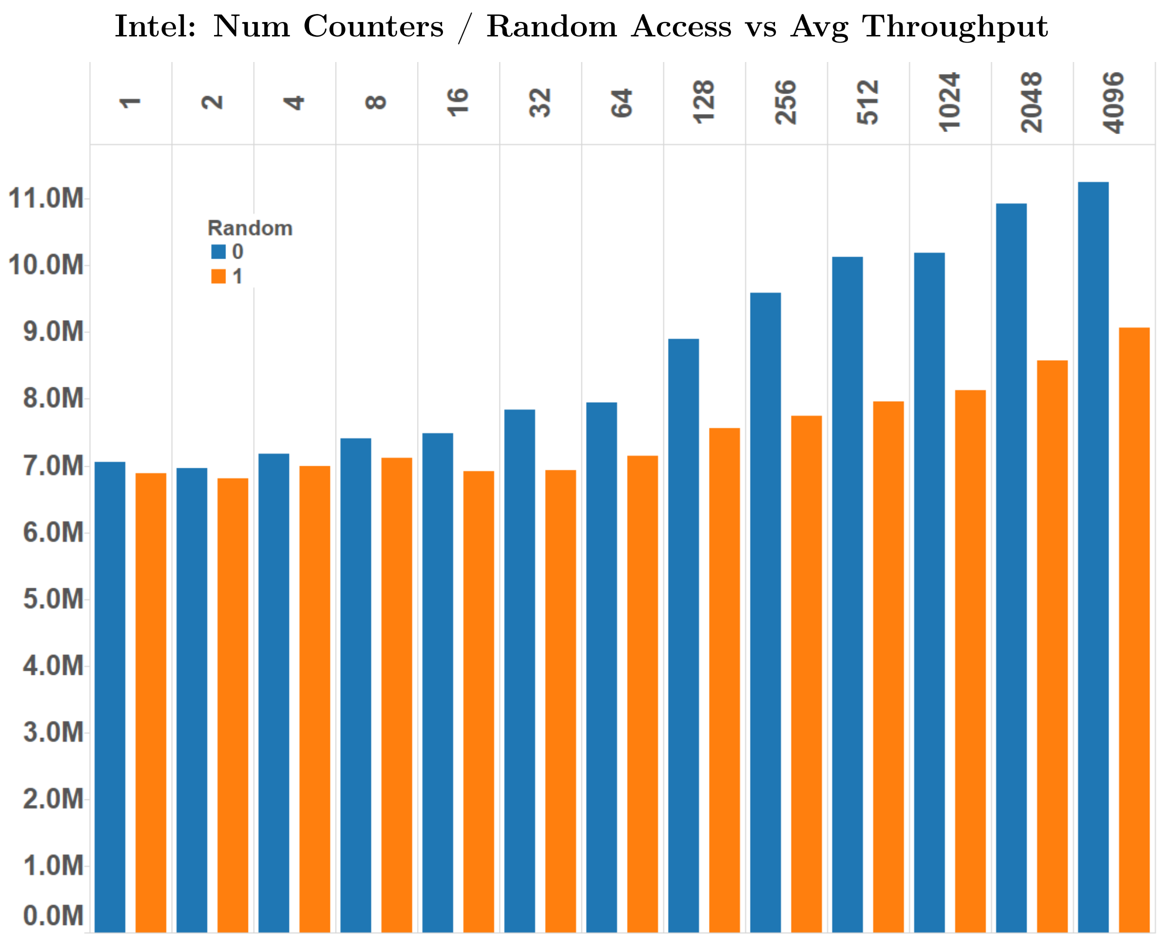


Figure 3-1: Sequential memory access results in higher transaction throughput than random memory access on the Intel machine

In Figure 3-4 we actually see distinct regions in the IBM results when one phenomenon dominates the other. When the probability of conflicting memory accesses between threads is sufficiently high due to contention ($\leq$ 1024 counters), padding memory accesses results in higher throughput because false sharing is reduced and the rate of conflict aborts is reduced. However, when the probability of conflicting accesses is lower ($\geq$ 2048 counters), we see the performance penalty of unbatched memory accesses overcome the performance benefit of reduced false sharing. The IBM machine cache line is 128 bytes wide, which means that up to 32x more cache lines may need to be fetched in the padded case than in the unpadded case when accessing the same number of counters; this makes the unbatched access penalty much more significant on the IBM machine than the Intel machine.

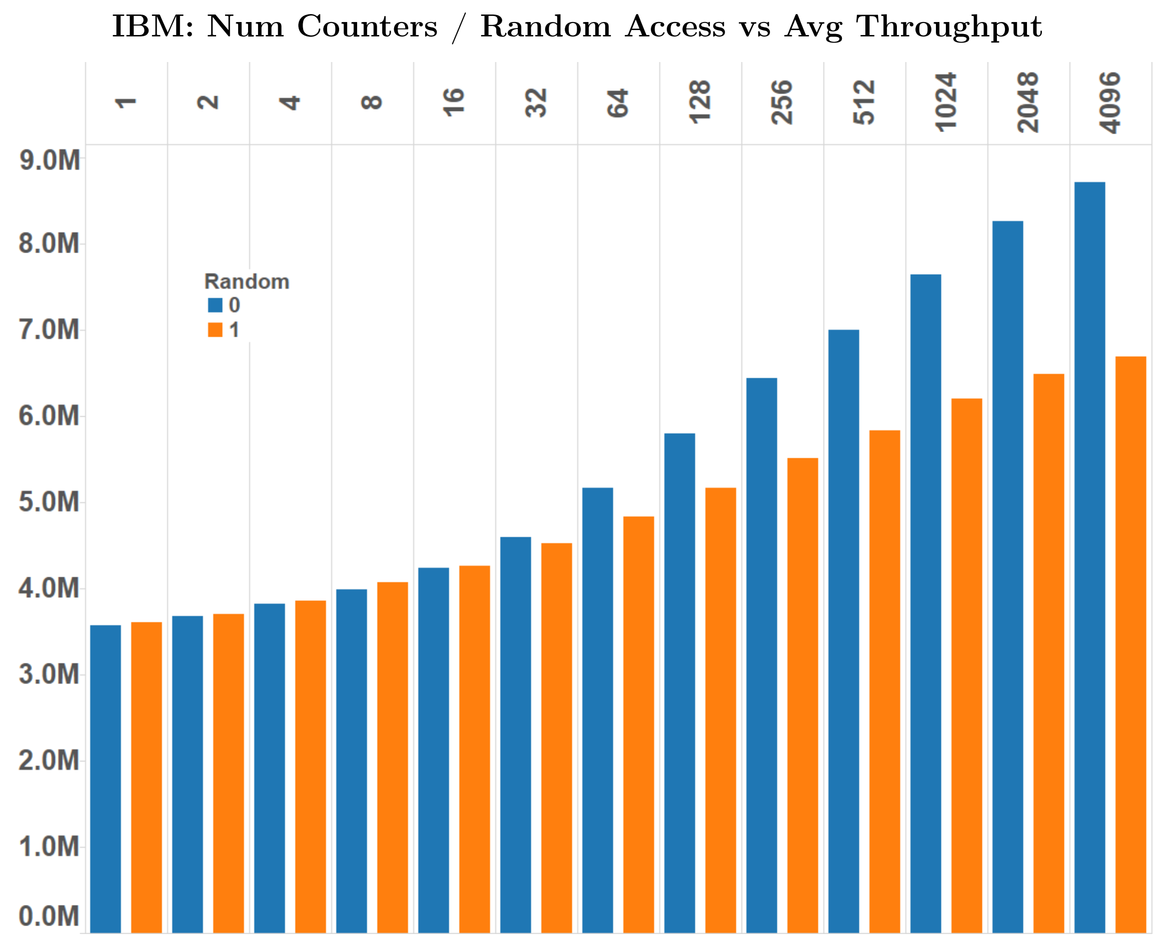


Figure 3-2: Sequential memory access also results in higher transaction throughput than random memory access on the IBM machine

From these observations, we conclude that padding memory accesses generally improves transaction performance on the Intel machine, but the effect on the IBM machine depends on the level of contention.

Next we examine the effect of modulating the write ratio while also varying the number of threads. Each individual block in Figure 3-5 is labeled with the specified number of threads and write ratio, along with the measured throughput and success rate on the Intel machine. Throughput is visually depicted by the size of the block–the larger the block, the higher the throughput. We see that as we increase the number of threads, which is visualized by the color of the blocks, throughput increases and success rate decreases. The increased performance makes sense because more work can be done with more concurrent threads; this increase is sublinear, however, as

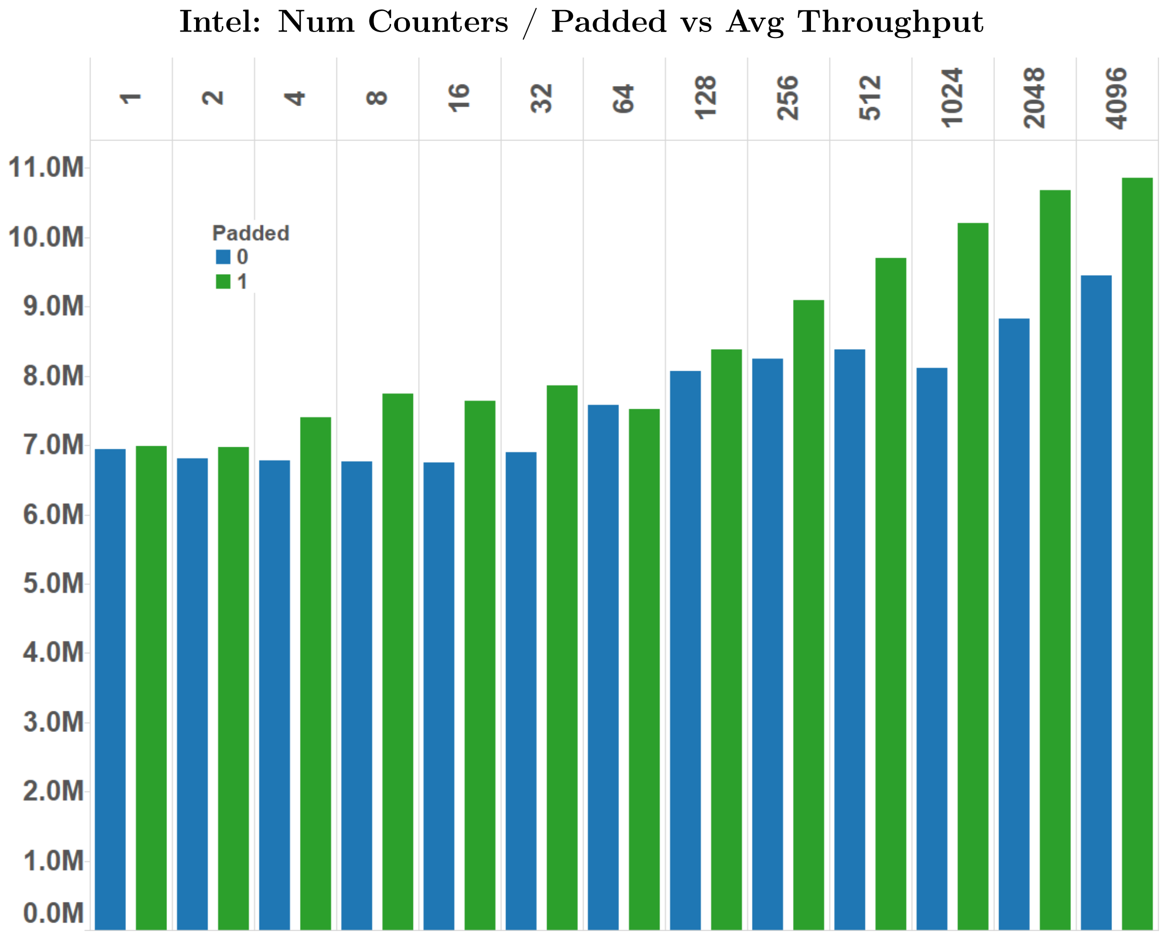


Figure 3-3: Padding memory accesses on the Intel machine generally improves transaction performance

IBM: Num Counters / Padded vs Avg Throughput

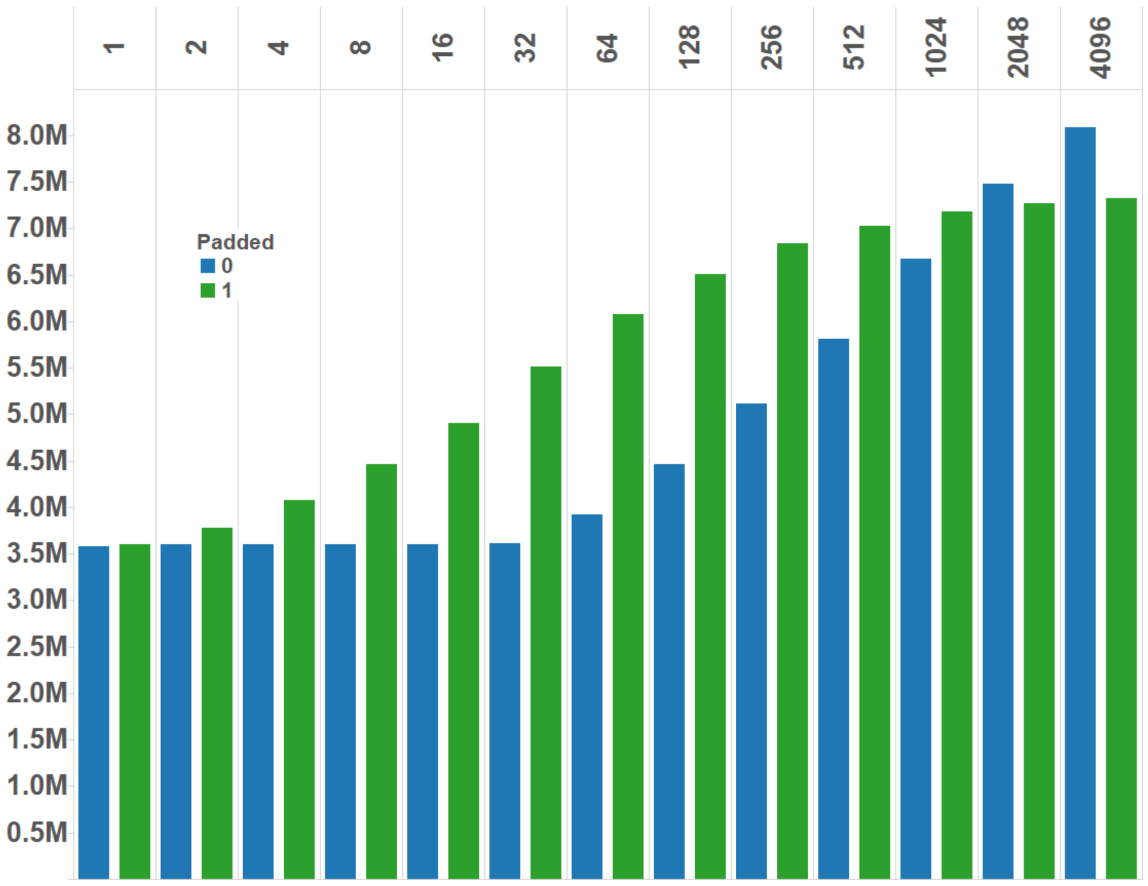


Figure 3-4: Padding memory accesses on the IBM machine improves transaction performance when contention is high, but it reduces performance when contention is low

increasing the contention by adding more threads also has the effect of increasing conflict aborts which lowers throughput.

We can further break down Figure 3-5 by examining the effect that the write ratio, which is visually represented by the color gradient of the blocks, has on transaction performance across different numbers of threads. For a single thread, the percentage of writes to reads generally has no effect on the throughput or success rate. For any number of threads greater than 1, however, we see that the higher the write ratio is and the darker the block is, the lower the throughput and success rate is and the smaller the block is. There is a clear trend indicating that transaction performance of a concurrent program is negatively correlated with the write ratio, and the magnitude of this negative relationship increases as the number of threads increases: the performance for 0% writes is 2.27x the performance for 100% writes in the case of 2 threads, 2.84x in the case of 3 threads, and 3.43x in the case of 4 threads.

In Figure 3-6 we have an analagous picture for the IBM machine. Each block is again labeled with thread count / write ratio / throughput (M tx/s) / sucess rate. We omit the data corresponding to the cases of fewer than 4 running threads for lack of space in the figure. We similarly observe that as we increase the number of threads, throughput increases and success rate decreases, and the negative correlation between performance and write ratio increases in magnitude as the number of threads increases. The IBM results contain data for very large thread counts, and the effects of modulating write ratio is much more evident than when analyzing results on the Intel machine. On the IBM machine, the marginal difference between 0% writes and 1% writes results in a huge performance difference, 1.39x throughput, for the case of 32 threads, and the difference is even more significant, 1.9x, for the case of 64 threads. With so many concurrently running threads, even the slightest increase in contention causes conflict aborts to surge, thus reducing performance greatly. For multicore programs with sufficiently high write ratios, the throughput gain from increasing the number of threads might hardly be worth the cost. For instance, the performance for 64 threads is 2.2x the performance for 4 threads in the case of 25% writes, despite the 16x increase in resources used.

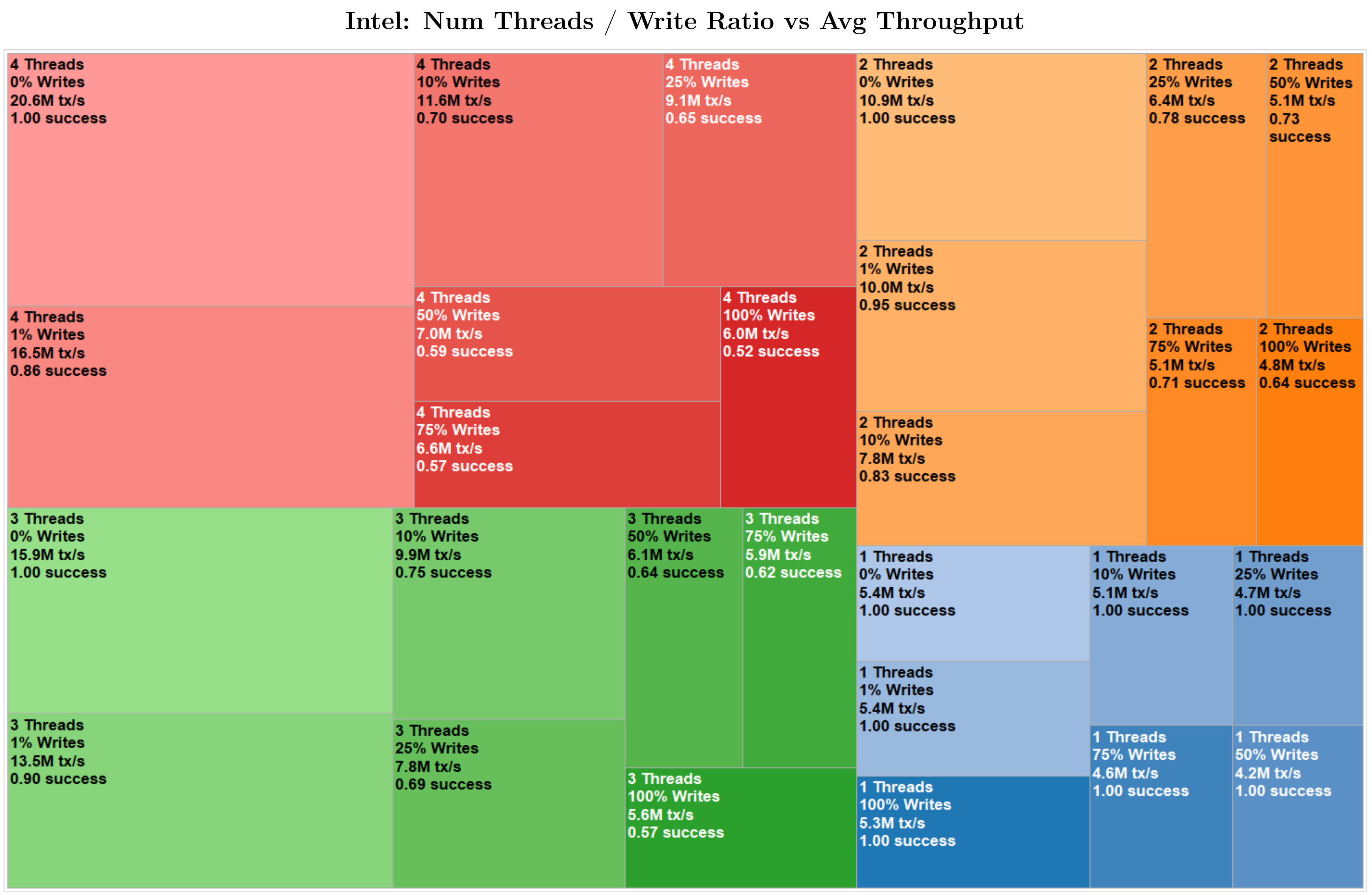


Figure 3-5: Performance is negatively correlated with write ratio, and the magnitude of this relationship increases with thread count

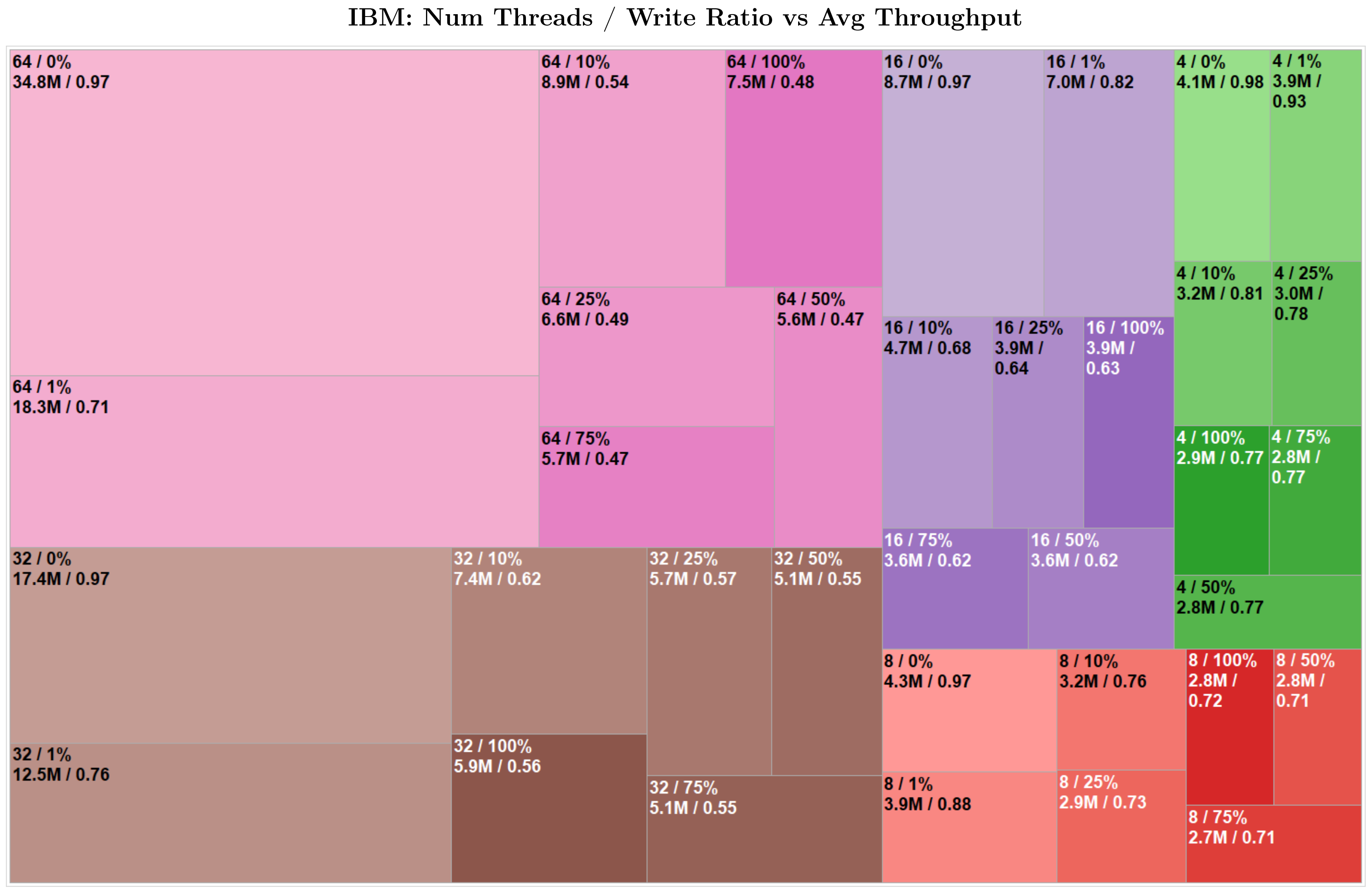


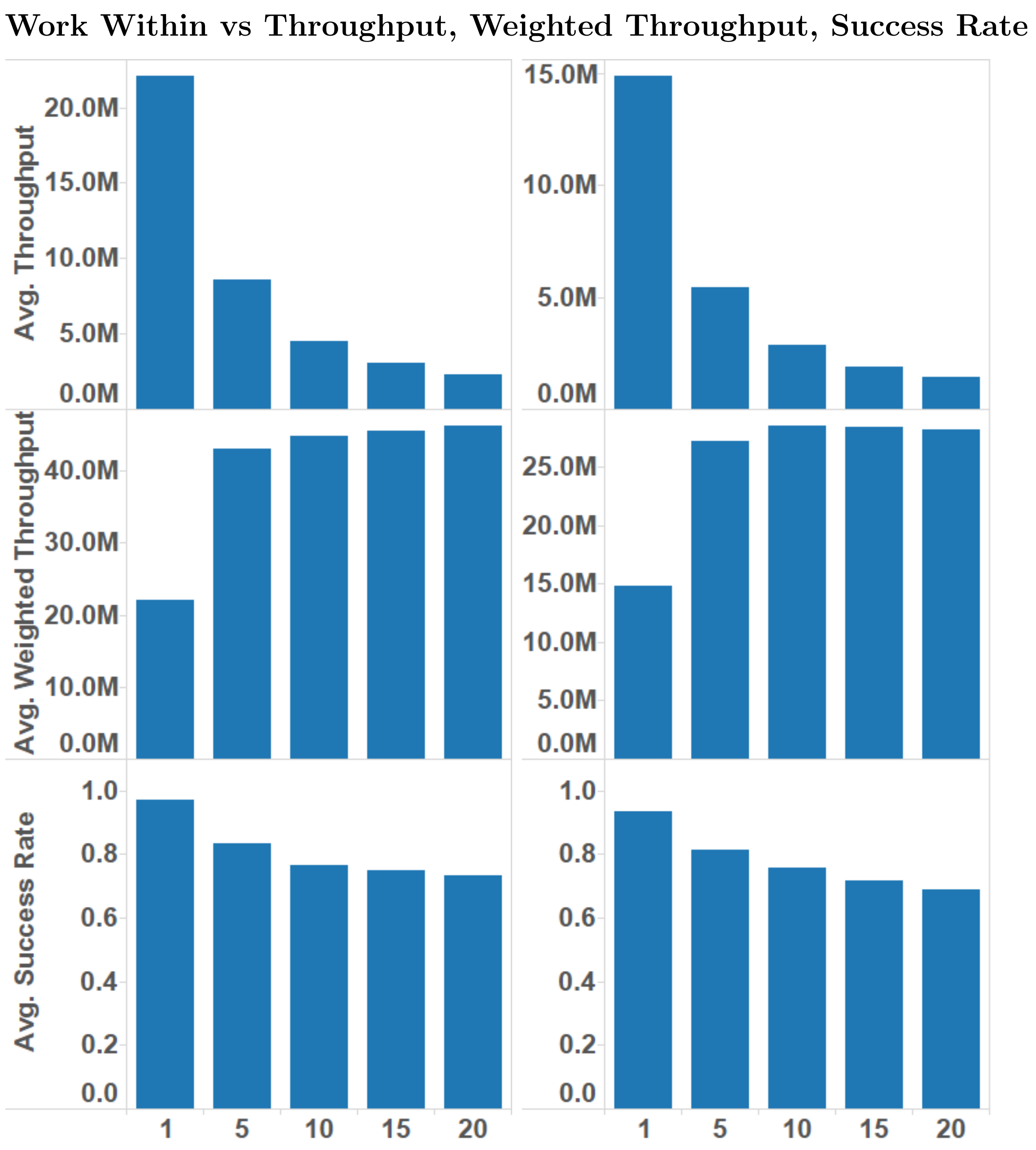
Figure 3-6: Performance is negatively correlated with write ratio, and the magnitude of this relationship increases with thread count

A peculiar observation about the IBM results is that performance is slightly better in the case of 100% writes than 50% writes or 75% writes. It could be that the mechanisms in place for implementing transactions in the IBM Power8 hardware favor homogenous (i.e. read-only or write-only) transactions, but this is pure speculation.

Note that even when there are no writes and no conflict aborts, the success rate is not 1.00 because the IBM machine is sensitive to capacity aborts, as we previously discovered. Even for fairly small transaction sizes of 20 counters, it is possible for these 20 counters to reside on at least 16 different cache lines that map to the same cache set, which will cause a capacity abort because the dedicated 4-set, 16-way cache will not be able to fit that transaction. This point serves to illustrate the significance of understanding the capacity constraints of hardware transactions.

When we increase transaction sizes, we naturally expect lower throughput, which is measured as *transactions* completed per second, because there is simply more work being done within each transaction. The top two plots in Figure 3-7 exactly illustrate this intuition for both the Intel and IBM machines. However, when analyzing weighted throughput–which is calculated as $throughput * workWithin$–we actually see an increase in the number of *operations* completed per second in the middle two plots. There is inherent overhead to implementing a hardware transaction, regardless of the amount of work done within it, so when we increase the transaction size, the fixed cost is amortized. The large jump in weighted throughput from $workWithin = 1$ to $workWithin = 5$ suggests that there is increased efficiency in batching operations within a transaction. Beyond $workWithin = 5$, however, there are diminishing gains to weighted throughput because larger transactions also raise the probability of conflict aborts, thus lowering transaction success rate, as depicted in the lower plots of Figure 3-7. From these observations we anticipate that an optimal value for hardware transaction size is around 5, because this value seems to balance the performance benefit of batching operations with the performance penalty of increased conflict aborts.

The work a program does between critical sections is inherent to the program and significantly affects the transaction throughput of that program. We can draw



(a) Intel (b) IBM

Figure 3-7: Increasing the work within transactions decreases throughput, measured as transactions completed per second, but increases weighted throughput, measured as operations completed per second

meaningful insights by viewing how *workWithin* interacts with *workBetween* to affect performance because these two parameters together determine the ratio of critical to noncritical sections in a program. We plot, for different values of *workBetween*, the effect of modulating transaction size on the Intel machine in Figure 3-8. When the amount of work that threads do between transactions is minimal and the program is frequently in the critical section, as is the case when $workBetween \leq 5$, increasing transaction size significantly decreases throughput as we observed before. On the other hand when there is more time between critical sections, such as when $workBetween \geq 10$, we see that the amount of work done within each transaction has less of an influence on performance because less of the program runtime is spent executing transactional code. The results on the IBM machine are very similar and have thus been omitted. While these observations fall in line with our expectations and may not appear novel, it is still meaningful to empirically validate our intuitions in this effort to fully understand the performance characteristics of HTM under different cases of contention.

Intel: Work Between / Work Within vs Avg Throughput, Success Rate

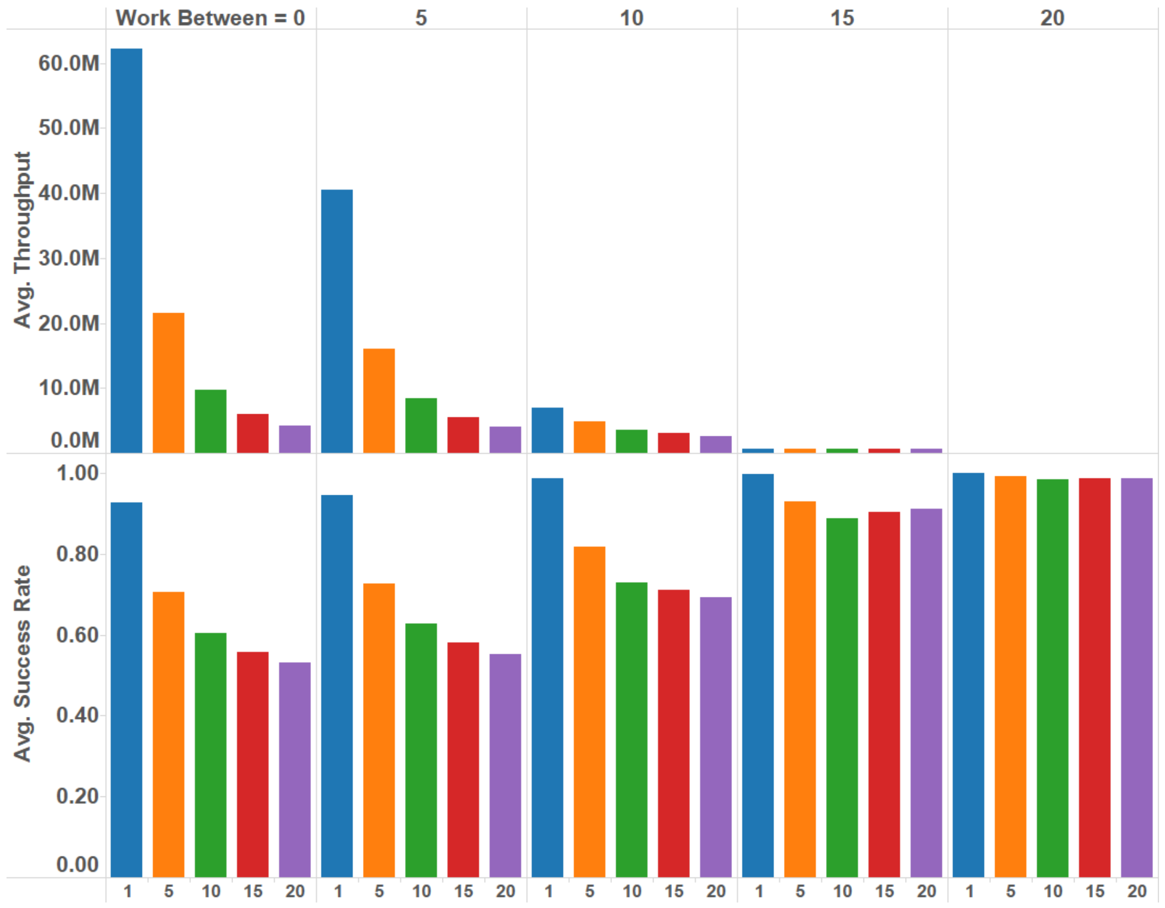


Figure 3-8: The amount of work done within a transaction significantly affects performance when concurrent threads do little work between transactions and are frequently in the critical section. The work within does not matter as much when the amount of work between transactions is large and relatively little time is spent in the critical section

Chapter 4

Throughput Prediction

In analyzing the experimental results from our synthetically generated contention experiments, we found many compelling performance trends that suggest the potential for predicting HTM performance in programs that are not precisely defined in our multicore basis set. We wanted to leverage the large amount of performance data we collected to somehow enable a prediction about other points in the infinite multicore programming space; Figure 4-1 illustrates our goal to predict the throughput of any arbitrary, real program.

To this end, we trained[1] a multivariate linear regression model on each of our Intel and IBM experimental result sets. The goal of these two models is to be able to predict the throughput of any multicore program that synchronizes using HTM.

4.1 Multivariate Linear Regression Models

To train the multivariate linear regression models, we first transformed the Intel and IBM result sets using a radial basis function [16] with $\gamma = 0.0001$ in order to improve the fit, because some of the first degree relations were found to be nonlinear. These transformed results were then used as input training data for the models.[2] To mitigate the problem of overfitting to the training data, we methodically generalized

[1]We used a supervised learning algorithm.
[2]We used the python scikit-learn module.

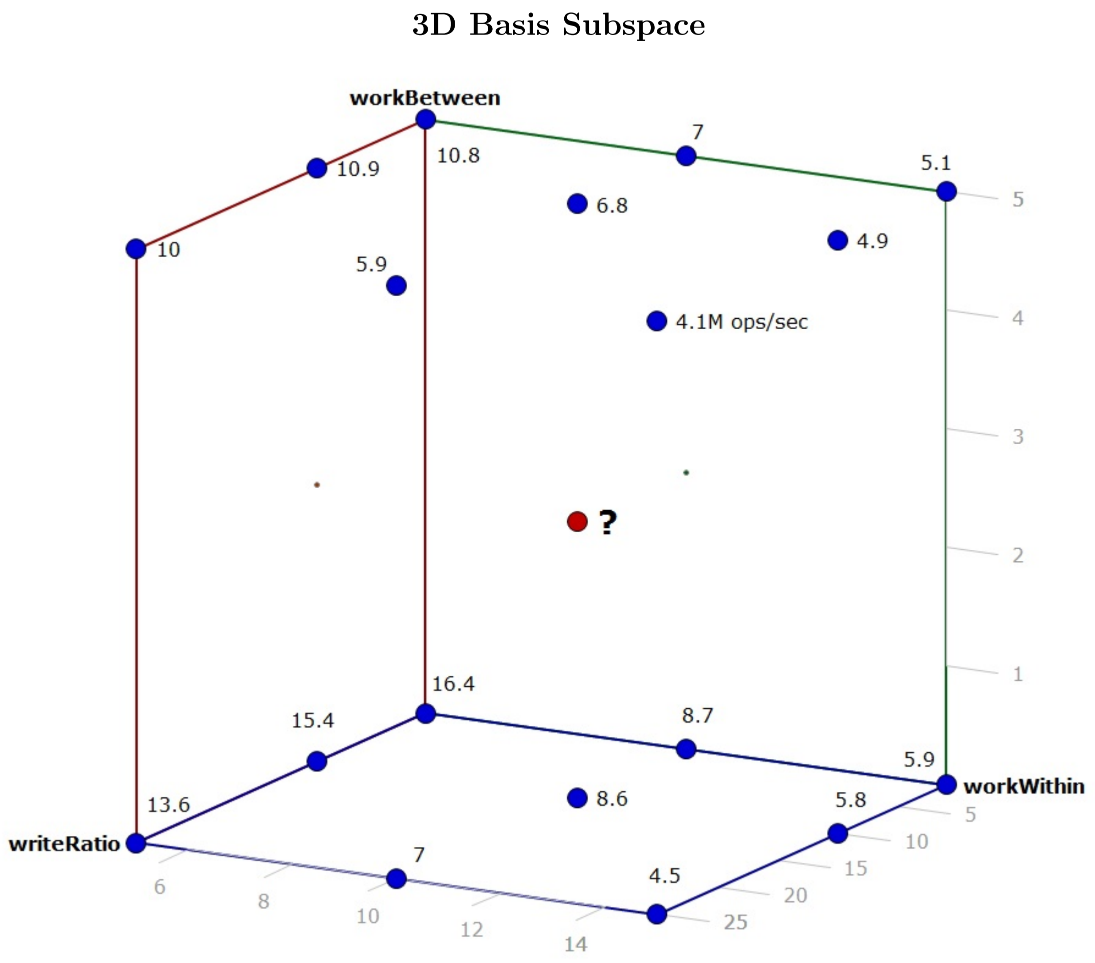


Figure 4-1: Projection of a linear probing hash table into a 3D subspace of our parameter set. Knowing the throughput of adjacent points in our multicore basis set should somehow inform us of the throughput of the unknown point

the models with 5-folds cross validation [18]. The resulting goodness-of-fit [1] values, $R^2_{intel} = 0.96$ and $R^2_{ibm} = 0.90$, for both models were quite high, and this reinforces our intuition about the potential for predicting the throughput of programs that utilize hardware transactions.

4.2 Basis Parameters Decomposition

In order to use the multivariate linear regression models to predict the throughput of an arbitrary multicore program, one must first decompose the program into a vector of parameters that matches the dimensionality of our basis set:

<random, padded, counters, workBetween, workWithin, writeRatio, threads>

Most parameters are either binary (*random, padded*), or they are straightforward approximations (*counters, workWithin, writeRatio, threads*); the one confounding parameter is *workBetween*. To measure the work done between critical sections of a program, a simple Intel pintool [2] can be used to instrument the program to count the number of CPU instructions both inside and outside of a program critical section. Recall that we modeled this parameter in our experiment as the execution of a naive recursive fibonacci, *fib(workBetween)*, between transactions. Considering the algorithmic complexity of naive fibonacci is $O(2^N)$, the parameter *workBetween* can thus be calculated by the formula:

$$workBetween = log_2(\frac{\#outside}{\#inside} \cdot workWithin)$$

Figure 4-2 captures the process of decomposing an actual program and using the resulting vector of parameters to produce a throughput prediction from the multivariate linear regression.

4.3 Empirical Validation

To empirically validate the regression models, we compared the predicted throughput values to actual measured values for three concurrent data structures: a stack, a

Throughput Prediction Flow

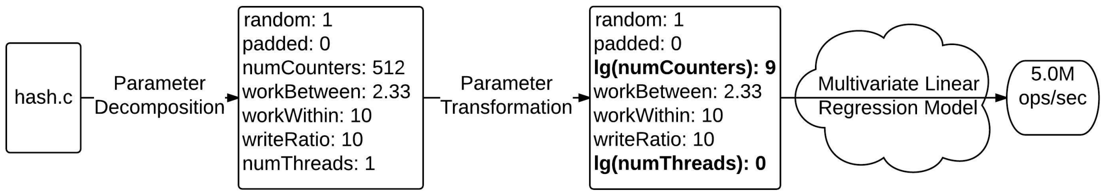


Figure 4-2: Decomposing a hash table implementation into a vector of parameters to pass into the multivariate linear regression model for throughput prediction. The parameter transformation step is an artifact of our model training process when we further transformed the input data to maximize regression fit

linear probing hash table, and a skiplist. Table 4.1 shows the results comparing the predicted and measured values for the Intel machine, and Table 4.2 shows the results for the IBM machine. In each case, we decomposed the concurrent data structure into a vector of parameters using the described method before applying the multivariate linear regression models to predict throughput.

While the predictions on the Intel machine are not 100% accurate, they are at least a reasonable approximation from the actual stack, linear probing hash table, and skiplist measurements. The same is true of the stack and hash table measurements on the IBM machine. These measurements empirically validate the accuracy of our regression models.

We were unable to record an actual measurement for the concurrent skiplist implementation on the IBM machine. When the skiplist became sufficiently large, the critical section accessed too many different memory locations, thus exceeding the maximum HTM capacity. Infinitely repeating capacity aborts left the program in a state of livelock [10] and the execution never finished. While failure to predict this livelock scenario is a failure of the IBM regression model, we still consider this example to be constructive because it validates the need for an understanding of the limitations of hardware transactions–without the knowledge we found from our capacity constraint experiments, we may have never realized the problem in this execution.

	stack	hash	skiplist
random	0	1	1
padded	1	0	0
$log_2(counters)$	0	9	12
workBetween	5.19	2.33	2.15
workWithin	2	10	18
writeRatio	50%	10%	20%
$log_2(threads)$	0	0	0
predicted	15.2M ops/sec	5.0M ops/sec	3.3M ops/sec
actual	17.2M ops/sec	4.1M ops/sec	2.6M ops/sec
error	13.2%	22.0%	26.9%

Table 4.1: Intel: Comparing Predicted and Measured Throughput of Concurrent Data Structures

	stack	hash	skiplist
random	0	1	1
padded	1	0	0
$log_2(counters)$	0	9	12
workBetween	5.19	2.33	2.15
workWithin	2	10	18
writeRatio	50%	10%	20%
$log_2(threads)$	0	0	0
predicted	4.7M ops/sec	1.5M ops/sec	1.7M ops/sec
actual	5.3M ops/sec	1.9M ops/sec	N/A
error	12.8%	26.7%	N/A

Table 4.2: IBM: Comparing Predicted and Measured Throughput of Concurrent Data Structures

4.4 Use Case

With these models, a programmer can now simplify the design of a multicore program that synchronizes with hardware transactions. An illustrative use case is to leverage a model to predict the throughput of different design iterations of a program (each with different respective parameter decompositions), compare the predicted throughputs, and select the design iteration that yields the best predicted performance. While the regression models may not predict *absolute* performance metrics well, they will be able to sufficiently capture the *relative* performance difference between iterations.

This distinction is enough to inform a decision about the most high-performing design, even before any programming investment is made.

Chapter 5

Closing

5.1 Future Work

Due to limitations of the existing Intel hardware, the Intel machine experiments did not involve more than 4 non-hyperthreaded hardware threads. It will be meaningful to further explore the Intel performance characteristics with more hardware threads once larger chips with HTM support are developed.

GCC optimization level `-O0` was used in our experiments because we were interested in investigating the pure performance characteristics of hardware transactional memory without the confounding effects that would come with different compiler optimization levels. That said, compiler optimizations are necessary to build the fastest programs. A future study of high-performing multicore C programs using HTM should include different optimization levels.

5.2 Conclusion

With the advent of hardware transactional memory in new Intel and IBM microprocessors, a new solution to the synchronization problem in multicore programs is available. We ran capacity constraint benchmarks to expose the hardware implementations that dictate the limits of HTM. We gathered synthetically generated performance data that informed us about how different cases of contention correlate with hardware transaction performance. We captured these performance trends in multivariate linear regression models that we have shown to be useful in predicting the throughput of arbitrary concurrent programs and facilitating their design. We anticipate that the contributions from this investigation will provide a much needed understanding of HTM, ultimately enabling its proliferation into future high-performing multicore programs.

References

[1] P. Bentler and D. Bonett. Significance tests and goodness of fit in the analysis of covariance structures. *Psychological Bulletin*, 88(3):588–606, November 1980.

[2] S. Berkowits. Pin - a dynamic binary instrumentation tool. Intel Developer Zone, June 2012.

[3] 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. ACM.

[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, Sept. 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. ACM.

[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. ACM.

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

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

[9] 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. ACM.

[10] A. Ho, S. Smith, and S. Hand. On deadlock, livelock, and forward progress. Technical Report UCAM-CL-TR-633, Cambridge University Computer Laboratory, May 2005.

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

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

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

[14] 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. ACM.

[15] R. Merritt. IBM plants transactional memory in cpu, August 2011.

[16] M. J. L. Orr. Introduction to radial basis function networks, April 1996.

[17] J. Reinders. Transactional synchronization in haswell. Intel Developer Zone, February 2012.

[18] J. D. Rodríguez, A. P. Martínez, and J. A. Lozano. Sensitivity analysis of k-fold cross validation in prediction error estimation. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, pages 569–575, January 2010.

[19] N. Shavit and D. Touitou. Software transactional memory. In *Proceedings of the Fourteenth Annual ACM Symposium on Principles of Distributed Computing*, PODC '95, pages 204–213, New York, NY, USA, 1995. ACM.

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

[21] M. D. Wang, M. Burcea, L. Li, S. Sharifymoghaddam, G. Steffan, and C. Amza. Exploring the performance and programmability design space of hardware transactional memory. In *TRANSACT '14*, 2014.

[22] 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. ACM.

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

[24] 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. ACM.

[25] 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. ACM.