# A Case for Using Value Prediction to Improve Performance of Transactional Memory

Salil Pant
Center for Efficient, Scalable and Reliable Computing
North Carolina State University
smpant@ncsu.edu

Gregory Byrd
Center for Efficient, Scalable and Reliable Computing
North Carolina State University
gbyrd@ncsu.edu

## Abstract

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. TM avoids many of the problems associated with mutual-exclusion-based synchronization and can make writing parallel programs relatively easier. 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 paper, we look into the nature of the shared data involved in conflicts for TM systems. We find that for a certain class of applications, shared-conflicting data is often updated in a predictable manner by different transactions. We propose using a value predictor to capture this predictability for such data structures and increase overall concurrency by satisfying loads from conflicting transactions with predicted values, instead of stalling. In this paper, we present a possible design and implementation of TM system with a value predictor to avoid conflicts between concurrent transactions. 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 minimizing stalls and rollbacks due to conflicts.

## 1. Introduction

Current lock-based mutual exclusion schemes are not suitable for fine-grain synchronization because of the overhead they tend to generate (Kagi et al. 2-4 Jun 1997; Mellor-Crummey and Scott 1991). Transactional Memory (Herlihy and Moss 1993; Rajwar and Goodman 2002; Hammond et al. 2004; Ananian et al. 2006; Moore et al. 11-15 Feb. 2006) 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. If accesses from different threads conflict, all but one thread, abort/stall speculative execution and restart the critical section. If a transaction reaches the end of the critical section without encountering any conflicts, it can "commit" its speculative data and turn non-speculative.

For parallel programmers, a transaction is presented as *an abstraction that provides for atomic execution of a block of code with atomicity, consistency and durability guarantees*[1]. TM systems also provide wait-free properties to critical sections, that avoids dangerous scenarios like priority inversion, thread-starvation and deadlocks etc. These scenarios are very difficult to reason about during parallel programming. The above features of TM systems make writing parallel programs relatively easier than locks. In most cases the programmer has to simply identify critical sections and label them as transactions.

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 (Herlihy et al. 2003) using special data structures in memory or by custom hardware (Herlihy and Moss 1993; Hammond et al. 2004; Ananian et al. 2006; Moore et al. 11-15 Feb. 2006). 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.

Performance of TM 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 conflicts (Bobba et al. 2007; Hammond et al. 2004). For critical sections that are heavily contended, TM systems can suffer frequent rollbacks due to conflicts and lead to degraded performance. This problem only gets worse with increasing number of processors. Past research work has often proposed switching to a conservative locking scheme for heavily contended critical sections to reduce critical section memory traffic (Rajwar and Goodman 2002). In this paper, we consider a different approach for solving this problem.

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 and allocating memory to different processes. Within the Splash suite (Woo et al. 1995), benchmarks like Raytrace, Radiosity, Cholesky are among the few that use a queue structure for task/memory scheduling. The STAMP benchmark suite (Minh et al. 2008) contains many benchmarks that traverse queues or linked-lists during parallel execution. 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



[1] Fundamentally TM does not provide these guarantees, they are provided by the underlying system or the programming language. Conceptually speaking, TM only guarantees atomicity.

the queue size etc. Due to this, concurrent transactions conflict and operations can get serialized.

Upon closely observing the shared data involved in bookkeeping, we find that it is often updated by each transaction in a predictable manner. In this research, we observe 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 the order they were predicted 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.

This research makes the following contributions:

- We present a case for implementing a memory-level value predictor that can parallelize conflicting transactions for a TM system.
- We present a possible design of the value predictor and highlight the issues that can arise during the design of a TM system with the value predictor.
- We present a limit study using a queue-based microbenchmark to get an idea of the best case and worst case performances.
- We also present benchmark results to show that opportunities to exploit predictable values exists and can be used to improve performance over base TM systems.

Our initial results look promising; Overall performance of the TM system improves with the addition of the value predictor. We get significant speedups ( in most cases over 100%) by using a value-predictor-based TM model over the base LogTM model. We observe most of our speedup arises by reducing the number of conflicts and aborts in TM systems. Also, mispredictions are not very expensive and do not degrade performance significantly. The rest of the paper is organized as follows: We discuss transactions in detail in the next section. In section 3 we put forward our case for having a value predictor. In the next section we present one possible design of such an enhanced TM system. Section 5 presents our initial simulation results. Finally we present our conclusion and future work.

## 2. Transactional Memory

Transactional Memory was first proposed by Herlihy et al. (Herlihy and Moss 1993) as a lock-free synchronization mechanism supported by hardware. Since then, 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 (Rajwar and Goodman 2002; Rajwar et al. 2005; Hammond et al. 2004; Ananian et al. 2006; Moore et al. 11-15 Feb. 2006; Herlihy et al. 2003).

In this work, we look mainly at hardware-supported TM (HTM) systems since they provide better performance and demand lesser effort from the programmer compared to STM systems (Rajwar et al. 2005; Herlihy et al. 2003; Kumar et al. 2006). HTM systems use hardware to perform the two important functions of Conflict Detection and Version Management. 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.
2. A history buffer or checkpoint capability to restart the processor from the beginning of the transaction.
3. Extra read and write bits in the cache to detect transactional read and write accesses. Modifications in the cache coherence protocol to detect and resolve conflicts.
4. A buffer to hold multiple versions of transactional data.

Performance of HTM systems often exceeds performance of lock-based synchronization for many 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, we have avoided spinning on the lock and improved concurrency by optimistic execution. However, performance of TM system suffers if transactions incur frequent conflicts which may lead to aborts. 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 possibly reissue requests for all shared lines that it needs ownership, to the memory system. To minimize the number of aborts, some researchers have proposed back-off schemes to reduce the number of conflicts (Herlihy and Moss 1993; Moore et al. 11-15 Feb. 2006). In this work, we take a different approach. Instead of reducing concurrency by slowing down retries, we propose to increase concurrency by running transactions concurrently with predicted values. We describe our proposal in the next section.

## 3. Case for Value Prediction

We notice that many parallel scientific applications consist of a data structure such as a linked list or a queue that performs task management, memory management etc. For example, Cholesky and Raytrace from the SPLASH (Woo et al. 1995) suite of benchmarks maintain a linked list for memory management purposes. Radiosity maintains queues for dynamic task scheduling. 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 using locks. We explain this further with the help of a simple example as shown in Figure 1. Note that the data structure used by a real application would be a modified version of our example.

Typical members of a queue data structure are the head and tail pointers and the size of the queue (Cormen et al. 2001). Common operations provided by the queue object are inserting new elements and deleting elements, typically from the head. Operations on the queue are as shown in Figure 1. Both the insert and delete operations can potentially manipulate the head and tail pointers and the queue size, while inserting and deleting elements that may be distinct. This means that every operation on the queue needs to be serialized with every other operation to maintain atomicity. A parallel version of the queue object using transactions is as shown in Figure 2.

The transactional version avoids conservative locking, but performs no better since within both the operations, insert and delete, at least one shared member of the queue object is updated, which will lead to conflicts for concurrent transactions leading to stalls or aborts which reduce concurrency.

Since different processors maybe inserting or deleting different elements, which can be done in parallel, the only thing limiting potential parallelism for the insert and delete operations is the update

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

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

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

**Figure 1.** Example showing shared operations on the Queue

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

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

**Figure 2.** Enqueue and Dequeue Operations using Transactions

of the shared members of the queue object. To improve scalability for such structures, we look into the nature of shared operations that create conflicts. We notice that the insert and delete operations for a queue object update the head, tail and size members of the shared queue object. The head and tail pointers move by a size equal to the size of element and the queue size increments or decrements by 1. Thus, each operation changes the value of shared data by a fixed amount. This behavior can be exploited to improve concurrency. If we can predict future values from the insert and delete operations and feed those values to concurrently running transactions, we can improve the concurrency of the program.

The above operations on the queue are simplified versions of the insert and delete operations that occur in most applications. If an application is using a linked-list, it may delete any element and not just the head. In such a case, there may be an extra loop in the delete procedure to search for the element to delete and extra previous or next shared pointer updates. Even in this case, we can improve concurrency because if the elements being operated upon are distinct, they may update different shared memory pointers which may not conflict. A conflict for any shared memory operation not being predicted will be handled as per the rules of the underlying TM model and may lead to a stall or aborts. However, we believe that such conflicts will be minimal and value prediction of the serializing members will lead to improved concurrency.

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 and commit and turn non-speculative, without interference from the value predictor. From initial microbenchmark results, we observe that we do not lose much performance in the worst case with many mispredicted values, 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.

Also, for such applications on a TM system, with increasing number of processors, the number of conflicts in the system tend to increase, thus, TM systems will not provide scalable performance for such applications. Our scheme tries to minimize the impact of conflicts in such cases and thus, has the potential to provide better performance. In the next section, we describe a possible design of such value predictor in detail.

## 4. Design of the enhanced TM system

### 4.1 Base LogTM System

The base multiprocessor system we assume is a shared memory system using directories to maintain cache coherence over a point-to-point interconnection network. We use the LogTM model proposed by Moore (Moore et al. 11-15 Feb. 2006) et al., built over on top of the Simics full system simulator (Magnusson et al. 2002) and the GEMS (Martin et al. 2005) memory model, to support transactions, as our base TM system over which we develop our value predictor. Borrowing terminology from Bobba (Bobba et al. 2007) 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 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 (Rajwar and Goodman 2002) for transactions that are retained during 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 the nacked transaction is aborted. In the LogTM model, an abort leads to a rollback of memory using the log and a processor checkpoint. Upon com-

pleting 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 because of two reasons:

- The LogTM model has been provided along with the GEMS suite and accurately models the timing of transaction aborts and commits.
- It uses a Nack-based coherence protocol which is more supportive of a value predictor than a deferral-based scheme (Rajwar and Goodman 2002). Nacks are visible to memory and can be used to trigger the value predictor to make a prediction.

### 4.2 Identifying values for predictions

Identifying an address for value prediction is easy; Whenever a directory receives a Nack, it inserts the conflicting address into the value predictor. This means subsequent stores to that address are captured by the value predictor to generate stride. If the value predictor is full, then we can have multiple ways of evicting an existing entry. We can either remove the oldest entry or the entry with the least number of accesses. In our experiments, we simply remove the entries in order because our VP profiling data shows that we never have more simultaneous sharers than the number of VP entries.

We assume a centrally-located memory-level 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.
- 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.
- The value predictor also performs the important function of avoiding "mutually dependent" transactions which can lead to a deadlock. This is the main reason the predictor is global and at the memory.
- As transactions commit, stores to addresses inside the value predictor are committed[2] to memory, in order, and their history is updated in the value predictor. Any predictions that were made for those entries are then verified.
- 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.

### 4.3 Organization of the Memory-level value predictor

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, as shown in fig 3b 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. The number of recent values to save depends on the amount of history desired to make correct predictions. For this research work, our value predictor is stride-based and so, we do not save more than 2 most recent written values. For a "confident" prediction, we can save the last 3 recent stores and predict only if the stride is preserved across the 3 stores. The maximum number of speculated values needed, per entry, is equal to the number of processors in the system minus 1. However, similar to the limited pointer directory structure (Chaiken et al. 1991), we can avoid having more than 5 speculated values per entry without sacrificing performance. If there are more than 5 sharers at a time, we can stall those transactions.

We also have an abort flag for each entry which is set if any of sharers that received a predicted value had to abort or if the prediction was incorrect. This is done to limit the number of mispredictions, in case, the stride behavior of the address changes. If the abort flag is set, there will be no more predictions made for the address. The abort flag is reset on any subsequent store to that address.

So far, in all of our experiments, the total number of data values predicted, simultaneously, has always been less than 5. Thus, we envision the number of entries in the value predictor to be fixed to around 5 or 10.

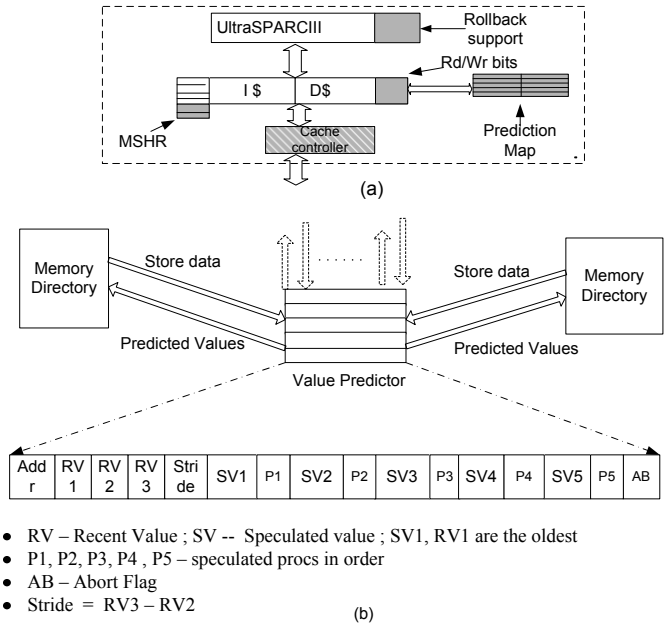



**Figure 3.** Part (a) shows the changes needed to the micro-architecture to support VP-TM. Part (b) shows the structure of the VP entry.

### 4.4 Operation of the Value Predictor

The VP-TM also demands a few changes to the Nack-based coherence protocol used by the LogTM model. Figure 5 shows the protocol actions for the VP-TM system. The sequence of steps to get a predicted value is as follows:

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 requestor load with a predicted value and the value predictor will add the cpu, and the value to the speculated values list in the entry corresponding to the address.

[2] In the LogTM model, on a commit, we discard the old values.

3. The MSHR entries for the load operation at the caches are not cleared. Thus, at the memory level, the load operation still "exists".
4. The memory system keeps retrying the request until the owner transaction commits or aborts.
5. All read/write requests coming from other processors are now forwarded to the latest speculated owner of the line as indicated by the value predictor.
6. The transaction with the predicted value can continue to execute but it cannot commit its data until the prediction is verified. In our scheme, we stall the transaction if it reaches the end and the predicted data has not been verified yet.
7. Once the owner commits, the retry can get back non-speculative data which is then compared with the predicted data and the processor is informed of the result.
8. If the prediction succeeded, the value is "passed on" and the processor that got the predicted value becomes the new owner of the line. If all predicted values have been verified, a processor can commit a transaction when it reaches the end of the transaction.
9. If the prediction was wrong, the data value is returned to the memory and the processor needs to abort the transaction. The value predictor also informs the directory to change the state of the line to invalid.

The logging-behavior of addresses that are present inside the value predictor is opposite of the LogTM model. Stores to these addresses are not propagated to memory until commit. This is done to ease the design because the *the load value predicted by the value predictor is the last store value for that address that will be seen at the end of the Nacking transaction* and if we update the memory on with multiple such stores, inside the value predictor, it will need to know when to use finally use the value. If a transaction received predicted values, the VP-TM system also need not log data for addresses because the predicted value may be incorrect and it should not be written back to memory during an abort. We implemented a small buffer of 5 entries, near the transaction log, that identifies predictable addresses accessed by the transaction and has new data that was generated by stores to these addresses. We call this the "prediction map" as shown in figure 3a. When the transaction commits, this map is used to flush store data back to memory. Since the size of the buffer is small, we do not anticipate that this will slow commits down significantly. Note that the number of entries in this buffer need not be the same as those in the value predictor. The value predictor can be bigger to allow more parallelism.

Value prediction creates an ordering among stores from different processors to a memory address. Whenever a transaction gets a predicted value, it becomes the "speculative owner" of the line, even if the prediction has not been verified yet. 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 multiple speculative owners, the directory consults the value predictor to get the processor to which the current request/retry must be forwarded. Figure 5d shows an example of requests from multiple speculative owners getting forwarded to their respective owners.

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 the directory. 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.

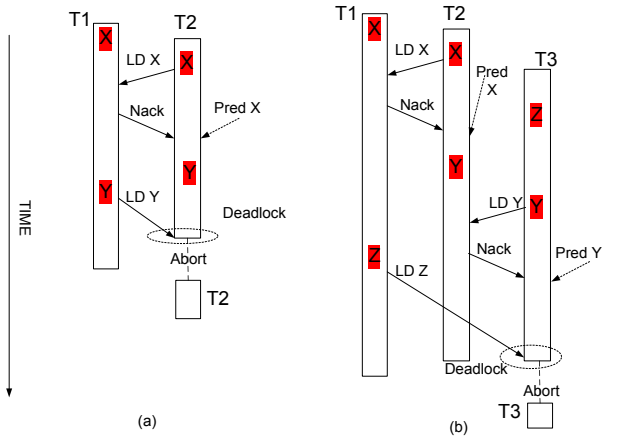
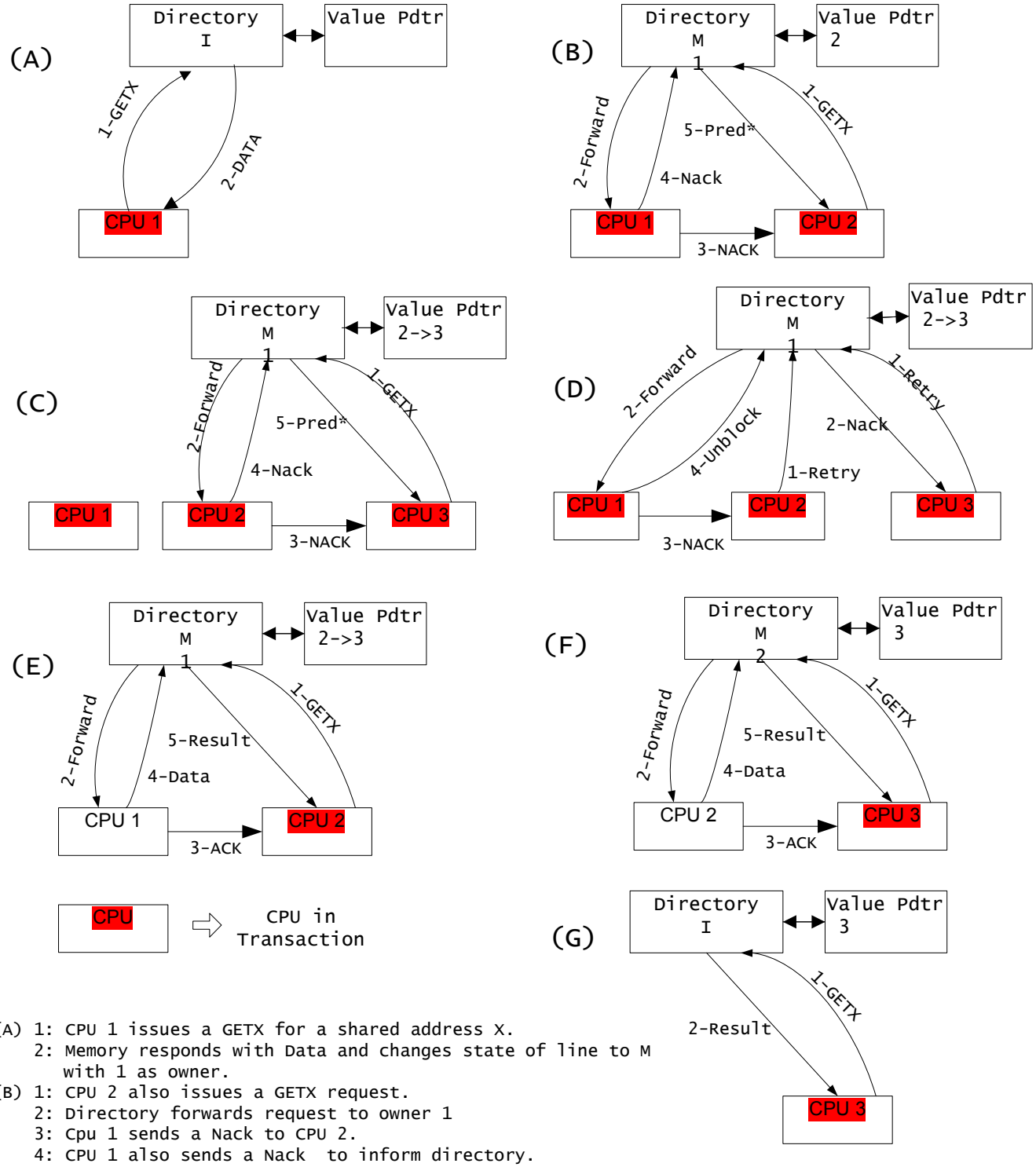



**Figure 4.** Figure showing the various deadlock cases that arise due to value prediction. X, Y and Z are shared conflicting addresses. Shading indicates load-exclusive accesses within transactions.

### 4.5 Dependencies Among Predicted Transactions

If a transaction receives a predicted value it cannot commit until its prediction is verified, usually by the transaction 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 to see if the nacking processor has ever got 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 recieved predicted values. This information is needed to decide whether a creating a predicted value for a given processor will lead to a deadlock.

In this design, it should be noted that we predict values for integers and pointers in parallel programs. While predictions are only made for up to 4 bytes of data, to merge the value predictor with the caches and 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 is because the memory system always maintains coherence for cache line sizes. This condition 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. We avoid this scenario by identifying such a load to predicted cache lines and stalling until the prediction is resolved. In our experiments, we do not see a significant slowdown due to such stalling. We can also pad the data




(A) 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 commmits 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.** Coherence Protocol Actions with VP-TM

structure elements, manually, such that they do not end up on the same cache line [3].

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:

- Not all shared conflicting data is predictable. This means that, while some of the shared values will be predicted, transactions will still stall or abort due to Nacks for the 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.

## 5. Performance Evaluation and Results

In this section, we try to verify our claim that a value predictor-based TM can improve concurrency for a TM system. We implement a value predictor on top of LogTM model provided by Moore (Moore et al. 11-15 Feb. 2006) et al. based on the GEMS (Martin et al. 2005) suite which is driven by Simics (Magnusson et al. 2002) 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 as shown in figure 3b.
- 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.

For evaluation purposes, we use a microbenchmark to isolate the performance benefits or losses from using a value predictor. Processors execute critical sections inside multiple loop iterations for the microbenchmark. To ensure load balancing, we introduce some delay after the critical section is exited by a processor. The amount of delay outside of the critical section is twice the delay inside the critical section. The microbenchmark performs $2^{10}/n$ operations on a shared queue. It is the same as the example outlined in the section 3. We have 2 different versions of the microbenchmark. The first version performs only insertions while the second version randomly inserts or deletes elements from the head. With this microbenchmark, we want to evaluate performance benefits in a scenario where we have serializing members of the queue object.

[3] This is a popular technique to avoid false sharing.

| Unit | Value |
|---|---|
| Processor | single-issue, in-order, 1GHz |
| L1 Cache | 16K, 4-way, split,1-cycle hit latency |
| L2 Cache | 2M, 8-way, unified,10-cycle hit latency |
| Directory | 80-cycle latency |
| Memory | 4GB |

**Table 1.** System Parameters for our simulations

Performance may still not be ideal because there might be other data conflicts that may limit speedup. For ex. conflicts due to updating next pointers of queue elements by concurrent transactions. The two versions of the queue microbenchmark help us in obtaining the best and worst case performance numbers for VP-TM.

We compare performance of our VP-TM with the LogTM model to get an idea of the performance by adding the value predictor. The system parameters are as shown in Table 1.

To demonstrate the potential of VP-TM, we picked up a few benchmarks from the SPLASH suite (Woo et al. 1995) and the STAMP suite (Minh et al. 2008) of benchmarks. For the splash applications, we simply replaced critical sections with begin and end transaction primitives. We used Simics "magic" instructions to for this purpose. These benchmarks have been picked because they use a linked data structure within the parallel section and, with the TM model, updates to the linked data structure has the potential of limiting speedup. With VP-TM, we wish to see if we can improve the performance of these benchmarks. Table 2 shows the benchmarks along with 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 due to simulation time concerns.

### 5.1 Predictions, Accuracy and Speedup

We show speedup numbers (over sequential execution) and number of aborts/restarts. For the microbenchmark, we also show the accuracy of prediction. Predictions are divided into 3 types:

- *Good Predictions* Predictions that are were 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 even before they were verified. Some of the causes of such aborts might be deadlocks 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 even an abort which can lead to performance degradation even with highly accurate VP. Similarly, aborted predictions does not always indicate poor performance since they represent wasted work when the transaction would have stalled anyway. In general however, 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 max number of simultaneous predictions on a single entry. While increased number of predictions are capable of increasing concurrency, they can also lead to cyclic dependencies as discussed in section 4.1. We vary

| Benchmark | Suite | Input | Units Measured | Serializing Object |
|---|---|---|---|---|
| RAYTRACE | SPLASH | car | parallel section | memory manager list |
| RADIOSITY | SPLASH | batch | 1 task | task-queue |
| Intruder | STAMP | -a10 -l4 -n1024 -s1 | parallel section | packet queue |
| Labyrinth | STAMP | random-x24-y24-z3-n1024 | parallel section | pathvector list |

**Table 2.** Benchmark Parameters

the max number of predictions from 1 to 5 to see if increased predictions always result in better performance. In our figures, we mention the different cases by the maximum number of predictions allowed in each case.

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 value predictor and, instead, using magic instructions in Simics, we pass these addresses to the value predictor. 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.

### 5.2 Performance Analysis

Figures 6a and b show the speedups for 1, 2, 3 and 5 predictions per address (y-axis on the left). It also shows percentage of total transactions that restarted for 1 of the VP cases[4] and the LogTM model, for 4, 8 and 16 processors(y-axis on the right). The dotted bar in the graph indicates prediction accuracy for the 2 predictions per address case using the right y-axis. As shown in figures, our VP-TM system has improved performance of the microbenchmark for both the cases. Looking at figure 6a, we can see that the value predictor can easily capture the predictability for this data structure and provide good performance by reducing restarts. However, even for the random insert/delete microbenchmark, our VP-TM system performs better, albeit not as good as the 1st case, than the base LogTM system even with a low prediction accuracy as seen from figure 6b. We attribute this performance benefit to the improved concurrency with the VP-TM system that can lead to a reduced number of aborts/restarts and thus 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 are limited to 2 or 3. This is because with increased number of sharers we can have a cascade of mispredictions[5] that get resolved in order, reducing concurrency and degrading performance in such a case. 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.

### 5.3 Application benchmark results

Now that we know that the value predictor works on this data structure, the next step is to evaluate the performance for a transactional benchmark that utilizes such a structure to see if speeding up accesses around the queue can lead to overall performance improvement. For Raytrace and Radiosity benchmarks, we only ran experiments up to 3 predicted sharers because of simulation time concerns.

For the benchmarks we studied, VP-TM performs better than the Log TM system for all cases even as the percentage of good predictions are as low as 13% in Radiosity and Intruder benchmarks to as high as almost 70% in Labyrinth benchmark from the STAMP suite (Minh et al. 2008). Figure 7 shows speedup numbers for the 16 processors case for the LogTM model and the VP-TM model with 1,2,3 and 5 max-predicted-sharers per address. Some predictions get aborted before they can be verified. Hence, the total percent of good and bad predictions does not add up to 100. Overall, the 2 max-predicted-sharers case is the best performer across all benchmarks. As shown in the figure, VP-TM improves the speedup for all benchmarks in most cases over 100%. With Labyrinth, VP-TM improves the speedup up to 11 times compared to the base LogTM case. The speedup numbers can be explained by looking at the transactional characteristics of these benchmarks from table 8. It can be seen that for the LogTM mode with no predictions, the number of Negative acknowledgments (Nacks) and aborts are high leading to poor performance. VP-TM succesfully manages to convert some of the Nacks to concurrent execution leading to better performance.

The coverage column indicates the percentage of conflicting accesses that were successfully predicted. As we can see Labyrinth has good coverage as well as high prediction accuracy which explains it stellar performance. Intruder, on the other hand, has poor coverage and low prediction accuracy which leads to poor performance. The benchmarks Raytrace and Radiosity have low coverage and accuracy, however, even that is sufficient to improve concurrency and reduce Nacks and Aborts for those cases. Performance is good also because Radiosity and Raytrace are not as transaction-intensive as the other two benchmarks from the STAMP suite.

The performance of all benchmarks with VP-TM is satisfactory except for the Intruder benchmark. The performance improvement of VP-TM over TM for Intruder is only 10 to 15%. We know that Intruder uses a queue-based structure for the incoming packet stream. To identify the predictability of this structure for this benchmark, we manually locate predictable members of this queue structure, namely the head, tail and the size, in this benchmark and use magic instructions as a way to add these members to the value predictor during program initialization. We also switch off the dynamic addition of entries at Nacks to the value predictor. We compare this static approach to the usual case for VP.

The results are as shown in figure 9. From the results 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

[4] 2 predicted sharers per address in our case, since it is the best performing.

[5] The abort flag in the VP per entry is specifically used to avoid such a scenario.

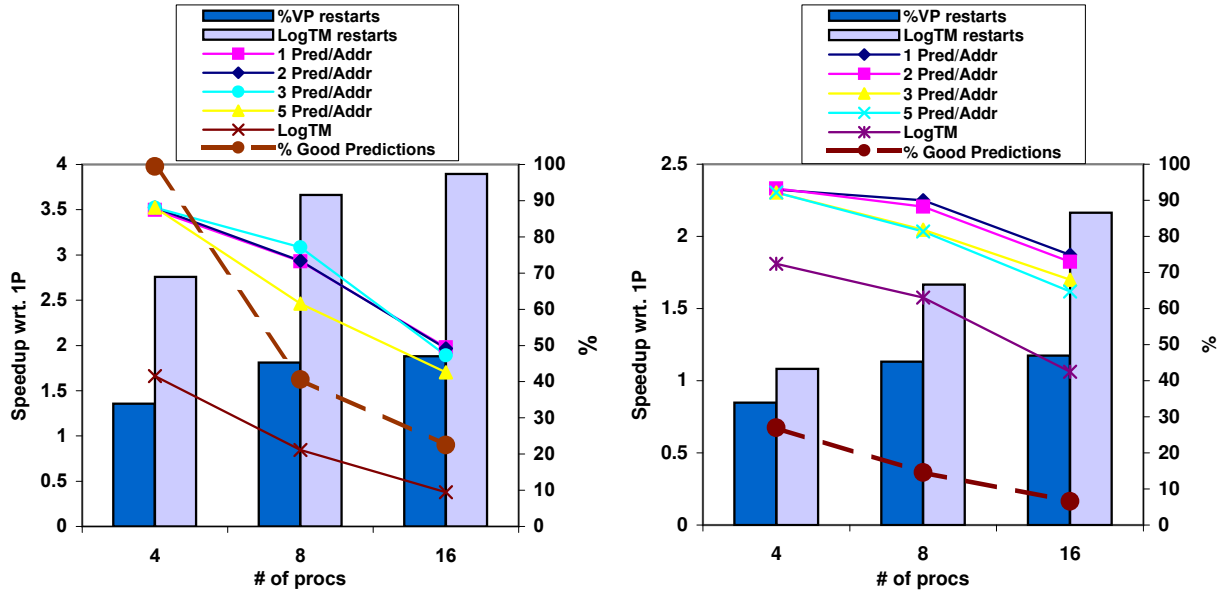



**Figure 6.** Results for the Queue Microbenchmark: Speedup (lines), % Restarts (bars) and % Prediction Accuracy (dotted-line).

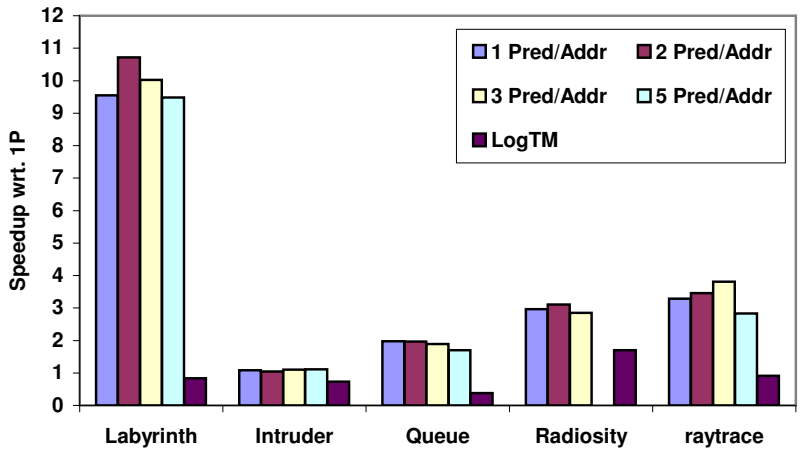



**Figure 7.** Speedup Results for 16 processors with different max-predicted-sharers

degradation is the addition of certain conflicting addresses to the value predictor in the dynamic case, 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, however, we show that in some cases, with manual identification, we can further improve the performance of VP-TM.

### 5.4 Rationale, Complexity and Usefulness of VP-TM

From the results section, we observe that a TM system with value prediction outperforms a Log-based TM system for almost all benchmarks even with low prediction accuracy. We believe the main reason for this is the reduction in the number of aborts with the value predictor and the improved concurrency at different points during parallel execution. This also means that bad and aborted predictions do not affect performance of TM severely. These two points are motivation to include the value predictor as part of HTM systems in spite of the increased hardware requirements. Also, with the advent of CMP architectures, parallel programming is expected to be used frequently. While TM makes writing parallel programs easier, with increasing number of processors, the TM model cannot be expected to scale up. TM relies on optimistic speculative execution, something which is not a safe bet with increasing number of processors. With value prediction in TM, we can not only improve concurrency, but we also reduce the number of aborts, that can reduce memory traffic over the system bus, further improving performance.

We do not expect VP-TM to benefit every application that is expected to run on an HTM system. For some applications, like

| | | | | | | Total predns | | % Good | %Bad | Addresses |
|---|---|---|---|---|---|---|---|---|---|---|
| | CPUS | TXNS | Commits | ABORTS | NACKS | & coverage | | Predictions | Predictions | predicted |
| Labyrinth | 16 | 15926 | 2048 | 13878 | 52561 | 0 | | 0 | 0 | |
| | 16 | 2266 | 2050 | 216 | 1021 | 698 | 99% | 69.77 | 3.1519 | 3 |
| | | | | | | | | | | |
| Intruder | 16 | 4635 | 1720 | 2915 | 7062 | 0 | | 0 | 0 | |
| | 16 | 2027 | 1720 | 307 | 3343 | 444 | 40% | 27.93 | 1.8018 | 17 |
| | | | | | | | | | | |
| Radiosity | 16 | 10026 | 2048 | 7978 | 41809 | 0 | | 0 | 0 | |
| | 16 | 2494 | 2048 | 446 | 19032 | 2928 | 45% | 13.35 | 31.182 | 5 |
| | | | | | | | | | | |
| Raytrace | 16 | 123039 | 44736 | 78300 | 139494 | 0 | | 0 | 0 | |
| | 16 | 55392 | 44260 | 11132 | 14233 | 1502 | 50% | 15.65 | 14.78 | 6 |

**Figure 8.** Table Showing transactional characteristics of application benchmarks for LogTM and VP-TM for each benchmark. The VP numbers are for the Max 2 predicted sharers.

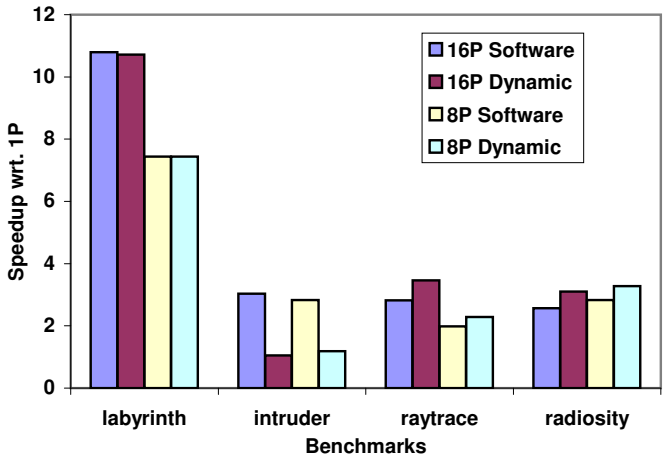



**Figure 9.** Static vs. Dynamic Identification of predictable values

SPECJBB 2000 (spe), TM has the potential to scale to a large number of processors. However, for many other applications, especially the queue-based applications that we have discussed, TM cannot provide any benefit over standard lock-based approaches. Such applications can be found be found in most standard parallel benchmark suites which means they do form an important set of applications that are expected to run on CMPs. For such applications, VP-TM can be used to provide improved performance.

## 6. Related Work

Although similar work has been performed for TLS (Cintra and Torrellas 2002; Steffan et al. 2002), there is no research work that performs value prediction to improve concurrency of transactional memory programs. Dependence-aware transactional memory (Ramadan et al. 2009) is the closest to our work. In their work, they forward data within transactions as it is produced. Although they do not need a value predictor, multiple writes to a value can lead to restarts. They also demand significant changes to the coherence protocol compared to VP-TM.

## 7. Conclusion and Future Work

Transactional Memory has been put forth as a scheme to improve performance of parallel programs, mainly database applications. With this paper, we try to put forth a case for using a value predictor that can extend the concurrency of transactions for a certain class of 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. With increasing number of aborts, we still manage to see speedups up to 16 processors.

Future work for this paper to evaluate back off techniques to improve speedup in presence or multiple aborts with increasing number of processors and control the number of predictions made to reduce the number of aborted predictions. We would also like to implement a way to identify predictable addresses in hardware without programmer intervention.

## Acknowledgments

This research was sponsored in part by the National Science Foundation, under contract CCF-0702294.

## References

Specjbb 2000. http://www.spec.org/jbb200.

C. Scott Ananian, Krste Asanovic, Bradley C. Kuszmaul, Charles E. Leiserson, and Sean Lie. Unbounded transactional memory. *IEEE Micro*, 26 (1):59–69, 2006. ISSN 0272-1732. doi: http://doi.ieeecomputersociety.org/10.1109/MM.2006.26.

Jayaram Bobba, Kevin E. Moore, Haris Volos, Luke Yen, Mark D. Hill, Michael M. Swift, and David A. Wood. Performance pathologies in hardware transactional memory. *SIGARCH Comput. Archit. News*, 35 (2):81–91, 2007. ISSN 0163-5964. doi: http://doi.acm.org/10.1145/1273440.1250674.

David Chaiken, John Kubiatowicz, and Anant Agarwal. Limitless directories: A scalable cache coherence scheme. *SIGPLAN Not.*, 26(4):224–234, 1991. ISSN 0362-1340.

Marcelo Cintra and Josep Torrellas. Eliminating squashes through learning cross-thread violations in speculative parallelization for multiprocessors. In *HPCA '02: Proceedings of the 8th International Symposium on High-Performance Computer Architecture*, page 43, Washington, DC, USA, 2002. IEEE Computer Society.

Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. *Introduction to Algorithms, Second Edition*. The MIT Press, September 2001. ISBN 0262531968. URL `http://www.amazon.ca/exec/obidos/redirect?tag=citeulike09-20\&path=ASIN/0262531968`.

Lance Hammond, Brian D. Carlstrom, Vicky Wong, Michael Chen, Christos Kozyrakis, and Kunle Olukotun. Transactional coherence and consistency: Simplifying parallel hardware and software. *IEEE Micro*, 24 (6):92–103, 2004. ISSN 0272-1732. doi: http://dx.doi.org/10.1109/MM.2004.91.

M. Herlihy and J. E. B. Moss. Transactional Memory: Architectural Support For Lock-Free Data Structures. In *Proceedings of theTwen-*

*tiethAnnual International Symposium on Computer Architecture*, 1993. URL `citeseer.ist.psu.edu/herlihy93transactional.html`.

Maurice Herlihy, Victor Luchango, Mark Moir, and III William N. Scherer. Software transactional memory for dynamic-sized data structures. pages 92–101, Jul 2003.

A. Kagi, D. Burger, and J.R. Goodman. Efficient Synchronization: Let them eat QOLB. *Computer Architecture, 1997. Conference Proceedings. The 24th Annual International Symposium on*, pages 170–180, 2-4 Jun 1997.

Sanjeev Kumar, Michael Chu, Christopher J. Hughes, Partha Kundu, and Anthony Nguyen. Hybrid transactional memory. In *Proceedings of Symposium on Principles and Practice of Parallel Programming*, Mar 2006.

Peter S. Magnusson, Magnus Christensson, Jesper Eskilson, Daniel Forsgren, Gustav Hallberg, Johan Hogberg, Fredrik Larsson, Andreas Moestedt, and Bengt Werner. Simics: A full system simulation platform. *Computer*, 35(2):50–58, 2002. ISSN 0018-9162. doi: http://doi.ieeecomputersociety.org/10.1109/2.982916.

Milo M. K. Martin, Daniel J. Sorin, Bradford M. Beckmann, Michael R. Marty, Min Xu, Alaa R. Alameldeen, Kevin E. Moore, Mark D. Hill, and David A. Wood. Multifacet's general execution-driven multiprocessor simulator (gems) toolset. *SIGARCH Comput. Archit. News*, 33(4):92–99, 2005. ISSN 0163-5964.

John M. Mellor-Crummey and Michael L. Scott. Algorithms for scalable synchronization on shared-memory multiprocessors. *ACM Trans. Comput. Syst.*, 9(1):21–65, 1991. ISSN 0734-2071. doi: http://doi.acm.org/10.1145/103727.103729.

Chi Cao Minh, JaeWoong Chung, Christos Kozyrakis, and Kunle Olukotun. Stamp: Stanford transactional applications for multi-processing. In *Proceedings of The IEEE International Symposium on Workload Characterization*, Sep 2008.

K.E. Moore, J. Bobba, M.J. Moravan, M.D. Hill, and D.A. Wood. Logtm: log-based transactional memory. *High-Performance Computer Architecture, 2006. The Twelfth International Symposium on*, pages 254–265, 11-15 Feb. 2006. ISSN 1530-0897. doi: 10.1109/HPCA.2006.1598134.

R. Rajwar, M. Herlihy, and K. Lai. Virtualizing transactional memory. pages 494–505, 2005. URL `http://ieeexplore.ieee.org/xpls/abs\_all.jsp?arnumber=1431581`.

Ravi Rajwar and James R. Goodman. Transactional lock-free execution of lock-based programs. *SIGPLAN Not.*, 37(10):5–17, 2002. ISSN 0362-1340. doi: http://doi.acm.org/10.1145/605432.605399.

Hany E. Ramadan, Christopher J. Rossbach, Owen S. Hofmann, and Emmett Witchel. Dependence-aware transactions for increased concurrency. Accepted for publication at the 41st Annual ACM/IEEE International Symposium on Microarchitecture, 2009.

J. Gregory Steffan, Christopher B. Colohan, Antonia Zhai, and Todd C. Mowry. Improving value communication for thread-level speculation. In *HPCA*, pages 65–72, 2002.

S. Woo, M. Ohara, E. Torrie, J. Singh, and A. Gupta. Methodological considerations and characterization of the splash-2 parallel application suite. In *Proceedings of the 23rd Annual Symposium on Computer Architecture*, May 1995.