Understanding Transactional Memory Performance

Donald E. Porter and Emmett Witchel
The University of Texas at Austin
{porterde,witchel}@cs.utexas.edu

***Abstract*—Transactional memory promises to generalize transactional programming to mainstream languages and data structures. The purported benefit of transactions is that they are easier to program correctly than fine-grained locking and perform just as well. This performance claim is not always borne out because an application may violate a common-case assumption of the TM designer or because of external system effects. This paper carefully studies a range of factors that can adversely influence transactional memory performance.**

In order to help programmers assess the suitability of their code for transactional memory, this paper introduces a formal model of transactional memory as well as a tool, called Syncchar. Syncchar can predict the speedup of a conversion from locks to transactions within 25% for the STAMP benchmarks. We also use the Syncchar tool to diagnose and eliminate a starvation pathology in the TxLinux kernel, improving the performance of the Modified Andrew Benchmark by 55% over Linux.

The paper also presents the first detailed study of how the performance of user-level transactional programs (from the STAMP benchmarks) are influenced by factors outside of the transactional memory system. The study includes data about the interaction of transactional programs with the architecture, memory allocator, and compiler. Because many factors influence the performance of transactional programs, getting good performance from transactions is more difficult than commonly appreciated.

I. Introduction

Transactional memory [11, 14] is a promising paradigm to simplify concurrent programming. Transactions relieve the programmer from worry about deadlock, making it easier to code correctly. However, most real-life programs need to be both correct and efficient, and the introduction of transactions to an application can have negative, counterintuitive consequences for performance. Based on several years' experience working with transactional memory, this paper analyzes a range of issues that lead to performance problems in transactional applications, with a focus on helping developers and applications transition from using locks to transactions.

Designers of transactional memory systems make implementation decisions based on an implicit model of common-case application behavior. For instance, TM designs often trade faster commits for slower aborts, and as a result applications with high-contention transactions can perform much worse than with locking. In many cases, application data structures can be reorganized to improve transactional performance, but these opportunities are not necessarily obvious upon inspection of the code. Thus, it is important for developers to have tools that help them to diagnose and to correct performance problems.

To help developers characterize and tune the performance of transactional programs, this paper develops and validates *Syncchar*, or synchronization characterization, a formal model of transactional memory performance and a tool based on the model. The Syncchar model starts with a lock-based or transactional parallel application and samples the sets of addresses read and written during critical sections. It then builds a model of the program's execution that can predict the performance of the application if it used transactions. The model has two key metrics, *data independence* and *conflict density* of the critical regions. Data independence measures the likelihood that threads will access disjoint data. Conflict density measures how many threads are likely to be involved in a data conflict should one occur. Because most large-scale parallel applications use locking, a key use for the Syncchar model is identifying which applications could benefit from using transactions before investing the engineering effort to make such a conversion.

The Syncchar model is approximate, but useful, both for the critical regions that it predicts will be performance problems and for those that it predicts will perform well. Programmers can focus time reorganizing code in critical regions that will serialize when using transactions. Working with a performance model helps programmers tune transactional performance, which requires different intuitions from tuning lock-based code. For instance, common lock-based programming techniques, such as walking a linked list or incrementing a shared counter inside a critical region, can reduce transactional performance.

This paper also presents a case study of performance tuning the TxLinux kernel [29, 31]. TxLinux is a version of the Linux kernel converted to use hardware transactional memory. This study demonstrates the often counterintuitive nature of tuning transactional performance and how Syncchar analysis can help focus the tuning effort.

Performance problems for transactional applications can also arise from the interaction of the TM system with unrelated portions of the system. For instance, compiler optimizations that avoid branches on a deeply pipelined processor can dramatically increase the conflict rate for a critical region. These issues are difficult for the application developer to anticipate and to debug. Ultimately, these issues must be resolved with better integration of the TM implementation with the rest of the system stack, but the Syncchar tool can help developers assess whether their code needs tuning or a performance problem is attributable to system effects.

This paper is the first to present data on how the system influences the performance of user-level transactional programs. The paper discusses the effect of various factors on performance, including the input size, architectural features, standard libraries and the compiler.

This paper contributes the following:

1) A new analytic model that predicts how application performance will scale with transactions.
2) Experimental validation of the analytic model, characterizing the performance of workloads from the STAMP benchmark suite [20] with an average error rate of 25%.
3) Detailed measurements and case studies of whole-system HTM behavior, including architectural effects, the memory allocator, and the compiler.
4) A case study of performance tuning the TxLinux kernel, in which we improve the performance of the Modified Andrew Benchmark at 8 CPUs by 55% relative to unmodified Linux.

This paper describes the need for a transactional memory performance prediction and tuning tool (Section II). The paper then presents novel techniques and a tool for investigating the potential of transactional memory, through measuring data independence and conflict density of critical regions protected by the same lock. (Section III). Next, the paper validates the model with a set of microbenchmarks and the STAMP benchmark suite (Section IV). The paper then presents detailed measurements and case studies of system effects on transactional memory performance (Section V). Finally, the paper applies the methodology and tools to tune the performance of the TxLinux kernel [29, 31] (Section VI). Section VII describes implementation details of the Syncchar tool, Section VIII presents related work, and Section IX concludes.

II. Tuning transactional performance is difficult

Understanding and tuning transactional performance requires different skills and intuitions from tuning lock-based concurrent code because the factors that influence performance are different. Tuning the performance of lock-based programs generally involves identifying highly contended locks and breaking them down into smaller locks, or restructuring the data to avoid synchronization (e.g., per-thread data structures and read-copy update [19]). Tuning transactions, on the other hand, requires a reduction in conflicting memory operations (concurrent transactions writing the same data). Although there are some known techniques for avoiding memory conflicts, the process can be difficult and counterintuitive.

Many transactional memory implementations also have substantial limitations that complicate a conversion from locking to transactions. For instance, HTM systems that use cache coherence either require that the address set of a transaction fit in the L1 cache or fall back on a software-based mutual exclusion mechanism. TM systems also have difficulty isolating and rolling back the effects of system calls inside of a transaction. While many transactional applications can work around the limits of a TM implementation in practice, these issues can make adoption of transactions more complicated than a rote replacement of locks with atomic blocks.

Given these challenges to adopting transactional memory, a predictive performance model such as Syncchar can help developers assess whether a migration to transactional memory is worth the investment. Syncchar is also useful as a standard profiling tool for transactional applications, giving developers insight into which code is most likely to become a scalability bottleneck.

If performance of a converted system does not meet expectations, Syncchar can help assess whether the problem lies within the application or the transactional memory implementation. As we show in Section V, many parts of the overall computer system can affect the performance of a transactional program. The transactional memory programming community can also benefit from a predictor of performance in developing standard benchmarks for transactional memory implementations and reasoning about the performance of a particular system.

A. Transaction tuning ≠ lock tuning

A common optimization in lock-based programming is caching attributes, such as the number of elements in a data structure. Caching attributes saves work when they are needed, and updating them introduces no synchronization overhead under locking, since the lock protecting the data structure is already held when the data structure is updated. With transactional memory, however, multiple threads updating the same memory location serialize concurrent execution. Cached attributes can be particularly prone to becoming a bottleneck because of the work wasted on a transaction restart. If updates are made late in the critical section and if values are read early in others, the work lost on a transaction restart increases further.

Splitting counters that cannot be eliminated into per-thread counters avoids conflicts on counter writes, but introduces conflicts on reads, as each CPU's value must be read to provide a correct sum. Per-thread counters thus work well if they are updated more frequently than they are read, but are pointless otherwise.

Linked lists are commonly used as a generic container because they are simple to implement, have a low memory overhead, and don't have problems with resizing. Linked lists, however, can be pathologically bad for transactional memory, because each thread traverses the exact same pointer path inside a critical region. Any pointer update will conflict with all other critical regions that have walked further down the list, despite the fact that concurrent execution of the operations is semantically safe.

B. System effects

Transactional programs are influenced by the program input size, hardware architecture, operating system, and sometimes the memory allocator and the compiler. Each subsystem can affect performance, and their interplay can also be significant.

For instance, the responsiveness of the operating system scheduler can lead to load imbalance or failure to concurrently schedule transactional threads. Poor scheduling can cause traditional transaction metrics, like restart rate, to improve despite worsening execution time. Similarly, limitations of a hardware system, such as overflowing the cache or taking a TLB miss, can cause a transaction to restart and fall back on a slower software path. These effects are often difficult to predict by code inspection.

Despite a seemingly straightforward appearance, tuning the memory access patterns of a transactional program can be subtle and non-intuitive. Even simple rules, like splitting counters shared by threads into per-thread counters, can be pointless or counterproductive when applied by rote. In a large, complicated system, programmers will need tools like Syncchar to identify where their tuning efforts are best spent.

III. The Syncchar model

The Syncchar model formalizes the intuition that transactional performance is primarily determined by the number of conflicting transactions, and provides the basis for the Syncchar performance tuning tool. This section explains the intuitions behind the Syncchar model and provides a rigorous treatment of how the model predicts transactional performance based on sampling the address sets of dynamic critical regions from a parallel execution.

A. Syncchar approach

Assessing the likelihood that dynamic instances of critical regions will conflict is at the heart of the Syncchar approach. The performance scalability of a transactional memory program hinges on the number of critical sections that can execute concurrently.

Transactional memory systems generally rely on *conflict serializability* as their safety condition, implemented as an abstract form of two-phase locking [30]. We call the set of addresses read during an instance of a critical section the *read set*, the set of addresses written the *write set*, and the union of read and write set the *address set*. For critical sections A and B, A conflicts with B if: $A_W \cap (B_R \cup B_W) \neq \emptyset$. Informally, conflict serializability says that the write set of one critical region must be disjoint from the other's address set to guarantee safety. Conflict serializability is efficient to compute so it is used widely in transactional memory systems.

A group of critical region executions are *data independent* if each write set is disjoint from the others' address sets. If critical regions concurrently modify the same data, or have *data conflicts*[1], the transactional memory system will serialize execution of the critical sections. In such cases, transactional memory can perform much worse than conservative locking due to the overhead required to detect and to resolve conflicts.

Conflict density is a measure of how long the serial schedule resulting from a conflict is likely to be. Assume a conflict among N threads. In the best case, a single thread might write a datum read by $N-1$ other threads. This is a low density conflict that produces a short serialized execution schedule ($N-1$ readers commit, and then the writer). In the worst case, each thread can write a datum written by each of the other $N-1$ threads, yielding a high density conflict that necessitates a completely sequential schedule (the threads must run serially, one after the next).

Syncchar estimates the data independence and conflict density of critical regions by sampling their address sets. Syncchar samples address sets of critical regions that could potentially execute concurrently using transactional memory and determines which of them can conflict. This process, described in detail below, is effectively an implementation of the safety property of the TM system. Sampling incorporates the dynamic behavior of the application and its potential data conflicts.

A similar model was developed by von Praun et al. [36], which calculates the *dependence density* of an application and broadly categorizes tasks within a program as low, medium, and high concurrency. Dependence density is essentially an aggregation of data independence and conflict density; our experience shows that this decomposition provides valuable insight into performance debugging. Syncchar closes the loop by making more concrete performance predictions (Section IV), and providing insights into performance tuning applications (Section VI). Section VIII provides additional comparison between these papers.

B. Data independence

Data independence provides a high-level profile of the likelihood that a set of critical regions will conflict. The data independence of a lock, I_n, is formally defined as the mean number of threads that will not conflict when n threads are concurrently executing critical sections protected by the same lock. Data independence is a function of the number of threads that can execute concurrently, and not a simple mean. Intuitively, one expects the probability of conflict when only a single thread is scheduled to be zero (data independence of 1), and the likelihood of conflict to increase as the number of threads grows (unless all threads access completely disjoint data).

Thus, when Syncchar calculates data independence for n CPUs, it begins with a sample of critical regions from up to n different threads. Syncchar then compares each address set in the sample to all other address sets in the sample, determining which threads are involved in a conflict C_n and which are not. It keeps a running mean of the number of data independent threads:

$$I_n = n - |C_n|$$

C. Conflict density

The key metric in the Syncchar model is the *density* of the conflicts. A set of critical regions with high conflict density will be prone to conflicts that involve many threads and form a long serial schedule, whereas low density conflicts can be resolved more quickly.

If one thinks of conflicts as forming a graph, with critical regions as nodes and conflicts as edges, the density of the conflict is the number of edges per node. Intuitively, if every transaction conflicts with every other, one expects them to execute sequentially, which means lower performance for transactional memory. On the other hand, if for example, one thread writes a memory location read by 31 other threads, all readers should complete once the writer completes, resulting in a performance gain that is commonly experienced in practice. This scenario would have a star topology if represented as a graph; removing the most connected node causes all others

[1] We selected the term data conflicts over data dependence to avoid confusion with other meanings.

to become disconnected (and able to proceed concurrently). Hence, modeling this phenomenon is crucial to the accuracy of predicting performance.

Syncchar calculates the conflict density of a sample of address sets as follows. For each address set in C_n, we measure the number of conflicts with other address sets in C_n and divide it by $(|C_n| - 1)$. The sum of each of these terms is the density (D_n). Formally, this is expressed as

$$D_n = \sum_{x \in C_n} \frac{\sum_{y \in \{C_n - x\}} \mathit{conflicts}(x, y)}{|C_n - x|}$$

where $\mathit{conflicts}(x, y)$ evaluates to 1 if address sets x and y can conflict and 0 otherwise. In the case where all conflicting address sets conflict with all others, the density equals the size of the conflicts $(D_n == |C_n|)$. In the star topology case, D_n should equal 2.

D. Predicting transactional performance

The intuition behind our approach to predicting transactional performance is that performance will be limited by the serial schedule transactions must form to ensure correctness. In the Syncchar model, this is the conflict density. In the common case, the speedup of n threads concurrently executing a set of transactions is expressed as $\frac{n}{max(D_n,1)}$.

When used with a lock-based parallel program, Syncchar tracks both the acquisition time (acq) for a lock and the length of a critical region after the lock is acquired $(held)$, as well as unrelated cycles $other$. According to Amdahl's law, the overall speedup obtained from concurrent execution of a critical section is constrained by the amount of time spent in that section. Thus, the speedup term above is applied only to the cycles spent acquiring or holding a lock, leaving the unrelated cycles $(other)$ unaffected by transaction behavior. Formally, then, our calculation becomes:

$$\mathit{Execution\ Time} = \frac{(acq + held) * max(D_n, 1)}{n} + other$$

E. Limitations

Like any model, Syncchar makes certain simplifying assumptions that balance complexity against accuracy. Bobba et al. [4] provide an analysis of performance pathologies for various hardware transactional memory designs. In our experience working with an eager/eager HTM implementation, the most common deviation from the Syncchar model is that HTM can perform substantially worse than locking under high contention. This can be attributed to a number of factors, including restart costs, suboptimal back-off policy, and bus contention. Ideally, when a transaction is restarted to resolve a conflict, it will retry execution as soon as the winner of the conflict commits. In an HTM with a high restart cost or a high back-off, there may be a period of needless latency after the winner commits and before the losing transaction resumes, lowering transaction throughput. In the worst case, this can lead to a conflict with a new transaction. On the other end of the spectrum, a back-off policy that retries too quickly can increase contention for cache lines and the memory bus, degrading system performance. A goal of our model is generality, so we have avoided including implementation-specific parameters in our model, as these contention pathologies can vary widely across HTM implementations.

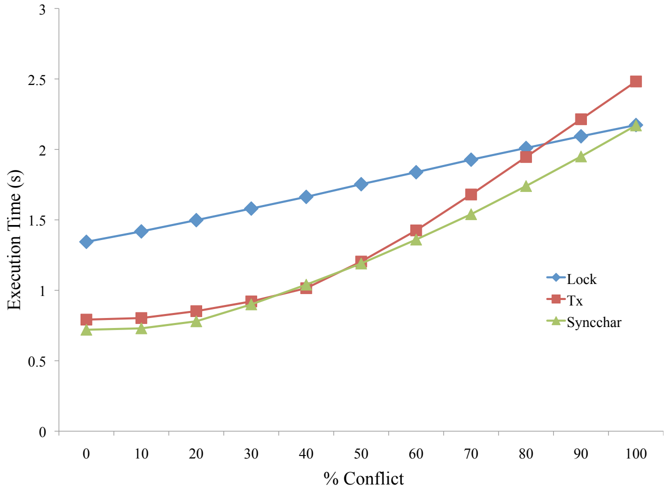

Fig. 1. Performance of locks vs. transactions for a microbenchmark over a range of probabilities of conflict. The Syncchar line illustrates how the model loses prediction precision, but correctly identifies the performance trend at high contention due to implementation-specific effects in the HTM.

Figure 1 shows the execution time of a simple microbenchmark at 8 CPUs, where critical regions write to a shared variable with a configurable probability, with the probability ranging from 0 to 100. The Tx line crosses the Lock line between a probability of 80 and 90. This leads to deviation in the accuracy of the Syncchar model beginning at about 60% probability of dense conflicts. Syncchar correctly identifies that transactions will give little to no performance improvement as contention increases, but it cannot predict how poorly a given HTM will perform.

The Syncchar model might also underestimate performance. An application that has a freqently executed critical region that is also highly data independent will have excellent TM performance, but might experience lock contention. Threads using locks mutually exclude even if a critical region is data independent. In such a case, a more accurate prediction may be realized by dropping the acq cycles and only applying the speedup term to the $held$ cycles. Rather than apply heuristic weights to the acq cycles, we leave the issue for future work and use a more approximate prediction.

IV. Model validation

In this section we validate the Syncchar model by implementing it as a module for Virtutech Simics [17] and comparing the predicted performance with transactions to the measured performance on an HTM model. We validate the Syncchar model against seven benchmarks from STAMP version 0.9.9, listed with input parameters in Table I. All experiments in this and the following sections are performed modeling 8, 16, and 32 1GHz, x86 processors. As Simics supports only a fixed IPC,

bayes	Learns a randomly generated bayesian network. `-v32 -r4096 -n5 -p30 -s1 -i2 -e4`
genome	Reconstructs a larger gene sequence from segments of the gene. `-g16384 -s48 -n4194304`
intruder	Emulates a network intrusion detection system; transactionalizes the packet capture and reassembly phases. `-a10 -l32 -n65216 -s1`
kmeans	Implements the K-means clustering technique. `-m40 -n40 -t0.0009 -i random-n65536-d32-c16.txt`
ssca2	Scalable Synthetic Compact Applications 2, kernel 1. Constructs a graph data structure using adjacency and auxiliary arrays. `-s17 -i1.0 -u1.0 -l3 -p3`
vacation	Implements a travel reservation system. Transactions protect updates to a local database implemented with a red-black tree. `-n4 -q60 -r1048576 -u90 -t1048576`
yada	Ruppert's algorithm for Delaunay mesh refinement, similar to Kulkarni et al. [13]. `-a20 -i ttimeu10000.2`

TABLE I
DESCRIPTION OF THE BENCHMARKS USED IN SECTIONS IV AND V, FROM THE STAMP BENCHMARK SUITE [20]. WE INCLUDE OUR CUSTOM INPUT PARAMETERS FOR REFERENCE, AS WE COMPARE WITH THE DEFAULT PARAMETERS IN SECTION V.

Workload		Tx	Syncchar	% Err	DI	CD
bayes	8 CPU	-	-	-	-	-
	16 CPU	.29	.29	0	0.00	3.10
	32 CPU	.20	.15	25	0.21	3.39
genome	8 CPU	1.35	1.11	19	5.89	0.01
	16 CPU	.79	.94	16	11.44	0.04
	32 CPU	.50	.84	40	28.40	0.13
intruder	8 CPU	1.06	1.52	43	2.20	2.72
	16 CPU	1.33	1.63	22	2.60	4.51
	32 CPU	1.67	1.72	3	2.68	8.45
kmeans	8 CPU	7.72	8.69	13	5.28	1.98
	16 CPU	4.40	5.98	37	8.43	2.92
	32 CPU	3.05	3.06	1	11.93	4.19
ssca2	8 CPU	.64	.64	13	7.98	0.00
	16 CPU	.40	.36	13	15.94	0.00
	32 CPU	.27	.21	11	31.94	0.01
vacation	8 CPU	1.39	1.16	17	5.03	.90
	16 CPU	.72	.53	26	6.92	1.74
	32 CPU	.39	.21	46	7.70	3.18
yada	8 CPU	.38	.39	26	4.76	2.51
	16 CPU	.21	.37	19	7.94	4.71
	32 CPU	.15	.37	153	15.44	9.43

TABLE II
THE EXECUTION TIME IN SECONDS FOR THE STAMP BENCHMARKS IN SECONDS (LABELED TX), THE PROJECTED EXECUTION TIME IN SECONDS, LABELED SYNCCHAR, AND ACCURACY (% ERROR). DI IS THE DATA INDEPENDENCE VALUE FOR THE BENCHMARK AND CD IS THE CONFLICT DENSITY. 8 CPU BAYES DATA WAS NOT AVAILABLE AT THE TIME OF SUBMISSION.

the simulations used an IPC of 1, which is a reasonable choice for a moderate superscalar implementation. Each processor has a 32 KB, 4-way set associative private L1 cache with 64 byte lines and an access time of one cycle. L2 caches are also private with 64 byte lines. Each L2 cache is 4MB, 8-way set associative. L2 cache accesses cost 16 cycles, and are kept coherent using a transaction-aware MESI snooping protocol. Our coherence model is implemented by the Simics `txcache` module [32] . Main memory is a single, shared 1 GB, with an access time of 200 cycles. Each CPU has two 4-way set associative, 64 entry TLB's–one for instructions and one for data.

All lock-based experiments run on Linux version 2.6.16.1. We used the MetaTM hardware transactional memory model and TxLinux version 2.6.16.1 for the transactional memory experiments [29, 31]. Main memory access latency is pseudo-randomly perturbed by 0-4 ns to get accurate performance measurements for multi-threaded programs. Multi-threaded programs are sensitive to scheduling, so several randomly perturbed runs give a more accurate picture of performance, as described by Alameldeen and Wood [1]. Measurements presented are a mean of 4 simulated executions.

A. Predicting speedup for STAMP

We applied the Syncchar model to programs from the STAMP transactional memory benchmark suite [20]. As we will discuss further in Section V-C, we do not use the labyrinth benchmark in any experiments due to a memory leak that we could not resolve.

Table II shows the predicted and actual execution times of the parallel phases of these benchmarks. The measured exeuction times are the mean of four runs, and the standard deviations are all within 5% of the mean, except for bayes, which has a variable number of transactions and standard deviations as high as 23% of execution time. The geometric mean error in our predictions across all benchmarks is 25%. Syncchar tracks the scalability trends very closely, both for high-contention workloads that have poor scalability, like intruder, and for low-contention workloads that have good scalability, like kmeans. Syncchar's precision decreases as the benchmarks become very short, particularly for benchmarks that run for .3 seconds or less. In the worst cases, Syncchar predicts the scaling trends offset by a factor of 40-153%.

Data independence and conflict density prove to be interesting metrics, with widely varying values across STAMP applications. Ssca2 is highly data independent, while the high-contention intruder is not. Bayes has very light conflict density while yada's conflicts are quite dense, with an average of nine densely conflicting threads per transactional conflict.

Figure 2 shows the projected speedup of a representative sample of benchmarks, broken down into the portion attributable to data independence and the portion attributable to low conflict density. The percent of the speedup attributable to data independence is the same as the fraction of critical regions that do not conflict, and the rest is attributable to low conflict density. The projection in Figure 2 for ssca2, at one end of the spectrum, is entirely due to data independence. Like many transactional benchmarks, ssca2's critical regions rarely conflict, yielding good scalability and a substantial speedup over locking. On the other extreme, nearly every critical section in bayes is likely to conflict at least once, yet the measured speedup is still substantial because relatively few threads are involved in each conflict. Low conflict density is the key to bayes' scalability. Similarly, the kmeans benchmark has high

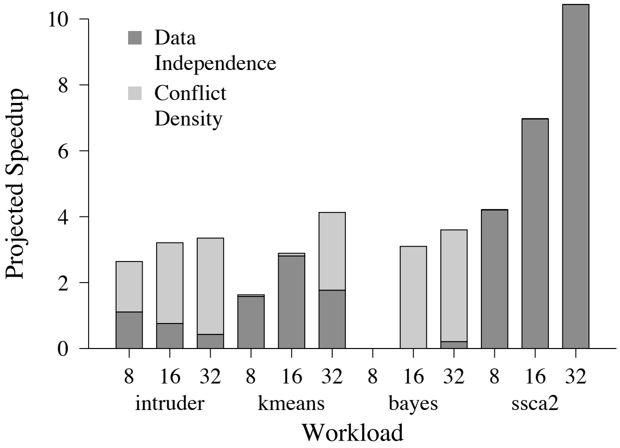

Fig. 2. The projected speedups of selected benchmarks using transactions versus locking, decomposed into the portion attributable to data independence and conflict density.

data independence at moderate CPU counts, but increasingly relies on non-dense conflicts for its performance improvement at higher CPU counts. The kmeans benchmark shows a reasonable mix of both data independence and non-dense conflicts, which we expect to be the case with realistic workloads.

These experiments show that the Syncchar model strikes a good balance between accuracy and complexity. While there are outliers in the model's predictions, most are off by 26% or less, and all capture the scaling trends of the workload. These performance estimates can help the programmer understand a transactional program's performance.

B. Scheduling model and sampling

Syncchar measures data independence and conflict density by checking for conflicts between critical regions that are likely to be scheduled concurrently. This Subsection describes how Syncchar samples critical regions to detect conflicts. The Syncchar approach respects the schedule of the original execution, but generalizes it significantly to simplify the model.

Syncchar records the address sets of critical regions within a sliding time window. After Syncchar has seen a certain number of critical regions from the same lock, it samples the desired number of critical regions and measures data independence and conflict density as if those critical regions were scheduled together. In our experiments, the sliding window consists of 512 critical regions, a value selected based on experimentation with different sizes. In selecting address sets to compare, Syncchar is careful to select only one address set per process. After calculating data independence and conflict density, the recorded address sets are discarded.

Limiting the number of address sets that Syncchar can consider concurrently captures many likely execution schedules with a simple concurrency model and implementation. For instance, a newly forked thread cannot execute concurrently with critical region executions that occurred before its creation. Syncchar's sliding window reduces comparisons between critical regions that cannot be scheduled concurrently due to some other synchronization mechanism, such as a join after a fork, without having an exhaustive scheduling model.

Data Independence						
Workload		**10%**	**%Err**	**25%**	**%Err**	**100%**
intruder	8	2.71	23	2.47	12	2.20
	16	2.95	13	2.73	5	2.60
	32	2.39	11	2.48	4	2.68
kmeans	8	5.37	2	5.34	1	5.28
	16	8.51	1	8.47	0	8.43
	32	11.95	0	11.95	0	11.93
ssca2	8	7.98	0	7.98	0	7.98
	16	15.92	0	15.94	0	15.94
	32	31.96	0	31.96	0	31.94
yada	8	4.08	14	4.52	5	4.76
	16	8.33	5	7.52	5	7.94
	32	16.22	5	15.42	0	15.44
Conflict Density						
Workload		**10%**	**%Err**	**25%**	**%Err**	**100%**
intruder	8	2.71	0	2.73	0	2.72
	16	4.62	2	4.62	2	4.51
	32	8.66	2	8.67	2	8.45
kmeans	8	1.91	4	1.93	3	1.98
	16	2.99	2	2.94	1	2.92
	32	4.23	1	4.22	1	4.19
ssca2	8	0.00	0	0.00	0	0.00
	16	0.00	0	0.00	0	0.00
	32	0.01	0	0.01	0	0.01
yada	8	4.08	63	2.71	8	2.51
	16	4.72	0	5.09	8	4.71
	32	8.69	8	9.08	4	9.43

TABLE III
SENSITIVITY OF DATA INDEPENDENCE AND CONFLICT DENSITY TO DIFFERENT SAMPLING RATES (10%, 25% AND 100%).

Syncchar samples address sets to reduce the number of address sets recorded, improving the efficiency of the tool. In our previous experiments, we used a 100% sampling rate for the highest fidelity data. Table III shows the data independence and conflict density measurements at a sampling rate of 10%, 25%, and 100%. A representative sample of benchmarks was selected; others were omitted for space. At 25%, measurements are within 8% of the exhaustive measurement. While there are some outliers at 10% sampling rate, the result is within 15% of exhaustive measurement for most benchmarks. Syncchar's insensitivity to sampling indicates that the algorithms are practical for resource-constrained systems.

V. System Influence on Transaction Performance

In situations where a transactional application ought to perform well and does not, the problem may be poor interaction with the full system and the transactional memory implementation. This section analyzes situations where the system effects transactional performance of user-level programs, using 7 of the STAMP benchmark suite as a case study.

A. Input size

Table IV compares the speedup for STAMP benchmarks for different input sizes. Because detailed HTM simulation infrastructures tend to be slow, the STAMP benchmarks are

Workload		8 CPU	16 CPU	32 CPU
bayes	sim	3.9	3.9	3.5
	big	3.3	3.0	4.3
genome	sim	1.3	1.3	1.1
	big	7.3	14.5	26.7
intruder	sim	1.4	1.4	1.2
	big	2.4	1.9	1.5
kmeans	sim	5.1	5.0	5.5
	big	7.5	13.0	18.9
ssca2	sim	0.9	1.0	1.2
	big	2.3	3.8	5.5
vacation	sim	1.1	1.4	1.4
	big	7.0	12.1	18.9
yada	sim	2.8	3.5	3.9
	big	3.0	4.8	7.0

TABLE IV
THE SPEEDUP FOR THE PARALLEL PHASE OF A SELECTION OF STAMP BENCHMARKS RELATIVE TO 1 CPU WITH THE STANDARD SIMULATOR INPUTS AND BIG INPUTS.

distributed with simulator inputs, which are much smaller. Simulator inputs are much smaller than the kinds of inputs for which the programs were designed.

As the table makes clear, the simulator inputs are too small to show accurate scaling trends. For example, performance for genome's simulator inputs does not scale at all for 32 CPUs while our larger inputs scale by 26×. Ahmdahl's law [2] explains the problem. Even though the STAMP benchmarks only measure performance during the parallel parts of the code, the parallel section still includes serial setup. Some of the setup is implicit; for example, the OS must schedule all of the threads before they all execute. Because the parallel section is so short, the performance scaling of the simulator inputs does not reflect scaling on more realistic inputs. Also, the overhead of synchronizing at a barrier towards the end of the benchmark grows with higher processor counts.

While benchmark size and startup effects are a known problem for simulation [25], they continue to impede accurate measurement. We note that researchers can compensate for the brevity of the simulator inputs and still accurately calculate the gains in performance. Authors generally omit the details of how speedup curves are generated, so we could not verify that any particular methodology for measuring simulator inputs matches the scaling trends on non-simulator inputs. Unless otherwise noted, all STAMP data in this paper uses the "Big" inputs, listed in Table I.

Table VI shows detailed breakdowns of the simulator inputs and our resized inputs. The most important result from the table is that common transactional metrics, like restart rate and average backoff, generally are a poor indicator of execution time. For instance, execution time and restart rate are correlated as CPU counts increase for intruder and bayes with the simulator inputs, but are inversely correlated for yada and vacation. This is appreciated by the transactional memory community, but the lack of good performance indicators is a serious problem for developers who must performance-tune transactional programs. Simple event counters or runtime statistics provide insufficient data on which to base performance tuning decisions.

Workload		TLB in TX	TX that Miss		Miss/TX
bayes	8	1,005,978	1,301	79.48%	612
	16	1,350,564	979	69.30%	908
	32	800,294	752	56.59%	594
genome	8	44,300,488	1,170,001	99.15%	37
	16	45,963,325	1,169,923	99.15%	38
	32	52,167,445	1,169,967	99.15%	44
intruder	8	9,151,084	785,974	49.53%	5
	16	10,800,831	936,367	58.96%	6
	32	12,089,317	854,040	53.80%	8
kmeans	8	94,684	74,317	1.44%	0
	16	131,538	95,272	1.81%	0
	32	158,377	111,672	1.92%	0
ssca2	8	4,437,694	1,416,004	99.81%	3
	16	4,435,489	1,415,954	99.80%	3
	32	4,438,800	1,416,070	99.81%	3
vacation	8	55,804,649	1,048,576	100.00%	53
	16	59,479,544	1,048,576	100.00%	56
	32	70,203,295	1,048,576	100.00%	66
yada	8	412,759	39,966	62.72%	7
	16	446,156	42,194	65.00%	7
	32	493,853	43,563	66.16%	8

TABLE V
TOTAL TLB MISSES IN TRANSACTIONS FOR OUR BENCHMARKS AT 8, 16, AND 32 CPUS, AND THE NUMBER AND PERCENT OF TRANSACTIONS THAT INCUR AT LEAST ONE TLB MISS OVER THE COURSE OF THEIR EXECUTION.

Simply reducing restarts or backoff cycles might not affect performance. The actual dynamics are much more complicated. For instance, it might be more important to reduce the latency of the critical path of the computation than to reduce the restarts that occur off the critical path. The complexity of tuning transactional performance motivated us to develop the Syncchar model.

The table also shows that the idle time for some of the benchmarks is high, and noticeably higher for the simulator inputs. Load imbalance significantly affects the simulator inputs, causing unrealistic amounts of idle time. For instance, kmeans with big inputs has only 1% idle time at 32 processors, but it has 52% idle time for the simulated inputs. While the STAMP benchmarks are designed to minimize system time, several of them still spend up to 10% of their execution time in the operating system, mostly in memory management and page fault handling routines.

B. TLB Misses

We measure TLB miss rates within transactions in Table V. TLB misses are very common within transactions. In fact, for almost every run of genome, ssca2 and vacation on big inputs, over 88% of transactions take at least one TLB miss. The TLB miss rates on the simulator inputs are generally lower because these inputs touch so little data that they do not establish the steady-state TLB miss rate of the application. Only for kmeans does the simulator TLB miss rate match the big inputs. It has only 3 brief critical regions.

Knowing that TLB misses will be common in transactional programs is important for hardware designers. The Sun Rock processor [6] is reported to support HTM primitives, but it will not tolerate TLB misses during a transaction. Kmeans is a good fit for designs like the Sun Rock because it has a low TLB

Workload		Big						Sim					
		total	U/S/I			Restart %	Back	total	U/S/I			Restart %	Back
bayes	8cpu	.288	55	9	34	7.85	77,310	.016	46	3	49	3.66	877
	16cpu	.289	30	4	65	12.61	96,270	.016	25	4	69	6.16	4,344
	32cpu	.223	14	2	83	15.85	62,373	.018	14	2	83	9.90	4,093
genome	8cpu	1.387	91	1	8	.30	0	.013	25	12	61	.72	13
	16cpu	.722	89	1	10	.00	1	.013	15	9	75	1.09	10
	32cpu	.387	88	1	12	.00	0	.016	8	10	81	1.70	13
intruder	8cpu	1.063	98	1	0	42.51	134	.016	86	0	12	53.88	922
	16cpu	1.339	99	0	0	47.96	1,038	.017	76	0	23	63.68	2,500
	32cpu	1.668	99	0	0	46.73	1,985	.019	78	0	21	70.46	8,647
kmeans	8cpu	7.716	99	0	0	2.01	5	.012	90	0	9	6.04	35
	16cpu	4.404	99	0	0	10.05	47	.012	67	1	30	11.83	1,219
	32cpu	3.046	98	0	1	20.70	245	.011	44	3	52	16.24	1,706
ssca2	8cpu	.638	83	7	9	.00	0	.026	60	6	33	.06	0
	16cpu	.390	77	5	17	.00	0	.024	35	7	56	.09	0
	32cpu	.266	67	4	28	.01	0	.021	28	6	64	.20	8
vacation	8cpu	1.637	98	1	1	20.62	28	.012	52	0	47	14.24	57
	16cpu	.786	98	1	1	37.22	67	.013	45	1	52	22.61	741
	32cpu	.506	98	1	2	57.24	170	.009	38	1	60	36.62	1,156
yada	8cpu	.347	88	10	1	14.00	3,859	.029	87	2	9	15.87	3,340
	16cpu	.219	90	7	1	18.97	4,645	.023	82	1	15	17.81	4,408
	32cpu	.152	91	5	2	27.38	5,881	.021	79	0	19	20.17	7,523

TABLE VI
THE EXECUTION TIME (TOTAL) FOR SELECT STAMP BENCHMARKS USING BIG AND SIMULATED INPUTS, BROKEN DOWN INTO USER, SYSTEM, AND IDLE TIME (U/S/I), RESTART RATE (RESTART %) , AND MEAN BACKOFF CYCLES (BACK).

miss rate, none of its three critical regions make function calls, and only one critical region contains any control flow at all (a loop). It is unknown if Azul Systems' HTM can tolerate a TLB miss [5]. Architectures with a software reloaded TLB would likely require the TLB load sequence to run outside the context of the current transaction. Avoiding an abort of the current transaction for a TLB miss on these architectures might be challenging.

C. Memory allocation

Performing system calls, including memory allocation, within a transaction can violate isolation. While most memory allocators are mostly implemented at user level, they sometimes need to make system calls to get more memory from the OS (either `mmap` or `sbrk`), and they might make other system calls. This problem is well known [37, 39], and there are several ways to allow system calls within transactions, including open nesting [21, 22] and transactional pause [37, 39].

The STAMP benchmarks allocate memory and are not written with any special hooks to deal with allocation. A transaction can call the memory allocator, which calls the OS for more memory. Such a transaction can restart many times, leaking memory if there are no hooks for compensating actions on transaction restart. The labyrinth benchmark is missing from this paper because it pathologically leaks and cannot complete, and we have experienced pathological memory leaks in genome.

Surprisingly, memory allocation on a complete Linux environment can actually cause deadlock. Recent versions of glibc use a blocking lock to protect the memory allocator's data structures. This lock is implemented using the `futex()` system call [7], which can block the current thread until the lock is released. When a process releases a futex that has other threads waiting on it, it must trap into the kernel to notify the waiters. In the context of a hardware transaction, a transactional process can acquire the lock, a second can block on the lock, and then the transaction holding the lock restarts. The hardware releases the lock variable, but does not notify the kernel of the change and blocked threads can sleep indefinitely, effectively deadlocking the application. Many standard system libraries are written to be threadsafe and will still use locks, condition variables, and other synchronization mechanisms in a transactional memory application, potentially causing adverse effects.

We discovered this deadlock behavior in the vacation benchmark, which often allocates memory within transactions to insert a new node into its red-black tree. We work around the problem by conservatively preallocating all the memory a transaction might need before beginning a transaction. This *ad hoc* solution will not apply to general-purpose programming, especially in the presence of composing transactions with standard library routines. Calling standard libraries and system calls, such as `futex`, within a transaction will likely require a combination of new standard libraries [35], OS support [28], or a TM implementation-specific solution like making a single transaction non-speculative [3, 8, 12].

D. Compiler effects

A final challenge for optimizing transactional code is compiler optimizations for deeply pipelined machines. Figure 3 illustrates a simple conditional statement the programmer would expect to reduce conflicting accesses to a shared variable. Yet, the code generated by gcc always reads the value from memory and always writes it back. It only uses the condition to determine whether to update the register before writing it back. The compiler is trying to avoid branching around the

```
if(a < threshold)
   shared_variable = new_value;
```

```
;; edx holds a, 0xc03e6008 holds threshold
;; eax holds new_value, 0xc03e600c holds shared_variable

cmp    0xc03e6008,%edx  ;; compare a and threshold
cmovge 0xc03e600c,%eax  ;; conditionally load old value
mov    %eax,0xc03e600c  ;; unconditionally store to
                        ;; shared_variable
```

Fig. 3. A simple code sequence and the annotated x86 assembly produced by gcc.

bonnie++	Simulates file system bottleneck activity on Squid and INN servers stressing create/stat/unlink. 32 instances of: `bonnie++ -d /var/local -n 1`
MAB	File system benchmark simulating a software development workload[23]. Runs one instance per processor of the Modified Andrew Benchmark, without the compile phase.

TABLE VII
PARALLEL APPLICATIONS USED TO EXERCISE THE CONCURRENCY IN LINUX AND TXLINUX.

Lock	**j_state_lock**	**j_list_lock**
0-19%	420	242
20-39%	2	0
40-59%	0	0
60-79%	0	0
80-99%	0	0
100%	0	0
Avg. Workset Size	13	33
Total Conflicts Sampled	111,118	35,600
DI	1.03	1.15
CD	14.37	7.26
Asym	88%	92%

TABLE VIII
DISTRIBUTION OF CONFLICTING ADDRESSES FOR THE JOURNAL LOCKS, AS WELL AS OTHER KEY SYNCCHAR METRICS. THE RANGE OF PERCENTAGES SHOWS THE NUMBER OF WORKSET ADDRESSES INVOLVED IN THAT PERCENTAGE OF CONFLICTS. SAMPLES ARE TAKEN FROM BONNIE++ AT 16 CPUS. THE AVERAGE WORKSET SIZE IS IN WORDS. DI IS DATA INDEPENDENCE, CD IS CONFLICT DENSITY, AND ASYM IS THE PROBABILITY OF BEING INVOLVED IN AN ASYMMETRIC CONFLICT.

load and store, which makes sense on a superscalar platform. On a hardware transactional memory system, however, the performance lost to a coherence conflict is much larger than that lost to a mispredicted branch.

For a critical section such as the one above, the Syncchar tool identifies it as having no data independence and flags the shared variable as a contention hot-spot. Counterintuitive results such as these hint to the programmer that the code generated by the compiler should be inspected.

VI. SYNCCHAR AS A TUNING TOOL

In addition to predicting the performance of a transactional memory system, the Syncchar model can provide clues as to which critical sections are likely to have performance problems. This section demonstrates the utility of Syncchar for tuning transactional performance, using the (Tx)Linux 2.6.16 kernel as a case study.

A. *Tuning features*

Syncchar provides a number of useful metrics for the transactional memory developer. Data independence and conflict density give the developer an indication of how much parallelism they can expect from the code. Syncchar reports the distribution of conflicting addresses in a critical region, allowing the developer to identify contention hot-spots.

Syncchar also provides statistics on the workset size as illustrated in Table VIII, which can help developers identify critical regions that are likely to overflow the cache in a coherence-based HTM. Finally Syncchar provides traditional profiling metrics, such as the most frequently executed critical region and the most contended critical regions.

1) Asymmetric Conflicts: Conflicts between transactional and non-transactional threads are known as *asymmetric conflicts* [29] and can introduce performance pathologies in transactional applications. In resolving asymmetric conflicts, the transaction must always restart because a non-transactional thread cannot be rolled back. In many TM implementations, if non-transactional accesses occur faster than transactions can commit, transactions are starved and the application can hang.

Performance problems caused by asymmetric conflicts are difficult to diagnose and to resolve. Simple profiling can indicate in which transaction the program is spending too much time, but it cannot find the unrelated code that is starving the transaction with asymmetric conflicts. Even TM researchers with good intuition about transactional performance tend to scrutinize writes, yet a variable that is only read in a transaction can cause problems if it is frequently written outside of a transaction.

Syncchar helps developers by indicating critical regions and addresses that are prone to asymmetric conflicts. This is implemented by tracking reads and writes outside of critical regions and checking for conflicts with critical regions. For each critical region, Syncchar reports the percentage of address sets that conflict with the non-transactional address set, estimating the probability of an asymmetric conflict. In the next Subsection we use Syncchar to diagnose and to correct such a pathology in TxLinux.

B. *TxLinux case study*

Rossbach et al. [31] report comparable performance for locking and transactions for the TxLinux kernel. Transactions do not help performance because the locks in Linux 2.6 are already fine-grained, highly tuned, and because several benchmarks spend relatively little time in the kernel. The primary exception was the bonnie++ filesystem benchmark, which shows severely pathological behavior under transactions. We reproduced this pathology and used simple program counter profiling to indicate that most of the time was being spent in the `ext3` journaling code, protected by the `journal_t.j_state` and `journal_t.j_list` locks.

Workload		TxLinux	TxLinux-Opt	Speedup
bonnie++	8	HANG	1.55	-
	16	HANG	.98	-
	32	HANG	.54	-
MAB	8	.68	.48	1.4
	16	.24	.13	1.8
	32	.07	.07	1.0

TABLE IX
System time in seconds for TxLinux workloads under unmodified TxLinux and our optimized version of TxLinux (TxLinux-Opt), at 8, 16, and 32 CPUs (lower is better). Speedup is TxLinux-Opt relative to the TxLinux baseline (higher is better).

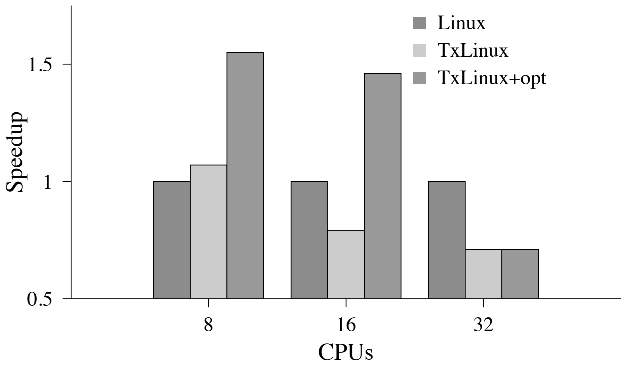

Fig. 4. Speedup of unmodified TxLinux and our optimized version of TxLinux (TxLinux-Opt) over unmodified Linux for the MAB benchmark at 8, 16, and 32 CPUs (larger is better).

We then used Syncchar to provide additional insight into the problem by sampling these critical regions by running bonnie++ on unmodified Linux. Table VIII shows the distribution of conflicting addresses for these journal locks. These critical regions tend to be short—13 and 33 words in the working set, respectively. They have relatively high conflict density, indicating that there is temporal locality in the data accessed within these critical regions, but no one "hotspot" address. This matched our initial experience in auditing the journal code, as we found no immediately obvious datum causing the high transaction restarts.

The red flag Syncchar raised for these critical regions was that they had a 92% probability of an asymmetric conflict. Upon further review, we found an assertion in the code that a bit in the `buffer_head.b_state` field is set. This bit is used as a spinlock and always set before entering the critical regions that were converted to transactions. Moreover, other bit fields in the same byte are used by non-transactional tasks such as journal writeback. Interestingly, these bits are not modified inside a transaction, so the intuitive approach of looking for conflicting writes inside a transaction is not enough. Developers need tools to help find these sorts of bugs in a large code base.

We addressed the problem by converting the bit spinlock to a cooperative transactional spinlock (cxspinlock) [31]. A cxspinlock uses transactions optimistically, but can fall back on locking to mutually exclude I/O. Cxspinlock acquisition is arbitrated by the HTM contention manager, so locks and transactions can fairly contend for a critical region. By using cxspinlocks and separating this bit into its own word (or cache line, depending on the HTM conflict granularity), we were able to eliminate the asymmetric conflicts that were starving the transaction.

C. Experimental results

To evaluate these optimizations, measurements of 8, 16, and 32 CPU systems were taken as described in Section IV. Because TxLinux simply idles when left undisturbed, we need a set of parallel applications to exercise the concurrency within the TxLinux subsystems, such as the file system and memory allocator. We used the full suite of 6 benchmarks from the TxLinux paper [31] and verified that our optimizations did not introduce any performance regressions. For the sake of space, we only report the two benchmarks with improved performance, listed in Table VII. All other benchmarks performed identically to unmodified TxLinux.

Table IX presents the time spent in the kernel for each of these benchmarks on both unmodified TxLinux and our optimized version, respectively. Kernel time is presented because our tuning efforts are for the kernel itself and cannot be expected to improve the non-kernel portions of the execution time. Further, this approach eliminates noise introduced by load imbalance in the workloads.

We reproduced the same pathological behavior on the bonnie benchmarks under TxLinux. After over a month of simulating bonnie on TxLinux (compared with 1–3 days on average for bonnie on unmodified Linux), the benchmarks had not terminated, so we list the baseline TxLinux performance as hung.

Our optimization allows bonnie to complete, and also improves the performance of MAB. For other workloads, including MAB at 32 CPUs, the journal locks did not affect execution time substantially, and the performance remains flat. At this CPU count the performance bottleneck shifts from the journal spinlocks to other critical regions in the kernel. This demonstrates that our optimizations do not introduce regressions or other performance anomalies for other workloads.

The bit spinlock optimization also improves the performance of TxLinux relative to unmodified Linux at 8 and 16 processors, as shown in Figure 4. Tuning TxLinux 2.6 with programmer intuition and simple profiling has yielded marginal ($< 10\%$) improvement over Linux, but Syncchar led us to the fix for the bonnie++ pathology and sped up MAB by over 50%.

This case study indicates that tuning the performance of transactional memory applications can be tricky and nonintuitive, as the cause for a given critical region dominating execution time can be in an entirely different part of the application. Programmers facing this challenge will benefit from tools that help them understand their system with quantitative rigor.

VII. Implementation Details

This section describes key implementation details and features of the Syncchar tool, which can be used to performance tune applications as complex as the TxLinux kernel [31] (Section VI).

The Syncchar prototype is a module for the Simics full-system, execution-driven simulator [17] that generates output that is post-processed. By implementing Syncchar in the machine simulator, it is able to measure user-level code and the operating system. Nearly all instrumentation is performed using simulator breakpoints, which are performance transparent and minimally affect the execution behavior of the program being tested.

The requirements of the Syncchar model are modest enough that it could be implemented for user-level applications using a binary instrumentation tool such as Pin [16], or in a virtual machine monitor for kernel instrumentation on a live machine.

When tracking the address set of a critical region in lock-based code, Syncchar ignores reads and writes to lock variables, as locks would be removed in a transactional version of the application. Similarly, because stack addresses are generally private, Syncchar filters stack addresses to avoid conflicts due to reuse of the same stack address in different activation frames.

Syncchar needs to know about dynamically allocated lock variables. If an object containing a lock is freed and reallocated, it is treated as a new lock. In our current implementation, this information is communicated to Syncchar through annotations to the lock initialization routines.

Many locking implementations, including spin locks, do not ensure fairness and are more susceptible to load imbalance in parallel programs than transactional memory, which can have sophisticated contention management policies [33]. Syncchar accounts for this effect in its predictions by discarding idle cycles at the end of a benchmark when there is gross load imbalance; in practice, one might use predictions with and without these cycles to establish a range for the performance prediction.

Finally, some spinlocks in the Linux kernel protect against concurrent attempts to execute I/O operations, but do not actually conflict on non-I/O memory addresses. When run on the Linux kernel, Syncchar detects I/O operations, and marks those critical regions as performing I/O. This annotation ensures that these critical regions will not be incorrectly reported as non-conflicting.

VIII. Related work

There is very little published about performance tuning transactional programs. However, there is a large body of work about tools used to debug and to performance tune lock-based concurrent programs [9, 38]. These tools are similar to Syncchar in that they use a runtime system to track locking and data access patterns, but are different in their analysis goals. The collected information is used to identify possible data race conditions rather than data independence and conflict density.

In previous work [27], we introduce the definition of data independent critical sections and a tool for measuring data independence of critical regions. This paper substantially modifies the data independence metric to be more practical (both in measurement and utility), introduces the new metric of conflict density, and provides empirical evaluation of its application to prediction and tuning.

Von Praun et al.[36] provide a definition of *dependence density* that is similar to the aggregation of data independence and conflict density under Syncchar. Von Praun focuses on programs that apply similar operations to large data sets and provides a coarse classification of the workloads. They do not use the results of their model to predict speedups, and do not report any experience using the result of their model to tune the measured programs. Syncchar closes the loop by generating and validating performance predictions from its model. Further, decomposing data independence and conflict density provides more insight into tuning the behavior of real-world applications with irregular parallelism, such as the Linux kernel.

Heindl and Pokam [10] present a framework for analytical performance modeling of STM implementations. The focus of the work is on assessing the performance impact of certain STM design decisions, such as lazy versus eager conflict detection or version management. To this end, they make certain simplifying assumptions about the general behavior of workloads, such as all transactions are the same length and all data are equally likely to cause conflicts (i.e., no contention "hotspots"). In contrast, the Syncchar model is designed to help TM users understand if and how their particular workload deviates from the common-case assumptions of the TM designer. For instance, many workloads have data that create contention hotspots, which may be tuned to improve performance.

Several papers [24, 30, 34] discuss transaction conflict behavior in the context of improving HTM implementations, but to the best of our knowledge none of these papers provides a formal model of transactional memory performance and no prior work closes the loop by validating such a model against measured performance.

When memory performance was an important issue in the early 90's, programmers needed tools like *memspy* [18] to reason about performance problems related to memory behavior. Now that parallel architectures are unavoidable, programmers will need tools to assist in tuning the performance of optimistically synchronized systems. Lev and Moir discuss the necessity for and challenges of debugging tools for transactional memory systems [15], and Zyulkyarov et al. introduce debugging extensions for atomic blocks in C# and the Windows Debugger [40]. This paper is distinguished by addressing performance, whereas their work addresses correctness problems. Perfumo et al. introduce a Haskell runtime with transactional memory instrumentation support for performance profiling [26]. This work is complementary to the Syncchar model, which provides limits that are not tied to specific schedules.

IX. Conclusion

This paper introduces Syncchar, a novel method and tool for reasoning about the performance of transactional memory. We have validated Syncchar's performance predictions and demonstrated its usefulness in guiding performance tuning on the TxLinux kernel. This paper also presents a detailed characterization of how the whole system, including architecture, libraries, and compiler, can affect the performance of transactional applications, often in a way that befuddles performance tuning. We see Syncchar as one in an array of profiling and debugging tools that must be developed to help application developers leverage transactional memory more effectively.

X. Acknowledgements

We would like to thank the anonymous reviewers and Jeff Napper for helpful comments on earlier versions of this paper. We thank Owen Hofmann for help developing the Syncchar tool. We also thank Virtutech AB for the Simics academic license program. This research is supported by NSF CISE Research Infrastructure Grant EIA-0303609, NSF Career Award 0644205, and the DARPA computer science study group.

References

[1] A. Alameldeen and D. Wood. Variability in architectural simulations of multi-threaded workloads. In *HPCA*, 2003.

[2] G. Amdahl. Validity of the single processor approach to achieving large-scale computing capabilities. In *AFIPS*, 1967.

[3] C. Blundell, J. Devietti, E. Lewis, and M. M. K. Martin. Making the fast case common and the uncommon case simple in unbounded transactional memory. In *ISCA*, 2007.

[4] J. Bobba, K. E. Moore, H. Volos, L. Yen, M. D. Hill, M. M. Swift, and D. A. Wood. Performance pathologies in hardware transactional memory. *ISCA*, 2007.

[5] C. Click. http://blogs.azulsystems.com/cliff.

[6] D. Dice, Y. Lev, M. Moir, and D. Nussbaum. Early experiences with a commercial hardware transactional memory implementation. *ASPLOS*, 2009.

[7] H. Franke, R. Russel, and M. Kirkwood. Fuss, futexes and furwocks: Fast userlevel locking in Linux. In *Proceedings of the Ottawa Linux Symposium*, 2002.

[8] L. Hammond, V. Wong, M. Chen, B. D. Carlstrom, J. D. Davis, B. Hertzberg, M. K. Prabhu, H. Wijaya, C. Kozyrakis, and K. Olukotun. Transactional memory coherence and consistency. In *ISCA*, page 102. IEEE Computer Society, Jun 2004.

[9] J. Harrow. Runtime checking of multithreaded applications with visual threads. In *SPIN*, pages 331–342, 2000.

[10] A. Heindl and G. Pokam. An analytic framework for performance modeling of software transactional memory. *Comput. Netw.*, 53(8):1202–1214, 2009.

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

[12] O. Hofmann, C. Rossbach, and E. Witchel. Maximum benefit from a minimal HTM. In *ASPLOS*, 2009.

[13] M. Kulkarni, K. Pingali, B. Walter, G. Ramanarayanan, K. Bala, and L. P. Chew. Optimistic parallelism requires abstractions. In *PLDI*, 2007.

[14] J. R. Larus and R. Rajwar. *Transactional Memory*. Morgan & Claypool, 2006.

[15] Y. Lev and M. Moir. Debugging with transactional memory. In *TRANSACT*, 2006.

[16] C. Luk, R. Cohn, R. Muth, H. Patil, A. Klauser, G. Lowney, S. Wallace, V. J. Reddi, and K. Hazelwood. Pin: Building customized program analysis tools with dynamic instrumentation. In *PLDI*, 2005.

[17] P. Magnusson, M. Christianson, J. Eskilson, D. Forsgren, G. Hallberg, J. Hogberg, F. Larsson, A. Moestedt, and B. Werner. Simics: A full system simulation platform. In *IEEE Computer vol.35 no.2*, Feb 2002.

[18] M. Martonosi, A. Gupta, and T. A. Anderson. Memspy: Analyzing memory system bottlenecks in programs. In *SIGMETRICS*, 1992.

[19] P. E. McKenney. *Exploiting Deferred Destruction: An Analysis of Read-Copy Update Techniques in Operating System Kernels*. PhD thesis, Oregon Health and Science University, 2004.

[20] C. C. Minh, J. Chung, C. Kozyrakis, and K. Olukotun. STAMP: Stanford transactional applications for multi-processing. In *IISWC*, 2008.

[21] M. J. Moravan, J. Bobba, K. E. Moore, L. Yen, M. D. Hill, B. Liblit, M. M. Swift, and D. A. Wood. Supporting nested transactional memory in LogTM. In *ASPLOS-XII*. 2006.

[22] J. E. B. Moss. *Nested Transactions: An Approach to Reliable Distributed Computing*. PhD thesis, Massachusetts Institute of Technology, 1981.

[23] J. K. Ousterhout. Why aren't operating systems getting faster as fast as hardware? In *USENIX Summer*, 1990.

[24] S. M. Pant and G. T. Byrd. Limited early value communication to improve performance of transactional memory. In *ICS*, 2009.

[25] E. Perelman, G. Hamerly, M. V. Biesbrouck, T. Sherwood, and B. Calder. Using SimPoint for accurate and efficient simulation. In *SIGMETRICS*, 2003.

[26] C. Perfumo, N. Sonmez, A. Cristal, O. Unsal, M. Valero, and T. Harris. Dissecting transactional executions in Haskell. In *TRANSACT*, 2007.

[27] D. Porter, O. Hofmann, and E. Witchel. Is the optimism in optimistic concurrency warranted? In *HotOS*, 2007.

[28] D. E. Porter, O. S. Hofmann, C. J. Rossbach, A. Benn, and E. Witchel. Operating system transactions. In *SOSP*, 2009.

[29] H. Ramadan, C. Rossbach, D. Porter, O. Hofmann, A. Bhandari, and E. Witchel. MetaTM/TxLinux: Transactional memory for an operating system. In *ISCA*, 2007.

[30] H. E. Ramadan, C. J. Rossbach, and E. Witchel. Dependence-aware transactions for increased concurrency. In *MICRO*, 2008.

[31] C. Rossbach, O. Hofmann, D. Porter, H. Ramadan, A. Bhandari, and E. Witchel. TxLinux: Using and managing transactional memory in an operating system. In *SOSP*, 2007.

[32] C. J. Rossbach. *Hardware Transactional Memory: A Systems Perspective*. PhD thesis, The University of Texas at Austin, 2009.

[33] W. N. Scherer III and M. L. Scott. Advanced contention management for dynamic software transactional memory. In *PODC*, 2005.

[34] A. Shriraman and S. Dwarkadas. Refereeing conflicts in hardware transactional memory. In *ICS*, 2009.

[35] H. Volos, A. J. Tack, N. Goyal, M. M. Swift, and A. Welc. xCalls: Safe I/O in memory transactions. In *EuroSys*, 2009.

[36] C. von Praun, R. Bordawekar, and C. Cascaval. Modeling optimistic concurrency using quantitative dependence analysis. In *PPoPP*, 2008.

[37] L. Yen, J. Bobba, M. Marty, K. E. Moore, H. Volos, M. D. Hill, M. M. Swift, and D. A. Wood. LogTM-SE: Decoupling hardware transactional memory from caches. In *HPC*. Feb 2007.

[38] Y. Yu, T. Rodeheffer, and W. Chen. Racetrack: Efficient detection of data race conditions via adaptive tracking. In *SOSP*, 2005.

[39] C. Zilles and L. Baugh. Extending hardware transactional memory to support non-busy waiting and non-transactional actions. In *TRANSACT*, 2006.

[40] F. Zyulkyarov, T. Harris, O. S. Unsal, A. Cristal, and M. Valero. Debugging programs that use atomic blocks and transactional memory. *PPoPP*, 2010.