A Consistency Checking Optimization Algorithm for Memory-Intensive Transactions

Justin E. Gottschlich, Daniel A. Connors, Jeremy G. Siek

University of Colorado at Boulder

University of Colorado at Boulder
Technical Report CU-CS 1049-08

Dept. of Computer Science
Campus Box 430
University of Colorado
Boulder, Colorado 80309-0430

Dept. of Electrical and Computer Engineering
Campus Box 425
University of Colorado
Boulder, Colorado 80309-0425

Abstract

Transactional memory (TM), a recent parallel programming concept, aims to simplify parallel programming while simultaneously maintaining performance benefits found in concurrent applications. Consistency checking, the manner in which memory conflicts are identified in transactional memory, is a critical aspect to TM system performance. We present a theoretical, analytical and empirical view of our novel consistency checking algorithm which is optimized for spatially wide transactional workloads. Initial tests show our algorithm yields super linear performance improvements over other alternatives as transaction size grows, resulting in performance gains between $5x - 250x$ for experimental benchmarks.

1. Introduction

Conventional parallel programming synchronization mechanisms, such as locks, monitors and semaphores are exceptionally difficult to program correctly [1, 2, 6, 8, 9, 11]. Yet, as computer manufacturers continue to mass produce chip multi-processors (CMPs), more parallel programming is needed to fully utilize CMP resources [2, 6]. Without other parallel programming alternatives, the nondeterministic nature of parallel programming continues to be a prime deterrent to software developers who must ensure their released software does not exhibit some of the crippling behavior parallel programs are notorious for, such as deadlocking, livelocking and lock convoys [5].

Transactional memory (TM) [12], a new alternative for parallel programming, aims to reduce the complexity of writing parallel code while simultaneously maintaining the performance benefit found in multi-threaded programs. Transactions, unlike other parallel programming mechanisms, are easier for the programmer to reason about and place the synchronization complexity of the software into a system abstracted away from the view of the programmer.

1.1 Optimizing Memory-Intensive Transactions

The focus of this paper is on the optimization of memory-intensive transactions. We have found five components which play a critical role in optimizing large transactions. They are as follows.

(1) Primary: Consistency Checking. Consistency checking is the way a transaction verifies its state is consistent. Consistency checking plays a vital role in making memory-intensive transactions fast. We are not the first to make this assertion, Scott [17] and Spear et al. [18] have also found consistency checking to play a critical role in software transactional memory (STM) performance.

There are two ways to perform consistency checking: *validation* and *invalidation*. A validating STM system requires a transaction check its own read and write sets against global memory to verify it is consistent. An *invalidating* STM system requires a transaction analyze all other in-flight transactions and flag them as invalid if they conflict with its state. Since TM systems derived from database systems (which perform validation) most existing STM systems implement validation as a means of consistency checking [13]. Our algorithm uses invalidation consistency checking and is able to achieve super linear performance improvements over validation as explained in the following section.

(2) Conflict Detection. The time consistency checking is performed is called *conflict detection* [13, 18]. Conflict detection can be performed at two times: prior to a transaction committing or when a transaction is committing (also known as commit-time). Most STM systems require conflict detection be performed at least at commit-time. Prior research of Marathe et al. [14] has shown that performing conflict detection at various times in addition to commit-time can yield performance benefits. We do not contest these findings. However, in a lock-based system where commit-time invalidation is provably correct [7], we have found that commit-time only conflict detection performs better than other alternatives. The recent findings of Dice et al. and their use of a commit-time validation locking scheme within TL2 further supports this argument [3].

By performing conflict detection only at commit-time, comparisons are reduced to $O(N)$, where N is the number of required comparisons for consistency. Performing conflict detection any time prior to commit-time requires $\geq O(N)$ operations.

(3) Updating. Updating is the process of committing transactional writes to global memory [13] and is performed in either *direct* or *deferred* manners. Deferred updating creates a local copy of global memory, performs modifications to the local copy and then writes those changes to global memory if the transaction commits. If the transaction aborts, no additional work is done. Direct updating makes an original backup copy

of global memory and then writes directly to global memory. If the transaction commits, the transaction does nothing. If the transaction aborts, the transaction restores global memory with its backup copy. Recent TM systems are beginning to favor direct updating due to its natural optimization of commits (BSTM [10], McRT-STM [16] and LogTM [15]).

Our algorithm uses both direct and deferred updating, but performs best when using deferred updating because deferred updating allows commit-time consistency checking to proceed without speculative contention management for faster committing, but later arriving, writer transactions (not possible in direct updating).

(4) Synchronization. Our algorithm's lock-based design uses thread-level locking, unlike all other lock-based and non-blocking systems we are aware of, such as: Ennals [4], TL2 [3] and RSTM [14] (to name a few), which use memory-level locking (or CAS in the case of RSTM). Thread-level locking is preferred to memory-level locking for memory-intensive transactions as it requires F_i atomic locking operations, where F_i is the number of in-flight transactions when the ith transaction is committing. For committing transactions in which $r_i + w_i > F_i$, where w_i and r_i are the committing transaction's read and write set size and F_i are the in-flight transactions when the ith transaction is committing, thread-level locking requires less atomic operations than memory-level locking. By our definition, *memory-intensive transactions* are ones in which $r_i + w_i > F_i$ and perform faster under thread-level locking systems than under memory-level locking systems.

(5) Data Sets. Read and write sets are stored within transactions, not within memory locations directly. Storing reads and writes within the transaction reduces cache pressure, indirection and maintenance time that is otherwise necessary if transactional read and write sets were stored and maintained directly within memory locations.

1.2 Contributions

The technical contributions of this paper are:

- The main contribution of this work is the introduction of our invalidation algorithm and a theoretical and mathematical analysis of its behavior.
- A secondary contribution of this work is the presentation of analytical and experimental results detailing the performance of our invalidation algorithm in contrast with DracoSTM's and RSTM's validation algorithm.

2. Consistency Checking Theory

This section presents a theoretical and mathematical analysis of validation and invalidation consistency checking.

Assumptions. We assume conflict detection is done at commit-time and the updating policy of the system is deferred. Furthermore, our equations use worst-case complexity which (1) assumes no conflict or (2) that conflicts are found during the last consistency checking operation.

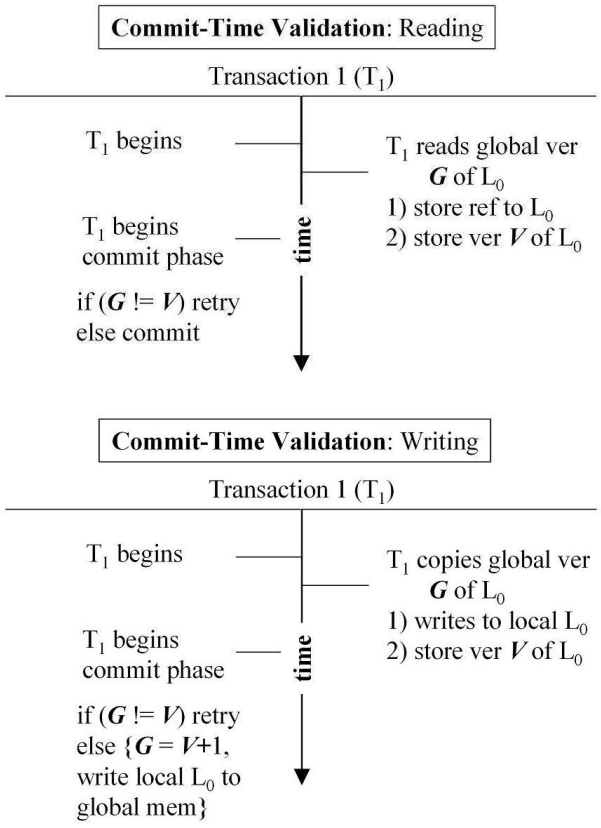

Figure 1. A commit-time validating system.

2.1 Validation

Validating STM systems ensure consistency by comparing transactional read and write data against global memory [13]. Validating transactions use versioning of memory to verify consistency. When the transaction's memory versions do not match global memory,

the transaction is aborted. When the transaction's memory versions do match global memory, global memory is updated with the transaction's write data.

A high-level diagram detailing how an STM system can implement validation is shown in Figure 1. Validation for read memory begins with storing a reference to the read memory location and the current global version number associated with it. When committing, the transaction then verifies the global version number of the read memory has not changed. This ensures the read memory is in the same state as it was when the transaction originally read the data. If the version number of the read data does not match global memory, the transaction must restart as the transaction is inconsistent. When a write is performed by a validating system, written to memory is copied as is the version number associated with it. When committing, the transaction verifies the global version number has not changed and updates global memory with its local changes. If global memory has changed, as indicated by a version number mismatch, the transaction restarts due to inconsistency.

Equation 1 (Validation). Given a non-unique series of M committing transactions, where w_i and r_i are the ith committing transaction's write and read set size, the commit-time consistency checking operations required for validation.

$$c(M) = \sum_{i=1}^{M} w_i + r_i \tag{1}$$

2.2 Invalidation

Invalidating STM systems assume their memory is consistent and flag conflicting in-flight transactions as invalid [13]. Invalidation functions on the premise that committing transactions not notified to abort are consistent. Committing invalidating transactions must guarantee conflicting in-flight transactions are identified as invalidated before they commit. As such, an invalidating transaction does not check its own state for consistency. Invalidating transaction only check other in-flight transactions for conflicts with itself and flags the conflicting transactions as invalid. Figure 2 illustrates the invalidation process. The difference in approach between invalidation and validation, leads to a significant change in the operational overhead required by each method.

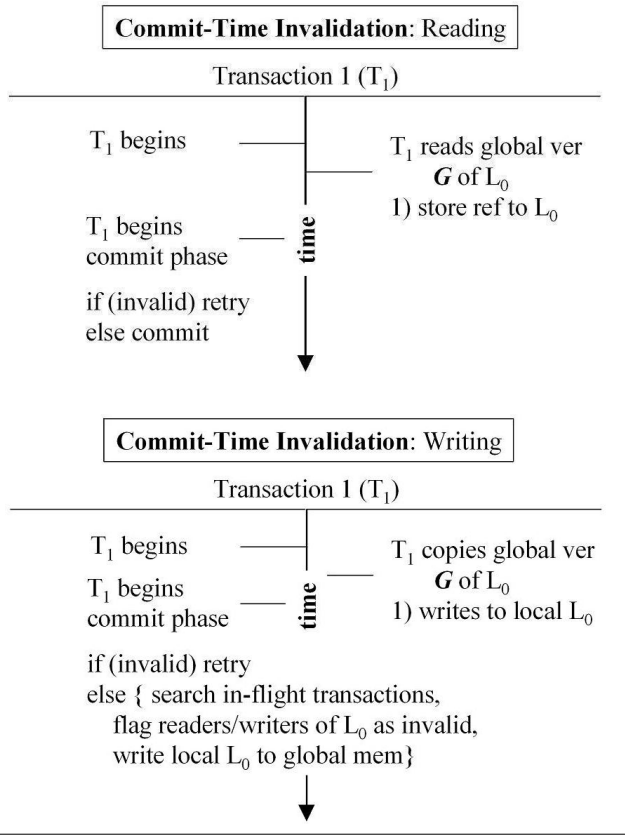

Figure 2. A commit-time invalidating system.

For writing, the invalidating transaction copies the data and writes to it – no version number is required. Prior to committing, the invalidating transaction checks to see if it has been notified to abort. If it has, it restarts the transaction, otherwise, it begins the commit phase. The commit phase of an invalidating transaction is quite different from a validating transaction. Committing invalidating transactions check other in-flight transactions for conflicts with its write data. If the committing, invalidating transaction locates another transaction that is reading or writing to data in its write set it must flag the in-flight transaction as invalid (or it must flag itself as invalid). The identified conflict requires that at least one of the transactions abort. Without considering priority inversion or other consequences, the committing transaction generally aborts the conflicting in-flight transaction.

Consistency checking overhead for read data is zero (optimal). The process of read data invalidation is shown in Figure 2. Prior to a transaction committing, the transaction checks to see if it has been flagged to

abort. If it has not been flagged to abort it begins the commit phase. *If the transaction is only reading, zero consistency checking is performed - the transaction simply commits.* This is a key observation and a fundamental basis for high performance for invalidating STM systems.

Equation 2 (Invalidation). Given a non-unique series of M committing transactions, where w_i is the ith committing transaction's write set size, F_i are the in-flight transactions at the time of the ith committing transaction and w_j and r_j are the jth in-flight transaction's write and read set sizes, and s is the asymptotic search function speed associated with the jth transaction, the commit-time consistency checking operations required for invalidation is shown in Equation 2.

$$c(M) = \sum_{i=1}^{M} \left(\sum_{j=1}^{F_i} w_i(s(w_j) + s(r_j)) \right) \quad (2)$$

Equation 3 (Optimized Invalidation). A naive implementation of Equation 2 is to implement the read and write sets in linear-time algorithms, such as linked lists. While the resulting performance of such an implementation can still outperform validation (Equation 1) and is shown in the following section, a search optimal container used for read and write sets can drastically reduce the overall required comparison operations of invalidation. By using binary search trees to store read and write sets for all transactions, Equation 2 can optimize the s search function from a linear-time operation to a logarithmic-time ($log_2()$) operation as shown in Equation 3.

$$c(M) = \sum_{i=1}^{M} \left(\sum_{j=1}^{F_i} w_i(log_2(w_j) + log_2(r_j)) \right) \quad (3)$$

2.3 Summary

The above differences in validation and invalidation lead to the following observations.

- Invalidation consistency checking requires verification of write sets. Validation consistency checking requires verification of both read and write sets.
- Invalidation has optimal transactional consistency checking (zero operations) for read-only transactions. Validation requires N consistency checking operations for the same transactions, where N is the number of elements in the transaction's read set.
- Invalidation has optimal transactional consistency checking (zero operations) when no other in-flight transactions exist, irrespective of the transaction's read and write set sizes. Validation requires N consistency checking operations for the same transactions, where N is the number of elements in the transaction's read and write sets.
- Invalidation has read and write set search-space optimizations. No counterpart search-space optimization exists in validation; all transactional read and write elements must be individually checked for consistency.

3. Consistency Checking Analysis

We now analyze the application of the validation (1) and invalidation (2) Equations on a linked list with two transactions (T_1 and T_2) and two operations (insert and lookup) on a pre-existing list containing ten nodes (0-9) as shown in Figure 3.

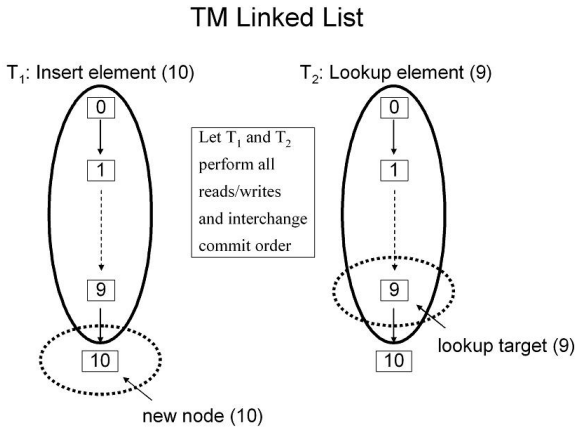

Figure 3. Linked list example.

T_1 inserts a node of value 10 by walking elements 0-9 and adding them to its read set. When T_1 inserts element 10 to the list, element 9 is promoted to its write

set. T_2 performs a lookup of value 9 by walking elements 0-9 and adding them to its read set. T_1 has 1 write element and 9 read elements, while T_2 has 10 read elements. We assume both T_1 and T_2 complete all transactional operations and just swap their commit orders: T_1 then T_2 and T_2 then T_1. From Equations 1, 2 and 3, the T_1, T_2 commit order results in $M = T_1, T_2, T_2$, while the T_2, T_1 commit order results in $M = T_2, T_1$. The non-unique M series is the same for both commit-time validation and invalidation equations. The memory conflict when node 9 is updated on T_1's insert, causes T_2 to abort on the T_1, T_2 commit order, resulting in the second T_2 seen in the M series.

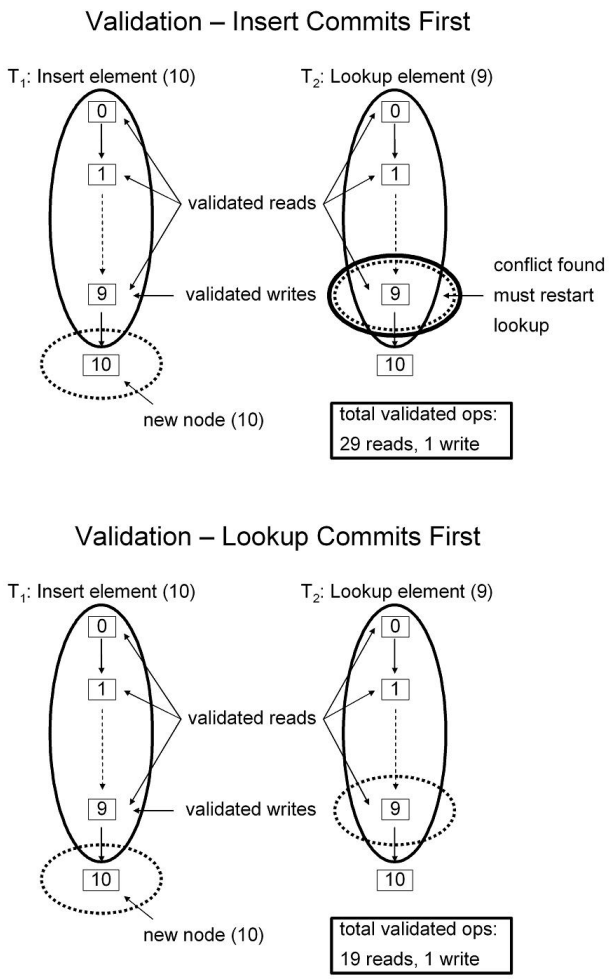

Figure 4. Validation operations for linked list.

For commit-time validation the T_1, T_2 commit order results in $M = T_1, T_2, T_2$. Applying Equation 1, 30 consistency checking operations are needed; w_1 + r_1 + w_2 + r_2 + w_2 + r_2 = 30. When the T_2, T_1 commit order occurs, $M = T_2, T_1$ and results in 20 operations as shown in Figure 4.

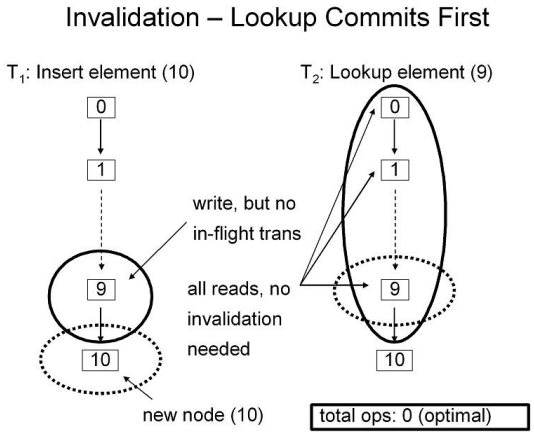

Figure 5. Invalidation operations for linked list.

For commit-time invalidation (Figure 5) the T_1, T_2 commit order also results in $M = T_1, T_2, T_2$. Applying Equation 2, 10 consistency checking operations are needed: $w_1(1 \times w_2 + 1 \times r_2) + w_2(0) = 10$. DracoSTM's implementation optimizes Equation 2 by storing read and write sets in binary search trees within the transactions (Equation 3), reducing their search time from linear to logarithmic; resulting a reduction of the T_1, T_2 commit order to 4 operations ($w_1(log_2(w_2) + log_2(r_2)) + w_2(0) = 4$). For the T_2, T_1 commit order, where $M = T_2, T_1, 0$ consistency checking operations are needed; when T_2 commits, its write set size is 0 and when T_1 commits afterward, no in-flight transactions exist and requires no operations.

Using DracoSTM's invalidation for the linked list with 10^1 elements, invalidation outperforms validation by $7.5x$ (30/4) for T_1, T_2 commit order and infinitely (20/0) for T_2, T_1 commit order. Yet, the performance differential in validation and invalidation are not constant as transaction size grows. The same T_1, T_2 commit order for a linked list which has 10^3 nodes (insert of 1000, lookup of 999), results in 3000 validations and 10 invalidations; $300x$ performance difference. The same scenario with a 10^5 sized linked list results in a $17,650x$ performance difference. As memory

increases, the performance differential between validation and invalidation widens at a super linear rate ($7.5x$, $300x$, $17,650x$), leading to fast performance for memory-intensive transactions when using DracoSTM's optimized invalidation algorithm.

4. Experimental Results

Our experiments compare DracoSTM's deferred and direct commit-time invalidation with RSTM's (University of Rochester STM) eager validation and DracoSTM's commit-time validation. We compare DracoSTM's deferred commit-time invalidation, DracoSTM's deferred commit-time validation and RSTM's eager validation to analyze how DracoSTM's commit-time invalidation performs against its own validating system and another validating system. We also compare DracoSTM's deferred updating system against its own direct updating system to draw conclusions on which type of updating policy usually performs better when aligned with commit-time invalidation.

For small transactions with a low thread count, RSTM is sometimes faster than DracoSTM. These results are as expected. According to Equations 1 and 3 transactions with small memory footprints show little performance differences in their consistency operations - the logarithmic benefits of optimized-search invalidation are not as visible until memory sizes grow. Furthermore, for all of the below transactional data structure algorithms many factors affecting speed. In the linked list benchmark, the workload is clearly dominated by the consistency checking algorithmic portion of the transaction. As such, the linked list case clearly illustrates the benefits of invalidation. However, in both the red-black tree and the hash tables, the parallel distribution of the workload and the smaller read and write set sizes, result in a reduced asymptotic algorithmic data structure workload. The reduced workload caused by low worst-case asymptotic overhead creates a nearly consistent performance differential for the overall benchmark due to the consistency checking performance playing a smaller role in overall performance. These results do not suggest the consistency checking algorithm is not bringing performance advantages, rather the performance advantages are not as well exposed as they are in linear-time algorithms.

Overview. Our comparisons of DracoSTM's various configurations and RSTM consist of inserts and inserts with lookups on three different data structures: red-black trees, hash tables and linked lists. All tests were performed on a 3.2 GHz 4-processor Intel Xeon with 16 GB of RAM. The graph data shows results for 4 and 8 threaded tests. In our benchmarks, we show commit-time invalidation with DracoSTM using direct and deferred updating, labeled iDraco_dir and iDraco_def, respectively. DracoSTM using deferred updating with validation is labeled vDraco_def. Finally, RSTM's algorithm using the polka contention manager is labeled RSTM_pol.

Inserts. Inserts into the data structures (red-black tree, hash table or linked list) are performed iteratively. For example, if 100 elements are inserted into a red-black tree, and the first element begins with a value of 0, then the next element's value will be 1, and then 2 and so on. To guarantee uniqueness, each thread inserts unique values of a specific range and the other threads are guaranteed to not insert on this range. For example, if thread 1 inserts elements 0-99, thread 2 will insert elements 100-199, thread 3 will insert elements 200-299, and so forth.

Lookups. Lookup operations are performed in an identical fashion as inserts. Values are iterated through sequentially as the values are known for each container. No attempts to retrieve values which do not exist within the containers are performed. While testing for non-existing values is a good test for sanity checking, it is not as importance for testing STM lookup time.

4.1 Data Interpretation

Each of the figures show a different container; red-black trees in figure 7, hash tables in figure 8 and finally linked lists in figure 6. Furthermore, each of the three figures show four graphs: (1) 4 threaded insert-only test, (2) 4 threaded insert + lookup test, (3) 8 threaded insert-only test and (4) 8 threaded insert + lookup test. The y-axis shows the transactions per second on a *logarithmic* scale, the x-axis shows the total number of elements inserted into the container (not the number inserted per thread).

Linked Lists. The linked list data structure and experimental results most clearly demonstrate the algorithmic performance differential of invalidation and validation. Linked list performance results are shown in figure 6. For the 4 threaded tests, DracoSTM's invalidation, validation and RSTM start with nearly identical performance. RSTM's performance quickly degrades,

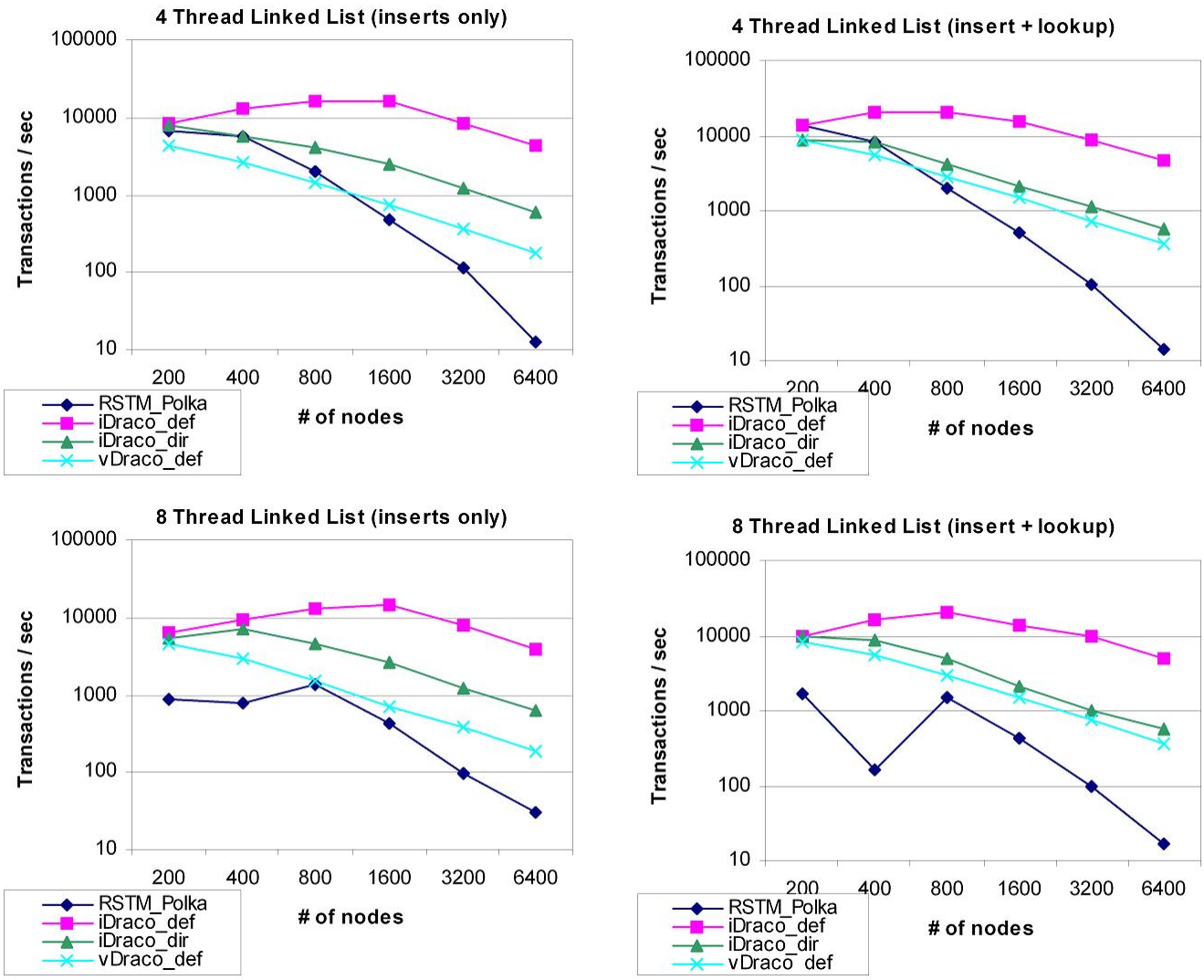

Figure 6. Linked List Benchmarks.

as does DracoSTM's validation, while DracoSTM's invalidation performance degrades slowly. For the 8 threaded tests, DracoSTM's invalidation and validation performance begins at roughly an order of magnitude faster than RSTM. However, as the linked list size doubles, the DracoSTM's validation slows significantly while DracoSTM's invalidation slows more gradually.

The resulting performance difference for the 4 threaded tests leaves DracoSTM outperforming RSTM by $\approx 250x$. For the 8 threaded test, RSTM starts $\approx 10x$ slower than DracoSTM and finishes $\approx 100x$ slower than DracoSTM. The DracoSTM invalidation algorithm also outperforms its own validation algorithm by $12x$ for the 4 threaded test and $13x$ for the 8 threaded test.

Red-Black Trees. Figure 7 shows the red-black tree performance of DracoSTM and RSTM. RSTM is faster than DracoSTM for both initial 4 threaded tests. However, as the size of the container doubles, DracoSTM widens the performance gap eventually leading to an order of magnitude performance difference between DracoSTM and RSTM. For the 8 threaded tests, RSTM is slightly slower than DracoSTM's deferred updating, but faster than DracoSTM's direct updating. At the final test interval, DracoSTM is roughly an order of magnitude faster than RSTM. DracoSTM's validating algorithm performs roughly an order of magnitude slower than DracoSTM's invalidating algorithm, continually. These results indicate that the workload overhead of the red-black tree container are higher than the consistency

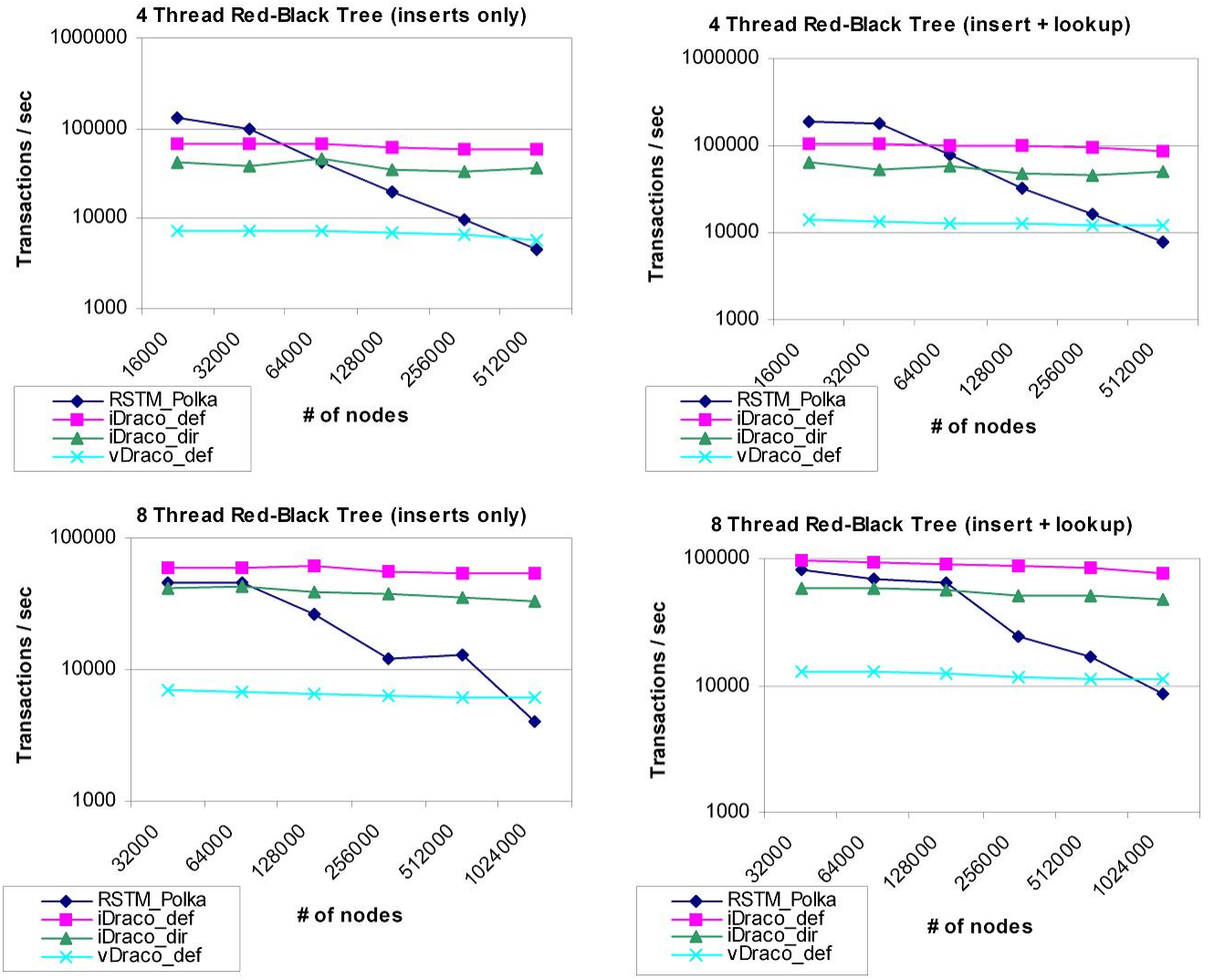

Figure 7. Red-Black Tree Benchmarks.

checking operations for DracoSTM transactions. However, for small red-black container sizes (1000 nodes per thread), not shown here, the DracoSTM performance difference between invalidation and validation is less than $5x$. The increased rate of difference for the larger sized red-black trees demonstrates the affects of the consistency checking algorithm's logarithmic performance.

Hash Table. Hash tables performance numbers for DracoSTM and RSTM are shown in figure 8. For the 4 threaded tests, RSTM begins $\approx 5x$ faster than DracoSTM and ends with DracoSTM $\approx 5x$ faster than RSTM. For the 8 threaded test, DracoSTM begins $\approx 2x$ faster than RSTM and ends with DracoSTM $\approx 5x$ faster than RSTM.

Hash tables as implemented by RSTM and DracoSTM, are bucketed linked lists. As such, they are innately parallel yielding low conflict rates and small read and write sets, due to highly distributed small containers. As such, the consistency checking algorithmic affects are less visible in hash tables than the others. Still, DracoSTM's $5x$ performance difference with RSTM is impressive for such an innately parallel structure. Furthermore, DracoSTM's invalidation algorithm continues to perform $2x$ faster than its own validation algorithm even for higher hash table workloads which are likely to remain highly distributed, demonstrating that even in highly distributed workloads the performance savings of invalidation is visible with large transactions.

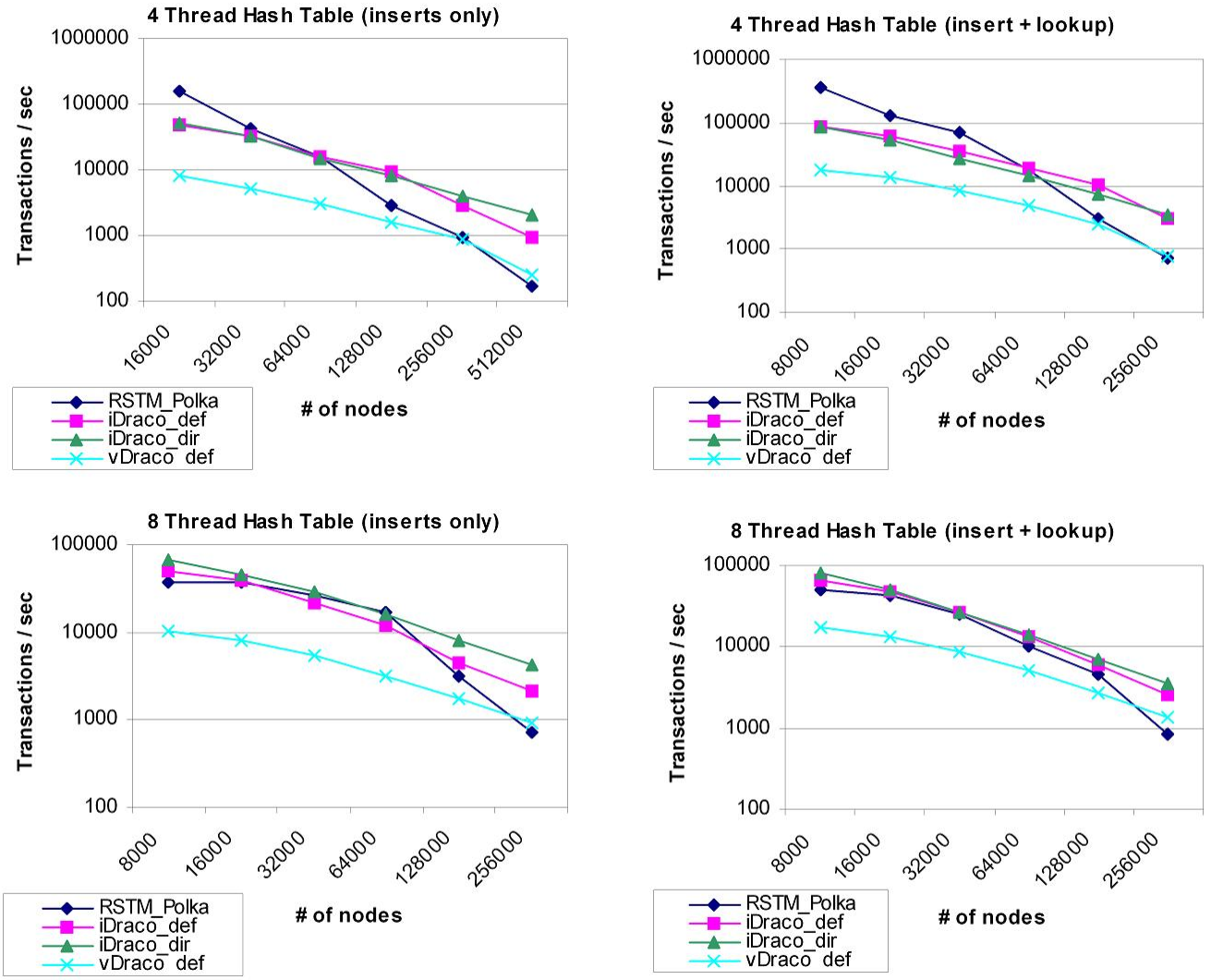

Figure 8. Hash Table Benchmarks.

5. Conclusion

This paper presented a theoretical, analytical and experimental view of our optimized consistency checking algorithm. Our high-level analysis of validation and invalidation showed that invalidating systems: (1) have read and write set search-space optimizations in which no counterpart exists for validating systems, (2) require zero consistency checking for reads, creating optimally performing read-only transactions and (3) require zero consistency checking for isolated transactions. After presenting the theoretical and mathematical view, we iterated through a linked list analysis, which revealed super linear performance differences for invalidation and validation as transactional memory size increased. We then presented experimental data comparing our invalidation algorithm implemented within DracoSTM against DracoSTM's own commit-time validation and RSTM's eager validation. For some small-memory footprint cases, all three systems perform similarly. As transaction size increases, RSTM and DracoSTM's commit-time validation performance degrades at a quicker rate than DracoSTM's invalidation for algorithms with linear-time complexity and at a more constant rate for algorithms with logarithmic complexity while still performing universally faster for all memory-intensive experiments. These observations suggest future STM systems which work with memory-intensive transactions should consider a similar approach to ours in order to reduce consis-

tency checking operations and optimize transactional throughput.

References

[1] C. S. Ananian, K. Asanovic, B. C. Kuszmaul, C. E. Leiserson, and S. Lie. Unbounded transactional memory. In *Proceedings of the Eleventh International Symposium on High-Performance Computer Architecture*, pages 316–327. Feb 2005.

[2] B. Carlstrom, J. Chung, H. Chafi, A. McDonald, C. Minh, L. Hammond, C. Kozyrakis, and K. Olukotun. Transactional execution of java programs, 2005.

[3] D. Dice, O. Shalev, and N. Shavit. Transactional locking II. In S. Dolev, editor, *Distributed Computing (20th DISC'06)*, volume 4167 of *Lecture Notes in Computer Science (LNCS)*, pages 194–208. Springer-Verlag (New York), Stockholm, Sept. 2006.

[4] R. Ennals. Software transactional memory should not be obstruction-free. Technical report, Intel Research Tech Report, Jan 2006.

[5] K. Fraser and T. Harris. Concurrent programming without locks. *ACM Trans. Comput. Syst*, 25(2), 2007.

[6] C. Gniady, B. Falsafi, and T. N. Vijaykumar. Is SC + ILP=RC? In *ISCA*, pages 162–171, 1999.

[7] J. E. Gottschlich. Exploration of lock-based software transactional memory. Master's thesis, University of Colorado at Boulder, Oct. 2007.

[8] L. Hammond, B. D. Carlstrom, V. Wong, B. Hertzberg, M. Chen, C. Kozyrakis, and K. Olukotun. Programming with transactional coherence and consistency (TCC). *j-SIGPLAN*, 39(11):1–13, Nov. 2004.

[9] T. Harris and K. Fraser. Language support for lightweight transactions. *j-SIGPLAN*, 38(11):388–402, Nov. 2003.

[10] T. Harris, M. Plesko, A. Shinnar, and D. Tarditi. Optimizing memory transactions. *ACM SIGPLAN Notices*, 41(6):14–25, June 2006.

[11] Herlihy, Luchangco, Moir, and Scherer. Software transactional memory for dynamic-sized data structures. In *PODC*, 2003.

[12] M. Herlihy and J. E. B. Moss. Transactional memory: Architectural support for lock-free data structures. In *ISCA*, pages 289–300, 1993.

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

[14] V. J. Marathe, M. F. Spear, C. Heriot, A. Acharya, D. Eisenstat, W. N. Scherer, III, and M. L. Scott. Lowering the overhead of nonblocking software transactional memory. revised, University of Rochester, Computer Science Department, May 2006.

[15] K. E. Moore, J. Bobba, M. J. Moravan, M. D. Hill, and D. A. Wood. Logtm: Log-based transactional memory. In *Proceedings of the 12th International Symposium on High-Performance Computer Architecture*, pages 254–265. IEEE Computer Society, Feb. 2006.

[16] B. Saha, A.-R. Adl-Tabatabai, R. L. Hudson, C. C. Minh, and B. Hertzberg. McRT-STM: a high performance software transactional memory system for a multi-core runtime. In *PPOPP*. ACM SIGPLAN 2006, Mar. 2006.

[17] M. L. Scott. Sequential specification of transactional memory semantics. In *Proceedings of the First ACM SIGPLAN Workshop on Languages, Compilers, and Hardware Support for Transactional Computing*. Jun 2006.

[18] M. F. Spear, V. J. Marathe, W. N. Scherer III, and M. L. Scott. Conflict detection and validation strategies for softwaretransactional memory. In *Proceedings of the Twentieth International Symposium on Distributed Computing*, Sep 2006.