Modeling Transactional Memory Workload Performance

Donald E. Porter and Emmett Witchel

Department of Computer Sciences, The University of Texas at Austin, Austin, TX 78712

{porterde,witchel}@cs.utexas.edu

Abstract

Transactional memory promises to make parallel programming easier than with fine-grained locking, while performing 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. In order to help programmers assess the suitability of their code for transactional memory, this work 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. Because getting good performance from transactions is more difficult than commonly appreciated, developers need tools to tune transactional performance.

Categories and Subject Descriptors C.1.4 [*Processor Architectures*]: [Parallel Architecture]; C.4 [*Processor Architectures*]: Performance of Systems—Modeling Techniques

General Terms Design, Measurement, Performance

1. Introduction

Transactional memory [3] 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.

Designers of transactional memory systems make implementation decisions based on an implicit model of common-case application behavior, which may not be true for a given application and cause poor performance. For instance, TM designs often trade faster commits for slower aborts, causing applications with high-contention transactions to 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 then diagnose and correct performance problems.

Performance problems for transactional programs may also arise from the interaction of the TM system with unrelated portions of the system. Figure 1 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 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. These system effects are difficult for the application developer to anticipate and debug. Ultimately, these issues must be resolved with better integration of the TM implementation with the rest of the system stack, but the a tuning tool can help developers distinguish whether a performance problem originates with their code or a is attributable to system effects.

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

```
mov    0x8(%ebp),%edx
cmp    0xc03e6008,%edx
mov    %edx,%eax
cmovge 0xc03e600c,%eax
mov    %eax,0xc03e600c
```

Figure 1. A simple code sequence and the x86 assembly produced by gcc.

To help developers characterize and tune the performance of transactional programs, this work proposes and validates *Syncchar*, a formal model of transactional memory performance and a tool based on the model. The Syncchar, or synchronization characterization, 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.

2. 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. Transactional memory systems generally rely on *conflict serializability* as their safety condition. We call the set of addresses read during 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:

PPoPP'10, January 9–14, 2010, Bangalore, India.
ACM 978-1-60558-708-0/10/01.

$A_W \cap (B_R \cup B_W) \neq \emptyset$. 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.

Critical regions are *data independent* if their write set is disjoint from the other's address set. If critical sections concurrently modify the same data, or have *data conflicts*[1], the transactional memory system will serialize access to critical sections. In such cases, transactional memory can perform much worse than conservative locking due to the overhead required to detect and resolve conflicts.

Conflict density is a measure of the connectedness of the graph for a data conflict. 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.

There is very little published about performance tuning transactional programs. Heindl and Pokam [2] present a framework for analytical performance modeling of STM implementations, whereas this work models performance of applications. Perfumo et al. introduce a Haskell runtime with transactional memory instrumentation support for performance profiling [6]. This work is complementary to the Syncchar model, which provides limits that are not tied to specific schedules. The closest model is presented by von Praun et al.[9], which provides a definition of *dependence density* that is similar to the aggregation of data independence and conflict density under Syncchar. This work closes the loop by applying the model to making concrete performance predictions.

3. Model validation

In this section we validate the Syncchar model by implementing it as a module for Virtutech Simics [4] 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 [5]. All experiments model 8, 16, and 32 1GHz, x86 processors. Additional simulation parameters and results are available in a companion technical report [7]. All lock-based experiments are run on Linux version 2.6.16.1, and TM experiments use MetaTM [8]. Measurements presented are a mean of 4 simulated executions, with main memory access latency pseudo-randomly perturbed to account for simulator determinism [1].

Table 1 shows the predicted and actual execution times of the parallel phases of these benchmarks. The geometric mean error across 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%.

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

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 1. 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.

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.

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. Thus, the Syncchar model is an important tool that application developers will need to leverage transactional memory more effectively.

References

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

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

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

[4] P. Magnusson, M. Christianson, and J. E. et al. Simics: A full system simulation platform. In *IEEE Computer vol.35 no.2*, Feb 2002.

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

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

[7] D. E. Porter and E. Witchel. Understanding transactional memory performance. Technical Report TR-09-31, UTCS, 2009.

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

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