ABSTRACT

PANT, SALIL M. Value Communication Techniques to Improve Scalability of Transactional Memory Systems. (Under the direction of Dr. Gregory T. Byrd).

Transactional Memory(TM) is an optimistic speculative synchronization scheme that provides atomic execution for a region of code marked as a transaction by the programmer. Programs with critical sections that are not heavily contended benefit from the optimistic nature of TM systems. However, for heavily contended critical sections, performance for TM systems can degrade due to conflicts leading to stalls and expensive rollbacks. In this thesis, we investigate methods to improve scalability of TM systems using early value communication (EVC) and understand its mechanisms and hardware complexity.

We look into the nature of the shared data involved in conflicts for TM systems and find that most transactions have conflicts around a few shared addresses and shared-conflicting data is often updated in a predictable manner by different transactions. We propose using a memory-level value predictor (VP-TM) to capture this predictability for such data structures and increase overall concurrency by satisfying loads from conflicting transactions with predicted values, instead of stalling. We present one possible design and implementation of TM system with a value predictor. Our benchmark results show us that the value predictor can capture this predictable behavior for most benchmarks and can improve performance of TM programs by improving concurrency and minimizing stalls and rollbacks due to conflicts.

To reduce the hardware complexity, we present another design that can provide performance from EVC without the extra hardware costs. Finally, we present a realistic design of the VP-TM system that shows the full impact of the extra hardware and messages and gives us a better idea of performance with VP-TM. Our goal is to increase the adoption of TM for parallel programming by extending the number of applications that can benefit from TM.
Value Communication Techniques to Improve Scalability of Transactional Memory Systems

by

Salil Mohan Pant

A dissertation submitted to the Graduate Faculty of North Carolina State University in partial fulfillment of the requirements for the Degree of Doctor of Philosophy

Computer Engineering

Raleigh, North Carolina

2010

APPROVED BY:

Dr. Xiaosong Ma

Dr. Thomas Conte

Dr. Gregory T. Byrd

Chair of Advisory Committee

Dr. Eric Rotenberg
DEDICATION

To

Mom, Dad and my lovely wife, Allison
BIOGRAPHY

Salil Mohan Pant was born on 27\textsuperscript{th} May 1981 in Thane, India. He received the Bachelor of Engineering (B.E.) Degree in Electronics Engineering from K. J. Somaiya College of Engineering, University of Mumbai, in 2002.

Salil has been a graduate student in the Computer Science Department at North Carolina State University, Raleigh, NC since Fall 2002. He is a student member of the Institute of Electrical and Electronics Engineers (IEEE). Since Fall 2003, he has been working as a Research Assistant with Dr G. Byrd in the field of Parallel Processing. He graduated with a Masters Degree in Computer Science in Spring 2004 and a PhD degree in Electrical and Computer Engineering in Spring 2010. Somewhere during his PhD, he managed to meet the most beautiful woman ever, woo her and marry her in September 2009.
ACKNOWLEDGMENTS

First and foremost I would like to thank God because of the inner strength and perserverance that he has provided me with. It is because of these qualities that I have been able to carry out my research work and finish my degree.

Next, I would like to thank my parents and my sisters for everything they have given me in life. It is only due to their love, support and encouragement that I could achieve whatever I have. Special thanks to my father for his continuous guidance, inspiration and motivation to aim big and work hard towards my goals.

I would like to thank my lovely wife Allison who had to tolerate all my weird working hours during deadlines. I would not have been able to graduate without her.

I sincerely thank my advisor Dr Greg. Byrd for giving me the opportunity to work under his guidance. He has been very understanding, supportive and inspiring throughout my work.

I would also like to thank my committee members for agreeing to be on my committee and providing valuable insight for this work from time to time.

A special mention has to be made of the coffee shop that my friends and I have been to regularly for almost 7 years - Cup-a-Joe. The time spent there with all the discussions and their coffee definitely made my stay in Raleigh more enjoyable.
TABLE OF CONTENTS

List of Tables ................................................................. viii
List of Figures ................................................................. ix

Chapter 1 Introduction ....................................................... 1
  1.1 Synchronization and Conventional Locking
      Techniques ............................................................... 1
  1.2 Transactional Memory .................................................. 2
  1.3 Problems with Hardware Transactional Memory ................. 5
  1.4 Memory-level Value Predictor ...................................... 6
  1.5 Value-Predicted Transactional Memory ......................... 6
  1.6 Hardware Complexity of VP-TM ................................. 7
  1.7 Limited Early Value Communication ............................. 8
  1.8 Realistic Value Predictor Design .............................. 8

Chapter 2 Conventional Synchronization and Transactional Memory 11
  2.1 The problem of Synchronization .................................. 11
      2.1.1 Synchronization using Atomic instructions .............. 12
      2.1.2 Problems with Locking ..................................... 14
  2.2 Transactional Memory .............................................. 16
  2.3 Hardware-supported TM .......................................... 18

Chapter 3 Related Work .................................................... 20
  3.1 Transaction Design Space ......................................... 20
      3.1.1 Classification of HTM systems ........................... 21
  3.2 Evolution of HTM systems ...................................... 22
      3.2.1 Herlihy’s proposal ......................................... 22
      3.2.2 Transactional Lock Removal .............................. 23
      3.2.3 Transactional Coherence and Consistency - TCC ........ 24
      3.2.4 Unbounded/Large Transactional Memory ............... 25
  3.3 Software Transactional Memory .................................. 26
<table>
<thead>
<tr>
<th>Chapter 3</th>
<th>Section</th>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>3.3.1</td>
<td>Shavit’s STM system</td>
<td>27</td>
<td></td>
</tr>
<tr>
<td>3.3.2</td>
<td>Dynamic STM</td>
<td>27</td>
<td></td>
</tr>
<tr>
<td>3.3.3</td>
<td>Hybrid TM</td>
<td>28</td>
<td></td>
</tr>
<tr>
<td>3.4</td>
<td>Value Predictors</td>
<td>28</td>
<td></td>
</tr>
<tr>
<td>3.5</td>
<td>Thread-level Speculation</td>
<td>30</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Chapter 4</th>
<th>Section</th>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>4.1</td>
<td>Issues with TM</td>
<td>32</td>
<td></td>
</tr>
<tr>
<td>4.2</td>
<td>Scalability of applications with TM</td>
<td>34</td>
<td></td>
</tr>
<tr>
<td>4.2.1</td>
<td>Evaluating the Scalability problem</td>
<td>35</td>
<td></td>
</tr>
<tr>
<td>4.2.2</td>
<td>Analysis</td>
<td>37</td>
<td></td>
</tr>
<tr>
<td>4.2.3</td>
<td>Problems with conflicts on large systems</td>
<td>38</td>
<td></td>
</tr>
<tr>
<td>4.3</td>
<td>Dependence-Aware Transactional Memory</td>
<td>39</td>
<td></td>
</tr>
<tr>
<td>4.4</td>
<td>Analysis of TM benchmarks</td>
<td>40</td>
<td></td>
</tr>
<tr>
<td>4.5</td>
<td>A Case for Value Prediction</td>
<td>43</td>
<td></td>
</tr>
<tr>
<td>4.6</td>
<td>Final Motivation - Parallel Programming Advantage</td>
<td>47</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Chapter 5</th>
<th>Section</th>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>5.1</td>
<td>Description of the base LogTM system</td>
<td>48</td>
<td></td>
</tr>
<tr>
<td>5.2</td>
<td>Value-Predicted TM</td>
<td>49</td>
<td></td>
</tr>
<tr>
<td>5.3</td>
<td>Coherence protocol for the VP-TM system</td>
<td>53</td>
<td></td>
</tr>
<tr>
<td>5.3.1</td>
<td>Dependencies Among Predicted Transactions</td>
<td>53</td>
<td></td>
</tr>
<tr>
<td>5.4</td>
<td>Performance Evaluation and Results</td>
<td>57</td>
<td></td>
</tr>
<tr>
<td>5.4.1</td>
<td>Predictions, Accuracy and Speedup</td>
<td>58</td>
<td></td>
</tr>
<tr>
<td>5.4.2</td>
<td>Static vs. Dynamic Analysis</td>
<td>59</td>
<td></td>
</tr>
<tr>
<td>5.5</td>
<td>Evaluation using Microbenchmarks</td>
<td>59</td>
<td></td>
</tr>
<tr>
<td>5.5.1</td>
<td>Performance Analysis</td>
<td>60</td>
<td></td>
</tr>
<tr>
<td>5.5.2</td>
<td>Evaluation using benchmark suites</td>
<td>62</td>
<td></td>
</tr>
<tr>
<td>5.5.3</td>
<td>Benchmark results</td>
<td>63</td>
<td></td>
</tr>
<tr>
<td>5.5.4</td>
<td>Static vs. Dynamic results</td>
<td>65</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Chapter 6</th>
<th>Section</th>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>6.1</td>
<td>Introduction to LEVC</td>
<td>67</td>
<td></td>
</tr>
<tr>
<td>6.1.1</td>
<td>Problem of Dependent Transactions</td>
<td>69</td>
<td></td>
</tr>
<tr>
<td>6.2</td>
<td>Analysis of TM benchmarks</td>
<td>71</td>
<td></td>
</tr>
<tr>
<td>6.3</td>
<td>Design of LEVC system</td>
<td>73</td>
<td></td>
</tr>
<tr>
<td>6.4</td>
<td>Performance Evaluation and Results</td>
<td>76</td>
<td></td>
</tr>
<tr>
<td>6.4.1</td>
<td>Benchmark results</td>
<td>78</td>
<td></td>
</tr>
</tbody>
</table>
Chapter 7 Realistic Design for VP-TM .............................................. 80
  7.1 VP-TM design ............................................................... 80
      7.1.1 Structure of the Value Predictor ............................... 81
  7.2 Changes to VP-TM .......................................................... 83
      7.2.1 Optimizations and Timing Issues .................................. 84
  7.3 Experiments and Results ................................................. 86

Chapter 8 Conclusion and Future Work ........................................... 89
  8.1 Comparison of Early Value Communication Schemes ....................... 90
  8.2 Conclusion ......................................................................... 90
  8.3 Future Work from Early Value Communication .............................. 91

Bibliography ................................................................................. 93
LIST OF TABLES

Table 5.1  System Parameters for our simulations . . . . . . . . . . . . . . 58
Table 5.2  Parameters: 1024 transactions - Insert-only version . . . . . 62
Table 5.3  Parameters: 1024 transactions - Random version . . . . . . . 62
Table 5.4  Benchmark Parameters . . . . . . . . . . . . . . . . . . . . . . . 63

Table 6.1  System Parameters for our simulations . . . . . . . . . . . . . 77
Table 6.2  Benchmark Parameters . . . . . . . . . . . . . . . . . . . . . . . 77

Table 8.1  Table comparing early value communication schemes . . . . . . 89
LIST OF FIGURES

Figure 1.1 Example using locks ........................................ 3
Figure 1.2 Example using TM ........................................... 4
Figure 1.3 Contention in TM ............................................ 5

Figure 2.1 Example using locks ........................................ 12
Figure 2.2 Example using locks ........................................ 12
Figure 2.3 Example using locks ........................................ 13
Figure 2.4 Use of locks .................................................. 14
Figure 2.5 Race conditions with locks ................................. 15

Figure 4.1 Scalability of Queue microbenchmark .................... 36
Figure 4.2 Scalability of Cholesky and Radiosity ..................... 37
Figure 4.3 Example using DATM ........................................ 38
Figure 4.4 Table showing conflict characteristics .................... 41
Figure 4.5 Queue data structure ...................................... 43
Figure 4.6 Queue data structure ...................................... 43
Figure 4.7 Stamp Examples from Vacation and Labyrinth benchmarks .. 44

Figure 5.1 Our base simulated system ................................. 50
Figure 5.2 Structure of the VP entry .................................. 51
Figure 5.3 Coherence Protocol Actions with VP-TM ................... 54
Figure 5.4 Deadlock cases ............................................. 55
Figure 5.5 Results for Insert-only version of Queue ................ 60
Figure 5.6 Results for Random Inserts and Delete ver. of Queue .... 61
Figure 5.7 Speedup Results for 16 processors ......................... 63
Figure 5.8 Benchmark characteristics for LogTM and VP-TM ......... 64
Figure 5.9 Static vs. Dynamic speedup ................................ 66

Figure 6.1 Deadlock with Early Communication ....................... 70
Figure 6.2 Table showing conflict characteristics ................... 72
Figure 6.3 Nacks in Different models ................................ 74
Figure 6.4 Speedup Results for 16 processors ......................... 78
Figure 6.5 Characteristics of TM benchmarks ......................... 79
Figure 7.1  Deadlock for VP-TM  
Figure 7.2  Races with UPDATE and ADD  
Figure 7.3  Optimized VP-TM results  

82  
85  
86
Chapter 1

Introduction

1.1 Synchronization and Conventional Locking Techniques

Parallel programs often need synchronization for correct execution [17]. Insufficient synchronization can lead to data races in the program. Unnecessary synchronization may or may not affect correctness, but it restricts concurrency. Ideally, synchronization should be fine-grained; only those threads that need to synchronize for correct execution should be stalled. The goal of a parallel programmer is to always keep the stall time due to synchronization at a minimum while maintaining correctness in all cases.

Current high performance architectures generally provide support for synchronization in the form of special read-update primitives like Test-and-Set or Fetch-and-Add. The programmer or the library writer can implement any kind of synchronization event using these basic primitives. One disadvantage of using atomic read-update primitives is the overhead generated during synchronization events. A naive implementation of mutual exclusion lock using test-and-set can produce a worst case $O(P)$ traffic on the bus for $P$ processors since every failed test and set invalidates the copy of the synchronization variable in other caches [2, 27]. Many different array- and
queue-based lock schemes have been proposed to avoid the extra traffic generated during locking [27, 16]. These schemes successfully reduce traffic for a contended lock but they increase the overhead for uncontested locks. In general, there is always some synchronization overhead with locking and it is recommended to use synchronization at a coarser granularity, when there is enough work within the critical section to amortize the cost of obtaining the lock.

Thus, although synchronization at a fine-grain exposes concurrency in a parallel program, it has the following issues:

1. It can make programming and debugging very hard, especially in presence of nested locks. It can also increase deadlock/livelock and starvation scenarios.

2. Fine-grained locking can lead an to increased number of locking instances, which can increase overhead during synchronization due to contention. This can reduce the perceived benefits from fine-grain synchronization.

Coarse-grained synchronization may not provide as much concurrency as fine-grained synchronization but it is often preferred because of ease of programming. This is a classic trade-off with parallel programming.

1.2 Transactional Memory

Transactional Memory (TM) [14, 37, 12, 1, 31, 45] has been proposed as one way of providing atomic execution for multiple memory addresses without using locks. TM systems try to avoid some of the performance problems associated with locking by transactional execution, namely optimistic speculative execution of the critical section in absence of “conflicting accesses”. Concurrent accesses are said to “conflict” if they are made to the same memory address and at least one of them is a write [44]. If accesses from different threads conflict, all but one threads abort speculative execution and restart the critical section. If a transaction reaches the end of the critical section
Thread 1

parallel_region()
{
    .
    .
    LOCK(X);
    update_A();
    LOCK(Y);
    update_B();
    UNLOCK(Y);
    UNLOCK(Z);
    .
    LOCK(X);
    update_A();
    LOCK(Z);
    update_C();
    LOCK(Z);
    update_C();
    LOCK(Y);
    update_B();
    UNLOCK(Y);
    UNLOCK(Z);
    UNLOCK(X);
}

Thread 2

parallel_region()
{
    .
    .
    LOCK(Z);
    update_C();
    UNLOCK(Z);
    .
    LOCK(Z);
    update_C();
    LOCK(Y);
    update_B();
    UNLOCK(Y);
    UNLOCK(Z);
    .
    LOCK(Z);
    update_B();
    LOCK(Z);
    update_B();
    UNLOCK(Y);
    UNLOCK(Z);
}

Thread 3

parallel_region()
{
    .
    .
    LOCK(X);
    update_A();
    LOCK(Y);
    update_B();
    UNLOCK(Y);
    UNLOCK(Z);
    .
    LOCK(Z);
    update_C();
    LOCK(Y);
    update_B();
    UNLOCK(Y);
    UNLOCK(Z);
    .
    LOCK(Z);
    update_B();
    LOCK(Z);
    update_B();
    UNLOCK(Y);
    UNLOCK(Z);
}

Figure 1.1: Figure showing an example of how locks can be used for synchronization. Lock variables X, Y and Z “protect” shared variables A, B and C respectively.

without encountering any conflicts, it can “commit” its speculative data and turn non-speculative. Atomic execution is achieved by not exposing speculative data to other threads until commit and instantly turning all speculative data to non-speculative upon commit.

Perhaps, the biggest advantage of TM systems comes from the simple programming model it offers to parallel programmers compared to locks. For parallel programmers, a transaction is *an abstraction that provides for atomic execution of a block of code along with isolation, consistency and durability guarantees*. In most cases, TM requires the parallel programmers to only enclose critical sections as transactions and provides the guarantee of atomic execution without using any named variables like conventional locks as shown in figures 1.1 and 1.2. This frees parallel programmers from the burden of carefully associating every shared variable with the correct lock variables. This has the advantage of making programming as well as debugging easier. Programming is easier because there are no lock variables. Atomicity is automati-
Figure 1.2: Figure showing how the same program can be written using transactions. The nested locks have been flattened here and this may reduce concurrency. There are models of TM which use nested transactions but we do not show them here for simplicity.

cally provided for any region of code marked as transaction and each transaction is executed atomically with respect to other transactions. This can also make debugging easier and with less correctness bugs to deal with, the programmer can now focus more on performance issues. TM systems can also provide wait-free properties to critical sections, that avoids dangerous scenarios like priority inversion, thread-starvation, deadlocks etc. These scenarios are very difficult to reason about during parallel programming. The above features of TM systems makes writing parallel programs relatively easier than locks.

With TM, the burden of providing atomicity and wait-free properties falls on the system designer. The underlying implementation for a TM system can be either provided by software [45, 15, 24, 40, 10, 54, 42, 39] using special data structures or by hardware [14, 12, 1, 31, 32, 33, 4, 43]. In this work, we consider hardware-supported transactional memory (HTM) systems only since they incur lower overheads and
provide better performance than software transactional memory systems (STM).

1.3 Problems with Hardware Transactional Memory

Performance of HTM is best under mild or medium contention, when the performance benefit from speculative execution with no conflicts exceeds the performance loss due to rollback on aborts [5, 12]. Aborts are wasteful because they not only waste cycles rolling back processor and memory state, they also increase the amount of traffic on the network. Every time a transaction restarts, it will reissue requests for all lines for which it needs ownership, to the memory system. Conceptually, the memory traffic that may be produced by repeated conflicts among transactions is equal to or worse than the traffic produced by using locks based on test-and-set atomic instructions, which performs repeated writes to shared memory to get the lock. This problem will only get worse with an increasing number of cores on multi-core systems. With increasing cores the number of transactions concurrently executing also increases and so does the probability of conflicts.
1.4 Memory-level Value Predictor

As one solution for reducing conflicts in a TM system, we present a Memory-level Value Predictor (VP-TM) that can reduce conflicts for certain applications. We observe that parallel programs often need to maintain a queue or a linked-list of elements for various activities like scheduling tasks, keeping records or allocating memory to different processes. Operations on such structures mainly involve insertions, searching and deletions of elements. These operations are often difficult to parallelize, even if they are carried out on different elements, because of the bookkeeping associated with each operation like incrementing/decrementing the shared head or tail pointer, updating the queue size, etc. Due to this, concurrent transactions conflict and queue operations can get serialized.

1.5 Value-Predicted Transactional Memory

In this research, we closely observe the shared data involved in bookkeeping and find that it is often updated by each transaction in a predictable manner. We study this predictability of shared conflicting data in various benchmarks and explore the idea of having a value predictor in memory to learn and predict shared data that can be used by conflicting transactions to run in parallel. As commits occur, the predictor validates each predicted value in order and instructs each transaction to continue or abort. With correct predictions, we can run multiple transactions in parallel with very little communication, improving speedup. Wrongly predicted values lead to aborts and utilize the rollback mechanism already present in TM systems to restart the critical section. We evaluate our proposal with the help of micro-benchmarks. Our initial results look promising: Overall performance of the TM system improves with the addition of the value predictor. We get significant speedups by using a value-predictor-based TM model over the base TM model. Also, mispredictions are not very expensive and do not degrade performance significantly.
1.6 Hardware Complexity of VP-TM

We also look into the issue of hardware complexity of the Value Predictor. The VP-TM system adds states and extra messages for prediction and verification, and changes at each processor subsystem (transaction state and caches), compared to TM. Also, speculatively communicating data within transactions can create a commit order among transactions. Transactions that have received uncommitted values (or predicted values) become “dependent” on the owner of the data and cannot commit until the owner commits. This can lead to cyclic dependencies which need to be detected to avoid deadlocks. To detect such cycles, the superset of all the processors that have received or forwarded data speculatively within a transaction must be available to all processors. We use a directory-based system along with a centralized value predictor so that all the cycles that can arise due to predictions can be detected to avoid a deadlock. Once a cycle is detected in a VP-TM system, at least one of the transactions needs to be rolled back to avoid a deadlock. Dependent transactions not only increase the hardware requirements, but also reduce the performance gains that can be obtained from the increased concurrency in VP-TM. Further complications can arise when a processor in this “chain” of dependent processors aborts. In this case, the chain needs to be updated to avoid starving the dependent transactions.

The above issues indicate that early communication of transactional data either by forwarding or prediction needs extensive additional hardware support in the form of changes to the coherence protocol, additional buffer space, bits to keep track of forwarded values etc. and changes to the deadlock detection mechanisms. Even with all the additional hardware, performance of these techniques can still be limited due to the additional deadlock cases that can arise. Thus, we may not be able to extract good performance with VP-TM but we definitely end up with significantly increased hardware because of the centralized value predictor to order transactions and the extra mechanisms to detect cycles.
1.7 Limited Early Value Communication

We believe that the hardware requirements for TM systems already are high enough and, therefore, we try to minimize the extra hardware that may be needed by VP-TM. We specifically want to avoid adding states to the coherence protocol since that increases verification time and, ultimately the cost of TM hardware. In this research, we take a step back and analyze the behavior of TM benchmarks from the STAMP suite [28] to find the number of concurrently stalled sharers of data. Our results indicate that the number of transactions simultaneously accessing or conflicting on shared addresses is usually low. We propose an alternative scheme, limited early value communication, that allows uncommitted transactional values to be forwarded in a limited manner without any changes in the coherence protocol and with minimal extra hardware at each processor to maintain transactional correctness and consistency while providing better performance than the base LogTM system under high contention. We reduce the hardware in the following ways compared to the previous approaches:

- We avoid the need to maintain a list of dependents and have to broadcast it like DATM or have a centralized hardware structure like VP-TM, to detect cycles. This also simplifies the memory directory.
- We avoid using extra states/changes in the coherence protocol to indicate latest “speculative owner” of the data (VP-TM) or to forward data to a concurrent sharer (DATM) of uncommitted data.

1.8 Realistic Value Predictor Design

Finally, we simulate a more realistic design for a VP-TM system using an extra virtual network dedicated to value predictor messages. We analyze the impact of these messages on overall system scalability and performance. We present optimizations that can reduce the communication impact of value predictor messages. We
then present speedup numbers for 16 processors and show that our VP-TM system with realistic value predictor still outperforms the base LogTM system for many applications. We point out the type of applications that we believe will benefit from using VP-TM as well as applications we believe will not see benefits from VP-TM. This is an effort to provide a final confirmation of our earlier results that show that adding a value predictor can benefit many applications by eliminating conflicts and improving scalability.

This research makes the following contributions:

- We discuss the issue of scalability for TM systems for large (16 or more cores) systems. We present data confirming that performance on TM systems can suffer due to excessive conflicts on such systems.

- We show that there often exists predictability in conflicting data that is written by different transactions, and that behavior can be exploited to improve performance and scalability of TM programs.

- We present one possible design of the value predictor to exploit this predictability of conflicting data and highlight the issues that can arise during the design of a TM system with the value predictor.

- We present results using a functional implementation to show that opportunities to exploit predictable values exists and can be used to improve performance over base TM systems.

- Next we present a limited value forwarding system that can reduce the hardware complexity of the above design.

- Finally, we present a realistic design of a VP-TM system and show the performance differences compared to the earlier functional implementation.

The rest of the paper is organized as follows: We discuss transactions in detail in the next chapter along with some related work in section 3. In chapter 4.5 we put
forward our case for having a value predictor. In chapter 5 we present one possible design of such an enhanced TM system. Section 5.4 presents our initial simulation results. Next, we present our limited forwarding approach. Chapter 7 discusses a realistic design of the VP-TM system. Finally, we present conclusions and possible future work directions that can be taken from this study.
Chapter 2

Conventional Synchronization and Transactional Memory

2.1 The problem of Synchronization

Shared Memory parallel programming uses concurrently executing threads that use shared data structures for communication and coordination. Parallel threads communicating via shared data structures can give rise to different execution orders. If these accesses are "conflicting"\(^1\), overlapping execution orders can produce incorrect, non-deterministic or undesirable results as shown in Figure 2.1. To avoid such results, parallel programmers orchestrate synchronization using shared variables that order concurrent events and avoid conflicts on other shared variables. Figure 2.2 shows how shared variables can be used to provide synchronization between two threads and avoid overlapping execution orders.

\(^1\)Conflicting accesses are accesses made to the same memory location such that there is at least one access which is a write [44].
Figure 2.1: Figure showing how conflicting reads and writes can overlap and produce incorrect results. In (b), we see that order of writes and reads overlaps so that even after both threads have incremented a, the final value of a in memory is 1.

Figure 2.2: Figure showing how flags can be used to coordinate access to shared data within critical sections in yellow.

2.1.1 Synchronization using Atomic instructions

Using shared variables does not provide adequate synchronization and can lead to indefinite waiting as shown in figure 2.3. The above problem arises because coordination of parallel threads relies on two operations, a read and a write to the shared location, and these operations are not atomically executed by the shared memory machines, leading to overlap. The conventional approach to this problem taken by most
Figure 2.3: Figure showing the sequence of instructions that can result in indefinite stalls for both threads.

current shared memory machines is to provide atomic read-test-update instructions that can be used to read, test and update a single memory location atomically. These instructions allow the programmer to express coordination directly in terms of memory operations and avoid overlap during parallel execution. Besides event ordering, synchronization can also be used to provide “mutual exclusion” during critical sections. Mutual exclusion for critical sections allows programmers to perform complex updates to shared data in a serialized manner beyond that which can be provided by simple atomic instructions.

Synchronization constructs can be of two types, depending on the behavior of threads that are waiting to access critical sections: busy-waiting and blocking. Busy-wait constructs cause idle threads to execute a loop that repeatedly tests whether the critical section can be accessed. Blocking constructs cause idle threads to go to sleep to avoid repeated memory operations. This thread is then awakened \(^2\) once the critical section is available. In this work, we focus on busy-wait constructs since blocking constructs are used mainly at the operating system level to synchronize or schedule processes. Mellor-Crummey and Scott [27] provide a good comparison of different busy-wait techniques and point out performance benefits and issues with

\(^2\)The thread is awakened by another thread as part of synchronization. At the OS level, blocked threads are awakened by the OS scheduler.
Thread 1
LOCK (X);
a = a + 1
UNLOCK (X);

Thread 2
LOCK(Y);
b = b + 1;
UNLOCK(Y);

Thread 3
LOCK(X)
a = a + 1;
LOCK(Y);
b = b + 1;
UNLOCK(Y);
UNLOCK(X);

Figure 2.4: Figure showing how locks are used in shared memory parallel programs.

Thread libraries provide higher-level synchronization primitives that have richer semantics than simple atomic instructions and hide many lower-level details from the programmers, improving compatibility of parallel programs across various architectures. In this work, we focus on only lock primitives that are used to provide mutual exclusion. An example of how to use locks is as shown in Figure 2.4. Lock variable X is used to read or write shared variable a and lock Y is used for b. All accesses to a and b have to be performed by acquiring locks X and Y respectively.

2.1.2 Problems with Locking

While locks can provide a uniform interface for parallel programs across different architectures, they can make parallel programming hard. One difficulty is the assignment of locks to shared structures in a parallel program. Since a lock implies serialization, our goal is to minimize the granularity of locking i.e. cover as small sections of data with a single lock as possible to expose greater concurrency. However, there are problems with that approach. Fine-grained locking can increase the instances of locking in the program. From the hardware point of view, lock-acquire latencies are exposed during synchronization and they limit the concurrency that can be achieved by fine-grained synchronization. Also, certain locking schemes like
Figure 2.5: (a) shows data races with incorrect use of locks. (1) shows lock x used for variable a. However, in (2), a is not inside critical section. In (3), the wrong lock is used to gain access to the variable a. (b) shows how nested locks can lead to a deadlock. In this case, each thread holds a lock and requests for another leading to a deadlock.

test-and-set [27, 2] can generate excessive coherence traffic during synchronization reducing the bandwidth available to other threads.

Using fine-grained locks in software can increase the number of lock variables to track and this increases the programing and debugging time. This also implies greater chance of “race conditions” [44, 17]. A race condition is a scenario that occurs because of inadequate synchronization which can expose a shared memory location to conflicting threads leading to unpredictable results, as shown in Figure 2.5a. Race conditions are hard to detect since data races may not get exposed under all scenarios. Fine-grained synchronization can also lead to nested locks. Nesting can lead to deadlock/livelock conditions. A deadlock is a scenario in parallel computation when two or more threads are waiting on a synchronization variable and there is no forward progress made, as shown in Figure 2.5b. A livelock scenario is similar to a deadlock, except that parallel threads are in a cycle requesting synchronization access repeatedly. Livelocks can be harder to detect than deadlocks.
There are also several reliability problems associated with locks due to their software-waiting nature. During parallel execution, gaining a lock implies control of a shared resource which can be anything: for example, a memory page that has been allocated to the thread, or an I/O device. All other threads will wait until this thread releases the shared resource via unlock. If the lock owner aborts, it may not give up its ownership of shared resources which implies that all other threads needing access will continue to wait forever. It is also hard to assign priorities to threads using locks. A lower priority thread can take control of resources using threads and make a higher-priority thread wait until it gives up the lock. The scenario worsens if the lock owning thread is de-scheduled. Other threads that are currently scheduled simply have to wait for the thread that owns the lock to be scheduled again and free the lock. This problem is often referred to as priority inversion [18].

The goal of synchronization is to provide correct program execution for all execution orders while exposing as much concurrency as possible. Synchronization performance can have a huge impact over scalability of parallel programs since synchronization directly decides the amount of concurrency in the program. With locks, programmers need to not only carefully assign locks to shared data structures to expose enough concurrency for good performance, they also need to orchestrate the entire process of synchronization across the entire program for correctness while avoiding problems like race conditions, livelocks, deadlocks and priority inversion. Because of this, parallel programmers often choose to use coarse-grained locks sacrificing performance for ease of programming, reliability and correctness.

### 2.2 Transactional Memory

While parallel programming using shared memory has been a difficult task for programmers, databases have been able to exploit tremendous concurrency by executing simultaneous queries as “transactions”. A transaction is a sequence of actions that appear instantaneous and indivisible to an outside observer. A transaction has
four specific attributes of atomicity, consistency, isolation and durability and it is the responsibility of the underlying transaction processing monitor to provide these guarantees. The advantage of using transactions for concurrent programming is that transactions free the database programmer from trying to orchestrate parallelism and he only needs to reason about results that are final values of transactions. The idea of using transactions for shared memory programming was proposed by Herlihy et al. [14]. In their work, Herlihy et al. propose “Transactional Memory” as a memory subsystem that provides transactional guarantees. Transactional Memory (TM) maintains the programmer’s model of critical sections and it avoids the need to assign synchronization variables to shared data structures and provides a simpler programming interface to use for synchronization. TM can provide minimal synchronization overhead for minimal to medium contention and uses hardware or software to convert conservative locks into optimistic transactions that dynamically preserve atomicity. There are a number of programming and performance benefits of TM systems compared to traditional lock-based schemes.

- TM systems are easier to program since the programmer does not have to deal with shared locks and their assignment to shared variables in the program.

- Since TM is optimistic and avoids arbitrating for a lock, mild to medium contended critical sections perform better with TM compared to traditional lock schemes.

- TM is not as sensitive to coarse-grained synchronization as locks. TM with fine-grained conflict detection has the potential for good performance even with large transaction sizes.

Since Herlihy’s paper, there have been a number of proposals that revolve around their idea of providing a “transactional memory” abstraction to the programmer supported either by hardware or software. In all cases, providing the TM abstraction needs the following:
• To support isolation and optimistic wait-free execution of transactions, we need a mechanism to log reads and writes performed by individual transactions.

• To maintain atomicity of transactions, we need a mechanism to detect whether there are conflicting accesses to a memory location within the log.

• To provide consistency, we need a mechanism to be able to roll back execution for any transactions that violate the atomicity principle detected using the above mechanism.

In this work, we look mainly at hardware-supported TM (HTM) systems since they provide better performance and demand less effort from the programmer compared to STM systems.

2.3 Hardware-supported TM

HTM systems use hardware to perform the two important functions of Conflict Detection and Version Management. Transaction begin and end is detected by special instructions supported by the ISA. Conflict detection is provided by minor modifications to the underlying coherence protocol. Whenever a processor tries to read or update a memory location that has been updated by another processor currently in a transaction, the coherence protocol signals a “conflict”. To manage different versions of transactional data, the hardware must provide a separate buffer or a log in memory and a mechanism to switch versions on abort or a commit. Standard modifications to the underlying hardware to support TM are:

1. Special instructions to mark the beginning and the end of the critical section as a transaction or a way to distinguish transactional read and write accesses.

2. A history buffer that supports rollback or checkpoint capability to restart the processor from the beginning of the transaction.

3. A buffer to hold multiple versions of transactional data.
4. Modifications in the cache coherence protocol to detect and resolve conflicts.

Upon entering a critical section labeled as a transaction, an HTM system saves processor state and initializes the transactional buffer or log. Loads and stores inside the critical section are marked as “transactional” and either the oldest or the latest version of transactional data is stored to the buffer. When execution reaches the end of the critical section, it signals a commit, at which point the data written by the transaction is “committed” to memory and the transaction ends. A conflicting access signaled by the coherence protocol will either stall or abort the transaction.

Conflicts can be signaled due to reads or writes. Conflicts due to reads from other processors can be resolved by deferring the read request or by sending negative acknowledgments until the transaction is committed. However, write conflicts that appear due to invalidations sent by the writer will cause either all the sharers or the writer to abort the ongoing transaction and restart the critical section. Restarting transactions involves rolling back processor state to re-execute the critical section. The memory state also needs to be rolled back to maintain atomicity and consistency of transactional data.

We now discuss some of the important TM proposals that have been put forth. Our goal is to highlight the problems that these schemes try to tackle and the key ideas put forth. We discuss the schemes along the following dimensions: transaction management (how is transactional execution triggered), speculative data management (including overflow) and conflict management.
Chapter 3

Related Work

3.1 Transaction Design Space

Implementational differences in HTM systems can have a huge impact on performance of individual applications. The differences in performance between various implementations arise because of the differences in transaction sizes and conflict behavior across various applications. Overall, there is no single implementation that outperforms all other implementations on all applications. Each system has a set of goals in addition to improving performance. For example, in the Transaction Coherence and Consistency paper [12], the authors state that a system providing delayed conflict detection and commit can not only reduce the memory bandwidth of the system, but it can automatically provide sequential consistency model without the need of fences. Rajwar et al. [35] and Ananian et al. [1] state their main goal as to try to accommodate arbitrarily large transactions without significant performance penalty.

In this chapter, we study some of these systems to gain insight and to be able to compare and better evaluate our system against existing proposals. We begin with a discussion of the basic conditions that need to be fulfilled to provide abstraction of transaction.

As stated in Chapter 2, there are four attributes of database transactions that
must be provided by any underlying transaction processing system: atomicity, consistency, isolation and durability. In parallel programming transactions need a TM system to provide only the properties of atomicity, consistency and isolation \(^1\). For atomicity, a TM system must be able to provide instantaneous commit and abort of transactional data. For isolation, a TM system must be able to provide a means to store transactional data locally and make it visible only after a commit. For consistency, the system must provide atomicity and isolation of transactional data and be able to discard modifications to data on an abort as if the transaction did not execute at all with respect to memory.

### 3.1.1 Classification of HTM systems

To satisfy the above requirements, HTM systems add the following functions to a shared memory multiprocessor system - Version Management and Conflict Management [5]. Version management has the following responsibilities

- Label data that has been accessed or produced by the transaction.
- Save the data that is overwritten by transactional stores.

Version management systems can be “eager” or “lazy” depending on whether new values that are produced during transaction are propagated to the memory system or not [5]. The benefits of eager update is that it reduces the latency of the commit operation. Aborts on the other hand are slowed down because transactional data needs to be rolled back to the state before the transaction. This can improve performance if most transactions are committed than aborted. An eager version management requires modifications to the cache lines and the coherence protocol since there is new data in the cache/memory that needs to be isolated from other threads until transaction commit. This may increase the complexity of the cache subsystem and the coherence protocol. The LogTM system [31] is an example of an eager version

\(^1\)Durability is not needed because parallel programming deals with memory and not disks.
management system. We study the LogTM system since it is the base system on which our scheme has been developed.

Conflict management deals with dynamic detection of conflicting reads and writes during transactions on different threads and how those conflicts are resolved. For HTM systems, conflict management utilizes version management and the coherence protocol already present in shared memory multiprocessor machines to detect conflicting accesses during transactions. Conflicts can be detected either directly when the conflicting access is performed or lazily at the end when the transaction commits. A lazy conflict detection mechanism has the disadvantage of unnecessary execution in presence of conflicts. However, it also has the potential to reduce the number of aborts since it avoids repeated conflicts. The Transactional Coherence and Consistency system [12] is an example of a system that uses lazy conflict detection. We discuss the system in greater detail in section 3.2.3. In the next section we study some HTM systems. For each system, we discuss a feature of the system that is important to understand the evolution of TM.

3.2 Evolution of HTM systems

3.2.1 Herlihy’s proposal

As stated in the earlier chapters, TM was first proposed by Herlihy [14] et al. The paper provides special instructions to indicate transactional execution and special “commit” and “abort” instructions to commit or abort transactions. For speculative data management purposes, the paper uses a separate transactional cache to hold both, the speculative, and the most recent committed version of the transactional data. Transactional memory instructions bring data into the transactional cache and no extra transactional read/write bits are needed. Upon commit, the speculative data is committed and the older version is invalidated. A transaction abort invalidates speculative data using “abort” instruction. Interrupts and transactional cache
overflows abort transactions.

Conflict management is done as follows. Conflicts are detected when a read or write access from one transaction conflicts with another ongoing transaction. Upon a conflict, the entries in the transactional cache are invalidated. The transaction continues to execute until commit. The commit instruction indicates a failure in such a case and the programmer can use this information to retry the critical section. To minimize deadlocks or livelocks, aborted transactions use software-level back-off techniques to delay retry and also avoid congesting the network in case of frequent aborts. The paper suggests using queuing techniques in the rare case of a deadlock. Overall, the paper provided a technique using which the programmer can define customized read-modify-write operations for arbitrary words in memory, using a special cache, without the drawbacks of conventional locking schemes.

3.2.2 Transactional Lock Removal

The next paper on Transactional Memory by Rajwar et al [37] is also an HTM design. It builds upon their previous work of speculative execution using lock elision (SLE) [36] by adding “timestamps” to improve concurrency in presence of data conflicts. Transactions are identified from critical sections using the same mechanism as SLE. The major contribution from this paper is the idea of forcing transactions to wait/stall based on timestamps, instead of always aborting on a conflict. Stalling transactions is implemented using a deferral-based coherence protocol that delays handling of certain coherence requests until transaction commit. Timestamps are generated locally, associated with a transaction at the start and used during conflict resolution to avoid deadlocks - earlier timestamp implies higher priority. In presence of a conflict, the requesting transaction is chosen as the victim to be stalled or aborted. The decision to stall or abort depends on the type of conflict and also on whether stalling the transaction might lead to a deadlock situation with other pending transactions. In general, a contender with an earlier timestamp wins the conflict. Aborted transactions use a processor checkpoint to rollback to the beginning of the
critical section and reuse the original timestamp to avoid starvation.

Stalling some transactions instead of aborting avoids the overhead of restart and reduces network traffic associated with reissuing shared memory requests within the transaction. The deadlock detection policy based on timestamps guarantees no deadlocks and that all transactions will eventually finish. Speculative data is managed by locally buffering speculative updates and keeping the old committed versions around until transaction commit. In case of a buffer overflow or I/O requests that cannot be undone, the execution reverts back to locks as in SLE.

3.2.3 Transactional Coherence and Consistency - TCC

This paper presents a new parallel programming paradigm based on purely transactional execution. A parallel program is now a series of transactions that can execute in parallel in absence of conflicts. With pure transactional execution, the main objective of this work is to avoid complicated hardware that is required for maintaining coherence and consistency for each and every memory operation that is issued by different processors. The begin and end of transactions are indicated by special programming constructs provided by the paper. At the beginning of a new transaction it is assumed that the processor takes a checkpoint to enable execution rollback.

During transactional execution, each processor buffers data locally to maintain atomicity and the processor caches do not need to maintain coherence for shared data until transaction commit. Upon commit, the write set (the set of addresses written to within the transaction) of the transaction is broadcast to all other processors. Any processor that has read from or has written to this set of addresses in the current transaction (and has not yet committed) detects a conflict and rolls back execution to the beginning of the transaction. A transaction that has already begun the commit procedure always commits. Also, commits are serialized since the write and read sets of transactions are local until commit. Thus, a transaction commit stalls all other transactions that are ready to commit. This unnecessary serialization, along with the broadcast of write set at commit, seem to be the biggest challenges to scalability.
for this scheme. Transactions can also have a “commit order” that can be used to parallelize serial codes, for ex. loops in serial programs. In this case, transactions that have higher phase number are not allowed to commit before a transaction that has a lower phase number. Buffer overflow of speculative data is not discussed in this paper.

3.2.4 Unbounded/Large Transactional Memory

This paper highlighted one of the biggest problems with HTM which the previous papers have so far neglected, namely, transactional data overflow. Transactional data maintained in speculative buffers or caches can overflow and lead to following problems:

1. The data that has overflowed cannot be written back to memory or invalidated until transaction commit for correctness purposes.

2. There can be no guarantees made about size of transactions and programmers cannot be expected to write transactions depending on the size of the speculative buffers.

3. Overflowed addresses need to be accessed frequently for conflict detection. This implies that the overflow locations should be quickly accessible in the midst of a transaction.

Besides transactional data overflow, transaction execution could also overflow beyond its time quantum leading to a context switch in the middle of the transaction. This paper advances the following thesis: Transactional Memory support should be “virtualized” to support transactions of arbitrary size and duration. This means that any form of overflow should be intercepted and dealt by the hardware/software without programmer interference. There should be no limitations on the size or duration of transactions in the model exposed to the programmer. One important performance
goal of such a model is also to make sure the performance of the HTM system does not degrade for small transactions which is the common case.

The UTM paper proposes to maintain bookkeeping information about transactions in a memory-resident data structure called the transaction log that uses global virtual addresses. This avoids the problem of transactional data overflow. To enable transactions to survive context switches, the authors add a pointer to this log as part of the process/thread state saved during a context switch. This makes it easier to suspend the transaction and even migrate it to a different processor. The cost of these techniques is very high, both in terms of performance and area. With the transaction log, every transactional load and store must access multiple memory locations for purposes of log-update, conflict detection or storing old committed versions. In their work, the authors chose to not use checkpointing and instead extend the ROB to handle rolling back to a program counter that has already been committed.

\section{3.3 Software Transactional Memory}

So far we have only looked at hardware transactional memory implementations. Transactional memory can also be implemented by software using specialized libraries that provide transactional atomicity \cite{45, 15, 24, 40, 10, 54, 42, 39}. Software transactional memory (STM) has a few advantages over hardware-only transactional memory \cite{10, 54}:

1. Maintaining transaction state in software has the advantage of offering flexibility in choosing various algorithms for speculative data management and conflict management. Most HTMs can offer only a single scheme for each of the above cases.

2. STMs can easily be used with larger or long running transactions because there is no fixed hardware log structure. Logging is done in memory in STM.
3. STMs make it easier to handle exceptions or faulty conditions during transactions.

3.3.1 Shavit’s STM system

We briefly describe software transactional memory. The first proposal for STM by Shavit et al. [45] could be used only for critical sections that accessed memory addresses which could be statically determined. Upon entering a critical section, a thread requests ownership for all addresses that will be accessed by the transaction, one by one in increasing order and retains them. If it succeeds in obtaining ownership for all addresses, the memory is updated by this transaction and upon exit the threads releases ownership for all addresses used in the critical section. If a thread cannot obtain ownership for any address that it requested, it releases all previously acquired ownerships for the current critical section to avoid deadlocks. Ownership for addresses are maintained in a single transaction table and the per-address status is changed atomically using load-locked and store-conditional operations. Non-blocking is achieved by helping other threads in case of failed ownerships for addresses. Although such a software-based TM model does not perform as well as an HTM system, it offers tremendous flexibility in terms of portability and resilience in the face of faults.

3.3.2 Dynamic STM

Another STM system worth describing is the Dynamic STM (DSTM) system by Herlihy et al. [15]. Instead of requesting permissions for statically known addresses before executing the critical section (or transaction), DSTM works by requesting permissions for addresses (or objects) as they are accessed inside the critical section. If a transaction successfully gets required permissions, DSTM creates a copy of the object and only updates the copy inside the critical section. When the end of the critical section is reached, DSTM atomically changes the status of all transactional
addresses (or objects) indicating a “commit” and the newer version (copy) for all addresses becomes the committed version. If the transaction cannot commit due to a conflict, the status of the transaction is atomically changed to “aborted” for all objects and the older version for all those objects becomes the last committed version.

3.3.3 Hybrid TM

We do not study STM further because HTM performs better than STM due to the latencies involved with conflict detection and logging in software in STM [25]. There is another approach that has been taken by researchers in an effort to get the performance benefits of HTM along with the flexibility of STM systems. Such systems are also called Hybrid TM systems [9, 46, 29, 11, 30, 51, 3]. Hybrid TM systems can also be described as best-effort TM systems whereby the hardware provides TM support for the common case. In presence of problems like system calls or I/O within transactions or buffer overflow during a transaction, the software layer provides TM support. This approach has also been taken by the Rock TM [30] implementation. Issues that need to be considered in Hybrid systems include but are not limited to

- There needs to be a trigger mechanism to switch to software. Typically, begin and end transaction primitives need to detect whether the transaction will be supported by hardware or software.
- The system must take into account that transactions can co-exist in hardware and software. This means the hardware log and the transactional read/write bits must be exposed to the software such that it can detect conflicts between transactions in hardware and software.

3.4 Value Predictors

In this section, we discuss value predictors since we will be motivating the use of a value predictor to eliminate conflicts in TM. Value predictors have been pro-
posed to tolerate cache miss latencies for loads in uniprocessors [21, 20, 22, 19, 6].

A value predictor predicts values that will be bought in by a load instruction from memory, allowing dependent instructions to speculatively proceed with the predicted value. Value prediction is based on locality of values written by store instructions to integer and floating-point registers. This can be considered a logical extension of branch prediction where the predictor predicts a single bit that indicates the direction a branch is going to take. The prediction mechanism works similar to branch prediction. A trace of old values is used to extrapolate and provide a predicted value to an outstanding load that has suffered a cache miss. There are two main techniques for extrapolating predictions from older values: computation-based (using the stride between recent store values) or context-based (identifying a pattern of reuse via correlation) [41, 13, 19, 50].

The value predictor is table-based and is often indexed by the program counter of load instruction being predicted. It is located near the micro-architecture since predicted values need to be provided to dependent instructions which are often close behind the load miss. Value Prediction is speculative so it needs a rollback mechanism in cases where prediction is incorrect. This can be accomplished by preventing any instructions that have used speculative values to commit architecturally until the prediction can be verified. If prediction fails, the instructions are squashed in the pipeline and the program counter is restored to the first dependent instruction that consumed the predicted value.

In our work, the value predictor is located at the memory because it needs to see messages coming from different processors. Similar to the uniprocessor case, the value predictor we use is a table-based and is indexed by the load-address. We use a stride-based scheme to generate predicted values because we are able to identify a stride pattern in conflicting data. We explain this in detail in the next chapter.
3.5 Thread-level Speculation

We now take a look at related work in the area of thread-level speculation (TLS). TLS is very similar to HTM in that both require the programmer to mark regions of code that will be executed concurrently by the hardware as long as there are no conflicts [47, 53, 7, 8, 48]. The goal of TLS is to be able to concurrently execute portions of sequential code, marked by the programmer, in absence of conflicts. TLS differs from TM in that it is used for sequential codes where programmers only identify independent tasks, whereas TM is used in programs that have been explicitly parallelized by the programmer and transactions have been identified as regions within parallel threads that need atomic execution. There is an ordering among speculative threads that has to be maintained by TLS because threads are spawned from sequential codes. Concurrent speculative threads in TLS only need to be executed in parallel whereas transactions in TM also need to be atomic in nature. This means that transactional data needs to be isolated from other transactions, but threads in TLS can expose their writes to more speculative threads. This is an extra requirement for any hardware or software that implements TLS as compared to TM, where transactions do not have an implicit order and can commit in any order in absence of conflicts.

The main reason for discussing TLS as related work is to be able to introduce an approach of using value prediction for TLS which is similar to our work. Steffan et al. [48] discuss the potential of using value predictors to improve the performance of TLS. In their work, the authors study different schemes to communicate values between speculative threads in TLS. The paper proposes a scheme whereby a more speculative thread can receive predicted values based on stores from less speculative threads. The authors show that value prediction with some throttling can be successfully used to communicate values between speculative threads and improve concurrency in TLS. They also show that mis-predictions can have a high cost in TLS. One difference between using value prediction in TLS and value prediction for TM is that in TM value prediction introduces extra ordering within threads based on
predicted values, whereas such an ordering is already present in TLS. This ordering implies that adding a value predictor to TLS is simpler than adding one to TM.

We turn our attention to some issues with HTM systems and look at scalability of TM in the next chapter.
Chapter 4

Motivation for our Work

4.1 Issues with TM

As stated, performance of HTM systems often exceeds performance of lock-based synchronization, for applications where synchronization contention is not severe and the fine-grained conflict detection provided by the coherence protocol avoids unnecessary conflicts. If a transaction can commit without any conflicts, it has reduced the number of shared accesses and possibly avoided spinning on the lock. However, for critical sections that are heavily contended, there is a performance loss from repeatedly retrying the critical section because:

1. The processor needs to rollback its state to the beginning of the transaction and that can take up some cycles. This may mean copying all register values from a checkpoint or simply switching to a different rename map table.

2. The memory state needs to be rolled back. If the newer version of transactional data has been privately stored, then this is easier, since we simply need to invalidate all speculative data produced by this transaction (and possibly clear some state in the caches). Otherwise, if the newer version has been propagated to the memory, then the memory rollback can be long depending, on the size of the transaction.
3. Each time the processor rolls back and restarts a critical section, it re-issues all shared requests to the memory system. Repeated retries may increase the traffic on the memory system and congest the interconnection network or the bus. They can also increase the coherence traffic since the requests can change ownership of shared data.

Different conflict management schemes handle conflicts differently, but in the presence of conflicting accesses they eventually only allow only a single winner to proceed. Eager conflict detection schemes try to detect conflicts as they happen, to improve concurrency, but sometimes they may lead to more than one conflict detected between two transactions for the same pair of conflicting accesses. This is because when a conflict is detected all but one transactions are forced to restart. The restarting transaction may issue the same conflicting access again before the winner commits and the same conflicting access will again be detected as a conflict. The transaction that “lost” during the conflict resolution process has no way of knowing whether the winning transaction has committed or not before retrying access. This problem is referred to as “Restart Convoy” by Bobba et al. [5]. Lazy conflict detection schemes may avoid such repeated conflicts, but they reduce concurrency by delaying conflict detection until commit.

TLR [37] and Log-TM [31] systems use deferred requests or Nacks in the coherence protocol to convert most of the write-read conflicts to stalls. This avoids having to abort and restart some transactions. However, the above two schemes also make the TM system prone to deadlocks, and thus Nacks/Stalls cannot always be applied. By using timestamps and recording Nacks to other processors, a potential for deadlock is detected and a Nack converts to an abort message. Some researchers have proposed back-off schemes [14, 31] as another way to alleviate this problem with TM systems. Back-off schemes can help reduce the number of “extraneous” conflicts, but they also reduce concurrency for TM systems due to the back-off times increasing exponentially over time.
A practical issue can arise during the design of an HTM system: since conflict management (which can have a tremendous impact on performance) is provided by modifying the underlying coherence protocol, it is permanent. This means that the designers would like to choose a conflict management scheme that works best for most applications. However, Bobba [5] in their work showed that no single conflict/data management scheme works for all applications, and a scheme that works well for some communication patterns may show poor performance for others. Any ideas that favor a certain class of applications over others may not prove profitable. There needs to be a conflict management scheme that can provide good performance for a majority of applications, while at a minimum not degrading performance for the rest.

4.2 Scalability of applications with TM

As stated earlier, transactional execution to provide synchronization of parallel programs has been inspired by transactions in database systems. There exists tremendous amounts of parallelism in databases. Theoretically, a database can support as many concurrent operations as the number of unique records present. Therefore, for databases, the purpose of transactions is to provide an abstraction for programmers such that they do not have to worry about the underlying details of the system or the amount of concurrency available. For shared memory parallel programs, TM is attractive because it makes programming easier, and it has the potential to perform better than conservative locks due to a dynamic fine-grained conflict detection mechanism that can be enabled by the underlying implementations. Thus, optimistic execution of critical sections using transactions is well rewarded in the realm of multiprocessors with 16 - 32 processors. In this work, we are mainly concerned with the scalability of programs that are written using the transactional memory paradigm. Given that shared memory parallel programs do not offer as much concurrency as databases and conflicts can have an adverse effect on performance, we believe that many programs using transactions will not scale because concurrency exposed using transactions will
be negated by the adverse impact of conflicts. We verify this claim in the next section.

4.2.1 Evaluating the Scalability problem

To evaluate the effect of TM on scalability, we measure conflicts with an increasing number of processors. To reduce the amount of simulation time, we use a modified version of the “tourmaline” transactional memory simulator present in the GEMS suite [26]. Tourmaline is a functional-only HTM simulator which provides the basic TM functions of data-logging, checkpoint creation at the beginning of a transaction, conflict detection and processor rollback on a conflict. The interconnection network is not modeled. All operations take a single cycle to complete and conflicts always lead to aborts. In case of excessive aborting, transactions are serialized to guarantee forward progress. The following changes were made to this basic model:

1. Added a timing model in the form of a 16KB data cache with read/write, hit and miss latencies as 10 and 200 cycles, respectively. All other operations, including instruction fetches, take 1 cycle.

2. Added “timestamps” to transactions. Each transaction gets a timestamp at the beginning which it maintains until it commits. This is used to decide the aborting processor on a conflict.

3. To avoid an inflated number of conflicts due to aborting and restarting on every conflict, we added a simple exponential back-off strategy of doubling the wait time after each abort to avoid excessive conflicts.

4. Added a rollback penalty of 40 cycles to aborting transactions. This is to account for the checkpoint and memory restoration after abort.

We measure conflicts and speedups for two versions of the queue microbenchmark with an increasing number of processors. One version uses large transactions (10k instructions per transaction) and the other version uses medium-sized transactions
(up to 1K instructions per transaction). This is done mainly to see the effect of increased transaction sizes on conflicts. We also used two benchmarks from the Splash suite [52], Cholesky and Raytrace. These benchmarks were chosen mainly because of their synchronization characteristics. Both contain large number of critical sections and synchronization is a major bottleneck for scalability [52]. The average size of a transaction for Cholesky is around 150 instructions and for Raytrace, around 600 instructions. We are using a smaller data set size for Cholesky than the earlier simulations due to limited time. This has an adverse effect on the scalability of the benchmark as shown in results.

![Graphs showing scalability](image)

Figure 4.1: Figure showing scalability of the Queue microbenchmark under medium and heavy contention.

One important thing to note is that the Tourmaline-based model does not have the “NACK” capability of the Log-TM model. This means that a conflict between two processors always aborts one of them, which may not be the case with the Log-TM model. We believe this will not significantly change the number of conflicts that we wish to measure.
Figure 4.2: Figure showing scalability of the Cholesky and Raytrace benchmarks.

4.2.2 Analysis

From figures 4.1 and 4.2, we see that large transactions reduce the scalability of the benchmark. The number of conflicts increases exponentially with the number of processors, irrespective of the size of the transactions. This indicates that TM may not be scalable, especially under heavy contention. The Cholesky benchmark in Figure 6.2a shows similar performance as the microbenchmark with medium-sized transactions albeit, with fewer conflicts. This may be due to the fact that some critical sections inside Cholesky were not converted to transactions since they deal with malloc and free system calls and their transaction versions led to memory faults. The Raytrace benchmark is more scalable and resilient to conflicts up to 16 processors. Only at 32 processors do we see the effect of conflicts on Raytrace. Raytrace is not significantly affected by conflicts because it is not as synchronization-intensive as Cholesky. Raytrace consists of mostly unpredictable communication in the form of shared reads; since the network is not being modeled in tourmaline, the performance
scales up with the number of processors.

4.2.3 Problems with conflicts on large systems

Frequent Conflicts and restarting transactions can have an adverse effect on the network/memory system, increasing bandwidth use as well as communication latency. This problem is worse on larger systems, since there may be other applications running simultaneously on other processors that will be slowed down. Also, in a large system, the usefulness of Nacks/Stalls may reduce, since it will lead to more stalls, thus increasing the chances of deadlock. Hence, more transactions will have to be aborted eventually reducing concurrency. The problem of restart convoys also becomes more pronounced in larger systems, where there may be many transactions that can conflict on shared data and simultaneously restart, leading to a convoy.

Figure 4.3: Figure showing sequence of actions that leads to forwarding in DATM protocol. Memory location X is owned by CPU 1 which is currently in a transaction. A read access to X from CPU 2 can lead to CPU 1 forwarding the cache line. CPU 2 then forwards the cache line to CPU 3.
4.3 Dependence-Aware Transactional Memory

Dependence-aware TM(DATM) by Ramadan et al. [38] is the closest to our work and it also tries to tackle the problem of poor scalability of TM systems in the presence of conflicts. In this work, the authors try to eliminate conflicts by forwarding uncommitted data to a conflicting load while maintaining conflict-serializability. DATM allows conflicting transactions that are conflict-serializable to share data and commit safely instead of stalling or restarting, while providing the guarantees of atomicity and isolation like regular transactions. Figure 4.3 shows how transactions in DATM can forward uncommitted data to eliminate conflicts. If any transaction forwards uncommitted data, it creates a dependency. The transaction that received the forwarded value is now dependent on the producer and cannot commit until the producer has committed. DATM system tracks all dependencies at cache line granularity. New states are added to the cache coherence protocol to enable forwarding of data and to maintain multiple versions of the same cache line. Speculative writes are indicated by new cache line states and are stored in the cache. Write back of speculatively forwarded data is done based on an ordered list of transaction IDs maintained at each cache. The list indicates the dependence relationships between all transactions involved in early communication and subsequently determines their commit order.

DATM works well if there are no subsequent updates to cache lines that have been forwarded speculatively between transactions. Any future writes to such a cache line by the source transaction will result in restarting the dependent transaction to maintain atomicity and consistency. Our work does not suffer from this shortcoming of DATM since we only predict the final value of a memory location regardless of where the write happened during the transaction. We compare our work with DATM in detail in section 4.3 In the next section, we discuss how we solve the problem of conflicts by analyzing benchmarks and trying to find a pattern that can be exploited to improve performance in presence of conflicts.
### 4.4 Analysis of TM benchmarks

To analyze the problems with conflicts in TM systems, we decided to study data communication between transactions. We choose two benchmarks from the Splash [52] suite - Raytrace and Radiosity and some others from the STAMP [28] suite. For the Splash benchmarks, lock-based critical sections are converted to transactions simply by replacing lock/unlock primitives with begin and end transaction markers.\(^1\) We choose these two benchmarks mainly because they show contention under the TM model (as Nacks or Restarts) such that their performance is limited with increasing number of processors. The STAMP [28] suite already uses transactions and so we use all benchmarks from that suite except the genome benchmark. In both the above suites, we look for characteristics that are independent of the type of conflict management scheme used, such as read/write behavior of transactions, number of unique addresses that are written to during transactions, percentage of transactions that conflict etc.

We use an execution-driven simulator Simics [23] to simulate a 16 processor system driving the GEMS [26] memory model. It models timing for a Log-based TM system [31] complete with extra coherence states and messages needed for TM. Transaction “begin” and “end” are indicated using Simics magic instructions. Borrowing terminology from Bobba [5] et al., the LogTM model uses an eager version management scheme along with early conflict detection. Conflicts are detected by the cache coherence protocol and result in conflicting transaction getting a negative-acknowledgment (NACK) and stalling the conflicting transaction, forcing it to retry later. For more information refer to the LogTM paper [31] and section 5.2.

There are some limitations with our LogTM-based HTM implementation, in that it cannot handle function calls/system calls and exceptions/interrupts from within transactions. This has to do with the register windows feature used by the Ultra-SPARC architecture [49] during function calls that interferes with the checkpointing

\(^1\)Barriers and Pauses were not converted to transactions.
and restart mechanism used in the LogTM system. We avoided function calls inside transactions as much as possible by inlining to enable these benchmarks to run with TM. Even though there was a slight increase in the size of the final binary due to inlining, we do not expect this to change our analysis results significantly. The genome benchmark from STAMP has very large functions being called within transactions and could not be inlined and thus, failed to complete our tests. Therefore, we do not include it in our analysis or results later.

<table>
<thead>
<tr>
<th>Benchmark</th>
<th>Transactional Characteristics</th>
<th>Conflict Characteristics</th>
<th>Predictability Characteristics</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td># Xns</td>
<td>Average Size of Xns</td>
<td>Xn Time</td>
</tr>
<tr>
<td>Kmeans</td>
<td>2746</td>
<td>244.2 (310)</td>
<td>8.70%</td>
</tr>
<tr>
<td>vacation</td>
<td>2048</td>
<td>2982.9 (7507)</td>
<td>31.40%</td>
</tr>
<tr>
<td>labyrinth</td>
<td>2048</td>
<td>3036.0 (24837)</td>
<td>52.30%</td>
</tr>
<tr>
<td>ssca2</td>
<td>47299</td>
<td>65.2 (258)</td>
<td>10.20%</td>
</tr>
<tr>
<td>yada</td>
<td>20400</td>
<td>188.8 (1229)</td>
<td>49.60%</td>
</tr>
<tr>
<td>intruder</td>
<td>1488</td>
<td>208.4 (1073)</td>
<td>48.90%</td>
</tr>
<tr>
<td>bayes</td>
<td>32</td>
<td>208.3 (612)</td>
<td>25%</td>
</tr>
<tr>
<td>raytrace</td>
<td>70682</td>
<td>25.6 (54)</td>
<td>44.6%</td>
</tr>
<tr>
<td>radiosity</td>
<td>104209</td>
<td>40.1 (1402)</td>
<td>25%</td>
</tr>
</tbody>
</table>

Figure 4.4: Table showing conflict characteristics for 16 processor system. The numbers in parenthesis in the average size column indicate the maximum transaction size found in benchmark.

Figure 4.4 shows a table with 3 different characteristics of the benchmarks on a 16 processor system. The first 3 columns show transactional characteristics of the benchmarks. The next 6 columns show what we call “Conflict characteristics”. The remaining columns are used to show a pattern that exists in the data written during conflicts that can help us reduce their performance impact. From the transactional characteristics of the various benchmarks, we see that the benchmarks Raytrace and
Radiosity have a large number of short running transactions. In the STAMP suite, vacation and labyrinth are benchmarks with long transactions and they spend significant amounts of program time inside of transactions, whereas kmeans and ssc2 spend only 8-10% of their time inside of transactions. The size of transaction can be used as an indicator of contention. As the average size of the transaction increases, the probability of conflicts increases.

The conflict characteristics gives us an idea of the nature of conflicts. The conflict characteristics will differ depending on the type of conflict management scheme used and the number of processors in the system. We look for frequency of conflicts within a transaction, how many transactions get conflicted, and finally what is the performance impact (Nacks and restarts) from the conflicts. Looking at the frequency of conflicts, we see that number of conflicts per transaction is low; in most cases it is less than 3. The log size indicates the average number of writes per transaction that occur in the benchmark. This gives us an idea of the maximum number of conflicts that can occur in the benchmark. \(^2\)

The percentage of transactions that got conflicted is always low for all the benchmarks. This means that conflicts do not occur throughout the parallel program. The conflict addresses column shows the number of unique addresses involved in conflicts. That number is also low for most benchmarks. These two numbers indicate that while TM is able to extract good parallelism from most parts of the program, there are some sections/data structures within transactions that are responsible for most of the conflicts. This is an important result which signifies that \textit{resolving a few localized conflicts can unlock the parallelism and provide better performance for TM programs.}\(^2\)

\(^2\)We do not use this number as potential conflicts per transaction because the log may consist of both shared and private data.
4.5 A Case for Value Prediction

To understand the source of conflicts, we look at the data structures that are sequential and used frequently within transactions. Figure 4.5 shows the queue data structure:

```c
Class Queue {
    int queue_size;
    element* head;
    element* tail;
    enqueue(element*);
    dequeue();
}
```

- **enqueue(element* newElem)**
  
  ```c
  XBEGIN
  if (tail != NULL)
      tail->next = newElem;
  else
      head = newElem;
  tail = newElem;
  queue_size++;
  XEND
  ```

- **dequeue()**
  
  ```c
  XBEGIN
  if (head == NULL)
      return;
  elem* temp = head;
  head = head->next;
  free(temp);
  queue_size--;
  if (queue_size == 0)
      head = NULL;
      tail = NULL;
  XEND
  ```

Figure 4.5: Example showing the Queue Data Structure.

Figure 4.6: Enqueue and Dequeue operations on the queue.

4.5 A Case for Value Prediction

To understand the source of conflicts, we look at the data structures that are sequential and used frequently within transactions. Figure 4.5 shows the queue data structure.
structure that is commonly used in benchmarks from STAMP and Splash suites that can lead to conflicts under contention. Many parallel scientific applications consist of such a data structure task management and memory management purposes. For example, Cholesky and Raytrace from SPLASH [52] maintain a linked list for memory management purposes. Radiosity maintains queues for dynamic task scheduling. STAMP provides the heap, list and queue data structures in their library and they are used by most benchmarks for various bookkeeping activities. In many of these benchmarks, the individual tasks or threads can execute in parallel, and parallelism is often limited by serialized access to the data structure. Operations on these structures are not only serialized but also difficult to synchronize at a fine-grain. Every operation on the queue/list data structure needs to be serialized, even with transac-

```c
TM_BEGIN
    if (numFree < 1) {
        return FALSE;
    }
    TM_SHARED_WRITE(reservationPtr->numUsed,
                     TM_SHARED_READ(reservationPtr->numUsed) + 1));
    TM_SHARED_WRITE(reservationPtr->numFree,
                     (numFree - 1));
    CHECK_RESERVATION(reservationPtr);
TM_END

TMgrid_addPath method - Labyrinth
```

```c
TM_BEGIN
    long n = vector_getSize(pointVectorPtr);
    for (long i = 1; i < (n-1); i++) {
        long* gridPointPtr =
            (long*)vector_at(pointVectorPtr, i);
        long value =
            (long)TM_SHARED_READ(*gridPointPtr);
        if (value != GRID_POINT_EMPTY) {
            TM_RESTART();
        }TM_SHARED_WRITE(*gridPointPtr,
                              GRID_POINT_FULL);
    }
TM_END
```

```c
TM_BEGIN
    if (numFree < 1) {
        return FALSE;
    }
    TM_SHARED_WRITE(reservationPtr->numUsed,
                     TM_SHARED_READ(reservationPtr->numUsed) + 1));
    TM_SHARED_WRITE(reservationPtr->numFree,
                     (numFree - 1));
    CHECK_RESERVATION(reservationPtr);
TM_END

reservation_make method - Vacation
```

Figure 4.7: Examples from Vacation and Labyrinth benchmarks
tions, and any two concurrent transactions will conflict due to update of the same data structure members (head, tail and size) in all instances as shown in figure 4.6. Another example of data updates within transactions that can be conflicting is shown in figure 4.4. Both these examples are from the STAMP suite.

The predictability percentage column indicates our strategy for resolving conflicts. Looking at the data structures, we find that the conflicting-write values are often updated in a predictable manner. In the queue/list case, the head, tail and size of queue always increment or decrement a fixed amount and so do the shared writes in figures 4.4. This behavior can be exploited to improve concurrency. If we can predict future values written by conflicting transactions and feed those values to concurrently running transactions, we can improve the concurrency of the program. We profile values written by transactions for all addresses that got backed and compute strides between the values. The “predictability” of conflicts in a benchmark is the number of instances when at least 3 contiguous values written by the conflicting address had the same stride maintained.

Based on the three characteristics of the benchmarks, we rate each benchmark as being suitable or likely to benefit using value prediction to reduce conflicts. Labyrinth, Intruder and Yada benchmarks have long transactions with few conflicting addresses and highly predictable conflicting data. Thus, they are likely to see improvement from value prediction. For Radiosity and Raytrace, we label them as highly suitable because of the high predictability of conflicting data and the large number of transactions in those benchmarks. The benchmark vacation has long transactions and spends one third of its time inside of transactions; however, its conflict-predictability is low and the number of writes in a single transaction (from the log writes/Xn column) is high. Thus, we label its suitability as “medium”. With ssca2, its predictability is high but the average size of transactions and the percentage of time it spends within transactions is low. Thus, it is also labeled as medium only. For the remaining benchmarks, we do not expect much performance benefit from value prediction; however, we use them to show that a value-predictor-based TM system does not degrade per-
formance even if the benchmark is not suitable for value prediction. We do not use bayes any further because it does not show any contention in our tests.

The suitability of a benchmark for prediction is still a very optimistic assumption. There are various factors that can reduce the effectiveness of the value predictor.

- A benchmark may have a conflict towards the very end of the transaction and the value predictor may not derive sufficient concurrency.

- Other conflicting addresses (than those that conflicted in the LogTM case) may get exposed due to prediction, and they may not be predictable.

Thus, the only way to recognize any benefits from this scheme is by actual implementation.

A big incentive of using predicted values to run transactions in parallel comes from the fact that HTM systems already include hardware to enable speculative execution. Thus, the only extra hardware that is needed is for the value predictor, along with coherence protocol support for value predicted transactions. Another compelling reason for having a value predictor is that we do not lose the benefits of TM systems. In absence of conflicts, transactions execute normally, commit and turn non-speculative, without any interference from the value predictor. We believe that mispredictions will not be very costly, since most conflicting transactions would have stalled or aborted anyway. Our value-predictor-based TM system simply tries to run transactions in parallel when they would have otherwise stalled. We believe having a value predictor in the memory system has the potential to exceed the performance of any TM system for such applications since we break serializing data dependencies between transactions.
4.6 Final Motivation - Parallel Programming

Advantage

With value prediction we intend to improve performance and scalability of TM for applications whose concurrency is limited due to sequential data structures. The biggest motivation for this work is that we intend to keep the transactional memory programming paradigm alive because it makes parallel programming easier. With the advent of CMPs, parallel programming is getting renewed attention and with this work we would like to make it easier for programmers to transition to parallel programming by making a case for use of the TM programming paradigm. We want to increase the adoption of TM by increasing the number of applications that can benefit from TM. In the next chapter we discuss the first proposed design of our system.
Chapter 5

Value-Predicted TM (VP-TM)

5.1 Description of the base LogTM system

The base multiprocessor system for our study is a shared memory system using directories to maintain cache coherence over a point-to-point interconnection network, as shown in figure 5.1a. We use the LogTM model proposed by Moore [31] et al. to support transactions as our base TM system over which we develop our value predictor. Borrowing terminology from Bobba [5] et al., the LogTM model uses an eager version management scheme along with early conflict detection. For version management, every transaction is provided with a LIFO “log” in memory to save old data that is overwritten by the transaction. Conflicts are detected by the cache coherence protocol and result in the conflicting transaction getting a negative-acknowledgment (NACK). Nacks help retain ownership and force conflicting transactions to retry at a later time. However, NACKs also introduce the possibility of deadlocks.

Deadlocks are avoided by maintaining timestamps [37] for transactions that are retained following an abort and recording nacked transactions during conflicts. When a NACK is received, the timestamps from the conflicting and nacking transactions are compared and a cycle is detected if:

• The Nack is from a transaction that has an earlier timestamp.
• The conflicting transaction had previously nacked a transaction with an earlier timestamp.

A cycle indicates a potential deadlock, and it forces the nacked transaction to abort. In the LogTM model, an abort leads to a rollback of memory using the log and a processor checkpoint. Upon completing the rollback, the transaction is restarted. When a transaction reaches the end of the critical section, it commits by erasing the log of old values and the processor checkpoint.

We use the above TM model for three reasons:

1. The LogTM model has been provided along with the GEMS suite and accurately models the timing of transaction aborts and commits.

2. Eager version management makes commits faster, which is the common case, and hence leads to better performance [31].

3. It uses a Nack-based coherence protocol which is more supportive of a value predictor than a deferral-based scheme [37]. Nacks are visible to memory and can be used to trigger the value predictor to make a prediction.

### 5.2 Value-Predicted TM

Our initial design for the value-predictor-based TM (VP-TM) system has many simplifications. We assume a centrally-located value-predictor that performs the following functions:

• For addresses identified as predictable, the value predictor creates entries, intercepts load/store requests to those addresses, and maintains an ordered history of data values from different processors/transactions.

---

1 We deal with these simplifications in a later chapter.
• For a Nack to a load request whose address is present in the value predictor, it tries to provide a predicted value to the nacked transaction.

• As transactions commit, it updates the history for all shared addresses that were updated inside the transaction and verifies any predictions that were made for those addresses.

• In case of a misprediction, the predictor informs the processor, which in turn aborts the transaction that received the bad value and restarts.

Addresses that are not present in the value predictor table are treated normally, as in the base LogTM model. Since the value predictor needs to see all write accesses made to the memory location that needs predicted values, we envision that it will be located near the memory or directory controller; hence we call it a “memory-level” value predictor. A similar predictor has been explored in [8]; however, they have not provided any implementation details.
As shown in figure 5.2, our value predictor consists of a table of entries, where an entry corresponds to an address that is conflicting and needs predicted values. Each entry consists of the address, a set of recent values that were written to the memory location (along with the processors IDs and a set of speculated values), if any, for this line, along with the ID of the CPU that received the predicted value and the CPU that nacked the transaction. The number of recent entries to save depends on the amount of history desired to make correct predictions. For this research work, our value predictor is stride-based, so, we do not save more than 2 most recent written values. The maximum number of speculated values needed is equal to the number of processors in the system.

The VP-TM also demands a few changes to the Nack-based coherence protocol used by the LogTM model. Figure 5.3 shows the protocol actions for the VP-TM system. The sequence of steps in a VP-TM is as follows:

<table>
<thead>
<tr>
<th>Addr</th>
<th>RV1</th>
<th>RV2</th>
<th>RV3</th>
<th>Stride</th>
<th>SV1</th>
<th>SV2</th>
<th>SV3</th>
<th>SV4</th>
<th>SV5</th>
<th>P1</th>
<th>P2</th>
<th>P3</th>
<th>P4</th>
<th>P5</th>
<th>AB</th>
</tr>
</thead>
</table>

- RV – Recent Value; RV1 is the oldest
- SV -- Speculated value; SV1 is the oldest
- P1, P2, P3, P4, P5 – speculated procs in order
- AB – Abort Flag
- Stride = RV3 – RV2

Figure 5.2: Part (a) shows the changes needed to the micro-architecture to support VP-TM. Part (b) shows the structure of the VP entry.
1. If a transaction receives a nack from another transaction for an address present in the value-predictor, the value predictor tries to generate a prediction.

2. If a prediction was successfully generated, the directory protocol supplies the load from the requesting transaction with a predicted value. It does not clear the cache entries for the load operation at the caches. Thus, at the memory level, the load operation still “exists”.

3. The memory system keeps retrying the request until the owner commits or aborts.

4. The transaction with the predicted value can continue to execute but it cannot commit its data until the prediction is verified. We stall the transaction if it reaches the end and the predicted data has not been verified yet.

5. Once the request from the non-speculative owner returns, the data from the request is compared with the predicted data and the processor is informed of the result.

6. If the prediction succeeded, the processor that got the predicted value becomes the new owner of the line.

7. If a processor that got a predicted value has to abort a transaction, it informs the directory to remove its entry from the value predictor.

The VP-TM system also need not log data for addresses that have obtained a predicted value from the value predictor. This is because the predicted value may be incorrect and it should not be written back to memory during an abort. Also, stores for these addresses are not propagated to memory. Instead, store data for such addresses needs to be buffered until the predicted load value for that address has been verified. We implemented a small buffer of about 10 entries for each processor to store data which was predicted by the predictor.
5.3 Coherence protocol for the VP-TM system

Whenever a transaction gets a predicted value, it becomes the “speculative owner” of the line. This means that future load requests or retries for that address, from other transactions, are forwarded to this processor instead of the original owner. Only requests coming from the oldest speculative owner are forwarded to the original owner for the line. Thus, for an address that may have speculative owners, the directory consults the value-predictor to get the processor to which the current request must be forwarded. Figure 5.3d shows an example of requests from multiple speculative owners getting forwarded to their respective owners.

In this design, it should be noted that we predict values for integers and pointers in parallel programs. While these values are predictable, to merge the value predictor with the TM system and the coherence protocol, we treat the entire cache line as predicted even though the prediction was made only for a part of the cache line. This implies that within a transaction that got a predicted value for a cache line, a load that tries to access any other data on that cache line, will get incorrect data. One way to avoid this scenario is by marking each byte in the cache line for prediction and stalling on such a load until the prediction can be verified and line can get correct data. Another way is to perform all loads to predicted lines to memory to verify the address with the value predictor. A Nack from the value predictor can abort the transaction since it consumed bad data.

5.3.1 Dependencies Among Predicted Transactions

If a transaction receives a predicted value, it cannot commit until its prediction is verified, usually by the transition that had NACKed its request. Thus, value prediction imposes an ordering of commits among concurrent transactions. This may lead to a deadlock, especially if two transactions have received predicted values from one another and are each waiting for the other to commit. To avoid this, when a request gets NACKed, before generating a prediction, the value predictor looks up
1: CPU 1 issues a GETX for a shared address X.
2: Memory responds with Data and changes state of line to M with 1 as owner.
(B) 1: CPU 2 also issues a GETX request.
2: Directory forwards request to owner 1
3: CPU 1 sends a Nack to CPU 2.
4: CPU 1 also sends a Nack to inform directory.
5: Directory gets a value from the predictor and sends it to CPU 2.
(C) CPU 3 gets a predicted value from CPU 2 because CPU2 is now the speculative owner of the line.
(D) Retries from different CPUs go to their respective owners.
(E) CPU 1 has committed the transaction.
1,2,3: CPU 1 gives up data in response to forward from CPU 2.
4: The data is sent to the directory.
5: The value predictor then verifies the prediction and sends result to CPU 2.
(F) If the prediction succeeded, the directory maintains its state and changes the owner.
Now, CPU 2 commits its transaction and similarly gives up data to CPU 3.
(G) If prediction fails, then the directory moves to the I state.
1,2: In response to a retry from CPU3, the directory responds with the result and changes state according to the result of the prediction.

Figure 5.3: Coherence Protocol Actions with VP-TM
to see if the NACKing processor has ever gotten a “future” predicted value from the requesting processor. If true, then one of the processors needs to be aborted because otherwise it will lead to a deadlock. Since the NACKing processor has already received a predicted value, we choose to abort it so that at least one of the transactions can make forward progress. This is the main reason why we need a memory-level and a global value predictor. Being at the memory enables the value predictor to see all load/store requests for all address that have been identified as predictable. Being global enables the predictor to see all processors that have received predicted values. This information is needed to decide whether a creating a predicted value for a given processor will lead to a deadlock.

The order among speculative owners also matters when non-speculative data is finally returned to the directory. The directory consults the value-predictor to get the processor whose predicted data should be verified. This step is important to maintain sequential consistency. In case the prediction succeeds, this speculative owner becomes the new non-speculative owner of the line. If the prediction fails, the
line is returned to Invalid state. At this point, the directory line is in Invalid state; however, it has speculative owners. It can choose to abort all speculative owners or it can allow requests from speculative owners to get the current data from the directory and verify it against the prediction made.

Ideally, with the value predictor we would like to have a scenario where one CPU is executing the transaction with non-predicted values and all other transactions are running with predicted values in the order determined by the value predictor. If the predictions turn out to be correct, we have managed to improve the concurrency of such a system. However, there are many reasons why a value predictor may not deliver such a speedup even with predictable shared values:

- The programmer has not identified all shared conflicting data. This means that, while some of the shared values will be predicted, transactions will still stall or abort due to Nacks for thepu i addresses that are not present in value predictor.

- Transactions can still abort due to out-of-order prediction of values for different processors leading to cyclic dependencies.

- An abort of a speculative owner for a memory address can cause all future speculative owners of the address to mispredict and restart. Thus, we may have a “chain reaction” of aborts.

We investigate the behavior of a value-predictor-based TM system in the next section.
5.4 Performance Evaluation and Results

In this section, we try to verify our claim that adding a value predictor and using it to order transactions can improve concurrency for a TM system. We implemented a value predictor on top of LogTM model provided by Moore [31] et al. based on the GEMS [26] suite which is driven by Simics [23] functional simulator. We implement all the necessary functionality as described in the previous section for the value predictor to enable correct and deadlock-free execution. Some of the highlights(caveats) of the design of our value predictor are:

- The value predictor is modeled as a table with 10 entries. It is located near the memory system and is capable of intercepting all memory accesses.

- Extra messages added by the value predictor have been timed as per the coherence protocol.

- There is no directory-to-value-predictor latency. This will not significantly affect timing of protocol messages. We only avoid a few race conditions with this simplification.

- For shared conflicting accesses, we always generate exclusive requests to memory. This is purely a performance decision. For fairness, we use this feature for all simulations of TM, with or without the value predictor.

- A cache line that has predicted data cannot be written back to memory in the event of a writeback. In our design we simply stall until the value prediction is verified.

We compare performance of our VP-TM with the LogTM model to get an idea of the performance change by adding the value predictor. The system parameters are as shown in Table 5.1.
### Table 5.1: System Parameters for our simulations

<table>
<thead>
<tr>
<th>Unit</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Processor</td>
<td>single-issue, in-order, 1GHz</td>
</tr>
<tr>
<td>L1 Cache</td>
<td>64K, 4-way, split, 1-cycle hit latency</td>
</tr>
<tr>
<td>L2 Cache</td>
<td>8M, 32-way, unified, 10-cycle hit latency</td>
</tr>
<tr>
<td>Directory</td>
<td>80-cycle latency</td>
</tr>
<tr>
<td>Memory</td>
<td>4GB</td>
</tr>
<tr>
<td>Restart Penalty</td>
<td>20 cycles + penalty per line rollback from log</td>
</tr>
</tbody>
</table>

#### 5.4.1 Predictions, Accuracy and Speedup

We classify predictions made by the value predictor into three types:

- **Good Predictions** Predictions that are verified to be correct and can lead to a transaction commit.

- **Bad Predictions** Predictions that turn out to be incorrect at the time of verification.

- **Aborted Predictions** These are predictions that had to be aborted before they were verified. One cause of such aborts might be a deadlock with other transactions which caused a transaction to be aborted.

Prediction accuracy may not provide all the reasons for performance speedup or degradation compared to TM systems. While correct predictions have the potential to improve concurrency, multiple correct predictions on a single transaction can be undone by a single bad/aborted prediction or an abort which can lead to performance degradation even with an accurate VP. Similarly, the number of aborted predictions does not always indicate poor performance, since they represent wasted work when the transaction would have stalled anyway. In general, we expect that a speedup over TM would be due to reduced number of aborts and a reasonably accurate predictor.

To get a better idea of how predictions affect performance, we add a feature to our value predictor to control the maximum number of simultaneous predictions on
a single entry. While an increased number of predictions are capable of increasing concurrency, it can also lead to cyclic dependencies as discussed in section 5.2. We vary the max number of predictions from 1 to 5, to see if increased predictions always results in better performance. In our figures, we mention the different cases by the maximum number of predictions allowed in each case. For Radiosity and Raytrace, we ran tests only up to 3 max-predicted addresses due to simulation times being very high.

5.4.2 Static vs. Dynamic Analysis

To understand predictability within benchmarks better, we also conduct experiments where we manually identified some of the values that can serialize transactions. We disable the dynamic addition of entries inside the value predictor and use magic instructions in Simics to pass these addresses to the value predictor via the programmer. We then compare the results with the standard dynamic addition. The purpose of comparing with this software approach is to find out if the queue structure is the only performance bottleneck in the application and whether other conflicts can be easily predicted.

5.5 Evaluation using Microbenchmarks

For evaluation purposes, we use a microbenchmark to isolate the performance benefits or losses from using a value predictor. The microbenchmark is the same as the example outlined in the section 4.5. It performs $2^{10}/n$ operations on a shared queue. We have two different versions of the microbenchmark. The first version performs only insertions while the second version randomly inserts or deletes elements from the head. The insert-only version gives us an idea of the best possible case for our value-predicted TM system. With the random insertions and deletions, we wish to analyze whether the presence of a value predictor can degrade performance of a TM system in presence of conflicts that are not very predictable.
Figure 5.5: Results for the Insert-only version of Queue Microbenchmark: Speedup (denoted by lines) % Restarts (denoted by bars) and % Prediction Accuracy (denoted by dotted-line).

5.5.1 Performance Analysis

The results from figures 5.5 and 5.6 show the speedup for the two versions of the microbenchmark against 4, 8 and 16 processors (y axis on the right). We vary the number of predictions per address (y-axis on the left) and also show the percentage of total transactions that restarted for one of the VP cases\(^2\) and the LogTM model. The dotted line in the graph indicates prediction accuracy for the two predictions per address case using the right y-axis. As shown in figures, our VP-TM system has improved the scalability of the microbenchmark for both cases. Looking at figure 5.5, we can see that the value predictor can easily capture the predictability for this data structure and provide good performance by reducing restarts. Even for the random

\(^2\)2 predicted sharers per address in our case, since it is the best performing.
insert/delete microbenchmark, our VP-TM system performs better, albeit not as good as the first case, even with a low prediction accuracy as seen from figure 5.6.

We attribute this performance benefit to the improved concurrency with the VP-TM system that can lead to a reduced number of aborts/restarts, reducing memory traffic. The reduction in the prediction accuracy is due to the increased number of aborts in the system that disturbs the stride of the value predictor. With each restart or mis-prediction, the value predictor has to recalculate a stride while the “abort flag” limits further predictions until a transaction commits. This reduces the performance benefits that can be obtained.

For the different VP cases, we find that increasing the number of predicted sharers leads to better performance for a smaller system. But for 16 processors we find that limiting the number of predicted sharers works best. For a 16-processor system, the
best performing VP cases are when the number of predicted sharers limited limited to 2 or 3. This is because with an increased number of sharers, we can have a cascade of mispredictions\(^3\) that get resolved in order, reducing concurrency and degrading performance. This also means that we can fix the number of speculated value entries within a single value predictor entry to around 3 to 5 and still get good performance.

Table 5.2: Parameters: 1024 transactions - Insert-only version

<table>
<thead>
<tr>
<th>Parameter</th>
<th>2P</th>
<th>4P</th>
<th>8P</th>
<th>16P</th>
</tr>
</thead>
<tbody>
<tr>
<td>Aborts</td>
<td>120</td>
<td>1132</td>
<td>1551</td>
<td>2610</td>
</tr>
<tr>
<td>Bad Predictions</td>
<td>120</td>
<td>362</td>
<td>745</td>
<td>894</td>
</tr>
<tr>
<td>Good Predictions</td>
<td>180</td>
<td>525</td>
<td>545</td>
<td>561</td>
</tr>
<tr>
<td>Total Predictions</td>
<td>410</td>
<td>1398</td>
<td>2137</td>
<td>4510</td>
</tr>
</tbody>
</table>

Table 5.3: Parameters: 1024 transactions - Random version

<table>
<thead>
<tr>
<th>Parameter</th>
<th>2P</th>
<th>4P</th>
<th>8P</th>
<th>16P</th>
</tr>
</thead>
<tbody>
<tr>
<td>Aborts</td>
<td>95</td>
<td>797</td>
<td>1911</td>
<td>3227</td>
</tr>
<tr>
<td>Bad Predictions</td>
<td>95</td>
<td>494</td>
<td>666</td>
<td>672</td>
</tr>
<tr>
<td>Good Predictions</td>
<td>120</td>
<td>601</td>
<td>428</td>
<td>334</td>
</tr>
<tr>
<td>Total Predictions</td>
<td>380</td>
<td>1551</td>
<td>2427</td>
<td>3560</td>
</tr>
</tbody>
</table>

5.5.2 Evaluation using benchmark suites

We use the same set of benchmarks that we analyzed in section 4.4 to see if we can get the performance improvement with VP-TM that our analysis has predicted. Table 5.4 shows the benchmarks and the sizes used for simulation purposes. To reduce the amount of simulation time for Radiosity, we do not measure all iterations in the parallel section. We believe this will not affect the results because each iteration performs the same computation. Thus, we can get the trends we need from a single iteration.

\(^3\)The abort flag in the VP per entry is specifically used to avoid such a scenario.
Table 5.4: Benchmark Parameters

<table>
<thead>
<tr>
<th>Benchmark</th>
<th>Suite</th>
<th>Input</th>
<th>Units Measured</th>
</tr>
</thead>
<tbody>
<tr>
<td>RAYTRACE</td>
<td>SPLASH</td>
<td>car</td>
<td>parallel section</td>
</tr>
<tr>
<td>RADIOSITY</td>
<td>SPLASH</td>
<td>batch</td>
<td>1 task</td>
</tr>
<tr>
<td>Intruder</td>
<td>STAMP</td>
<td>-a10 -l4 -n1024 -s1</td>
<td>parallel section</td>
</tr>
<tr>
<td>Labyrinth</td>
<td>STAMP</td>
<td>random-x24-y24-z3-n1024</td>
<td>parallel section</td>
</tr>
<tr>
<td>Kmeans</td>
<td>STAMP</td>
<td>random-n2048-d32-c32</td>
<td>parallel section</td>
</tr>
<tr>
<td>ssc2</td>
<td>STAMP</td>
<td>-s13 -i1.0 -u1.0 -l3 -p3</td>
<td>parallel section</td>
</tr>
<tr>
<td>Vacation</td>
<td>STAMP</td>
<td>-n8 -q50 -u98 -r2048 -t2048</td>
<td>parallel section</td>
</tr>
<tr>
<td>Yada</td>
<td>STAMP</td>
<td>-a20 -ittimeu10000.2</td>
<td>parallel section</td>
</tr>
</tbody>
</table>

Figure 5.7: Speedup Results for 16 processors with different max-predicted-sharers

5.5.3 Benchmark results

From the benchmark results in figure 5.7, we can see that VP-TM system performs better than the LogTM model for all benchmarks. The performance differences can be explained by looking at figure 5.8 for successful predictions. For benchmarks such as labyrinth, Radiosity and Raytrace, the number of predictions and the percentage of good predictions is high enough that program concurrency improves to provide good
<table>
<thead>
<tr>
<th>Benchmark</th>
<th>VP-TM</th>
<th>LogTM</th>
<th>Best-case Speedup Over LogTM</th>
<th>Xns</th>
<th>Commits</th>
<th>Aborts</th>
<th>Nacks</th>
<th>Total Predictions</th>
<th>Good Predictions</th>
<th>Bad Predictions</th>
<th>Coverage</th>
</tr>
</thead>
<tbody>
<tr>
<td>Kmeans</td>
<td>2.06%</td>
<td>3806</td>
<td>2746</td>
<td>1060</td>
<td>3383</td>
<td>120</td>
<td>94.20%</td>
<td>4.20%</td>
<td>94.50%</td>
<td></td>
<td></td>
</tr>
<tr>
<td>vacation</td>
<td>5.86%</td>
<td>2089</td>
<td>2048</td>
<td>41</td>
<td>13809</td>
<td>28</td>
<td>32.10%</td>
<td>0.00%</td>
<td>70.00%</td>
<td></td>
<td></td>
</tr>
<tr>
<td>labyrinth</td>
<td>1027.75%</td>
<td>2332</td>
<td>2048</td>
<td>284</td>
<td>1499</td>
<td>710</td>
<td>69.00%</td>
<td>2.00%</td>
<td>38.00%</td>
<td></td>
<td></td>
</tr>
<tr>
<td>ssc2</td>
<td>44.79%</td>
<td>47659</td>
<td>47299</td>
<td>303</td>
<td>777</td>
<td>15</td>
<td>20.00%</td>
<td>13.30%</td>
<td>68.20%</td>
<td></td>
<td></td>
</tr>
<tr>
<td>yada</td>
<td>13.52%</td>
<td>11910</td>
<td>9695</td>
<td>2215</td>
<td>11086</td>
<td>1792</td>
<td>9.30%</td>
<td>2.70%</td>
<td>34.00%</td>
<td></td>
<td></td>
</tr>
<tr>
<td>intruder</td>
<td>12.47%</td>
<td>2647</td>
<td>1488</td>
<td>1159</td>
<td>7666</td>
<td>801</td>
<td>27.50%</td>
<td>13.70%</td>
<td>18.90%</td>
<td></td>
<td></td>
</tr>
<tr>
<td>raytrace</td>
<td>99.05%</td>
<td>57430</td>
<td>45391</td>
<td>12039</td>
<td>21029</td>
<td>2049</td>
<td>37.12%</td>
<td>17.21%</td>
<td>41.19%</td>
<td></td>
<td></td>
</tr>
<tr>
<td>radiosity</td>
<td>82.82%</td>
<td>76841</td>
<td>79281</td>
<td>23904</td>
<td>23432</td>
<td>3122</td>
<td>45.54%</td>
<td>33.2%</td>
<td>41.45%</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Figure 5.8: Table showing characteristics of application benchmarks for LogTM and VP-TM for each benchmark. The VP numbers are for the Max 2 predicted sharers, except the best case speedup numbers. The good and bad predictions do not add to 100% because some transactions abort even before predictions can be verified.

Speedup over the base case. Labyrinth shows the best performance with a speedup of 1000%. For benchmarks such as vacation, yada and ssc2, VP-TM system provides speedup improvements from 5% to 40%. The reason that vacation does not show much benefit from VP-TM is because of the low number of predictions. Although vacation has a predictability of around 30%, it has a large number of writes per transaction. Some of these writes get exposed as conflicts which are not very predictable (seen from the increased number of aborts and NACKs) and hence vacation does not benefit much from VP-TM. Ssca2, on the other hand, even with low prediction accuracy and total predictions, seems to benefit more from VP-TM because VP-TM reduces the number of aborts and NACKs, which leads to improved concurrency. This also means that, so far, our confidence mechanism inside the value predictor and the abort flag are doing their job of limiting predictions to avoid performance degradation.

While the job of VP-TM is to improve concurrency, thereby reducing NACKs and
restarts, in some cases in figure 5.8, we see that number of NACKs and restarts increases with VP-TM. The increase in NACKs can be easily explained because NACKs in VP-TM do not always signify serialization compared to LogTM model. The increase in restarts is due to the fact that we try to increase the concurrency of the transaction by eliminating conflicts. If not all conflicts get eliminated, the contention around the remaining conflicts increases, resulting in increased number of restarts around those conflicts. We may still see some performance improvement with VP-TM. But the resulting increase in restarts negates some of the performance improvement by increasing memory traffic as seen in kmeans. In all of the benchmarks we have studied so far, the improved concurrency and a good confidence mechanism (and abort flag) provides speedup over the base system.

The coverage column indicates the percentage of conflicting accesses that were successfully predicted. There are three reasons why a predictable conflict cannot be predicted:

- The abort flag is set.
- The number of simultaneous predictions on a single address is limited
- The prediction may lead to a cyclic dependency and subsequently a deadlock.

5.5.4 Static vs. Dynamic results

From the results as shown in figure 5.9, we see that for some benchmarks, the dynamic addition of entries to the value predictor provides better performance. With Labyrinth, we see that there is no performance difference in the static or dynamic cases, which means the only conflicting addresses are those that are present in the queue/list structure which were identified manually by the static scheme. That also explains why VP-TM performs so much better for this benchmark. With Intruder, the dynamic case performs worse than the static case. We believe the reason for this degradation is the addition of certain conflicting addresses to the value predictor that
are not predictable and lead to frequent mis-predictions. We noticed that with the static case, the total number of predicted addresses decreased but the percentage of good predictions improved compared to the dynamic case, leading to better performance. We know that manual identification of shared conflicting addresses cannot be expected from parallel programmers in all cases but we shows that in some cases with manual identification, we can further improve the performance of VP-TM.
Chapter 6

Limited Early Value Communication

6.1 Introduction to LEVC

As discussed earlier, our VP-TM work and work by Hany et al. [38] tries to tackle the problem of conflicts by either predicting conflicting data or forwarding data on a conflict within transactions. Both these techniques demand additional changes in memory subsystem. The Dependence-Aware TM model adds 8 extra states on top of the MSI coherence protocol to enable forwarding and verification of data. The VP-TM system from Pant et al. [34] system also adds states and extra messages for prediction and verification, and changes at each processor subsystem (transaction state and caches), compared to TM.

Besides these hardware requirements, speculatively communicating data within transactions can create a commit order among transactions. Transactions that have received uncommitted values (or predicted values) become “dependent” on the owner of the data and cannot commit until the owner commits. This can lead to cyclic dependencies which need to be detected to avoid deadlocks. To detect such cycles, the superset of all the processors that have received or forwarded data speculatively
within a transaction must be available to all processors. Ramadan et al. [38] use a bus-based system and an order vector that is broadcast to all processors to avoid cycles due to value forwarding. In our VP-TM work, we use a directory-based system along with a centralized value predictor so that all the cycles that can arise due to predictions can be detected to avoid a deadlock. Once a cycle is detected in a TM system, at least one of the transactions needs to be rolled back to avoid a deadlock. Dependent transactions can, thus, not only increase the hardware requirements, but also reduce the performance gains that can be obtained via any of the above techniques. Further complications can arise when a processor, in this “chain” of dependent processors, aborts. In this case, the chain needs to be updated to avoid starving the dependent transactions.

All these above issues indicate that early communication of transactional data either by forwarding or prediction needs extensive additional hardware support in the form of changes to the coherence protocol, additional buffer space, bits to keep track of forwarded values etc. and changes to the deadlock detection mechanisms. Even with all the additional hardware, performance of these techniques can still be limited due to the additional deadlock cases that can arise. Thus, we cannot extract performance from the above two techniques and we end up with significantly increased hardware because of the uncontrolled increase in concurrency and then simply providing mechanisms to detect cycles. Furthermore these techniques may not even scale with increasing number of processors since that will further increase the number of cycles that can arise.

We believe that the hardware requirements for TM systems already are high enough and, therefore, we try to minimize the extra hardware that may be needed by the above two techniques. We specifically want to avoid adding states to the coherence protocol since that increases verification time and ultimately, the cost of TM hardware. In this research, we take a step back and analyze the behavior of TM benchmarks from the STAMP suite [28] to find the number of concurrently stalled sharers of data. Our results indicate that the number of transactions simultaneously
accessing or conflicting on shared addresses is usually low. We propose an alternative scheme, based on the LogTM system, that allows uncommitted transactional values to be forwarded in a limited manner without any changes in the coherence protocol and with minimal extra hardware at each processor to maintain transactional correctness and consistency while providing better performance than the base TM system under high contention. We reduce the hardware in the following ways compared to the previous approaches:

- We avoid the need to maintain a list of dependents and have to broadcast it like DATM or have a centralized hardware structure like VP-TM, to detect cycles. This also simplifies the memory directory.

- We avoid using extra states/changes in the coherence protocol to indicate latest “speculative owner” of the data (VP-TM) or to forward data to a concurrent sharer (DATM) of uncommitted data.

In the next section, we describe the problem with any technique using forwarding or prediction in TM. Next we present an analysis of the benchmarks. Next, we present our implementation and results.

### 6.1.1 Problem of Dependent Transactions

Both the DATM and VP-TM systems try to solve the problem of conflicts by early communication of transactional data. They eliminate conflicts but create the problem of dependent transactions. Early communication of data necessitates a commit order among transactions to ensure conflict serializability. A transaction that has received an early value from another transaction cannot commit until the “owner” transaction commits, to verify that the forwarded/predicted value was correct. This can lead to cycles among transactions. The problem is further exacerbated if there are multiple such values communicated for a single transaction. This is shown in figure 6.1.

A value at X is forwarded from T1 to T2, making T2 dependent on T1 i.e. T2 cannot commit until T1 commits and can verify the correctness of the forwarded
Figure 6.1: Figure showing the various deadlock cases that arise due to early value communication. X, Y and Z are shared conflicting addresses. Shading indicates load-exclusive accesses within transactions. P/F indicates either prediction or forwarding.

value. Later on when T1 tries to load Y, it gets a NACK from T2. Thus, both T1 and T2 are now Nacking each other and none of them can proceed leading to a deadlock. This can be resolved by T2. When T2 sends a NACK to T1, it can check to see if it got a forwarded value from T1 and if true, it can choose to abort to lead T1 finish execution. Another example of a sequence of events that may lead to a deadlock is shown in figure 6.1b which involves 3 addresses. However, in this case, a deadlock cannot be detected simply based on just the two transactions involved in the conflict. In this case, the cycle is created with multiple transactions sharing values for multiple addresses. To detect such cycles, any processor that is involved in a conflict needs to know the entire set of processors involved in communication of an early value (receiving or forwarding/predicting). DATM achieves this by using an order set of dependent transactions that is broadcast each time a value is speculatively forwarded. The VP-TM system uses a centralized value predictor that can monitor all predictions and conflicts to detect deadlocks.
Both these solutions for detecting such cycles are fairly complex and needs extra hardware and mechanisms. These increased hardware requirements can increase the cost of implementation as well as verification. TM itself imposes significant changes in the coherence protocol and, thus, it will be very hard to convince system designers to add extra hardware to implement these ideas.

Dependencies between transactions in both these schemes not only lead to increased hardware requirements, they also reduce the performance gains that can be achieved simply because cycles created by dependent transactions can only be detected a-posteriori (once the cycle has been created). Thus, the only way to avoid them is by aborting transactions and rolling back and restarting from the beginning. This reduces the benefit that can be achieved by the above schemes. In this work we look into this aspect of early value communication. We try to analyze whether this increased concurrency is really required. We analyze benchmarks to find the number of concurrent sharers of conflicting data. In the next section, we present an analysis for the STAMP benchmark suite that enables us to reduce the hardware required for VP-TM.

6.2 Analysis of TM benchmarks

In this section, we present some characteristics of benchmarks that help us understand their behavior under contention. A similar table of characteristics is also presented in the VP-TM paper. We use benchmarks from the STAMP [28] suite which already uses transactions.

Figure 6.2 shows a table showing two different characteristics of the benchmarks on a 16 processor system. The first three columns show transactional characteristics of the benchmarks. The remaining columns show what we call as “Conflict characteristics” which is nothing but the behavior of conflicting accesses. Conflict characteristics give an idea of the contention in the program and analyzing this behavior helps us to improve concurrency. Nacks, Restarts, percent transactions that conflict and unique
Figure 6.2: Table showing conflict characteristics for 16 processor system. The numbers in parenthesis in the average size column indicate the maximum transaction size found in benchmark.

conflicting addresses are measured directly from execution. For conflicts per address, we count the number of Nacks sent per conflicting access (only one time for each Requestor and Nacker) from the owner. This way we avoid counting Nacks due to retries. Thus, “average conflicts per address” indicates the average number of Nacks for conflicting addresses per transaction (counting only transactions that conflicted).

From the table we make the following observations:

1. The percentage of conflicting transactions and the number of conflicting addresses is usually very low. This means that conflicts usually arise only from a few serializing structures in the program ¹.

2. Average number of threads/transactions involved in conflicts is only slightly greater than one for most benchmarks. Yada and intruder are the only benchmarks that have a value of average conflicts that is greater than one.

In this work, we focus on the last observation - most conflicting accesses only

¹This observation is from the VP-TM work.
involves one or two transactions. If we limit the prediction to only one processor, we can avoid the need to maintain an order of processors for an address either inside the value predictor as in VP-TM, or within the processor and having to broadcast it as in DATM [38]. We do not expect to get performance as good as VP-TM, however, our goal is to see if we can improve performance of LogTM on high contention benchmarks. Since we only demand a modest increase in hardware compared to the LogTM model, we can make a case for limited value forwarding to be used along with the LogTM model. In the next section, we present changes to the VP-TM that enable us to avoid having a memory-level value predictor entirely and thus reduce the hardware requirements of the VP-TM system.

### 6.3 Design of LEVC system

In figure 6.3, we compare the three TM models and describe the new system that we propose. In this section, we describe our new Limited Early Value Communication System (LEVC). We show that by limiting concurrency, we get rid of most of the extra hardware requirements that the DATM and VP-TM schemes can impose. In later sections, we evaluate the performance of LEVC and compare it to the base LogTM system and the VP-TM system [34]. The following are the changes made to the original LogTM model provided with the GEMS suite [26].

1. Whenever there is conflict, the NACK that is returned contains the value from the source cache. If there is no cycle detected, the transaction that receives the NACK can use this value to run ahead. The owner of the cache line is not changed at caches or the directory.

2. If a transaction chooses to use the early value returned by NACK, it stores this value in a buffer located at the processor. Subsequent loads and stores to the cache line will use this data.

3. The original memory transaction does not complete at the memory level and
Figure 6.3: In (a), we see that the LogTM model uses NACKs to resolve conflicting accesses. The VP-TM model has a value predictor at the directory that provides Nacked transactions with predicted values as seen in (b). The DATM scheme by Hany is similar as seen in (c). In the new scheme in (d), we forward the value from the Nacking transaction to only one processor. The initial state of the line is M with CPU-1 as owner. The shading on CPU indicates CPU is executing a transaction.
the transaction has to keep retrying the request until the committed value is available.

4. For any address, we allow only one such concurrent sharer using the value received from a NACK. This is where the scheme really differs from DATM. The source also notes down the line in its cache. This is done to avoid forwarding the value to other caches.

5. All requests to the line are forwarded to the owner of the line. Since a value has already been forwarded for the address, the owner returns a NACK with a NULL value. The Requestor will then retry the request.

6. Each time the NACK returns, the transaction checks to see if the value is different from the value in the buffer. If the value is different, it indicates that there was another write in the nacking transaction to the same address. The transaction running with the forwarded value must abort.

7. Once the committed value is available, it is compared with the forwarded value to verify the prediction. If the prediction fails, the transaction has to abort, otherwise it can commit.

8. On a commit, the value is written back to memory. On an abort, the value is discarded.

For LEVC, the LogTM directory protocol does not need any changes. The only extra hardware needed is a bit to indicate that this line was either forwarded or received to prevent further concurrent transactional sharing. However, to reduce the number of messages as an optimization, it can have an extra “speculation” bit at the directory to directly NACK requests from later sharers for the address. In this case, we will also need to inform the directory indicating start and end of speculation and an abort of the transaction that received a forwarded value to set/reset the speculative bit.
To avoid extra hardware for dependence checking, LEVC restricts the number of concurrent sharers of conflicting data. With just one processor, we avoid the requirement to know the entire set of processors involved in sharing. This avoids complicated broadcast mechanisms or centralized hardware that is needed by previous schemes. Another restriction that needs to be added when there may be multiple address values being shared is: A transaction that has received early data from another transaction cannot be permitted to forward any data until the forwarded data received is verified. This prevents a cycle from figure 6.1b from occurring. Since the number of unique conflicting addresses in most of our benchmark was low as seen in figure 6.2, we do not expect this to be a severe bottleneck.

Thus, with these minimal changes we can convert a stall-based LogTM model into a value forwarding TM system. Comparing this to the original LogTM system, we find that we can get improved concurrency from an early forwarding of values. However, value forwarding also increases the chance of deadlock which leads to increased rollbacks. In the next section, we try to compare performance of this system with the original LogTM model and see if this optimization can improve concurrency using forwarded values such that it can overcome the increased traffic and possible serialization from a potential increase in the number of rollbacks.

6.4 Performance Evaluation and Results

In this section, we try to evaluate our proposal of limited early value communication. We use the same LogTM model provided by Moore [31] et al. which is based on the GEMS [26] suite driven by the Simics [23] functional simulator. We implement all the necessary functionality as described in the previous section to enable correct and deadlock-free execution. We compare performance of our limited early value communication scheme with the LogTM model to see if our scheme can lead to increased concurrency and better performance compared to the base case. We also compare
Table 6.1: System Parameters for our simulations

<table>
<thead>
<tr>
<th>Unit</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Processor</td>
<td>single-issue, in-order, 1GHz</td>
</tr>
<tr>
<td>L1 Cache</td>
<td>64K, 4-way, split, 1-cycle hit latency</td>
</tr>
<tr>
<td>L2 Cache</td>
<td>8M, 32-way, unified, 10-cycle hit latency</td>
</tr>
<tr>
<td>Directory</td>
<td>80-cycle latency</td>
</tr>
<tr>
<td>Memory</td>
<td>4GB</td>
</tr>
<tr>
<td>Restart Penalty</td>
<td>20 cycles + penalty per line rollback from log</td>
</tr>
</tbody>
</table>

Table 6.2: Benchmark Parameters

<table>
<thead>
<tr>
<th>Benchmark</th>
<th>Input</th>
</tr>
</thead>
<tbody>
<tr>
<td>intruder</td>
<td>-a10 -l4 -n1024 -s1</td>
</tr>
<tr>
<td>labyrinth</td>
<td>random-x24-y24-z3-n1024</td>
</tr>
<tr>
<td>kmeans</td>
<td>random-n2048-d32-c32</td>
</tr>
<tr>
<td>sscap2</td>
<td>-s13 -i1.0 -u1.0 -l3 -p3</td>
</tr>
<tr>
<td>vacation</td>
<td>-n8 -q50 -u98 -r2048 -t2048</td>
</tr>
<tr>
<td>yada</td>
<td>-a20 -ittimeu10000.2</td>
</tr>
</tbody>
</table>

our scheme to VP-TM to measure how much performance is lost from the limited increase in concurrency. In our previous chapter, we describe a way to limit concurrency by statically controlling the number of predicted sharers per address inside the value predictor. We also show that different benchmarks need different levels of concurrency control and that no single “magic” number of sharers that performs best for all programs. In this work, we use the best case of VP-TM for all the benchmarks to get the upper limit of performance that can be achieved by early value communication with transactions. The system parameters are as shown in Table 6.1. We use the same set of benchmarks that we analyzed in section 6.2 to see if we can get the performance improvement with VP-TM that our analysis has shown. Table 6.2 shows the benchmarks and the sizes used for simulation purposes.
From the benchmark results in figure 6.4, we can see that our Limited Value Communication Scheme performs better than LogTM for all benchmarks except intruder (not visibly worse, -2.0%). The results can be explained by looking at figure 6.5. The performance improvement comes because of the increase in concurrency compared to the base case. For benchmark intruder, the number of forwarded values that were successfully predicted is very low. This means that early forwarding does not work well with intruder because of multiple writes per transaction. There is an increase in the number of restarts, however, and hence performance of intruder is
<table>
<thead>
<tr>
<th>Benchmark</th>
<th>LEVC</th>
<th>LogsTM</th>
</tr>
</thead>
<tbody>
<tr>
<td>kmeans</td>
<td>2796</td>
<td>2945</td>
</tr>
<tr>
<td>vacation</td>
<td>2096</td>
<td>2075</td>
</tr>
<tr>
<td>labyrinth</td>
<td>2332</td>
<td>15926</td>
</tr>
<tr>
<td>scca2</td>
<td>47615</td>
<td>47659</td>
</tr>
<tr>
<td>yada</td>
<td>21016</td>
<td>20498</td>
</tr>
<tr>
<td>intruder</td>
<td>1839</td>
<td>1582</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Benchmark</th>
<th>LEVC</th>
<th>LogsTM</th>
</tr>
</thead>
<tbody>
<tr>
<td>kmeans</td>
<td>2746</td>
<td>199</td>
</tr>
<tr>
<td>vacation</td>
<td>2048</td>
<td>48</td>
</tr>
<tr>
<td>labyrinth</td>
<td>2048</td>
<td>27</td>
</tr>
<tr>
<td>scca2</td>
<td>47299</td>
<td>360</td>
</tr>
<tr>
<td>yada</td>
<td>20400</td>
<td>98</td>
</tr>
<tr>
<td>intruder</td>
<td>1488</td>
<td>94</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Benchmarks</th>
<th>LEVC</th>
<th>LogsTM</th>
</tr>
</thead>
<tbody>
<tr>
<td>Vacation</td>
<td>50</td>
<td>48</td>
</tr>
<tr>
<td>Vacation</td>
<td>928</td>
<td>27</td>
</tr>
<tr>
<td>labyrinth</td>
<td>284</td>
<td>13878</td>
</tr>
<tr>
<td>scca2</td>
<td>316</td>
<td>360</td>
</tr>
<tr>
<td>yada</td>
<td>1616</td>
<td>98</td>
</tr>
<tr>
<td>intruder</td>
<td>351</td>
<td>94</td>
</tr>
</tbody>
</table>

Figure 6.5: Table showing execution characteristics of TM benchmarks. LEVC stands for limited early value communication.

slightly worse than LogTM. For most other benchmarks, our model always performs better than LogTM, approaching the performance of VP-TM in most cases. The best speedups are obtained for labyrinth (15%) and scca2 (30%).

For the vacation benchmark, our scheme does better than VP-TM. The reason for actually doing better than VP-TM even though VP-TM allows more concurrency is as follows: With VP-TM, increased concurrency can lead to increased number of cycles that can result in restarts (and/or mispredictions). Thus, the benefits from increase in concurrency are lost due to the increase in the number of restarts and speedup is limited. With LEVC, we control concurrency better, which results in lesser restarts (compared to VP-TM) and improved speedup. For other benchmarks like yada and kmeans, the increase in concurrency is almost matched by the loss in performance due to the increased restarts for LEVC and thus, we do not see significant performance different with LEVC, compared to the base LogTM system. Thus, LEVC can provide a controlled increase in concurrency to provide better performance than the base LogTM system with lesser hardware requirements than other proposed approaches.
Chapter 7

Realistic Design for VP-TM

7.1 VP-TM design

The VP-TM system from section 5.2 assumes the presence of a centralized memory-level value predictor. Some of the functions that it performs are:

• It can monitor load/store requests to memory addresses and create a history of recent store values.

• Upon detecting a pattern for store values, it can provide a predicted value on a NACK.

The above functions assume that the value predictor can see all requests and all NACK messages coming from all processors. This imposes severe requirements on the location as well as structure of the value predictor.

The goal of the VP-TM system was to determine if we could extend the concurrency of a TM system by detecting a pattern and providing predicted values to run conflicting transactions in parallel. During the design of the VP-TM system we did not simulate the latencies of messages that travel between the value predictor and the memory system because our objective was to figure out whether we can eliminate conflicts and improve scalability of TM by using the value predictor. The value pre-
The value predictor is present at the memory system because it needs to see all NACK messages and load/store requests to conflicting addresses. The value predictor also needs to be centralized to maintain correctness.

In this chapter, we focus on trying to simulate a value predictor-based TM system that has all these latencies accounted for. We try to simulate a more realistic scenario and figure out the performance impact of these extra messages. We first describe how we add various message latencies into our VP-TM system to make the simulations and results more realistic. Next, we describe our optimized system that tries to minimize the impact of these extra latencies on the performance of the system. Finally, we show performance results and improvements as we overlap prediction messages with other messages to reduce latency of predictions.

7.1.1 Structure of the Value Predictor

The value predictor is a single structure that resides at the memory level. To maintain conflict-serializability, the value predictor needs to know all conflicting addresses for all conflicting transactions. This is because when we allow transactions to forward uncommitted values, we are essentially creating an order for transactions to commit to maintain sequential consistency. An ordering among concurrent transactions implies ordering of reads and writes of one transaction with respect to other transactions in the order. This can be explained by figure 7.1. We can see that transaction 2 received value for Address X from transaction 1 and thus has to commit after 1 commits. Transactional correctness requires that load of Y from transaction 1 need to be complete with respect to memory before transaction 2. Thus, values can only be communicated from transaction 1 to 2. The value predictor needs to make sure that there is no value communication from 2 to 1. In the figure, we can see that since transaction 2 loads Y before 1 tries to read it in the same transaction, the value predictor signals an abort.

To maintain correctness for all transactions, the value predictor needs to make sure that any transaction that has received values from other transactions does not
end up being the predecessor to those transactions for any other address. This is the main reason that we need to have a centralized value predictor so that it can inspect all addresses that have speculative values communicated. All transactions that have received or communicated values need to be inspected to make a decision on whether it is safe for a transaction to communicate values to another without violating sequential consistency.
7.2 Changes to VP-TM

We add a new virtual network that is only used for messages to and from the value predictor. In this way, we keep the value predictor centralized and allow it to monitor all loads and stores to addresses and respond accordingly. We simulate the value predictor as being next to the directory that is in the middle of the system. For a 16 processor system, we place the value predictor at directory 7 in our configuration (0 - 15 directories). This is done to keep the maximum latency of message low and to avoid unfairly charging some processors higher latencies for value prediction.

We add a new virtual network in Ruby, the GEMS [26] memory model. This automatically accounts for bandwidth, latency and traffic considerations. The new design works as follows:

- Upon a NACK message, an ADD message is sent from the directory to the value predictor that contains the address, the requestor and the Nacking transaction processor id. The value predictor responds with PRED_DATA message for any conflict that can be predicted. This message contains the name of the processor that is the last predicted producer of the value.

- On a transactional store, an UPDATE message is sent from the directory to the value predictor to enable it to calculate stride for any address present in value predictor.

- During transaction commit, a VERIFY message is sent from the directory to the value predictor for all addresses that have values forwarded. This is in addition to the UPDATE message for all stores.

- Upon receiving a VERIFY, the value predictor sends out a PRED_SUCCESS or a PRED_FAIL message to the directory to indicate the status of the prediction. These messages are then forwarded to the respective processors.
7.2.1 Optimizations and Timing Issues

A naive implementation as described above can have a huge impact on the performance of the VP-TM system. The extra latencies of the predictor messages can potentially nullify performance gains from the improved concurrency with the VP-TM system compared to LogTM. In order to reduce their impact, we try to send these messages directly from the processors instead of sending them via the directories. This way we can overlap these messages with other coherence messages. However, this also gives rise to races between different value predictor messages that can lead to incorrect execution.

The ADD message can be sent directly from the Nacking processor. There are no races associated with sending an ADD message from different processors since once an address has been added, it will remain and all other ADD messages for the same address will be discarded. An UPDATE message can be sent directly from the processor that performs the store. This message also can be safely sent from any processor since the transaction model allows only a single transaction to write to a memory location and thus, only one processor will send an UPDATE message to the value predictor (the rest will try to get values predicted for loads.).

There may be a possibility of races between ADD and UPDATE messages. As shown in figure 7.2, a value predictor can receive an ADD message from another transaction before it receives an UPDATE from the predecessor. There are two solutions to this problem. The first solution is to have ACK messages from the value predictor to the memory system after UPDATES. The operation completes with respect to memory only after it completes with respect to the value predictor. This solution can slow down the commit of the store instruction. The second solution is easier to implement. If the value predictor receives an UPDATE message and it has predicted value for the same address, the value predictor triggers a misprediction for that address and sends out PRED_FAIL messages to the transactions that got the predicted values. In our implementation, we tried out the first solution first but there were no races detected. This may be due to the fact that any load behind the
Figure 7.2: Figure showing a sequence of operations (after a store (1) has performed by CPU 1) to show how messages can reach the value predictor in a different order. The UPDATE (1b), ADD (4b) and ADD (7b) messages can reach the value predictor in any order leading to incorrect execution.

updating store needs to go the predecessor transaction which then generates the Nack message. Hence, we implement the second solution that can perform better with no races detected even if it has a performance penalty detected for races.

The VERIFY message is in the critical path and its latency can slow down the commit and abort of transactions. We send this message directly from the processors that is performing the commit. since VERIFY messages do not need to slow down the current transaction which is trying to commit. There are no timing issues with sending the VERIFY message since verification of values takes place sequentially i.e. a processor can only commit once it receives a reply back from the VERIFY message and another processor will send a VERIFY only after its predecessor has committed.
7.3 Experiments and Results

For the value predictor simulated as explained above, we expect a slowdown due to the extra message latency that is incurred by predictor messages. We wish to evaluate the impact of these messages on the overall performance of the system. We present both the versions of the system. Our underlying system is the same as described in previous chapters.

Figure 7.3: Figure showing speedup results for 16 processors for different VP configurations and percent change in performance from naive to optimized version.

Figure 7.3 shows our results with the two versions of the value predictor. We call the VP-TM system described in Chapter four as the “Ideal VP” and the naive implementation of the value predictor which has the network latencies of value predictor messages simulated, as the “Naive VP”. The optimized version of the value predictor which tries to overlap value predictor messages with other messages is called as the “Optimal VP”. We compare all of these systems with the original LogTM system to better understand whether the presence of a realistic value predictor can help performance or whether the extra latencies from the messages nullifies gains made from
the value predictor.

We see from the results that performance drops for most applications compared to an Ideal system. With the exception of intruder and kmeans, however, we still see speedup in most applications. The line in the figure indicates the percent improvement from the optimized version of the value predictor compared to the naive implementation. We can analyze the applications better by dividing them into 3 categories

• Applications like yada and vacation that do not show significant improvement with the original VP-TM system. These apps do not show significant change in performance from the ideal system to the optimized system. This is mainly because there are not too many opportunities for value prediction and hence the small number of extra messages passed between the processors and the value predictor do make end up affecting performance.

• Applications like labyrinth, ssca2, raytrace and radiosity. These applications show significant performance improvements with VP-TM over the LogTM system. The performance for these applications drops due to extra latency of messages in the Naive VP design. These applications also show performance benefit from the optimized design implying that the overlapping predictor messages can really benefit a naive implementation.

• Finally, we have two applications kmeans and intruder which show minor performance improvement with the VP-TM system. The performance of these applications is worse than LogTM, even with the optimized design. Both the intruder and kmeans applications have a number of rollbacks and the accuracy of the value predictor is not good as can be seen from the characteristics in table 5.8. The loss in performance is primarily because due to the extra latency of messages in the optimized version compared to the ideal design. Transactions take longer to detect mispredictions which reduces concurrency and performance.
Thus, from the results, we can see that in most cases the optimized version of the value predictor can provide good performance compared to the LogTM system. This way, we make a case for including the value predictor to convert conflicting accesses among concurrent transactions to ordered accesses using predicted values. We present our conclusions in the next chapter along with a few ideas as part of future work.
Chapter 8

Conclusion and Future Work

Table 8.1: Table comparing early value communication approaches described in this paper.

<table>
<thead>
<tr>
<th>Characteristic</th>
<th>VP-TM</th>
<th>LEVC</th>
<th>DATM</th>
</tr>
</thead>
<tbody>
<tr>
<td>Enabling Mechanism</td>
<td>Last value prediction</td>
<td>Prediction only for first conflict</td>
<td>Cache line forwarding</td>
</tr>
<tr>
<td>Speculation Procedure</td>
<td>Predicts data from VP</td>
<td>Piggyback data on Nack</td>
<td>Forward cache line</td>
</tr>
<tr>
<td>Protocol changes</td>
<td>Extra Messages for predictor</td>
<td>Minimal changes: Piggyback data on Nack</td>
<td>New states to enable forwarding and broadcast a vector of processor ids</td>
</tr>
<tr>
<td>Extra Hardware</td>
<td>Centralized Value Predictor and extra buffer space on chip</td>
<td>Extra buffer space on chip</td>
<td>Extra buffer space on chip</td>
</tr>
<tr>
<td>Cycle Detection</td>
<td>Using centralized predictor</td>
<td>Prohibit a cycle by limiting forwarding</td>
<td>Vector of ids broadcast to every processor</td>
</tr>
<tr>
<td>Issues</td>
<td>(1) Unpredictable conflicts. (2) Unified VP</td>
<td>Limited Concurrency</td>
<td>(1) Depends on position of writes in transactions. (2) Modified coherence protocol</td>
</tr>
</tbody>
</table>
8.1 Comparison of Early Value Communication Schemes

Table 8.1 presents a comparison of the three early value communication schemes that we talk about in this paper. We differentiate between the schemes in terms of data communicated, coherence protocol changes required, additional hardware to maintain conflict-serializability and issues with each.

8.2 Conclusion

Multi-core processors are here to stay and we need to switch to parallel programming to make best use of the hundreds of cores that will be available soon. Transactional Memory [14, 12, 1, 37, 31] has been put forth as a scheme to improve performance of parallel programs, mainly database applications. With this work, we try to extend the scalability of existing TM systems and hope that it will increase the adoption of TM programming. We show that existing TM approaches are neither scalable nor good for certain scientific applications with sequential data structures.

We detect a pattern among conflicting accesses and try to put forth a case for using a value predictor that can extend the concurrency of transactions and usability for certain scientific applications. We find that the simple stride-based value predictor can boost the performance of a TM system even with a low prediction accuracy. Even with increased number of aborts, we still manage to see speedups at 16 processors due to reduced conflicting addresses in the program. We next present a limited value forwarding scheme whose objective is to reduce the hardware complexity from early value communications. We find that we can still get performance benefits with reduced hardware complexity of LEVC. Finally, we present a realistic design of a VP-TM system and evaluate its performance against the Ideal VP. With the realistic design, we wish to provide a final confirmation of the value of VP-TM showing that even with a realistic design, we can get performance benefits on most applications.
Thus, with this work, we try to extend the usefulness of transactional memory by eliminating conflicts and improving scalability of applications on transactional memory systems. We believe this can increase the adoption of TM for parallel programming. So far, hardware designers have not adopted transactional memory mainly because of the hardware requirements that it can impose. We believe that with a broader range of applications showing improved performance, we can make a case for implementing TM in hardware. Hopefully, the performance benefits from adding VP to a TM system are enough so that designers will implement TM in hardware with a value predictor.

8.3 Future Work from Early Value Communication

As part of future work, we try to make a case for compiler or programming language support to perform early value communication within TM systems. As mentioned earlier, early value communication results in a commit order among concurrent transactions. This is the order in which every load and store needs to complete with respect to memory to maintain sequential consistency. The VP-TM satisfies this condition by aborting transactions whenever there is a cycle detected. This reduces the performance benefits that can be obtained by early value communication and it also demands extra hardware for cycle detection. We believe that using some programmer/compiler help we can detect such cycles in advance and avoid early value communication. We envision some programmer help like in Shavit’s software transactional memory work [45] where they list the order of accesses within transaction. Otherwise, the compiler can be used to detect such accesses and provide hints to the hardware about transactions that are unsafe for early value communication.

We can also use hardware to solve the problem of cycles reducing performance with early value communication. We propose using a hardware dynamic transaction conflict table that provides the list and order of memory accesses that the transaction
is likely to access. This table can be provided at each processor and entries are added to it upon detecting a Nack. Once a transaction gets a predicted value, it uses the its own access table and the access table of the transaction it is waiting on, to decide whether to wait for this request or perform this request directly in memory. If the transaction notices the address on both lists and the producer has not performed the request yet, it will try to wait for the producing transaction to perform the request before placing its request. Otherwise, the request is made to the memory. This way we can reduce some of the cycles from early value communication. Our goal is to convert TM execution in hardware to a conflict-serializable execution that provides the illusion of TM style execution for ease of programming and maximum concurrency.
Bibliography


