Chapter 5

Evaluation

This chapter asseses the assumptions that underly the LogTM system (Chapter 3) and presents an evaluation of the LogTM implementations presented in Chapter 4. Section 5.1 describes the methodology used in the evaluation, including system model assumptions. Section 5.3 describes the workloads used in this evaluation. Section 5.4 presents the overall performance of LogTM. Section 5.5 discusses the performance impact of various implementation tradeoffs in LogTM.

5.1 Methods

The evaluation of LogTM presented in this chapter was performed using execution-driven full-system simulation of various LogTM implementations based on two basic system architectures: a symetric multiprocessor (SMP) and a chip-multiprocessor (CMP). The rest of this section describes the system model and simulation tools in detail.

	SMP System Model Settings
Processors	32, 1 GHz, single-issue, in-order, non-memory IPC=1
L1 Cache	16 kB 4-way split, 1-cycle latency
L2 Cache	4 MB 4-way unified, 12-cycle latency
Memory	4 GB 80-cycle latency
Directory	Full-bit vector sharers list; Directory cache, 6-cycle latency
Interconnection Network	Hierarchical switch topology, 14-cycle link latency

TABLE 5-1. . SMP System Model Parameters

5.2 SMP System Model

LogTM and the baseline system share the same basic multiprocessor architecture, summarized in Table 5-1. This setup involves 32 processors, each with two levels of private cache. A MESI directory protocol maintains coherence over a high-bandwidth switched interconnect. Though single-issue and in-order, the processor model includes an aggressive, single-cycle non-memory IPC. A detailed model of the memory system includes most timing intricacies of the transactional memory extensions.

5.2.1 CMP System Model

The LogTM CMP system model also uses 32 processors and two levels of cache. In the CMP model, however, the processors are all located on the same chip and share a single multi-banked level two cache. The system parameters for the CMP model are show in Table Table 5-2. Like the SMP model, the processors in the CMP model are in-order and have a non-memory CPI of 1, but memory system timing is simulated in detail.

	CMP System Model Settings
Processors	32, 1 GHz, single-issue, in-order, non-memory IPC=1
L1 Cache	16 kB 4-way split, 1-cycle latency
L2 Cache	8 MB 4-way unified, 12-cycle latency
Memory	4 GB 500-cycle latency
Directory	Full-bit vector sharers list; Directory cache, 6-cycle latency
Interconnection Network	On-chip crossbar, 4-cycle link latency

TABLE 5-2. . CMP System Model Parameters

5.2.2 Simulation Platform

The simulation framework uses Simics [28] in conjunction with customized memory models built with the Wisconsin GEMS toolset [51]. Simics, a full-system functional simulator, accurately models the SPARC architecture but does not support transactional memory. The LogTM interface was instead added using Simics "magic" instructions: special no-ops that Simics catches and passes to the memory model. To implement the *begin* instruction, the memory simulator uses a Simics call to read the thread's architectural registers and create a checkpoint. During a transaction, the memory simulator models the log updates. After an abort rolls back the log, the register checkpoint is written back to Simics, and the thread restarts the transaction.

The memory system simulator performs the rollback of transactional updates in simulated hardware and approximates the timing of software rollbacks. The execution time of a rollback is estimated using a fixed penalty to model the overhead of trapping to the abort handler plus a penalty for each entry in the transaction log to account for execution of the software handler itself. This abort penalty is applied in two ways. First, when a load or store instruction triggers an abort, execution on that processor is stalled for the abort penalty before the register state is restored and the transaction reexecuted. Second, the W bits remain set for the duration of the abort penalty, potentially stalling conflicting transactions. This approximation allows me to more easily measure the effect of abort latency on performance in LogTM (Section 5.5).

5.3 Workloads

LogTM's design is based on certain assumptions about the behavior of transactions. The wisdom of decisions such as favoring commits over aborts and providing support for large transactions, will depend on the characteristics of transactions run on such systems. Due to the lack of software written for transactional memory, however, I can only make rough estimates of the behavior of transactions in future software. In this evaluation, I use two strategies to select applications on which to test LogTM: (1) using microbenchmarks that execute simple operations on common data structures and (2) converting criticial sections in today's lock-based programs into LogTM transactions.

Benchmark	Input	Synchronization Methods
Barnes	512 bodies	locks on tree nodes
Cholesky	14	task queue locks
BkDB	512 Operations	locks
MP3D	128 molecules	locks
Radiosity	room	task queue & buffer locks
Raytrace	small image(teapot)	work list & counter locks

TABLE 5-4. . Benchmarks and Inputs

5.3.1 Microbenchmarks

	Description	Settings
Shared Counter	All threads repeatedly increment a single counter.	2500 cycle average think time between transactions
B-Tree	Threads alternate between insert and lookup operations on a single tree. Each lookup or insert is performed in a transaaction.	9-ary B-Tree, intially 5-levels deep. 20 % update, 80 % lookup

TABLE 5-3. Microbenchmarks

5.3.2 Benchmarks

This section evaluates LogTM on a subset of the SPLASH-2 benchmarks and a benchmark based on the lock manager of an open source database. The benchmarks described in Table 5-4 use locks in place of, or in addition to, barriers. The LogTM version of the SPLASH-2 benchmarks replaces locks with begin and end transaction calls. Barriers and other synchronization mechanisms were not changed. Our base SPLASH-2 benchmarks use PARMACS library locks, which use test-and-test-and-set locks but yield the processor after a pre-determined number of attempts (only one for these experiments). In one case (Raytrace), the benchmark has been optimized for transactions, by reorganizing a data structure to reduce false sharing. Reduced false sharing allows Raytrace to run much faster than the original program (Section 5.5).

5.4 LogTM Performance

In this section, I evaluate the performance of LogTM and examine the effectiveness of alternative implementations and optimizations. First, in order to assess the overall performance of LogTM, I compare the relative scalability of LogTM transactions to locks. Because the performance of LogTM relative to locks is dependent on many parameters, I choose one set of parameters for the overall comparison and analyze the effect of varying those parameters separately in detail.

The promise of transactional memory—in terms of application performance—is to increase the scalability of parallel programs. I evaluate LogTM's ability to deliver this promise by comparing the scalability of the workloads described above using locks and LogTM transactions. The results below demonstrate that LogTM improves scalability in these workloads, increasing the maximum achievable speedup for all benchmarks except Cholesky and BkDB, which do not scale with locks or transactions. In one case, Raytrace, LogTM also increases the number of processors used to achieve peak throughput.

The LogTM configuration in this experiment assumes the SMP machine model described in Section 5.2 and uses LOAD_PC writeset prediction, buffered logging with a 64-entry log buffer and assumes an abort handler trap latency of 200 cycles and a 40-cycle per-block overhead for restoring old values. Section 5.5 will evaluate the impact of these parameters in detail.

Figures 5-1 and 5-2 display the speedup of both lock-based and transaction-based benchmarks as the number of threads is increased from 1 to 31. For all of the benchmarks that scale to multiple processors using locks (all but Cholesky and BkDB) the LogTM version of the benchmark has a higher peak speedup than the lock-based version. For Barnes, the improvement is modest, but for Raytrace, Radiosity and MP3D, LogTM provides a dramatic increase in peak performance.

For most of the benchmarks, the LogTM program and the lock program reach their peak performance at the same number of threads—Barnes at 11, MP3D at 21 and Radiosity at 15. Raytrace, on the other hand, scales much more effectively with LogTM, not reaching peak performance on LogTM until 19 threads, while lock-based Raytrace reaches its peak at only 3 threads. These benchmarks, particularly those in the SPLASH-2 suite, have already been carefully tuned using locks.

5.5 Implementation Tradeoffs

As discussed in Section 4.5, LogTM may be implemented in many ways, using varying levels of hardware support for log creation, log rollback, conflict resolution and write set prediction. In this section, I measure the performance impact of several of these tradeoffs.

5.5.1 Write Set Prediction

One important factor in the performance of LogTM is the frequency of aborts. LogTM's eager version management is optimized for the case that aborts are rare and, as the results below demonstrate, its performance suffers when aborts are common. In this section, I explore the effectiveness of write set prediction (Section 4.5.1) at reducing aborts and improving performance in LogTM.

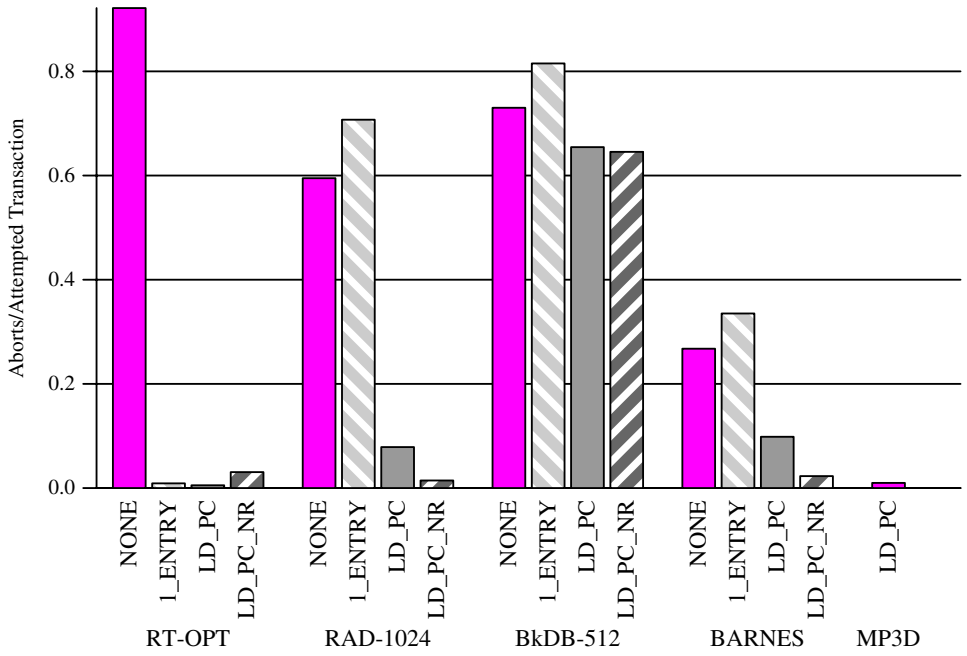

FIGURE 5-3. Abort rates for 3 write set predictors.

I compare the three schemes for predicting write sets in transactions described in Section 4.5.1: SINGLE-ENTRY, LOAD-PC and LOADPC-NORESET. Figure 5-3 displays the relative abort rate—the ratio of aborts to all attempted transactions. An abort rate of zero means that all transactions commit, whereas an abort rate of one indicates that all transactions abort (livelock). SINGLE-ENTRY suffices to reduce the abort rate for several benchmarks, especially Raytrace, which frequently updates shared counters inside transsactions. LOAD-PC and LOADPC-NORESET dramatically reduce the abort rate for all benchmarks. With LOAD_PC, only in BkDB do more than half of all transactions abort.

This reduction in abort rate, in most cases, leads to better performance in LogTM. FIGURE displays the performance of LogTM using each of the write set predictors normalized to the performance of the lock-based program. As shown in Figure 5-4, writeset prediction provides a substantial performance improvement in several workloads. In particular, writeset prediction speeds up LogTM on the benchmarks on which it performs the worst. Overall, LOAD-PC provides the most consistent performance improvements.

Although LOADPC-NORESET reduces aborts more than LOAD-PC, it also introduces more stalls, which erode the benefits of fewer aborts. BTREE?

5.5.2 Hardware Support for Logging

Section 4.1.4 describes the addition of a small buffer to aid logging in LogTM. The log buffer eliminates the need to perform a virtual to physical adddress translation on the log pointer for each store. When the buffer fills, however, software spills the contents of the hardware buffer to the in-memory log. Figure 5-5 examines the effect of the size of the log buffer on LogTM performance. For several benchmarks, the size of the log buffer has little effect on performance at all. Only Radiosity and BkDB seem particularly sensi-

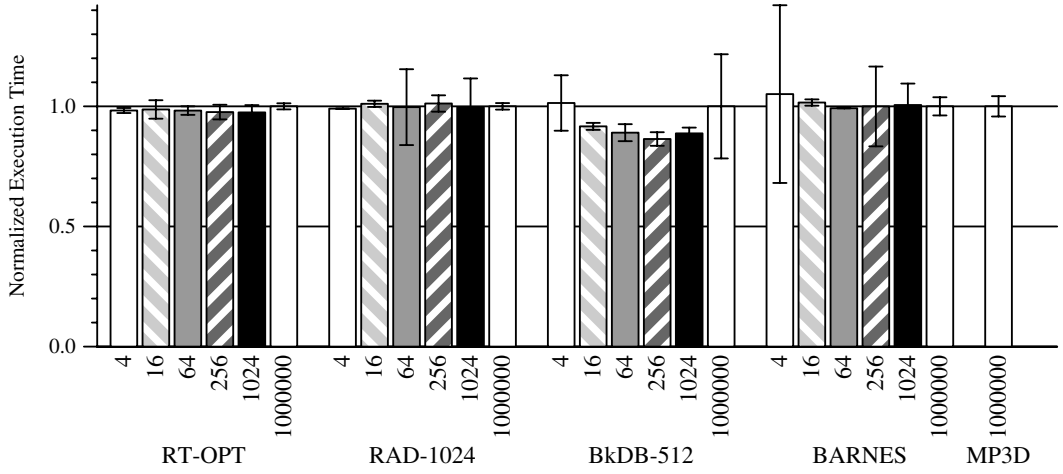

FIGURE 5-5. Performance impact of buffer-fill stalls.

tive to the size of the buffer. Even for those workloads, however, a reasonable size buffer (e.g., 64 entries) provides performance within 10% of that of an unlimited buffer.

5.5.3 Log Granularity

Section 4.1 discusses the trade-offs involved in selecting the granularity at which to build the transaction log in LogTM. The optimal granularity depends on both the size of transactions' write sets and the degree of spatial locality in them, both of which are dependent on the workload. For the workloads examined in this dissertation, the optimal logging granularity is quite small—likely 4 or 8-byte blocks.

Block Size	Barnes		BkDB		BTree		Cholesky		Mp3D		Radiosity		Raytrace	
	Blk/T	B/Blk	Blk/T	B/Blk	Blk/T	B/Blk	Blk/T	B/Blk	Blk/T	B/Blk	Blk/T	B/Blk	Blk/T	B/Blk
4	19.61	3.917	4.970	3.979	22.37	4.000	2.909	4.000	4.805	4.000	4.259	4.000	2.018	4.000
8	12.72	6.040	4.391	4.503	18.19	4.920	2.773	4.197	2.793	6.883	2.693	6.327	1.978	4.081
16	8.675	8.856	3.876	5.102	11.97	7.466	2.636	4.414	2.030	9.469	2.129	8.000	1.978	4.081
32	6.805	11.29	3.803	5.200	7.514	11.91	2.591	4.491	1.6248	11.83	1.903	8.950	1.978	4.081
64	5.506	13.95	3.317	5.962	4.765	18.78	2.000	5.881	1.468	13.09	1.753	9.718	1.978	4.082

TABLE 5-5. Log size/utilization at varying log granularities

Table 5-5 displays the average number of log entries per transaction (Blk/T) and average number of bytes used in each logged block (B/Blk) for logging at several different granularies. These results suggest that a

small logging block size will be more efficienct for these workloads. For many of the benchmarks, increasing the logging granularity (block size) does not result in a significant reduction of log entries per transaction. Of these benchmarks, only for Barnes and BTree does increasing the block size from 8 B to 64 B (an 8-fold increase) reduces the number of log entries by a factor of two or more. For Raytrace, increasing the block size beyond 4 B has almost no impact on the number of log entries. Interestingly, the benchmark for which a large block size results in the greatest reduction of log entries, BTree (22.4 to 4.76), was written using transactions from the outset rather than converted from a lock-based program. It may be the case that programs written specifically for transactional memory will use transactions that are longer-running and that write to more memory locations than the critical sections in programs written using lock-based mutual exclusion.

5.5.4 Conflict Detection and Resolution--INCOMPLETE

LogTM's innovation in conflict detection is its ability to maintain hardware detection of conflicts on memory blocks which are not cached by any processor. To evaluate the success of this mechanism, I consider two factors: the number of true conflicts that are not detected (false negatives) and the number of reported conflicts that are not actual conflicts (false positives). Any false negatives can lead to incorrect program behavior. Eliminating false negatives is necessary for any correct transactional memory implementation. False positives, on the other hand, impact performance, but not correctness.

TABLE: False positive rates for SMP/CMP design. SPLASH + BkDB.

TABLE: Conflict & Abort rates w/ different policies. Performance?

As mentioned in Chapter 3.4, one possible enhancement to LogTM would be to trap to a software contention manager [43] to resolve conflicts, possibly after waiting for a short time in hardware. The cumulative distribution of the length (in cycles) of the stalls in our benchmarks in Figure 5-6 shows that many stalls are short enough that it would be more efficient to simply wait than to re-schedule another thread on the

Benchmark	% < 256	% < 1K	% < 4K	% < 16K	% < 64K	Total Stalls
Barnes	33.9	60.1	85.5	96.5	99.6	1,400
Cholesky	17.3	40.2	95.5	99.9	100	1,482
Ocean	36.1	38.5	53.0	96.4	100	83
Radiosity	34.8	60.5	83.7	95.2	99.1	20,829
Raytrace-Base	5.57	6.16	9.04	34.1	93.7	13105
Raytrace-Opt	25.8	30.4	49.5	80.1	96.2	1,481
Water	34.8	40.6	65.2	98.5	100	69

TABLE 5-6. : Stall Duration Distribution in Cycles

stalled processor. Furthermore, if the processor is not re-scheduled, (e.g., if there are no idle threads) stalling can waste less work than repeatedly aborting and re-executing until the conflict is resolved. The distribution of stalls, however, has a long tail, so future work should consider trapping to a software contention manger in these cases.

5.5.5 False Sharing

This evaluation has confirmed the observation of Moore et al. [32] that *reducing false sharing with TM is even more important that reducing it with locks.* With TM, false sharing creates (apparent) conflicts that can stall or abort entire transactions. With locks, false sharing only slows progress with a few extra cache misses. For example, we created Raytrace-Opt from Raytrace-Base by eliminating false sharing between a global variable that provides a unique ray identifier and another that points to free blocks. Raytrace's most frequent (but short) transaction accesses the former, while a less frequent but long transaction accesses the later. Placing these variables in different memory blocks eliminated conflicts between the two transactions and greatly improved transactional execution time. The same action reduced cache misses in the lock-based version, but had a much smaller impact on execution time. LogTM shares this limitation with other

transactional memory implementations [4, 21, 41], except TCC [18], which mitigates this effect by optionally tracking transactions' read and write sets at word or byte granularity.