Boosting Performance of Transactional Memory through Transactional Read Tracking and Set Associative Locks

By

Amir Ghanbari Bavarsad

A Thesis submitted in partial fulfillment of the requirements of

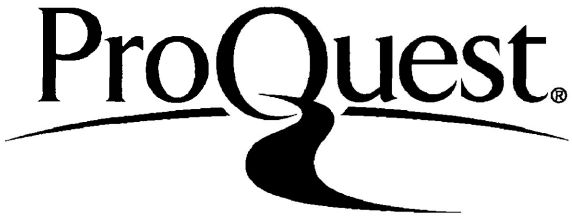
The Msc. Eng. Degree in

Electrical and Computer Engineering

Faculty of Engineering

Lakehead University

Thunder Bay, Ontario



Thunder Bay, Ontario, Canada

April 2013

ABSTRACT

Multi-core processors have become so prevalent in server, desktop, and even embedded systems that they are considered the norm for modern computing systems. The trend is likely toward many-core processors with many more than just 2, 4, or 8 cores per CPU. To benefit from the increasing number of cores per chip, application developers have to develop parallel programs [1]. Traditional lock-based programming is too difficult and error prone for most of programmers and is the domain of experts. Deadlock, race, and other synchronization bugs are some of the challenges of lock-based programming. To make parallel programming mainstream, it is necessary to adapt parallel programming by the majority of programmers and not just experts, and thus simplifying parallel programming has become an important challenge.

Transactional Memory (TM) is a promising programming model for managing concurrent accesses to the shared memory locations. Transactional memory allows a programmer to specify a section of a code to be "*transactional*", and the underlying system guarantees atomic execution of the code. This simplifies parallel programming and reduces the possibility of synchronization bugs.

This thesis develops several software- and hardware-based techniques to improve performance of existing transactional memory systems. The first technique is Transactional Read Tracking (TRT). TRT is a software-based approach that employs a locking mechanism for transactional read and write operations. The performance of TRT depends on memory access patterns of applications. In some cases, TRT falls behind the baseline scheme. To further improve performance of TRT, we introduce two hybrid methods that dynamically switches between TRT and the baseline scheme based on applications' behavior.

The second optimization technique is Set Associative Lock (SAL). Memory locations are mapped to a lock table in order to synchronize accesses to the shared memory locations. Direct mapped lock tables usually result in collision which leads to false aborts. In SAL, we increase associativity of the lock table to reduce false abort. While SAL improves performance in most of the applications, in some cases, it increases execution time due to overhead of lock tables in software. To cope with this problem, we propose Hardware-SAL (HW-SAL) which moves the set associative lock table to the hardware. As such, true power of set associativity will be harnessed without sacrificing performance.

ACKNOWLEDGEMENTS

I would first like to thank my advisor Professor Ehsan Atoofian for taking me on as a student in the fall of 2011 and provided funding for my research. His hands off advising style was a perfect fit that allowed me to pursue what I found interesting.

I could not have succeeded without the support of my family and friends. Thanks to my parents for supporting my decision to pursue this degree. Also I would like to thank my lab-mates with which I had interesting discussions, whether it was related to research or if it was just something to pass the time.

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

Chapter 1 Introduction

The computing industry has been trying to keep up with Moore's law [2] for almost half a century. Processors have become smaller and faster as Moore had predicted; however, limitations such as sub-threshold leakage [3] and other physical constrains in single-core processors [4] have forced manufacturers to shift their focus from designing complex single-core processors with higher frequency towards utilizing several simpler cores in a chip to boost performance and gain more processing power. As such, major chip manufacturers have shifted towards Chip Multiprocessors (CMPs) since 2000. A CMP provides several processing cores on the same die, increasing the total processing power. At the same time, the design complexity is reduced by using the same design for different cores. IBM was the first to release a commercial CMP: the POWER4 [5] which was followed by Intel [6] and AMD [7].

CMPs by IBM, Intel, AMD, and ARM made CMPs almost ubiquitous in most computing devices, ranging from workstations to mobile devices and tablets. One of the challenges in CMPs is that existing sequential programs are unable to utilize all the resources offered by CMPs, since the parallelism must be declared explicitly in the code and the legacy sequential programs were not designed with that concept in mind. This has led to creation of a demand for new programming methodologies to harness the ever-increasing computational power of CMPs.

1.1 Challenges in Programming for CMPs

Synchronization plays a pivotal role in parallel programming. Different threads of execution need to have a consistent view of the shared data; otherwise, the program would run into synchronization bugs such as deadlock, priority inversion and starvation [8]. Conventional parallel programming using locks allows a programmer to synchronize accesses to the shared data.

There are two types of locking mechanisms to ensure atomicity of parallel programs: Fine-grained locking and coarse-grained locking. In the next section we present these two types of locks and provide examples.

1.1.1 Fine-grained Locking

In fine-grained locking, in order to synchronize accesses to an object, we split the object into independently synchronized components, ensuring that at any moment at most one thread accesses each of those components. The problem with fine-grained locking is its complexity.

Figure 1.1 (a) shows a program for parallel version of "Sieve of Eratosthenes" using fine-grained locks. Sieve of Eratosthenes is an algorithm that finds prime numbers up to a certain limit. In this example, the limit is set to 100. Each entry in array A[] corresponds to a number. By the end of the program, if an entry is true, the corresponding number is a prime number; otherwise, it is a composite number. The whole array is divided among processors. Each processor goes through its share of the array and determines whether the corresponding numbers are prime numbers or not. In fine-grained locking, each entry of the array has to be protected by a lock. Before accessing the entry, lock is acquired. After processing the entry, the lock is released.

1.1.2 Coarse-grained Locking

In coarse-grained locking, we take a sequential implementation of an object, add a lock field, and ensure that each access to the shared object acquires and releases the lock. The problem with coarse-grained locking is its lack of scalability. Coarse-grained synchronization works well when concurrency level is low. However, if too many threads try to access the same object at the same time, then the object becomes a bottleneck, forcing threads to execute serially. Figure 1.1 (b) shows a program for parallel version of "Sieve of Eratosthenes" using coarse-grained locks. In coarse-grained locking, the critical section where the array is being accessed is protected by a single lock. Each thread needs to acquire the lock before it could access that part of the code. This reduces the concurrency, since only one thread could have exclusive access to the critical section at a time. On the other side, in fine-grained lock, each entry of the array is protected by a separate lock. As such, threads that access different entries of the array can run simultaneously. This increases concurrency level in the fine-grained lock programs.

The ever-increasing presence of CMPs in different areas of computing dictates the need for a system to provide a scalable, easy to program, and high performance mechanism for programmers in order to handle complexity of synchronization in their applications. This would make parallel programming mainstream, enabling average programmers write efficient and bug-

free parallel programs. One such method that provides all the aforementioned attributes in highly abstract level is Transactional Memory (TM).

```
//Let A be an array of Booleans initially set
//to true
bool A [100];
mutex locks[100];

void* thread_programs(int threadID)
{
   for (int i = 0; i<10; i+=threadID)
   {
      if (A[i] == true)
      {
         for (int j = i^2; j < n; j+=i)
         {
            mutex_lock(lock[j]);
            A[j] = false;
            mutex_unlock(lock[j]);
         }
      }
   }
}
```

(a) Fine-grained lock

```
//Let A be an array of Booleans initially set
//to true
bool A [100];
mutex lock;

void* thread_programs(int threadID)
{
   for (int i = 0; i<10; i+=threadID)
   {
      mutex_lock(lock);
      if (A[i] == true)
      {
         for (int j = i^2; j < n; j+=i)
         {
            A[j] = false;
         }
      }
      mutex_unlock(lock);
   }
}
```

(b) Coarse-grained lock

Figure 1.1 (a) Fine-grained and (b) coarse-grained lock-version of Sieve of Eratosthenes.

1.2 Transactional Memory

Transactional memory (TM) [9] is a new parallel programming model which is viewed by many as a replacement for lock-based programming. In TM, programmers mark sections of programs that access shared data as transactions and the underlying system ensures correct execution of the programs. Non-conflicting transactions execute in parallel and those transactions that conflict are aborted and restarted without the programmer having to worry about issues such as deadlock. Transactions are atomic [9]: each transaction either commits or aborts (its effects are discarded). Transactions are linearizable [9]: each committing transaction takes effect instantaneously at some point between start and end of the transaction.

Figure 1.2 shows the TM version of Sieve of Eratosthenes. Contrary to the lock versions, in transactional memory version of the program, only the critical section is defined using TM_BEGIN and TM_END and the programmer does not need to deal with the complexity of

locks and synchronization. If two transactions access an element of the array simultaneously, then the underlying system aborts and restarts one of the two transactions. As such, a programmer does not need to deal with synchronization of threads.

```
//Let A be an array of Booleans initially set
//to true
bool A [100];
void* thread_programs(int threadID)
{
   for (int i = 0; i<10; i+=threadID)
   {
      if (A[i] == true)
      {
         for (int j = i^2; j < n; j+=i)
         {
            TM_BEGIN();
            A[j] = false;
            TM_END();
         }
      }
   }
}
```

Figure 1.2 TM-version of Sieve of Eratosthenes.

TM provides several semantics for the programmers in order to make the synchronization of the code more abstract. These semantics are as follow:

1. Critical sections represented by transactions are atomic; they either successfully complete or abort.
2. Transactions are isolated and outside of transactions cannot see interim updates,
3. Transactions are placed in the code and there is no need to protect them with data types such as locks in an explicit manner by the programmers.

Transactional memory theoretically allows a programmer to concentrate more on defining where the critical sections should be in a coarse grained manner, but not have to worry about scalability and atomicity. The underlying system would guarantee those properties.

Transactional memory comes with three variants: Hardware Transactional Memory (HTM) [10] [11], Software Transactional Memory (STM) [12] [13], or Hybrid Transactional Memory [14]. Hardware Transactional Memory (HTM) exploits transactional caches to hold speculative data, track ownership of shared data, and detect conflicts among transactions. While HTM makes transactional memory fast, it increases hardware complexity and is not flexible. In addition, both

HTM and hybrid approaches require new features in hardware which do not exist in current processors, i.e. transactional cache. STM, however, can use available features of current processors and comes with fewer intrinsic limitations imposed by hardware structures, such as buffer size and caches. The downside of STM is that it is not as fast as HTM.

All Transactional Memory systems should offer some sort of non-blocking behaviour. A non-blocking behaviour guarantees that the threads that are contending for a shared resource do not have their execution postponed indefinitely. There are several forms of non-blocking behaviour:

- Wait-freedom [15]: is the strongest non-blocking guarantee of progress. An algorithm is wait-free if every operation has a limit on the number of steps the algorithm will take before the operation completes. This property is critical for real-time systems as long as the performance cost is not too high.
- Lock-freedom [16]: allows individual threads to starve but guarantees the system-wide progress. An algorithm is considered lock-free when the threads run sufficiently long enough that at least one of them makes progress. All wait-free algorithms are lock-free.
- Obstruction-freedom [17]: is possibly the weakest natural non-blocking progress guarantee. An algorithm is obstruction-free if at any point, a single thread executed in isolation (i.e., with all obstructing threads suspended) for a limited number of steps will complete its operation. All lock-free algorithms are obstruction-free.

1.3 Thesis Contributions

This thesis proposes two optimization techniques for STMs: Transactional Read Tracking (TRT) and Hardware Set Associative Lock (HW-SAL).

1.3.1 Transactional Read Tracking (TRT)

Current state of the art STM uses a global clock as a timestamp in order to maintain consistency among transactions. This causes contention over the global clock and dramatically impedes scalability, especially when the number of concurrent threads increases. TRT employs a distributed method that alleviates the need for the global clock, and so improves scalability and performance.

Although an effective method, TRT proves to be slower than baseline scheme, depending on the data access patterns of the benchmarks. A natural question would be whether the different advantages of TRT and its counterpart could be combined to gain maximum speed up. We

propose two adaptive methods that combine both techniques in order to avoid the shortcomings of each of the techniques and gain benefits of the two.

1.3.2 Hardware Set Associative Lock (HW-SAL)

In most recent STM systems, memory addresses are mapped to a lock table using a hash functions. Hash collision is an intrinsic property of a direct mapped lock table which would cause false conflicts. False conflict happens when two memory locations are mapped to the same entry of the lock table. An STM considers false conflict as memory conflict; whereas there is no actual conflict. False conflict reduces the concurrency of the STM systems. One way to circumvent this problem is using a set associative lock table. Much like a set associative cache, each entry of the lock table consists of several nodes. When an address is mapped to an entry, all nodes are searched for the address. As such, the probability of false conflict is reduced. However, implementation of SAL in software results in overhead which degrades performance in some of the benchmarks especially when associativity of the lock table increases.

The final contribution of this thesis is hardware support for SAL. In this work, we propose to move the lock table from software to hardware which in turn makes the lookup process fast and improves performance of SAL. There is a single hardware unit which contains the lock table and each processor can access the table. A programmer uses a set of new instructions provided by processors to access the table.

1.4 Thesis Organization

The rest of this thesis is organized as follows. In chapter 2, we explain the basic concepts of multiprocessor architecture and discuss transactional memory. We explain details of Transaction Locking II (TL2) as an example of state of the art STM. We also discuss related work in this chapter. In Chapter 3, we cover the motivation behind "Transactional Read Tracking" and explain details of implementations. Chapter 4 covers the "Set Associative Lock" and presents the "Hardware Support for SAL". Finally in chapter 5, we will offer the concluding remarks and future work for this thesis.

Chapter 2 Background and Related Work

In this chapter, we discuss architecture of CMPs and explain TM systems. In section 2.1, we review multiprocessor architectures. In section 2.2, we review the challenges that exist in programming for CMPs. Section 2.3 presents a brief overview of primitives provided by multiprocessors for parallel programming. In section 2.4, we explain Transactional Memory as a promising programming model for multiprocessor architectures. In section 2.5, we provide an overview of the related work to this thesis. We also describe details of TL2 as a state of the art STM in this section. Finally, in section 2.6, we will conclude this chapter with discussion on challenges of existing STMs.

2.1 Multiprocessor Architectures

The computing industry has gone into a drastic change since the early 2000's. Before 2000, chip manufacturers dedicated all transistors on a chip to a single-core processor. In each generation, the performance of processors was improved by increasing clock frequency. However, overheating and sub-threshold leakage [3] limits clock frequency. Therefore, chip manufacturers have shifted their focus to "multicore" architectures, where multiple cores are integrated on the same die. Figure 2.1 depicts the shift from single-core processors to multicore processors. As the figure shows, the clock frequency of the single core processors has increased rapidly until early 2000. Then, the rate at which performance increased diminished. Starting in 2005, the clock frequency was slightly reduced. This time marks the advent of multiprocessors. The manufacturers designed simpler processors with more core units, to provide parallelism.

The introduction of chip multiprocessors has changed the way we develop software. In single-core processors, increasing clock frequency results in reduction in execution time of programs and so programs were executed faster without any effort by the programmers. However, this trend in CMPs results in slow-down of programs since clock frequency of CMPs is reduced due to power budget.

Figure 2.2 shows architecture of a CMP with four cores. Each core executes an independent thread. One way to boost performance of applications in CMPs is parallel programming. A parallel program is composed of several threads and the threads are executed concurrently on a

CMP's cores. It is up to a programmer to utilize processor cores and harness computational power of the underlying hardware.

2.2 Challenges in Parallel Programming

Parallel programming introduces new challenges to programmers. As mentioned in Chapter 1, thread synchronization is one of the main concerns in parallel programming. Conventional constructs for synchronization such as locks and monitors are complicated and error-prone. Common problems associated with the conventional locking techniques are:

- **Priority inversion** [8]: occurs when a lower priority process is pre-empted while holding a lock needed by a higher-priority process.
- **Convoying** [8]: occurs when a process holding a lock is rescheduled, perhaps by exhausting its scheduling quantum time, by a page fault, or by some other kind of interrupt. When such an interruption occurs, other processes capable of running maybe unable to progress.
- **Deadlock** [8]**:** can occur if processes attempt to lock the same set of objects in different orders. Deadlock avoidance can be awkward if processes must lock multiple data objects, particularly if the set of objects is not known in advance.

In the next section, we review atomic primitives used in conventional lock-based parallel programs.

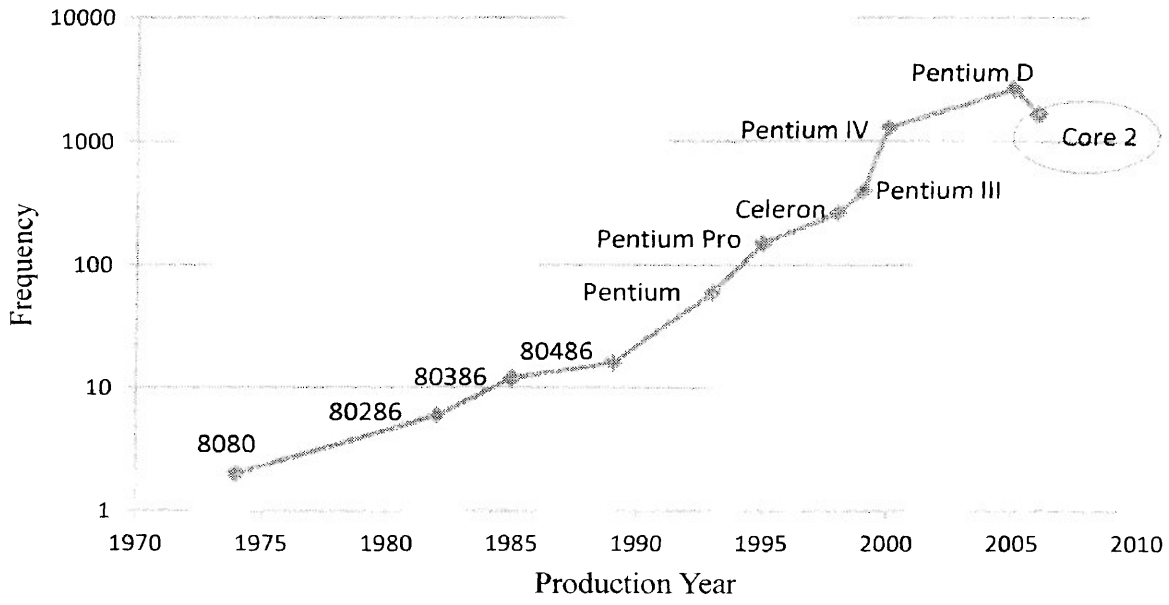


Figure 2.1 Clock frequency of different Intel processors over the years.

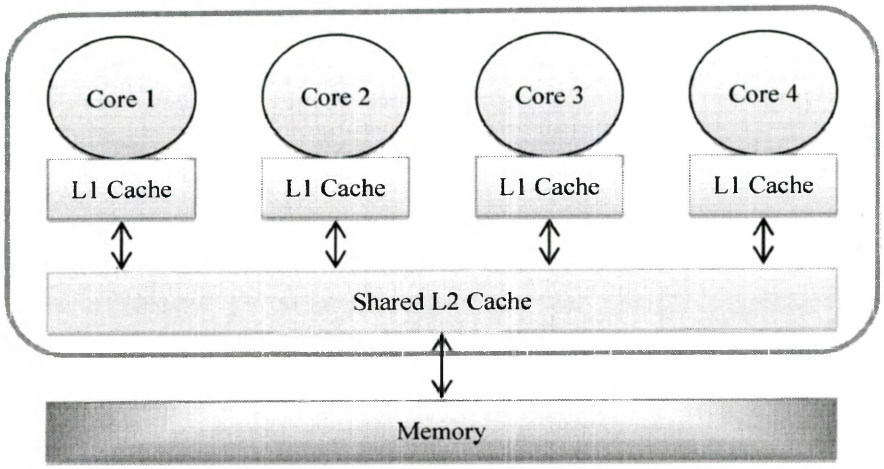


Figure 2.2 Architecture of a CMP.

2.3 Atomic Primitives

There has been extensive debate over the years on what primitives should be provided to support synchronization [18]. Some researchers have proposed that the user level synchronization operations, such as locks and barriers, should be supported at the machine level; meaning that the synchronization "algorithm" itself should be implemented in hardware [18]. As such, the system provides extensive hardware support which would make synchronization fast but not flexible. On the other side, software support would have the advantage of flexibility but it is slower than hardware primitives [18].

All practical synchronization operations rely on some sort of variation of atomic read-modify-write primitive. In this primitive, the value of a memory location is read, modified and written back atomically without intervening accesses to that location by other processors. Different synchronization algorithms can be built using this primitive. In the rest of this section, we discuss two variations of read-modify-write primitive.

2.3.1 Compare and Swap

Compare and Swap (CAS) operation takes three arguments: an address *"a"* in memory, an *expected* value *"e"*, and an *updated* value *"v"*. It atomically executes the following steps:

- If the memory at address *"a"* contains the expected value *"e"*, then write the new value *"v"* to that address and return the expected value, indicating successful CAS instruction.

- Otherwise leave the memory unchanged and return the value contained in the memory at address *"a"*, indicating failure of CAS instruction.

IBM 370 [18] was one of the first architectures to support a sophisticated atomic instruction, the *compare-and-swap* instruction [18]. This instruction supports synchronization for parallel programming on uniprocessor or multiprocessor systems. The *compare-and-swap* (CAS) instruction is now supported in many modern architectures such as AMD, Intel, and Sun. On Intel and AMD architectures, it is called CMPXCHG (compare and exchange) [19], while on SPARC™ it is called CAS [20].

The CAS instruction has one pitfall. Perhaps the most common use of CAS is the following. An application reads the value *"a"* from a given memory address, and computes a new value *"c"* for that location. It intends to store *"c"*, but only if the value *"a"* in the address has not been changed since it was read. One might think that applying a CAS with expected value *"a"* and updated value *"c"* would accomplish this goal. There is a problem: another thread could have overwritten the value *"a"* with another value *"b"*, and later write *"a"* again to the address. The CAS operation will replace *"a"* with *"c"*, but the application may not have done what it was intended to do (for example, if the address stores a pointer, the new value *"a"* may be the address of a recycled object). This problem is referred to as ABA problem [8].

2.3.2 Load-locked, Store-conditional

Some multiprocessors provide a pair of instructions called load-locked and store-conditional [18] to implement atomic operations, instead of atomic read-modify-write instructions such as compare and swap. The first instruction is commonly called *load-locked* or *load-linked* (LL). It loads a memory location into a register. It may be followed by any number of arbitrary instructions that modify the value in the register. Then the instruction *store-conditional* (SC) is called. It writes the register back to the memory location if and only if no other processor has written to that memory location since this processor completed its LL.

The LL/SC instruction is supported by a number of modern architectures: Alpha AXP (ldl_l/stl_c) [21], IBM PowerPC (lwarx/stwcx) [5] MIPS ll/sc, and ARM (ldrex/strex) [22]. LL/SC does not suffer from the ABA problem, but in practice there are often severe restrictions on what a thread can do between a LL and the matching SC. A context switch, another LL, or another load or store instruction may cause the SC to fail.

2.4 Transactional Memory (TM)

Transactional Memory (TM) was proposed to simplify parallel programming and improve scalability of programs. Software Transactional Memory (STM) has been a convenient way of providing necessary constructs for transactional programming. STMs do not require any modifications in hardware and are available to the programmers through a runtime or compile-time library.

In TM systems, conflict and version management are the two important aspects of the system design. Conflict management deals with when conflicts are detected, how they are detected and which actions should be taken to address them. Version management deals with where and how the transactional data are kept.

There are two policies for conflict management: *Eager* and *Lazy*. In eager policy, a TM system constantly monitors the transactional execution; as soon as it detects a conflict, it decides which transaction should be aborted. In this method, since the conflicts are detected at the earliest possible time, the wasted work due to aborted transactions is minimized. On the other side, lazy conflict management defers detection of conflicts until the end of transactions. At that point, the transaction checks all memory locations that are read or written in the transactional section. If another transaction has changed one or more of these memory locations, conflict happens and one of the two transactions should be aborted. Lazy conflict management can lead to wasted work, since a transaction is only aborted at the end of its execution. However, Tomic et al. [23] observed that it can allow more parallelism than eager policy.

There are also two main classes of version management policies in TMs: eager and lazy. Eager policy keeps new (speculative) values in-place (in the memory hierarchy) and holds pre-transactional values somewhere else. Most of TMs buffer the old values in a different location in memory, using a software-managed log [24] [25] . On the other hand, lazy policy keeps the old values in the memory hierarchy and holds new values in a log. The new values become visible to other transactions only during commit.

In the next section, we review previous research work in TM.

2.5 Related Work

The concept of Transactional Memory was first introduced by Herlihy and Moss [9]; the basic idea was to group shared-memory operations into atomic transactions. They changed cache coherence protocol in multiprocessors to support TM in hardware. A transactional cache accommodates speculative data generated by transactions. When a transaction successfully commits, the content of the transactional cache is written into the main memory. The transactional cache is able to snoop memory operations on the bus from other processors. As such, it can determine if another processor is trying to gain exclusive access to a cache line which is currently obtained by a local transaction; if so, the remote transaction would be aborted. This implies that the protocol is not non-blocking. Starvation is handled in software by backing off in the case of contention. The main disadvantage of this design is that it requires hardware modifications for cache coherency protocols.

Shavit and Touitou [26] proposed a lock-free STM. A notable difference from Herlihy is that they abort contending transactions, instead of recursively helping them; non-blocking behaviour is still guaranteed because aborted transactions help the transaction that aborted them before retrying. Their design only supports static transactions, in which the set of accessed memory locations is known in advance. However, this STM cannot support transactions that access memory locations that are allocated dynamically through operations such as malloc().

To alleviate the problem with static transactional programming, Moir [27] presents a lock-free and wait-free STM design which offers a dynamic transactional support, in contrast with Shavit and Touitou's static interface. The lock-free design divides the transactional memory into fixed-size blocks which form the unit of concurrency. A header array contains a word-size entry for each block in the memory, consisting of a block identifier and a version number. In their design, if two transactions conflict, then one of them aborts. In this scheme, transactions that do not access shared memory locations can execute simultaneously. The design also suffers from the same drawbacks as the conditionally wait-free MCAS [28] on which it builds: bookkeeping space is statically allocated for a fixed-size heap, and the read operation is potentially expensive. Also, the read-set of the transaction has to be validated in order to make sure that it has not been changed since the start of a transaction.

Herlihy *et al.* [29] have implemented an obstruction-free dynamically sized STM, meaning that memory blocks can be created and destroyed on the fly. Their implementation builds on a readily available form of the CAS primitive. A novel feature of their obstruction-free STM implementation is its use of modular contention managers to ensure progress in practice. Also, the design is disjoint-access-parallel [28]; meaning that processes that access disjoint sets of words in the shared memory, can progress concurrently as long as they do not interfere with each other. These features significantly decrease contentions in many multiprocessor applications.

Dice *et al.* [12] introduce Transactional Locking II, an STM based on commit-time locking and a novel global clock scheme for validation. Unlike all other STMs, TL2 fits seamlessly with any memory system life-cycle, including those using malloc()/free(). Also, unlike all other lock-based STMs it efficiently avoids periods of unsafe execution, that is, using its novel version-clock validation, user code is guaranteed to operate only on consistent memory states. It uses a table of Locks in order to maintain consistency of transactions. The overall performance improvement is notably higher compared to both lock-based and non-blocking STM systems. Due to aforementioned qualities of TL2, we use this system as a baseline for our optimization techniques.

Spear *et al.* [30] proposed global commit counter to reduce overhead of incremental validation. In invisible read policy, a reader detects read-write conflicts by incrementally validating open objects at the cost of quadratic number of objects. To reduce cost of validation, a global counter records the number of transactions that attempt to commit. When a transaction opens an object, the transaction skips incremental validation if the counter has not changed since the last time the reader checked its objects. As such, costly exhaustive validations can be avoided if transactional write operations happen rarely. However, this method does not distinguish access to different fields of an object and results in unnecessary serialization if transactions access disjoint memory locations.

Mannaeswamy and Govindarajan [31] studied cache behavior of STMs and showed that global clock misses are responsible for up to 38% of transactional misses in Stamp benchmark suite. They proposed a compiler optimization technique, which is called selective partition timestamp (SPTS), to reduce cost of the global clock. SPTS, partitions disjoint instances of data structures and assigns each partition to a separate pool. Each pool has its own local clock and data

structures within a pool compete for the local clock. As such, contention over the central global clock reduces. One of the restrictions of SPTS is that it decides on data structure partitions in compile-time and so needs to use a conservative approach to select candidate shared data structures for partitioning. As an example, if a data structure is passed to an external function, whose source code is not visible to the compiler, the compiler cannot apply SPTS to the data structure.

Riegel *et al.* [32] introduced a mechanism which uses real-time clocks to optimistically synchronize concurrent transactions. They also exploited externally synchronized clocks as the time base for better scalability. Their scheme requires external support and focuses on scalability of time base itself.

Avni and Shavit [13] introduced thread-local clock (TLC) technique to allow transactions to operate on consistent states without the global clock. In TLC, each thread has a local clock which is initialized to zero and is incremented by one at the start of every new transaction. There is also a thread local array that has an entry per thread recording timestamp of the thread. When a transaction commits it writes its thread id and timestamp into the associated lock. To validate read-set, all locks corresponding to the transactional read operations are checked to be unlocked. Then, the timestamp of each lock is checked to make sure that it is less than the associated thread j's entry in the thread local array. If the check fails then thread j's entry in the array is updated with the new timestamp. While TLC eliminates central global clock, it increases abort rate since the new timestamp of a committed transaction is not transferred to other transactions immediately. Instead, other transactions notice the new timestamp when their validations fail. As such, TLC may degrade performance despite of the fact that it eliminates the central clock. In addition, Avni and Shavit evaluated TLC with micro benchmarks which are not representative of real applications.

Atoofian [33] introduced "Set Associative Locks" (SAL) to reduce false aborts in STMs. In time-based STMs, different memory locations might be mapped to the same entry of the lock table. This creates false aborts and degrades performance. Atoofian proposed SAL which increases associativity of the lock table and reduces false aborts.

Many HTM implementations have been proposed since Herlihy and Moss introduced Transactional Memory. Ananian et al. [34] proposed UTM which was the first eager HTM

system that supports unbounded transactions. By placing transactional modifications across the memory hierarchy and storing metadata on the side, it is able to execute transactions of any size.

Hammond et al. [35] proposed a new shared memory model: Transactional memory Coherence and Consistency (TCC). TCC provides a model in which atomic transactions are always the basic unit of parallel work, communication, memory coherence, and memory reference consistency. TCC hardware combines all writes from each transaction region in a program into a single packet and broadcasts this packet to the permanent shared memory states atomically as a large block. This simplifies the coherence hardware because it reduces the need for small, low-latency messages and completely eliminates the need for conventional snoopy cache coherence protocols, as multiple speculatively written versions of a cache line may safely coexist within the system. Meanwhile, automatic, hardware-controlled rollback of speculative transactions resolves any correctness violations that may occur when several processors attempt to read and write the same data simultaneously. The cost of this simplified scheme is higher inter-processor bandwidth.

Chafi et al. [36] proposed Scalable TCC which presents a scalable TM implementation for directory-based distributed shared memory systems. This scheme is live lock free without the need for user-level intervention. The design is a scalable implementation of optimistic concurrency control that supports parallel commits with a two-phase commit protocol. It uses write-back caches and filters coherence messages. The scalable design is based on Transactional Coherence and Consistency (TCC), which supports continuous transactions and fault isolation.

Moore et al. [24] proposed LogTM in order to simplify the version management mechanism by keeping the pre-transactional state in a software-managed log; in case of an abort, the values in the software log will be written back to memory. Also using Read-Write cache bits [37] LogTM is able to eagerly detect conflicts. Yen et al. proposed LogTM-SE [25] in order to decouple the transactional state from caches by summarizing the memory accesses in signatures [38]. LogTM-SE uses signatures to summarize a transaction's read and write-sets and detects conflicts on coherence requests (eager conflict detection). Transactions update memory "in place" after saving the old values in a per-thread memory log (eager version management). Finally, a transaction commits locally by clearing its signature, resetting the log pointer, etc., while aborts must undo the log.

To accelerate the abort recovery and reduce the pressure on the write signature, FASTM [10] implements a hybrid version management mechanism. It takes advantage of the processor's cache hierarchy to provide fast abort recovery. FASTM uses a novel coherence protocol to buffer the transactional modifications in the first level cache and to keep the non-speculative values in the higher levels of the memory hierarchy. This mechanism allows fast abort recovery of transactions that do not overflow the first level cache. FASTM keeps the pre-transactional state in a software-managed log, which permits the eviction of speculative values and enables transparent execution even in the case of cache overflow.

DynTM [11] presents the first fully-flexible HTM system that permits the simultaneous execution of transactions using complementary version and conflict management strategies. DynTM utilizes a novel coherence protocol that allows tracking conflicts among eager and lazy transactions. Both the eager and the lazy execution modes of DynTM exhibit very high performance compared to modern HTM systems. Also, the DynTM lazy execution mode implements local commits, avoiding expensive commit arbitration. In addition, lazy transactions share the majority of hardware support with eager transactions, reducing cost of implementation substantially.

IBM zEC12 [39] is the first general purpose server which incorporates transactional memory. In zEC12, IBM used transactional memory to enable software to better support concurrent operations that use a shared set of data such as financial institutions processing transactions against the same set of accounts. zEC12 exploits up to 120 processing cores, supporting speeds of 5.5 GHz the highest clock speed CPU ever produced for commercial sale [40]. The architecture of the cores is a superscalar out-of-order pipeline with new instruction for transactional memory support.

BlueGene [41] is a supercomputing project that provides an ultra-scale technical computing platform to solve the most challenging problems facing engineers and scientists at faster, more energy efficient, and more reliable rates than ever before. Wang et al. [42] evaluated the performance of transactional memory applications on IBM's BlueGene/Q platform. They first provided a detailed description of the BG/Q HTM implementation and overhead. Then, they presented a thorough examination of parallel benchmarks for TM system on BG/Q. Also, they describe how the best-effort HTM support in BG/Q can be complemented with a software stack

that includes the kernel, the compiler, and the runtime system to deliver the simplicity of a TM programming model.

In this thesis, we use Transaction Locking II (TL2) [12] to evaluate our optimization techniques. It is important to note that although we use TL2 in this thesis, our optimization techniques are general and can be employed in any other time-based STMs [32] [43]. In the next section, we explain details of TL2.

2.5.1 Transactional Locking II (TL2)

Time-based transactional memories exploit a time base to impose order among transactions and reason about consistency of transactional data. In this section, we focus on TL2 as a time-based STM.

TL2 employs a validation scheme, called GV4, which relies on a global clock. The global clock is implemented as a shared counter and is incremented when a transaction commits. In addition, GV4 exploits a table of locks to synchronize accesses to the shared memory locations. The memory addresses are mapped to the locks using a hash function. Figure 2.3 shows the structure of memory and how it is mapped to the lock table entries. Each entry of the lock table has two sections: lock-bit and version number. The size of each entry in the lock table is equal to the size of address on the host machine. The least significant bit (LSB) of the lock shows whether the lock is free or acquired. If the LSB is zero (free), the rest of the lock shows the time stamp of the last transaction that wrote to a memory location covered by the lock. If the LSB is one (acquired), the rest of the lock holds the address of the owner transaction. Since the lock is word-aligned, the LSB can safely represent the status of the lock (free or acquired). Figure 2.4 shows the structure of an entry of the lock table. When a transaction commits, it updates version number of all locks corresponding to the memory locations that were written by the transaction.

TL2 supports both Eager and Lazy conflict management policies. Figure 2.5 shows the pseudo code for key functions in Eager GV4. For Eager Policy the following set of operations are executed:

- At the start of a transactional section, TL2 samples the global clock and stores it in a thread local variable called *read-version (rv)*.
- Then, it runs through a speculative execution and makes undo logs to restore state of the transaction in the event of failure. TL2 maintains two sets for each transaction: one

for transactional reads, called *read-set*, and the other one for transactional writes, called *write-set*. For each transactional read operation, TL2 checks consistency of the memory location to be read (line 3). TL2 compares *rv* and version field of the lock entry corresponding to the memory address. If *rv* is greater than or equal to the version field, then consistency check passes; otherwise, the consistency check fails since the memory address has been written by another transaction after current transaction has started. In addition, TL2 checks that the lock entry is free. If any of the aforementioned conditions are not met, the transaction aborts. In case of a write operation (line 9)Figure 2.6 TL2 tries to acquire the lock corresponding to that memory location at the same time. If lock acquisition is successful, the memory location will be updated immediately. If the lock is already acquired, the transaction aborts.

- At commit time (line 24), it increments the global version-clock using the atomic compare and swap (CAS) operation. Finally, TL2 goes through the *write-set* and releases acquired locks by writing the new value of the global clock in to the version field of the locks and clearing the lock bits.
- In case of abort (line 19), TL2 goes through the write-set and reverts all the changes made to the shared memory locations and releases all the locks it has acquired so far.

For read-only transactions, the process is much simpler; TL2 samples the global clock at the start of a transactional section and validates transactional read operations. For validation, TL2 checks that the read location's lock bit is clear and the timestamp in the lock field is less than the *rv*. If validation passes then the transaction commits; otherwise, the transaction aborts. Since there are no write operations, there is no need to increment the value of global clock.

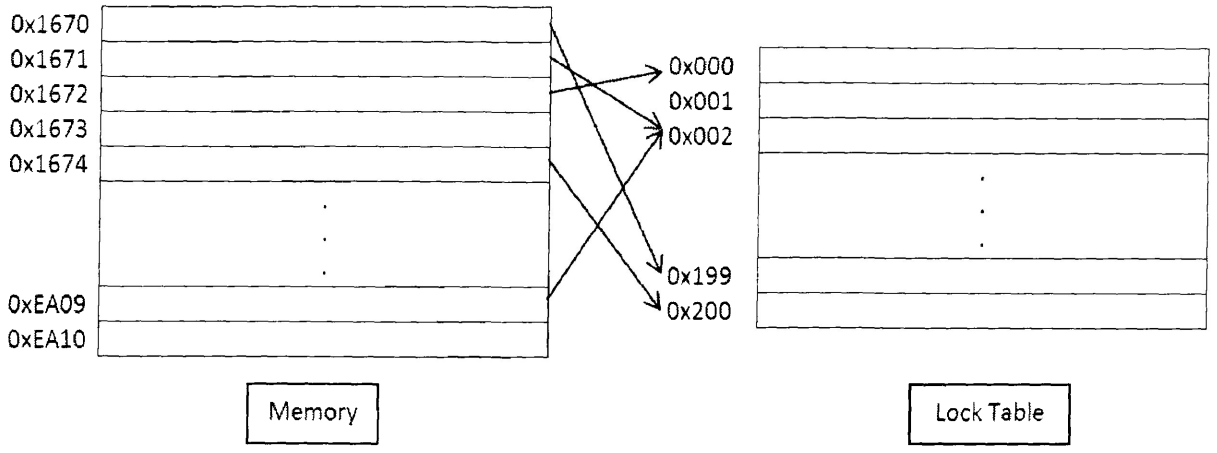


Figure 2.3 Lock table in TL2.

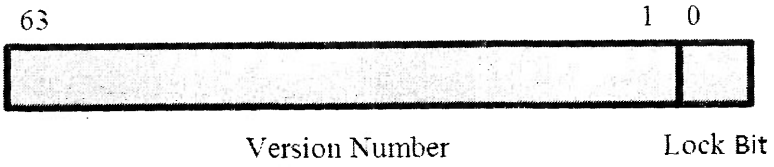


Figure 2.4 Structure of an entry of the lock table.

```
1  TxLoad(addr) {
2     //if lock entry corresponding to addr is free
3     if ( lock(addr) is free and readVersion >= lockVersion )
4           return (Mem[addr]);
5     } else
6        TxAbort();
7  }
8
9  TxStore (addr, value) {
10    If (lock(addr) is free ) {
11       Acquire( lock(addr) );
12       Write value in to addr;
13    }
14    else {
15       TxAbort();
16    }
17 }
18
19 TxAbort() {
20    DropLocks();
21    RevertMemoryChanges();
22 }
23
24 TxCommit() {
25    DropLocks();
26 }
```

Figure 2.5 Pseudo code for Eager GV4.

Figure 2.6 depicts the pseudo code for Lazy GV4. In Lazy policy, the following sequence of actions is employed:

- At the start of a transactional section, TL2 samples the global clock and stores it in a thread local variable called *read-version (rv)*.
- Then, it runs the transaction speculatively. For each transactional read operation (line 1), TL2 checks consistency of the memory location to be read. TL2 compares *rv* with version field of the lock entry corresponding to the memory address. If *rv* is greater than or equal to the version field, then consistency check passes; otherwise, the

consistency check fails since the memory address has been written by another transaction after current transaction has started. In addition, TL2 checks that the lock entry is free. If any of the aforementioned conditions are not met, transaction aborts. In case of a write operation (line 9), TL2 stores the new value in a private memory and defers updating the shared memory locations to commit time.

- At commit time (line 22), TL2 acquires locks corresponding to its *write-set*. If lock acquisition fails then the transaction aborts. When all locks have successfully been acquired, then it makes the tentative changes to the memory permanent. Then, TL2 releases all the locks and increments the global version-clock using an atomic compare and swap (CAS) operation.
- In case of abort (line 18), TL2 goes through the write-set and releases all the locks it has acquired so far.

Figure 2.6 depicts the pseudo code for main functions of GV4.

```
1   TxLoad(addr) {
2     //if lock entry corresponding to addr is free
3     if ( lock(addr) is free and readVersion >= lockVersion )
4           return (Mem[addr]);
5     } else
6       TxAbort();
7   }
8
9   TxStore (addr, value) {
10    If (lock(addr) is free ) {
11      Add the value to write-set
12    }
13    else {
14      TxAbort();
15    }
16  }
17
18  TxAbort() {
19    DropLocks();
20  }
21
22  TxCommit() {
23    AcquireLockForWrite-Set();
24    MakeTentativeChangesPermanent();
25    DropLocks()
26  }
```

Figure 2.6 Pseudo code for Lazy GV4.

TL2 reduces cost of validation relative to those STMs that require rescanning of the read-set on every transactional load [30]. However, there are some disadvantages in TL2. The global clock creates contention since each transaction increments the clock in commit, which results in costly cache coherency invalidation traffic. Also there is an extreme condition where TL2 leads to unnecessary aborts. Consider the case where Thread 0 (T_0) writes to a shared memory location and Thread 1 (T_1) reads from it. Assume that T_0 writes after T_1 has started, but before T_1 reads from the shared location. Since T_1 samples the clock at the start of the transactional section, the version of shared memory location is more than rv. Therefore, validation fails and T_1 aborts. However, T_0 and T_1 access the shared memory location at different timestamps and could have committed successfully. These unnecessary aborts waste processor resources and degrade performance.

2.5.2 Programming with TL2

TL2 provides several programming constructs for programmers to write transactional programs:

1) TM_STARTUP (): Initializes transactional states and the internal buffers. It also initiates the lock table.
2) TM_THREAD_ENTER (): creates a new thread of execution. It also initializes the newly created thread and sets the internal state registers, read-set and write-set buffers to appropriate values. This is placed at the beginning of the parallel section of a program.
3) TM_BEGIN (): Marks the beginning of a transaction. The thread saves a program's state at the start of the transactional section. If later, the transaction aborts, the state of the program is recovered.
4) TM_SHARED_READ (address): Loads shared memory value pointed by "address" into a temporary variable. This causes the memory address to be added to the read-set of the thread.
5) TM_SHARED_WRITE (address, value): Stores the "value" into the location in the shared memory pointed by "address". This causes the memory address to be added to the write-set of the thread. If eager policy is employed, the changes to the shared memory take place immediately. However, if lazy policy is employed, the changes to the shared memory are deferred until commit time.
6) TM_END (): Marks the end of a transaction. The transaction either successfully commits and finished execution, or aborts and re-executes from TM_BEGIN().
7) TM_THREAD_EXIT (): Deallocates memory for internal buffers of the thread. This is placed at the end of last parallel section of the code.

8) TM_SHUTDOWN (): Called at the end of the program in order to release all the internal buffers and transactional states allocated to the transactional memory system.

```
1   #define NUM_OF_THREADS 8
2   long max_count;
3   static int threads_arg[MAX_NUM_OF_THREADS];
4   void func_count (void* argPtr);
5   long  shared_counter = 0; //the shared variable
6   int main (int argc, char* argv)
7   {
8       TM_STARTUP (NUM_OF_THREADS);
9       thread_start(func_count, (void*)threads_arg);
10      TM_SHUTDOWN ();
11      return 0;
12  }
13
14  int func_count()
15  {
16      long myId = thread_getId();
17      long local_counter = 0;
18      long tmp = 0;
19      TM_THREAD_ENTER ();
20      while (local_counter++ < max_count)
21      {
22          TM_BEGIN();
23          tmp = (long)TM_SHARED_READ (shared_counter);
24          TM_SHARED_WRITE (shared_counter, tmp + 1);
25          TM_END();
26      }//end of while
27      TM_THREAD_EXIT ();
28  }
```

Figure 2.7 A sample code using transactional memory for implementation of a counter.

Figure 2.7 shows a shared counter implemented in TM. The shared counter is incremented by each thread. When the value of the thread exceeds a limit, the program finishes. TM_STARTUP creates the number of threads specified by the programmer (line 8) and initializes the internal data structures of TL2. Then, func_count is passed to thread_start() (line 9). func_count() is a function which is executed by multiple threads concurrently. In this function, TM_THREAD_ENTER is called to initialize the internal state of each thread (line 19) and create the *read-set* and *write-set* for each of them. In line 22, TM_BEGIN marks the beginning of the transactional section. In line 23, TM_SHARED_READ is used to read from shared_counter. shared_counter is a shared variable and is accessed by all transactions. So, it is necessary to protect this variable in the transactional section. In line 24, TM_SHARED_WRITE is called

which acquires the lock corresponding to shared_counter. TM_END (line 25) marks the end of transaction and causes TxCommit to be called. At the end of Transactional Section (line 27), TM_THREAD_EXIT() deallocates all data structures which were allocated by TM_BEGIN. TM_SHUTDOWN (line 10) is called at the end of the application, where all transactions terminate.

2.5.3 Benchmarks

We use STAMP v0.9.10 benchmark suite [44] to evaluate our work. A brief description of the benchmarks used in our evaluations is as follows:

- **Bayes:** A Bayesian network (or a belief network) is a way of representing probability distributions for a set of variables in a concise and comprehensible graphical manner.
- **Kmeans:** K-means is a partition-based method and is arguably the most commonly used clustering technique. K-means represents a cluster by the mean value of all objects contained in it.
- **Labyrinth:** Given a maze, this benchmark finds the shortest-distance paths between pairs of starting and ending points.
- **Ssca2:** The Scalable Synthetic Compact Applications (Ssca2) benchmark is comprised of four kernels that operate on a large, directed, weighted multi-graph. STAMP focuses on Kernel 1, which constructs an efficient graph data structure using adjacency arrays and auxiliary arrays.
- **Vacation:** This benchmark implements a travel reservation system powered by a non-distributed database.
- **Genome:** This benchmark implements a gene sequencing program that reconstructs the gene sequence from segments of a larger gene.

2.6 Summary

In this chapter, we presented background for TM systems and discussed related work. We explained details of TL2. TL2 utilizes a lock table and a novel global clock mechanism to provide the necessary functions for transactional memory. One of the limitations of the TL2 is global clock. Global clock is a central variable which is accessed by all transactions. As such, global clock restricts scalability of TL2 and increases overhead when the number of threads increases. In the next chapter, we propose TRT to alleviate the overhead of global clock and improve performance of TL2.

Chapter 3 Transactional Read Tracking

In STMs, all transactions should have a consistent view of shared memory locations at all times [32]. Some STMs [30] use validation technique to avoid inconsistency in transactions. In this technique, STM rescans all previously read elements on every new transactional load. Validation is an expensive operation and increases overhead of the implementation and degrades performance especially in applications with frequent transactional reads [32]. An alternative approach is a global clock which is used as a timestamp for shared memory locations. While this method is simple to implement, it results in contention over the global clock, especially when transactions commit frequently. In this chapter, we introduce Transactional Read Tracking (TRT) which tracks transactional read and write operations without using a central data structure such as global clock. TRT removes the burden of the global clock and improves scalability of STMs.

The rest of this chapter is organized as follows. In section 3.1, we present the motivation behind this work. In section 3.2, we introduce TRT and explain different aspects of it. In section 3.3, the performance of TRT is reported. In section 3.4, we present a hybrid method to increase the speedup provided by the system. In section 3.5, we introduce yet another hybrid method to further increase the performance.

3.1 Motivation

In time-based STMs, a transaction increments the clock when it commits. This results in broadcasting costly coherence invalidations over interconnection network and remote caches [45]. Therefore, global clock becomes a bottleneck when the number of concurrent transactions increases, even when there is no conflict among the transactions. *Transactional Read Tracking* (TRT) allows transactions to operate on consistent states without the need for a global notion of time. In TRT, each memory location is associated with a Lock. When a transaction reads a memory location, it sets a dedicated bit in the corresponding lock entry, indicating that the memory location has been read by a transaction. When a transaction writes into a memory location, it checks the read bits of the corresponding lock entry. If all read bits are zero and the lock has not been acquired, the write operation is successful and the thread continues; otherwise,

the thread has to abort. Hence, TRT maintains atomicity of transactions and validates transactional data without using a global clock.

3.2 Transactional Read Tracking

In this section, we explain details of TRT and discuss how TRT is implemented in software. Similar to GV4, TRT relies on locks to synchronize accesses to the shared memory locations. Figure 3.1 depicts the structure of a single lock entry in TRT. Each lock consists of a lock bit and a set of read bits. The lock bit indicates whether the corresponding memory location is locked by a transaction. When a transaction writes to a memory location it sets the corresponding lock bit. Read bit *i* corresponds to thread *i* and indicates whether thread *i* has read from the memory location. When thread *i* reads from a memory location it sets read bit *i* of the corresponding lock. The record of all transactional loads and stores is kept in two sets of logs: *read-set* and *write-set*.

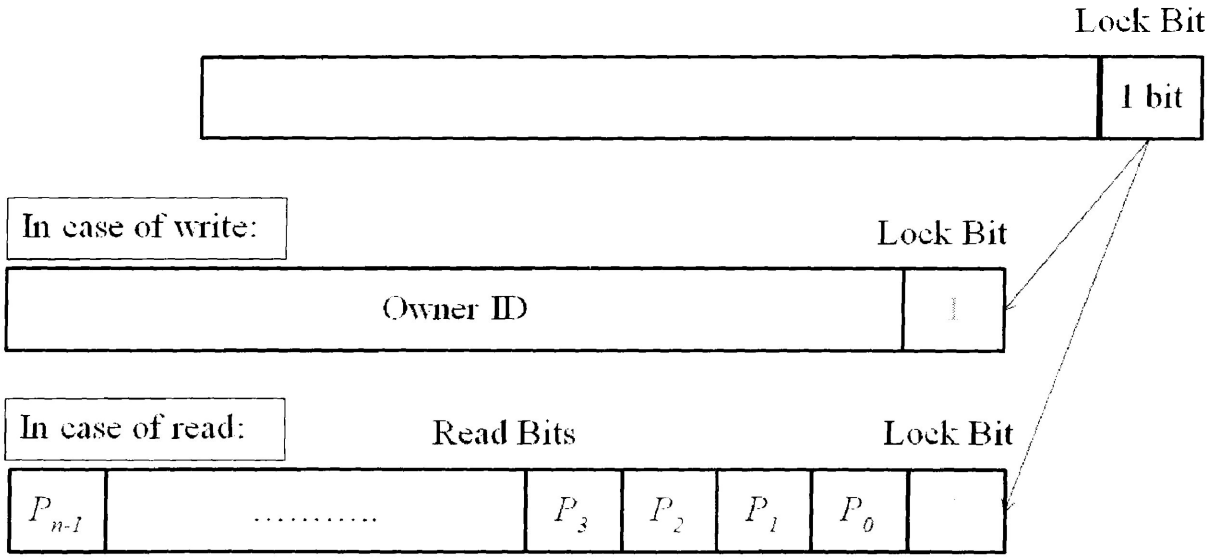


Figure 3.1 Structure of a single lock entry in TRT.

In TRT, the following set of operations is performed when a writing transaction executes:

- **Run transactional code:** Execute the transactional code, meanwhile, each time a shared memory location is accessed, the lock entry associated with the memory location is checked. For transactional read operations, if the lock is free, the lock is sampled into the thread's local *read-set* and the corresponding read bit in the lock is set using atomic Compare and Swap (CAS) operation [18]; if the lock is already acquired by another transaction then the current transaction aborts. For transactional write

operations, if the corresponding lock is free and all read bits are clear, then the lock is acquired using CAS. If the lock is already acquired, then two transactions conflict over a shared memory location. The one which acquires the lock sooner is the winner and writes into the shared memory location. The other should abort and restart its transactional section. If a read bit is set, then the writing transaction aborts. This is necessary to guarantee atomicity of transactional sections.

- **Commit:** When a transaction reaches the commit step, its transactional data are valid. The transaction requires clearing the read bits in the lock entries corresponding to its *read-set*. In addition, lock entries corresponding to its *write-set* should be dropped by clearing the lock bits.
- **Abort:** If a transaction needs to be aborted, it should revert all changes that it has made to the shared memory locations. The thread goes through *write-set* and undoes all write operations and clears the lock bits. Also, the transaction should clear its read bits in the lock entries corresponding to its *read-set*.

Figure 3.2 presents the pseudo code for TRT. When a transaction reads from an address in memory (TxLoad()), the corresponding lock is examined to ensure that it has not been acquired by another transaction (line 2). Then, it sets the read bit *i* to indicate that a transaction in thread *i* has read from the address (line 3). When a transaction writes into a memory location (TxStore()), it checks the lock entry corresponding to input address (line 10). If no other transaction has read/written from/to the address, then the lock entry is acquired (line 11) and the new value is written into the memory (line 12); otherwise, the transaction aborts (line 15). When a transaction aborts (TxAbort()), it goes through all nodes of its read-set and write-set to restore the state of the lock table (lines 20-21). In addition, all new values written to memory by TxStore() should be reverted (line 22). When a transaction commits (TxCommit()), all read bits of lock entries corresponding to read memory locations (line 26) and all lock bits corresponding to written memory locations should be cleared (line 27). However, contrary to TxAbort(), there is no need to revert the changes to the memory. Appendix A has the source code for TRT.

3.2.1 Proof of Correctness:

In this section, we prove that TRT algorithm maintains the atomicity and correctness of programs. The values resulting from any serial execution of transactions are assumed to be consistent. A parallel program is considered consistent, if a serial execution of transactions exists, and that generates the same result as the parallel program. Two concurrent transactions are defined as conflicting when they access the same memory location and at least one of them

writes to that location (Read-After-Write, Write-After-Read, and Write-After-Write hazard). Since TRT allows a thread to write to a memory location only if the location has not been read or written by any other thread, any of the aforementioned hazards will not happen. For transactional read operations, any number of threads can read a location as long as it has not been written by another transaction.

In term of atomicity, the key difference between GV4 and TRT is that TRT uses the first transactional read or write operation as the linearization point while GV4 uses the start of a transactional section as the linearization point. Hence, TRT, in contrast to GV4, does not generate unnecessary aborts as discussed in section 2.5.1.

```
1  TxLoad(addr) {
2     if ( lock(addr) is free ) //if lock entry corresponding to addr is free
3        set read bit in lock(addr);
4        return (Mem[addr]);
5     } else
6        TxAbort();
7  }
8
9  TxStore (addr, value) {
10    If (all read bits of lock(addr) are zero and lock(addr) is free ) {
11       Acquire( lock(addr) );
12       Write value in to addr;
13    }
14    else {
15       TxAbort();
16    }
17 }
18
19 TxAbort() {
20    RevertReadSetChanges();
21    DropLocks();
22    RevertMemoryChanges();
23 }
24
25 TxCommit() {
26    RevertReadSetChanges();
27    DropLocks();
28 }
```

Figure 3.2 Pseudo code for TRT.

3.3 Performance of TRT

In this section, we report performance of TRT. All tests were carried out on two Intel Xeon E5660 processors running at 2.8 GHz. Each processor has six cores and is capable of running up to 12 threads simultaneously. Each processor has a 12MB shared L3 cache with 64B cache lines. Each core has a 32KB instruction cache and a 32KB data cache. TABLE 3.1 presents the input arguments used for STAMP benchmarks.

TABLE 3.1 Input arguments for STAMP benchmarks

Benchmarks	**Input Parameters**
Bayes	-v32 -r4096 -n10 -p40 -i2 -e8 -s1
Kmeans	-m15 -n15 -t0.00001 -i inputs/random-n65536- d32-c16.txt
Labyrinth	-i inputs/random-x512-y512-z7-n512.txt
Ssca2	-s20 -i1.0 -u1.0 -l3 -p3
Vacation	-n4 -q60 -u90 -r1048576 -t4194304
Genome	-g16384 -s64 -n16777216

Figure 3.3 presents performance of TRT relative to GV4 in STAMP v0.9.10 benchmarks. The results were normalized to the execution time of GV4; hence bars less than 1 show performance improvement. For each benchmark, the number of threads varies between two and 16. While in some benchmarks, TRT is faster than GV4 on average, i.e. Labyrinth, GV4 works better than TRT in some others, i.e. Kmeans. The main reason that TRT falls behind GV4 in some benchmarks is associated with overhead of abort. In TRT, when a transaction aborts it traverses both *read-set* and *write-set* and reverts all the changes made to the lock table. Depending on the application's data access patterns, *read-set* or *write-set* might grow largely. However, GV4 has to check only its *write-set*. This causes extra overhead in some benchmarks and degrades performance.

To provide better insight into the overhead of abort in TRT, part of a code region taken from Genome is shown in Figure 3.4. TRT degrades performance of Genome by 83% when the number of threads is 16. This is the maximum slowdown across STAMP benchmarks. More than 79% of total aborts happen in the transaction shown in Figure 3.4. In previous parts of the code (not shown in the Figure), the algorithm removes duplicate segments using hash-set and in this specific part, it iterates over unique segments and computes hashes. Hashes are implemented as

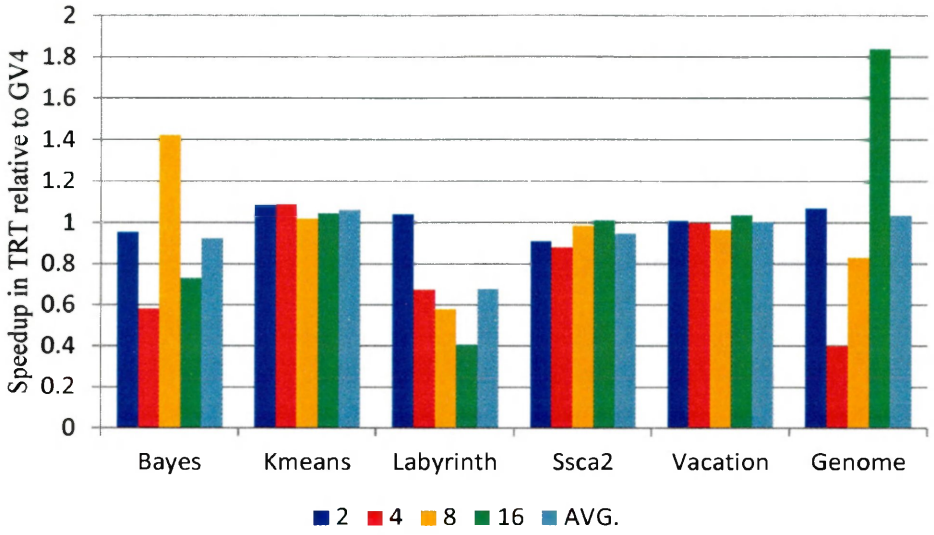


Figure 3.3 Speedup in TRT relative to GV4.

linked-lists and in function TMTABLE_INSERT(), all threads compete to insert nodes into the hashes. Hence, transactions create large *read-set*s and they try to write in one of them. This results in dramatic increase in read-write conflicts and the "for loop" in the program exacerbates this situation.

In the next section, we introduce RGVT which alleviates the overhead of abort and improves performance of TRT.

```
for (j = 1; j < segmentLength; j++) {
        ...
        TM_BEGIN();
        status = TMTABLE_INSERT(startHashTables[j], ...);
        TM_END();
        ...
}
```

Figure 3.4 Part of Genome program from STAMP v0.9.10 benchmark suite.

3.4 *rwConflict* Based GV4-TRT (RGVT)

TRT and GV4 have conflicting effects. While GV4 reduces overhead of abort, it unnecessarily serializes transactions and increases contention over the central global clock even if transactions do not conflict. On the other side, TRT eliminates cost of global clock but increases overhead of abort and introduces read-write conflicts. Hence, none of the two validation policies works well

across all applications. Depending on access pattern of shared data structures in a benchmark, one method may work better than the other.

3.4.1 Read-Write Conflict

One way to combine the best of TRT and GV4 is selecting one of the two techniques based on number of read-write conflicts. We count the number of read-write conflicts and store it in a local variable: *rwConflict*. If *rwConflict* is more than a pre-determined threshold, GV4 is the preferred scheme since TRT increases execution time due to overhead of abort. If *rwConflict* is less than the threshold, then the preferred validation scheme is TRT since GV4 unnecessarily serializes the committing transactions.

The reason for switching between GV4 and TRT based on read-write conflicts is that the number of read-write conflicts greatly affects the performance of TRT. If this number grows, it results in a large read-set and the likelihood of conflict is high. Aborting a transaction with a large read-set is a costly operation which degrades performance. It is worthwhile to mention that if the read-set size grows without creating conflict, TRT will provide acceptable speedup with respect to GV4.

To prevent change of validation scheme at the middle of a transaction, we use read-write lock [8] to synchronize transactions. Only one thread is allowed to change the validation scheme (i.e. thread zero). When a transaction in thread zero decides to change the validation scheme, it acquires the lock in write-mode, changes the granularity, and releases the lock. All the other transactions acquire the lock in read-mode at the start of transactional sections and release the lock when they commit or abort. Using the read-write lock, we guarantee that the validation scheme changes only when there is no running transaction.

3.4.2 Performance of RGVT

Figure 3.5 presents the performance of RGVT relative to GV4. The value of threshold is set to 10 and is used across all applications. We examined different values for threshold and found that threshold of 10 results in maximum speed-up. On average, RGVT improves performance across all benchmarks. Performance of Genome is considerably improved. Genome has a mixed structure: read-write conflicts are frequent in parts of the benchmark and in the other parts, they occur rarely. RGVT manages to utilize GV4 for the conflicting parts and TRT for the non-

conflicting parts. As such, RGVT improves performance of Genome for all variations of number of threads. For Labyrinth and Ssca2, TRT and RGVT are similar. In these two benchmarks, read-write conflicts occur rarely and so, RGVT quite often chooses TRT as the validation scheme. Kmeans and Vacation are not sensitive to the validation policies and so speedup of both TRT and RGVT are marginal. Bayes, similar to Genome, has a mixture of high and low read-write conflict transactions. RGVT adjusts validation policy dynamically and improves performance of Bayes over all variations of number of threads.

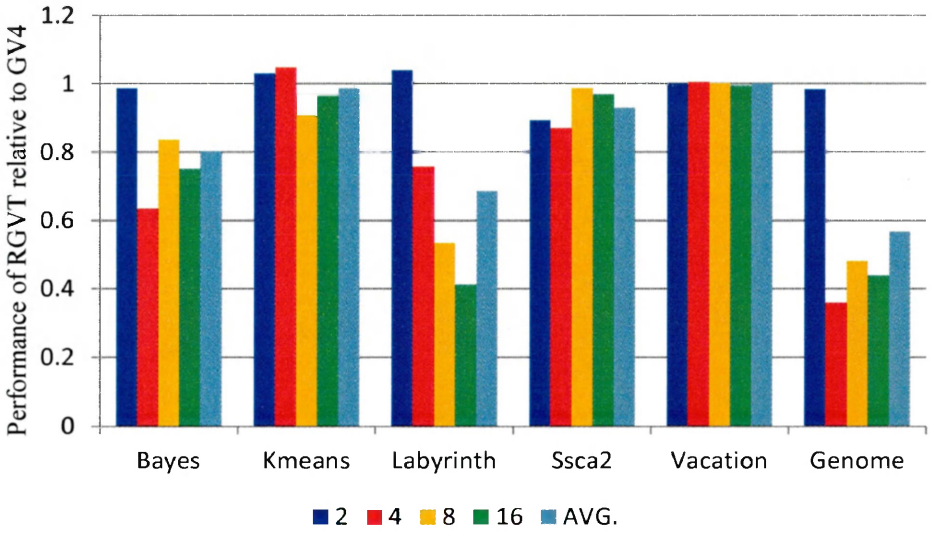


Figure 3.5 Performance of RGVT using rwConflict.

To provide better insight into RGVT, in Figure 3.6, we report how often transactions execute in GV4 and TRT modes. For each benchmark, the number of threads varies from two to 16. In Bayes, Labyrinth and Ssca2, virtually all transactions execute in TRT mode, which confirms the results presented in Figure 3.5, since speedup in these benchmarks is significant. In Genome, TRT is chosen as the initial validation method. When the size of the *read-set* and *write-set* grows the rate of read-write conflict increases. As such, *rwConflict* exceeds the threshold and RGVT selects GV4 for validation to avoid costly aborts. In Kmeans, quite often, GV4 is selected for validation. Therefore, sometimes there is performance degradation in this benchmark. This shows that *rwConflict* is not very accurate in predicting applications' behaviour. In the next section, we discuss a different technique to switch between GV4 and TRT.

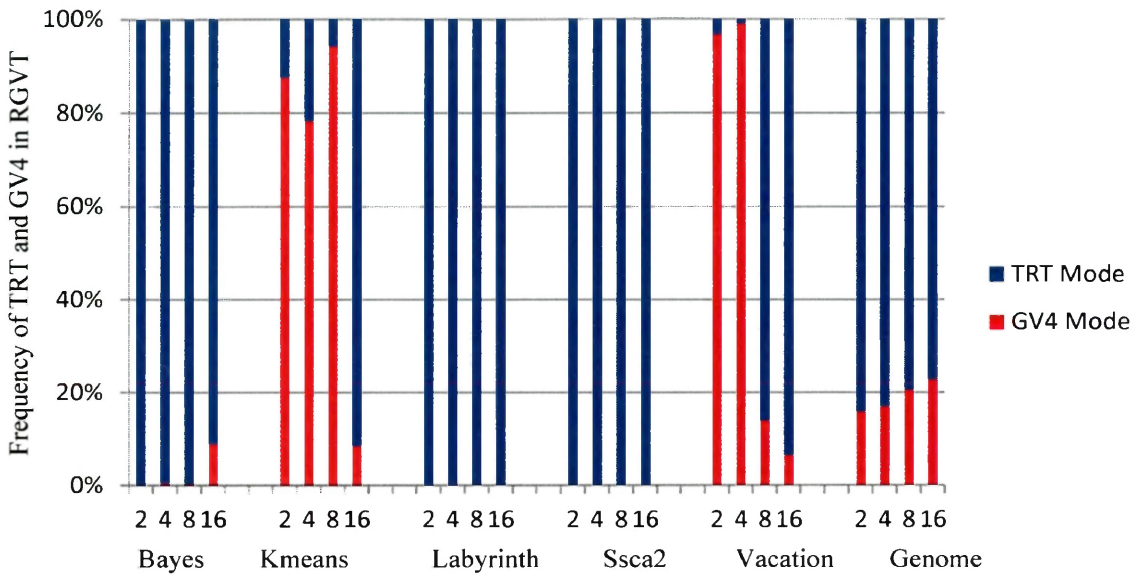


Figure 3.6 Frequency of GV4 and TRT in RGVT.

3.5 Perceptron GV4-TRT (PGVT)

In this section, we propose a new method which decides on validation policy based on probability of conflicts in transactions. As we will show later, this method improves performance more than *rwConflict*. To speculate the outcome of a transaction, we use a perceptron-based contention predictor (CP). Perceptron is a neural network which was introduced in 1962 to study brain function [46]. Jimenez and Lin exploited perceptron and proposed a highly accurate branch predictor [47]. We use perceptron to speculate outcome of transactions. A perceptron learns a Boolean function of n inputs. In our case, the function predicts whether a transaction commits or fails and inputs are global transaction history.

Figure 3.7 shows how a perceptron works. A perceptron is composed of a weight vector (w_i) and an input vector (x_i). Elements of the weight vector are signed integers and elements of the input vector are bipolar, *i.e.* each x_i is either -1, meaning transaction fails or 1, meaning transaction commits. The output y of a perceptron is computed as follows:

$$y = w_0 + \sum_{n=1}^{n} (x_i w_i)$$

A non-negative output is interpreted as predict commit. x_0 is always 1, so instead of learning correlation with a previous transaction, the bias weight, w_0, learns the bias of the transaction independent of other transactions.

Once a transaction executes and its real outcome is determined (commit or abort) the perceptron is trained. If the outcome of the transaction agrees with x_i, w_i is incremented; otherwise, w_i is decremented. Intuitively, when there is positive correlation between transaction i and the current transaction, the w_i becomes large. On the other side, if there is negative correlation between transaction i and the current transaction, then w_i becomes a large negative number. When there is a weak correlation, then weight remains close to zero and contributes little to the output.

Figure 3.8 shows the structure of the CP in a program with two threads. Each thread has a local CP and the local CP is composed of N link-lists. Each node of the link-list has the starting address of a transaction and a perceptron (weight vector and input vector). We use inline assembly to read Program Counter (PC). The PC is written into the starting address fields of the link-list nodes. When a transaction starts, the starting address of the transaction is hashed to an index $i \in 0...N-1$ into the table of link-lists. Then, a node with matching starting address in the link-list is selected (if there is no matching node, then we assume that the transaction commits successfully). The value of y is calculated as dot product of weight vector and input vector. If y is non-negative, then the transaction is predicted to commit; otherwise, it is predicted to abort. Once the actual outcome of the transaction is known, the weight vector is updated and the input vector is shifted with +1/-1 if the transaction commits/aborts.

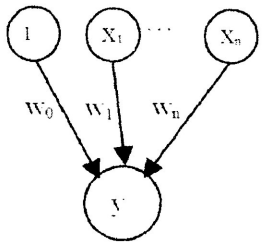


Figure 3.7 Weight vector and input vector in a perceptron.

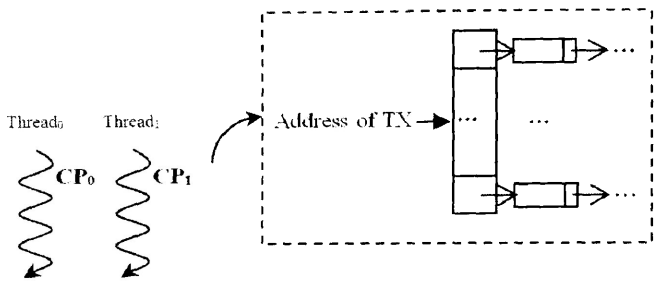


Figure 3.8 A program with two threads and two local contention predictors.

Figure 3.9 presents pseudo code for switching between TRT and GV4 in PGVT. Similar to adaptive TRT, decision is made by thread 0. We use read-write lock to guarantee that when thread 0 changes validation policy, there is no running transaction.

```
1    TxStart(){
2      if (threadID == 0) {
3        if (predConflict()==1)
4        {
5          //if perceptron predictor speculates conflict
6          rwLock.acquireForWrite();
7           //choose GV4 or TRT;
8           rwLock.release();
9        }
10     } else {
11       rwLock.acquireForRead();
12     }
13   }
```

Figure 3.9 Adaptive algorithm.

Figure 3.10 shows lookup and update in a perceptron predictor. When a transaction starts it accesses table of perceptrons using starting address of the transaction (line 1). Then, the CP computes the dot product of history vector (x[]) and weight vector (w[]) (lines 6-9). If the dot product is positive or zero, the transaction is predicted to commit; otherwise, it is predicted to abort. When a transaction commits or aborts, weight vector and history vector are updated in CPUpdate(). The weight vector is updated when a misprediction happens or when the output is less than a statically determined threshold (lines 17-25). When the outcome of a transaction agrees with x[i] (commit_nconflict=x[i]=1 or commit_nconflict=x[i]=-1) then w[i] is incremented; otherwise, w[i] is decremented (line 23). Intuitively, when there is positive correlation, the weight becomes large. On the other side, when there is negative correlation, the

weight becomes negative with large magnitude. In both cases, the weight has a large influence on the prediction. When there is weak correlation, the weight remains close to 0 and contributes little to the output of the CP (y). In addition, the history vector is updated (lines 26-29).

```
1  CPLookup(addr)//addr is starting address of a TX
2  {
3     w[]=Table[hash(addr)];//weight vector is read from table of  perceptrons
4     y = w[0];
5
6     for(i=1; i<historyLength; i++)
7     {
8        y+=x[i] . w[i]; //computes dot product of history vector and weight vector
9     }
10    return y >= 0 ;
11 }
12
13
14 CPUpdate(addr, commit_ncommit) //commit_nconflict=1 if a TX commits.
15                                    // commit_nconflict =-1 if a TX aborts.
16 {
17    if(y < θ or miprediction) //if y is less than θ or CP
18                              //mispredicted, then w[] is updated
19
20    {
21       for(i=0; i< historyLength; i++)
22       {
23          w[i] += commit_nconflict . x[i];
24       }
25    }
26    for(i= historyLength-1; i>1; i--)
27    {
28       x[i] = x[i-1];
29    }
30    x[1] = commit_nconflict;
31 }
```

Figure 3.10 Lookup and update in a perceptron predictor.

3.5.1 Accuracy of contention predictors

In Figure 3.11, we report accuracy of contention predictors. The predictors are implemented in the baseline GV4 but are not used for speculation. Note that in a parallel program with *n* threads, there are *n* predictors, one predictor for each thread. We report average data for the *n* predictors.

Accuracy shows how often speculated transactional conflicts turn to be the correct ones. Figure 3.11 shows accuracy for predictors with variable history lengths. The higher the bar, the better the prediction. For each benchmark, the number of threads is equal to 16. The history length changes between one and 32. Accuracy for other number of threads is similar. In Kmeans, Ssca2, Vacation, and Genome, the accuracy is more than 96%. On average, in Bayes and Labyrinth, accuracy is 64% and 75%, respectively.

3.5.2 Performance of PGVT

Figure 3.12 depicts the performance of PGVT relative to GV4. On average, PGVT improves performance across all benchmarks. On average, PGVT improves performance of Stamp benchmarks from 20% in Bayes to 47% in Labyrinth on average.

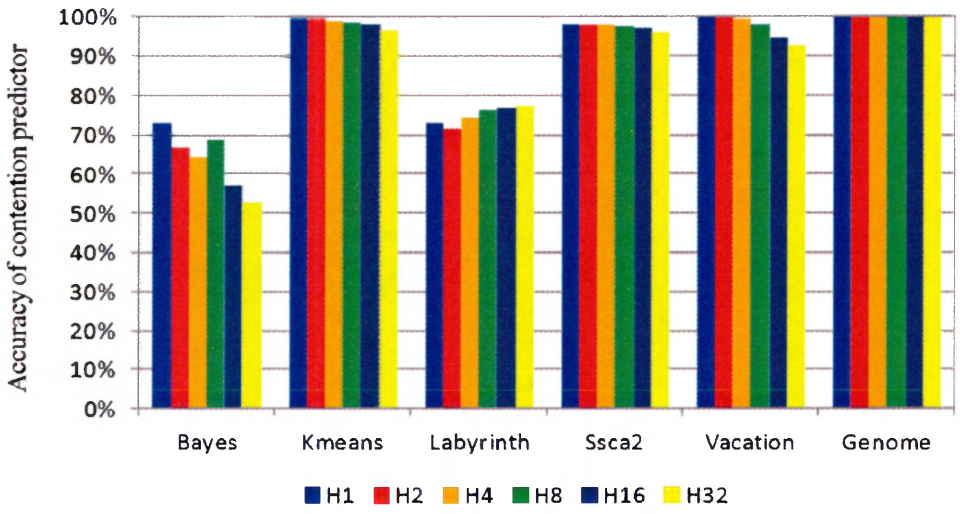


Figure 3.11 Accuracy of contention predictors with variable history lengths.

To provide better insight into PGVT, in Figure 3.13, we report how often transactions execute in GV4 and TRT modes. For each benchmark, the number of threads varies from two to 16. As shown in the Figure, quite often transactions run in TRT mode, which shows that TRT is the main source of performance.

A quick comparison between the two adaptive methods in this chapter suggests that PGVT offers a better performance improvement compared to the RGVT in most of the cases. The reason is that contention predictors tend to be more accurate than *rwConflct* approach. As such, PGVT improves performance more than *rwConflict* approach.

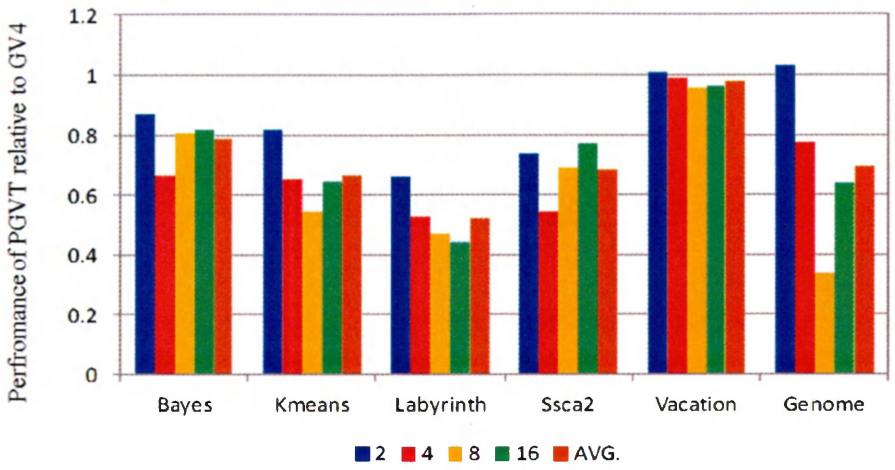


Figure 3.12 Performance of PGVT relative to GV4.

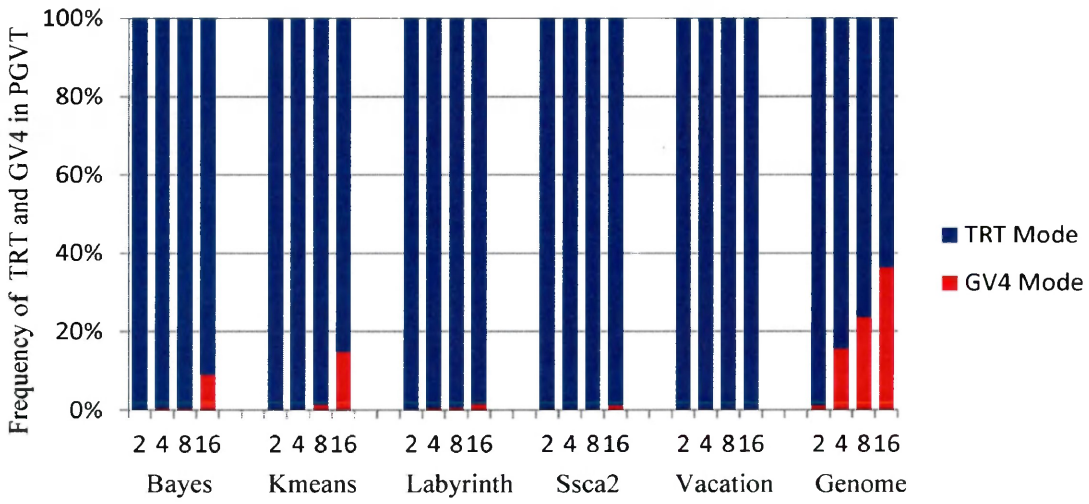


Figure 3.13 Frequency of GV4 and TRT in PGVT.

3.6 Summary

In this chapter, we proposed TRT to alleviate the overhead of global clock in time-based software transactional memory systems. TRT is a novel method that uses a distributed approach to eliminate the need for global clock; however, there is an overhead to this method. The overhead of abort impedes the performance of the system. In order to circumvent this problem, we proposed two techniques which dynamically select either GV4 or TRT. The first one is RGVT and relies on the number of *rwConflict*. The second one exploits perceptron predictors. Results indicate that both techniques improve the performance over the baseline GV4. However,

perceptron-based technique tends to be more accurate in term of predicting the behaviour of applications. Therefore, it is able to provide performance gain over RGVT.

Chapter 4 Hardware Support for Set Associative Lock (HW-SAL)

In this chapter, we present and evaluate "Hardware Support for Set Associative Lock" to boost performance of STMs. In section 4.1, we present the motivation behind this technique. In section 4.2, we briefly examine the Gem5 [48] simulator used to evaluate SAL and measure its performance impact. In section 4.3, a brief overview of false conflicts is presented and we explain how SAL is implemented in software (SW-SAL). In section 4.4, we discuss hardware SAL (HW-SAL). Finally, we report speed-up for SW-SAL and HW-SAL in section 4.5.

4.1 Motivation

Most STM systems rely on lock tables to maintain consistency of shared memory locations. A hash function maps memory locations to the lock table entries. Since memory is larger than the lock table, it is possible that two or more memory addresses are mapped to the same entry of the lock table. The aliasing in the lock table results in false conflicts which reduces the concurrency of the system and impedes performance. In order to address this issue, a mechanism should be employed that reduces false conflicts of a direct mapped table and has an adequate performance or better, compared to the direct mapped table. One such mechanism could be employing a better hash function in order to reduce the probability of false conflicts. However, a single hash function cannot address the complicated data access patterns of different applications. Additionally, other methods of hashing, such as open addressing with linear or quadratic probing [49] or Cuckoo hashing [50] lag performance and they do not eliminate false conflict completely. Set associative lock system in software was introduced, in order to reduce the false aborts [33]; however, the extra computational overhead of SW-SAL, combined with frequent accesses to the lock table, increases the cost of SW-SAL. Additionally, to harness the true power of set associative locks, associativity must be increased which exacerbates the overhead of the system in software. Therefore, it is prudent to exploit a hardware method to remove false abort and boost performance. In this chapter, we propose a mechanism that provides hardware support for Software transactional memory systems. In a nutshell, this system moves the lock table to the hardware in order to alleviate the computational overhead of SW-SAL and also reduce the false conflicts using high associative lock tables. As such, the true power of SAL could be harnessed without sacrificing performance.

4.2 Gem5 Simulator

We use Gem5 [48] simulator to evaluate performance impact of SAL. The Gem5 simulator provides a flexible and modular simulation system that is capable of evaluating a broad range of systems and is widely available to all researchers. This infrastructure provides flexibility by offering a diverse set of CPU models and memory system models. A commitment to modularity and clean interfaces allows researchers to focus on a particular aspect of the code without understanding the entire baseline code. The BSD based license makes the code available to all researchers without awkward legal restrictions.

The Gem5 [48] simulator is the combination of M5 [51] and GEMS [52] simulators. M5 provides a highly configurable simulation framework, multiple ISAs, and diverse CPU models. GEMS simulator complements these features with a detailed and flexible memory system, including support for multiple cache coherence protocols and interconnect models. Currently, Gem5 supports most of commercial ISAs (ARM, ALPHA, MIPS, Power, SPARC, and x86), including bootable Linux Operating System on three of them (ARM, ALPHA, and x86).

The Gem5 simulator currently provides four different CPU models, each of which lies at a unique point in the speed-vs.-accuracy spectrum. *AtomicSimple* is a minimal single IPC CPU model. *TimingSimple* is similar but also simulates the timing of memory references, *InOrder* is a pipelined, in-order CPU, and *O3* is a pipelined, out-of-order CPU model. Both the *O3* and *InOrder* models are "execute-in-execute" designs [51].

Each execution-driven CPU model can operate in either System-call Emulation (SE) mode or Full-system (FS) mode. System-call Emulation (SE) mode avoids the need to model devices or an operating system (OS) by emulating most of system-level services. Meanwhile, Full-System (FS) mode executes both user-level and kernel-level instructions and models a complete system including the OS and devices.

The Gem5 simulator includes two different memory system models, Classic and Ruby. The Classic model (from M5) provides a fast and easily configurable memory system, while the Ruby model (from GEMS) provides a flexible infrastructure capable of accurately simulating a wide variety of cache coherence memory systems.

In this thesis, we chose to use an ALPHA ISA, since it is a RISC architecture and is more flexible compared to x86. The simulations are performed using FS simulation model, running a Linux v2.6 and Timing CPU model with a classis memory model.

4.3 Set Associative Locks

As mentioned earlier, TL2 associates locks with shared memory locations to handle concurrent accesses to the shared data. The locks are organized as a large table and memory is striped using a hash function to map memory locations to the table entries. The benefit of the lock table is that it does not require any manual insertion of locks or modification of data structures. This is in contrast to per object scheme which requires manual or compiler-assisted insertion of locks in the data structures of programs [12]. Figure 4.1 depicts locking scheme in TL2.

The size of each entry in the lock table is equal to the size of address on the host machine. The least significant bit (LSB) of the lock shows whether the lock is free or acquired. If the LSB is zero (free), the rest of the lock shows the time stamp of the last transaction that wrote to a memory address covered by the lock. If the LSB is one (acquired), the rest of the lock holds the address of the owner transaction. Since the lock is word-aligned, the LSB can safely represent the status of the lock (free or acquired).

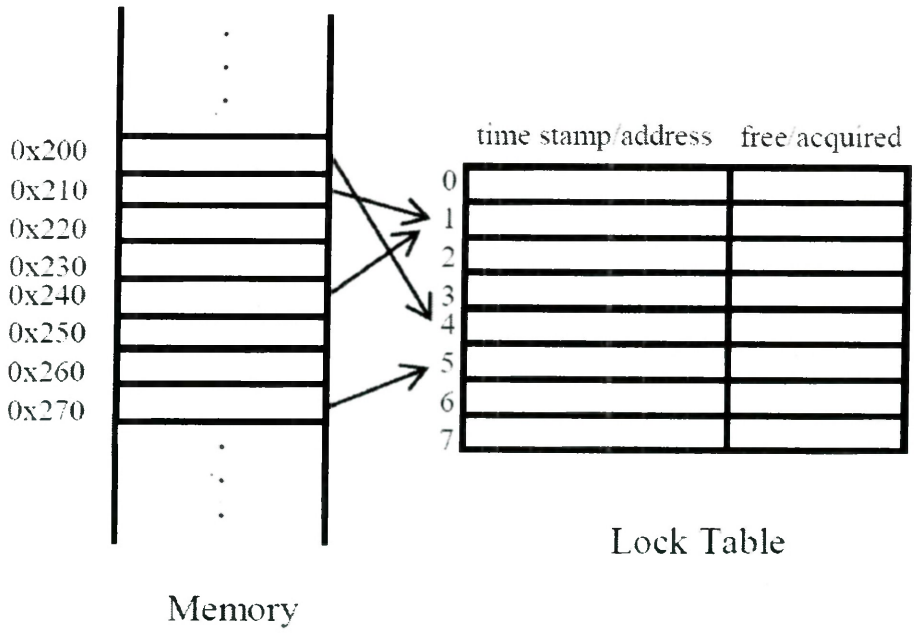


Figure 4.1 Memory address space is mapped to a lock table in TL2.

A conflict happens when two or more transactions access a memory location and at least one of the transactions write into the memory. By tracking entries of the lock table, transactions are able to detect conflicts. The first writing transaction acquires the lock which corresponds to the memory location. Other transactions find the lock acquired and abort. Since the lock table is smaller than applications' data memory footprint, different memory addresses may map to the same entry of the lock table. For example in Figure 4.1, memory blocks at both addresses 0x210 and 0x240 map to table entry one. This situation is called false conflict. In the event of false conflict, transactions are aborted conservatively which reduces concurrency level in programs.

4.3.1 Frequency of False Conflicts

In this section, we study the effect of aliasing-induced conflicts (false conflicts) on concurrency level of benchmarks. Figure 4.2 shows how often failure of transactions is due to false conflicts. The lock table used in our simulations has 2^{20} entries. For each benchmark, the number of threads varies from two to 16. False conflict rate varies over different benchmarks. In Labyrinth, false conflict is the dominant factor in failure of transactions. On average, 74% of aborts in Labyrinth are due to false conflicts. On the other side, Genome is the least sensitive benchmark to false conflicts. On average, 8% of aborts in Genome are induced by false conflicts. In Ssca2 and Vacation, false conflicts are responsible for 39% and 22% of aborts on average.

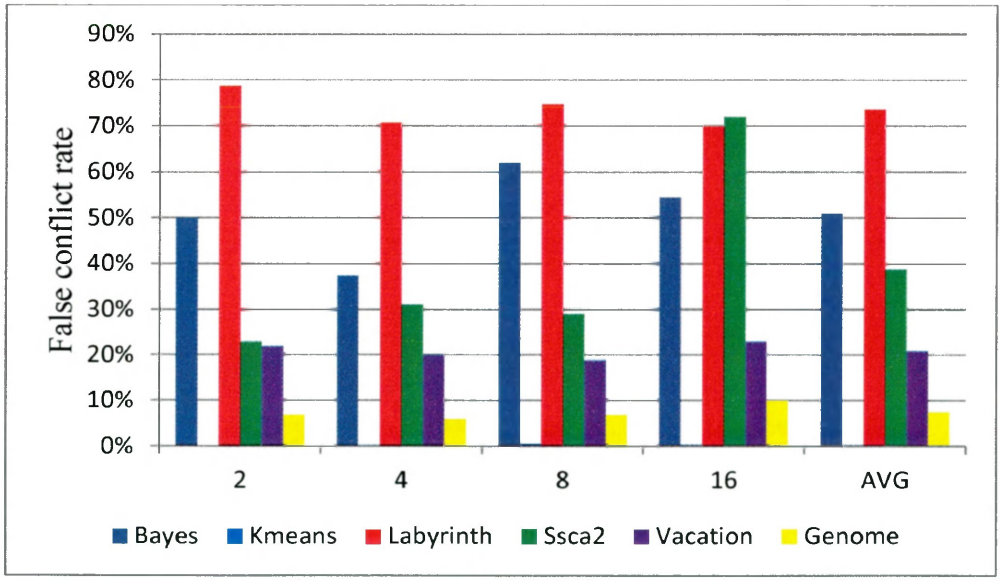


Figure 4.2 False conflicts in STAMP benchmarks.

Kmeans is not sensitive to false conflicts since the average rate for that benchmark is close to 0%. Bayes however, shows sensitivity to false conflicts in a rate close to 51%. False conflict degrades performance since transactions that experience false conflicts should abort. In the next section, we explain SW-SAL technique [33] to reduce aliasing-induced conflicts in STMs.

4.3.2 SW-SAL

The structure of the lock table in Figure 4.1 is similar to a direct-mapped cache in which a memory address is mapped to exactly one location in the cache (or lock table in the case of STM). Since there are more memory blocks than the lock entries, transactions compete for lock entries. To see how current mapping of memory addresses to locks degrades performance, consider transactions A and B accessing addresses 0x210 and 0x240, respectively and assume that the address mapping is according to Figure 4.1. To make the situation worse, suppose that transaction A writes into 0x210 *n* times and simultaneously, transaction B reads from 0x240. If transaction A acquires lock after transaction B reads from 0x240, then transaction B aborts since during validation, it finds the lock is acquired (if transaction A is not committed yet) or it detects mismatch in version number (if transaction A is already committed). All attempts by transaction B to read from 0x240 fails although A and B access different memory locations. Transaction B should wait until transaction A writes into 0x210 *n* times, then it can read from 0x240. This is known as false conflict. False conflicts may result in serious problem because they cannot be foreseen by programmers and may reduce performance significantly. As an example, Damron et al. [14] demonstrated that performance of their hybrid transactional memory in Berkeley DB lock subsystem benchmark decreases when scaling from 32 to 48 processors due to hash collisions in the lock table entries.

Set Associative Lock (SAL) [33] is a software technique that reduces aborts due to false conflicts. In SAL, a memory addresses is mapped to an entry of the lock. Each entry of the lock includes a set of nodes. Size of all sets in the lock table entries are the same. For example, in a 2-way SAL, there are two nodes per set. It is easier to see SAL logically as a two dimensional table. Figure 4.3 shows a 2-way SAL where each set consists of two nodes, thus giving us rows and columns. Similarly, a 4-way SAL has four nodes per set. In addition to the fields specified in Figure 4.1, each node in the lock table has address of memory location which is mapped to the node. To map a memory address to a node in the lock table, first a unique set of the lock table is

specified by a hash function. Then, all nodes in the set are searched for a matching address. If a matching address is found, then the corresponding node is returned; otherwise, one of the nodes in the set which is free (its lock bit is zero) is randomly selected and the address field of the node is overwritten. In the event that all nodes in the set are locked, then a null pointer is returned.

Consider transaction *W*, which writes to memory address *a* that is mapped to set *i* of the lock table. If address of one of the nodes in set *i* matches address *a*, then transaction *W* attempts to acquire lock field of the matching node; otherwise, transaction *W* attempts to acquire lock field of a free node (a free node is a node which its lock field is free). In this case, the most recent writing transaction to address *a* has already committed and transaction *W* can acquire the lock. If all nodes of set *i* are already acquired, then there is no available lock in set *i* to be acquired by transaction *W*. As such, transaction *W* aborts.

If a transaction reads a memory address, it allocates a node in the read-set for the corresponding memory location. In commit, the transaction validates all addresses in the read-set. Assume that transaction *R* reads memory address *a* that is mapped to set *i* of the lock table. If address of one of the nodes in set *i* matches address *a*, then transaction *R* checks the lock field of the matching node. It aborts if the lock is acquired or the version of the lock is more than the local *rv*. If there is no matching address in set *i* then transaction *R* finds a free node in set *i*. In this case, transaction *R* overwrites the address field of the node. If all nodes of set *i* are already acquired, then transaction *R* aborts.

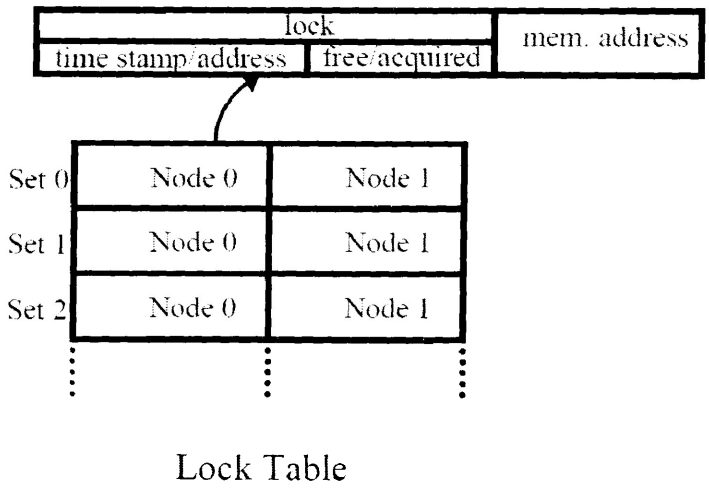


Figure 4.3 Structure of the lock table in a 2-way SAL.

In SAL, if two transactions access two different memory addresses that are mapped to the same set, then two different nodes of the set may be used for the two addresses. As such, the frequency of false conflicts is reduced in SAL. On the other side, in the baseline scheme, if the two memory addresses are mapped to the same entry of the lock table, then one of the two transactions aborts.

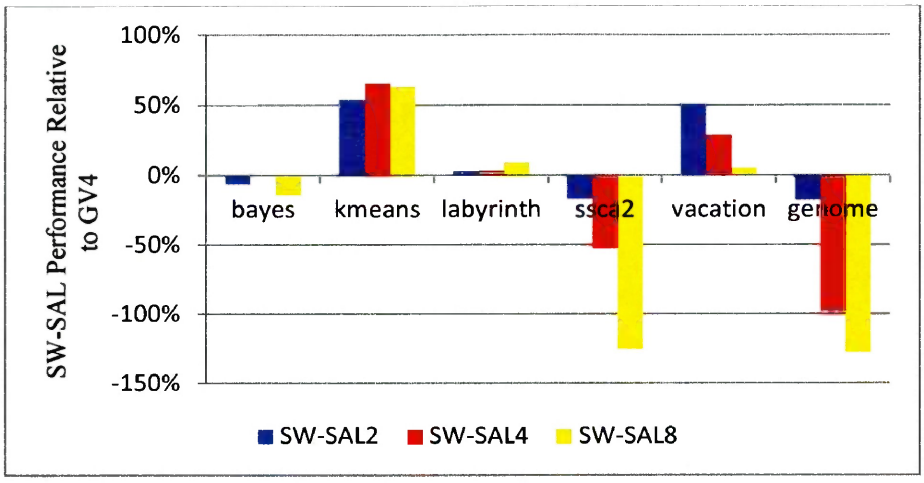
4.3.3 SW-SAL Performance

In this section, we present the performance of SW-SAL. For all our evaluations, we perform full-system simulations using Gem5 [48].We model a CMP based on Alpha 21264 architecture. Each core has a private instruction cache and data cache. TABLE 4.1 shows the configuration of the processor. Figure 4.4 presents the performance of SW-SAL relative to the baseline scheme. Positive bars represent speedup under SW-SAL. The associativity of the lock table changes from 2 to 8 and the number of threads varies from 2 to 16.

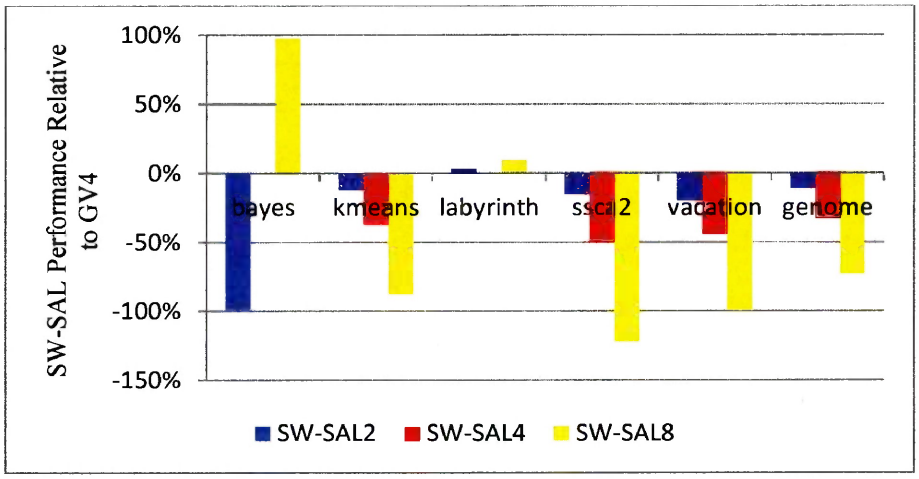
TABLE 4.1 Configuration of the processors in Gem5.

Benchmarks	Input Parameters
Processors	2-16 cores Alpha ISA, 2GHz
L_1 I&D Caches	64kB, 2-way associative, 64-byte line size, 1 cycle latency
L_2 Cache	Shared 2MB, 8-way associative, 64-byte line size, 10 cycles latency
Main Memory	2048MB, 100 cycles latency

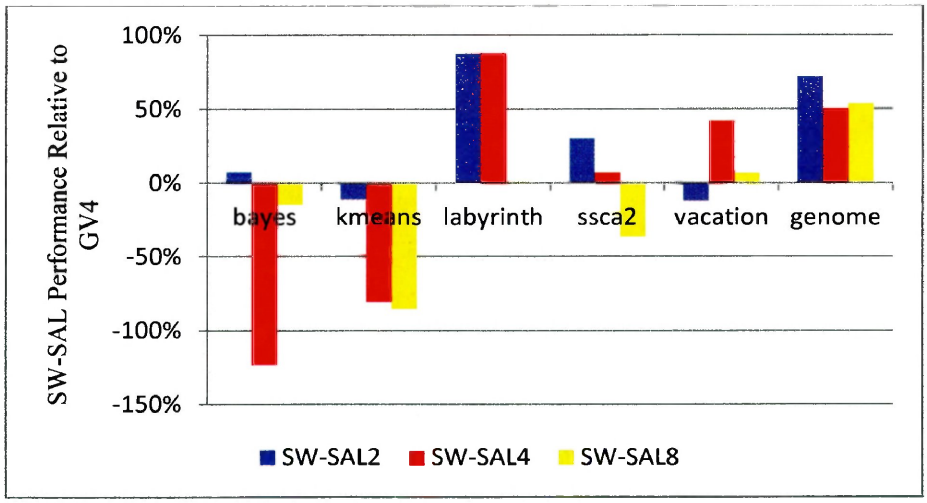
SW-SAL improves performance of Labyrinth. This aligns with frequency of false conflicts reported in Figure 4.2. On average, SAL2, SAL4, and SAL8 improve performance of Labyrinth by 26%, 27%, and 6% respectively. The associativity of SAL plays an important role in the performance of SW-SAL. SAL2 and SAL4 outperform GV4 in the Labyrinth benchmark. This is due to the fact that false conflict reduces as the associativity increases. However, SAL8 does not provide speedup comparable to SAL2 and SAL4. This is mainly due to the overhead of lock table. As associativity increases, it takes longer to find a matching node in the lock table. As such, timing overhead of SAL8 offsets false abort reduction.



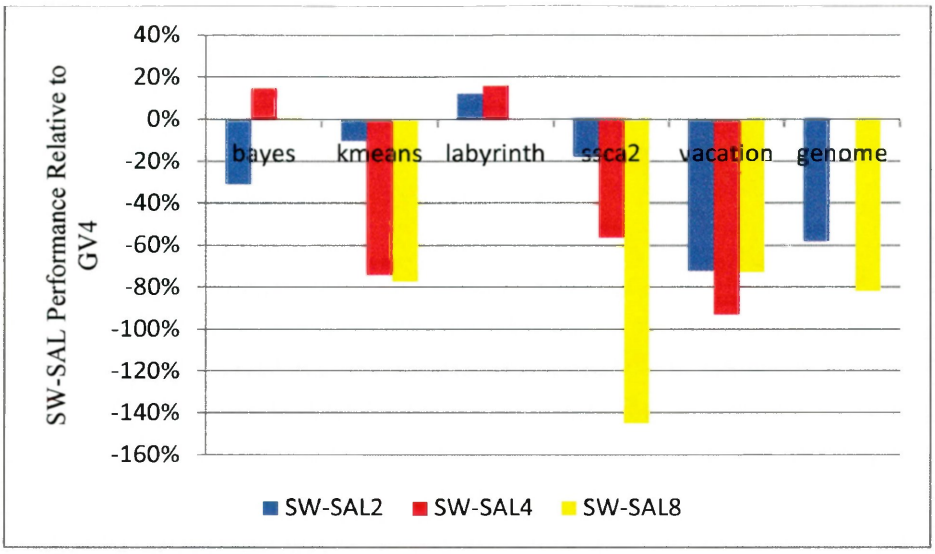
(a) 2 threads



(b) 4 threads



(c) 8 threads



(d) 16 threads

Figure 4.4 Performance improvements in SW-SAL relative to the baseline scheme.

In Ssca2, Vacation, and Genome, performance improvement is not as significant as in Labyrinth; in some cases there is performance degradation. On average, in Ssca2, SAL2, SAL4, and SAL8 degrade performance by 4%, 37%, and 107%, respectively. In Vacation, SAL2, SAL4, and SAL8 degrade performance by 13%, 16%, and 40%, respectively. In Genome, SAL2, SAL4, and SAL8 degrade performance by 3%, 19%, and 57%. The reason that SAL falls behind the baseline scheme for some of the configurations is that the overhead of SAL in lock table is more than the performance gain of SAL. From the performance improvement reported in Figure 4.4, we conclude that SAL2 is faster than SAL4 and SAL8 in STAMP v0.9.10 benchmarks. The simplicity of SAL2 reduces the overhead of the lock table and so, SAL2 is able to improve performance in some of the benchmarks with moderate and low false conflict rates.

In the next section, we present hardware SAL (HW-SAL) which moves the lock table from software to hardware to reduce the overhead of lock table in software.

4.4 Hardware SAL (HW-SAL)

Figure 4.5 depicts lock table in HW-SAL. Each entry of the table consists of a set of nodes. Each node corresponds to a location in the memory. Each node in the lock entry contains the lock which corresponds to the memory location it represents. It also contains the corresponding memory address. A timestamp is also added to each node for replacement. The last field of a

lock entry is an index which would act as a pointer in software space, pointing to a node in the lock table.

Lock table is a single memory unit on the processor chip and each core in the processor is connected to the lock table through a dedicated bus. Figure 4.6 depicts four processors connected to the lock table. Since the lock table is required to provide a very limited number of operations, the bus controller will be extremely simple and could be implemented at a very low cost.

4.4.1 ISA Augmentation

To communicate with the lock table in hardware, we add new instructions to the instruction set of the processor. ALPHA processor has a RISC architecture that makes definition of new instructions easier compared to CISC architectures such as x86. The Alpha ISA has a fixed instruction length of 32 bits. Alpha itself is a 64 bit machine; meaning that operands are 64 bits. Figure 4.7 shows the structure of the instructions in Alpha [21].

In order to add new instructions to Alpha, we used the opcode of integer instructions. We pass values to the HW-SAL through registers. Before calling a SAL-instruction in software, source operands of the instruction such as address and lock values are written in to the hardware registers. The compiler may use some of these registers and overwriting these registers will corrupt the logic of the program. Therefore, we checkpoint these registers in memory and then restore their contents after calling HW-SAL instructions.

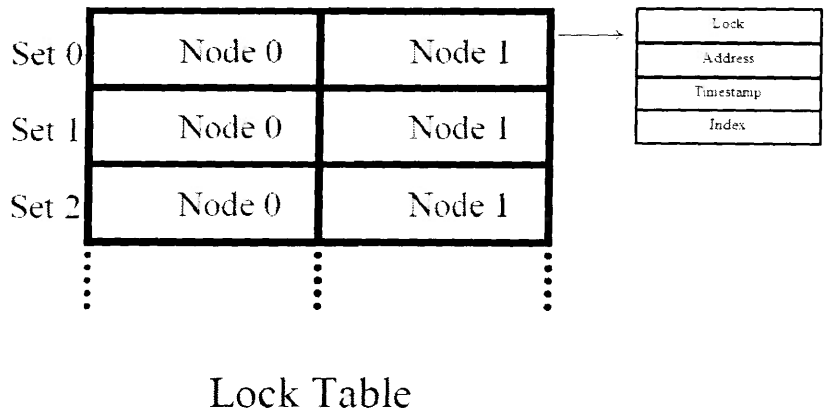


Figure 4.5 Structure of the lock table in HW-SAL.

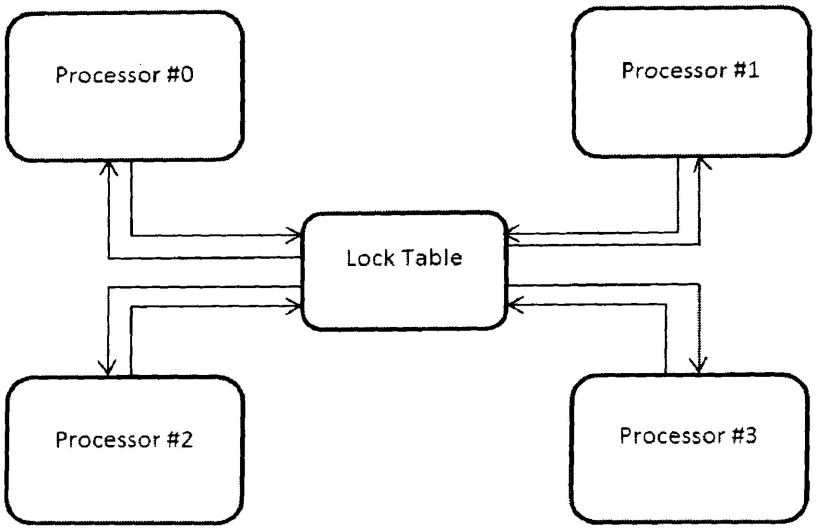


Figure 4.6 A HW-SAL with four cores.

31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0																																Type
Opcode					Ra					Rb					Unused			0			Function							Rc					Integer operate
Opcode					Ra					Literal								1			Function							Rc					Integer operate literal
Opcode					Ra					Rb								Function										Rc					Floating-point operate
Opcode					Ra					Rb											Displacement												Memory format
Opcode					Ra																Displacement												Branch format
Opcode																					Function												CALL_PAL format

Figure 4.7 Instruction format in Alpha architecture.

We add four new instructions in HW-SAL to access the lock table:

- XACTION_READLOCKENTRY: It is used to read a lock entry specified by an index. The index will act as a pointer in software space to the location of the lock in the hardware. The content of the lock is returned to the processor through the destination operand.
- XACTION_ACQUIRELOCK: It tries to acquire an entry of the lock specified by the address of the shared memory location. The index of the lock is returned through the destination field.
- XACTION_WRITETOLOCK: It writes to an entry of the lock that matches the provided index. This function returns nothing.
- XACTION_SALCAS: It tries to write to an entry of the lock, mimicking the behaviour of Compare and Swap operation in STM.

Appendix B contains the source code for SAL instructions. The opcode for all the instructions is the same. However, the function and the immediate fields of each instruction determine what operation should be performed by hardware. The definition of the SAL instructions is provided as a header file for the software. The assembly language is for the ALPHA processor and could be adapted for other architectures.

4.4.2 HW-SAL in Gem5

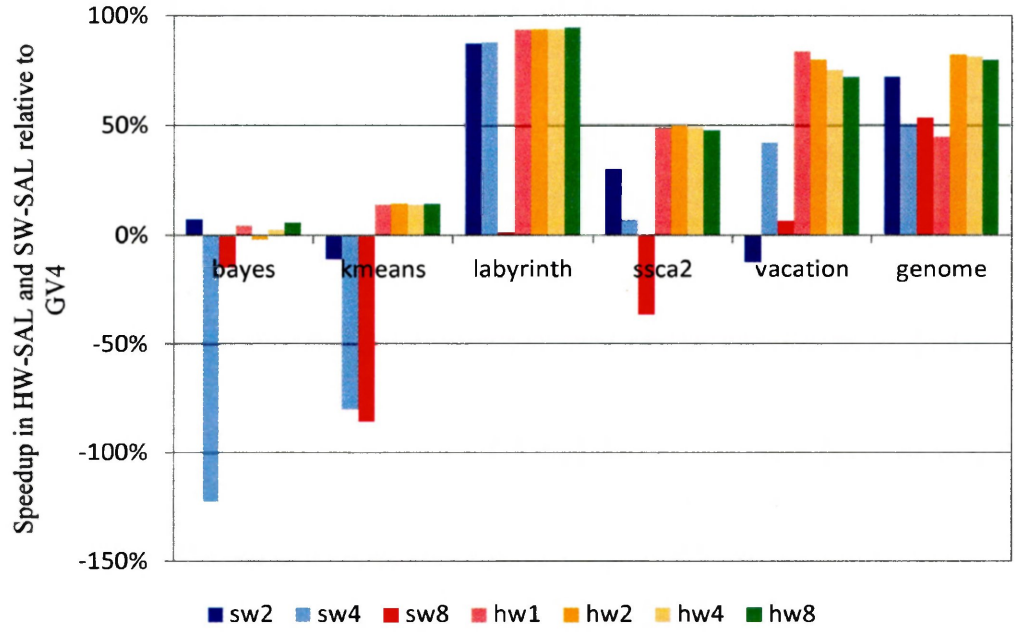
In order to simulate HW-SAL, the Gem5 code should be changed. Gem5 is a mixture of C++ programming language and Python scripting language; modules are implemented in C++ and the simulation scripts are in Python. Therefore, the HW-SAL modules are implemented in C++. Appendix B .1 contains the code where lock data structure is defined and contains a snippet of this code which defines data structure for lock entries and shows HW-SAL function declarations. HW-SAL functions are implemented as part of simple_thread class. Therefore, it is convenient to directly access them from the thread namespace of a processor. As shown in Appendix B.1, each lock node has the four fields as discussed earlier. The function headers are also provided in Appendix B.1.

Since there is only one lock table in hardware, lock table variable in Gem5 must be declared as a static variable; meaning that there is only one instance of the lock table during the simulation and the instance is accessible by other C++ objects.

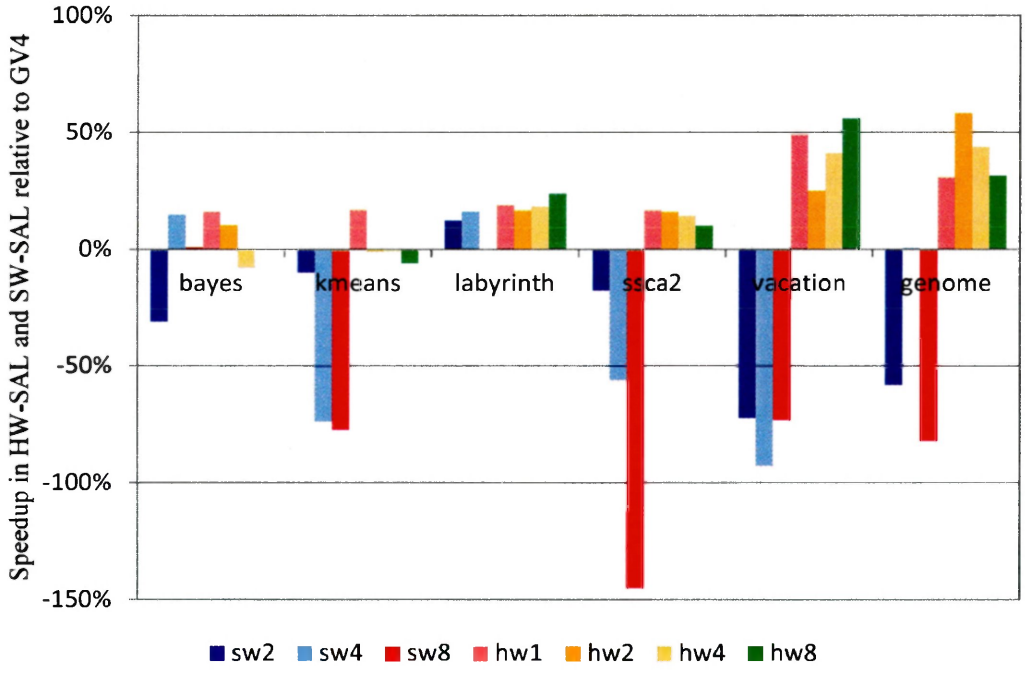
Associativity of the lock table can be determined through a configuration file. This greatly increases the flexibility of the simulator; otherwise, each time that a new associativity is desired, the whole system must be recompiled. Also, the size of the lock table can be changed through a configuration file at the beginning of the simulation.

4.5 Performance Evaluation

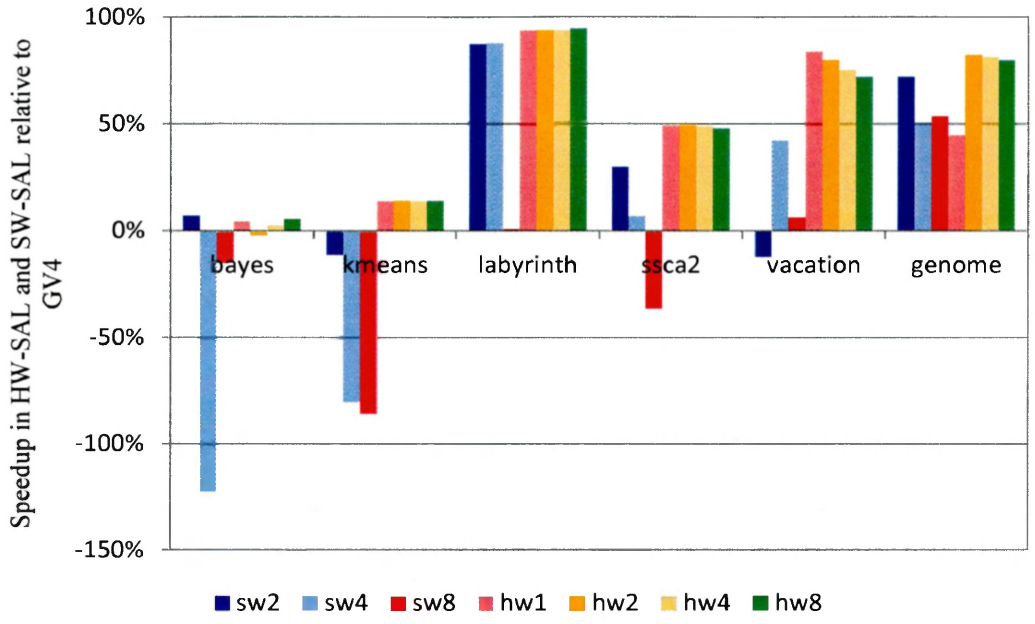
In this section, we report the performance of HW-SAL and SW-SAL compared to the baseline TL2. Figure 4.8 depicts the speedup gained by HW-SAL and SW-SAL with different number of threads, ranging from 2 to 16.



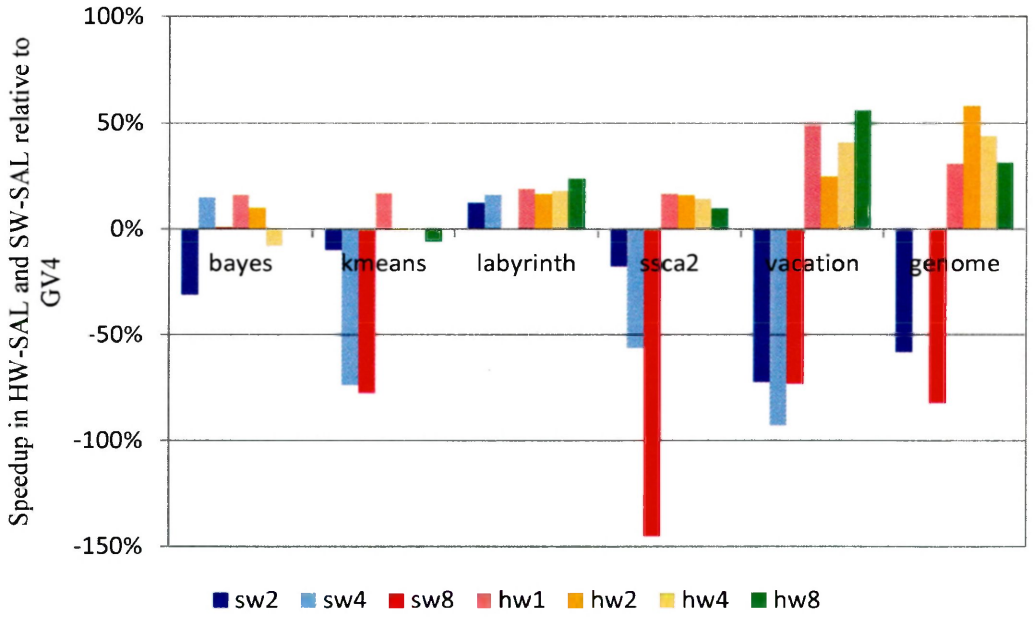
(a) 2-thread



(b) 4-thread



(c) 8-thread



(d) 16-thread

Figure 4.8 SW-SAL and HW-SAL speedup. (a) 2-thread. (b) 4-thread. (c) 8-thread. (d) 16-thread.

In most of the benchmarks, HW-SAL results in significant speedup over SW-SAL and GV4. SW-SAL works well with labyrinth, but does not perform well in Bayes and Kmeans. As the

associativity of the system grows the performance of SW-SAL reduces due to the timing overhead of the lock table. Labyrinth responds well to both SW-SAL and HW-SAL. According to Figure 4.2, the false conflict rate of Labyrinth is very high; therefore both hardware and software methods are able to provide considerable performance improvements by reducing false conflicts. For Ssca2, HW-SAL improves the performance because it does not have the overhead of software. SW-SAL lags behind, since the overhead of execution with high associativity is more than the performance gained by reducing the false conflicts. The same could be said for vacation. Genome however, shows a sporadic behaviour. For 2 and 8 threads, SW-SAL manages to provide performance; however, for 4 and 16 threads, there is a drastic degradation in performance. On the other hand, HW-SAL provides a steady performance across different number of threads. For Bayes and Kmeans, the overhead of SW-SAL is too high; hence, SW-SAL causes slow down across these benchmarks. HW-SAL however, shows great promise and provides moderate speedup.

On average, HW-SAL improves performance by 65% and 20% over SW_SAL and GV4, respectively.

4.6 Summary

In this chapter, we presented hardware support for software transactional memory. We provided a hardware mechanism in order to increase the performance of the state of the art STM by moving the lock table to the hardware. An associative lock table is crucial to reduce the false conflicts of STMs. However, an associative lock table in software may degrade performance especially when degree of associativity increases. On the other side, implementation of the lock table in hardware eliminates the overhead of the software table and is able to harness the true power of an associative lock table.

Chapter 5 Conclusions

Single core processors have been the focus of computer industry for years; however, due to the design complexity and power limitations, the industry cannot keep up the improvement pace as before. Therefore, all the major manufacturers have shifted their focus to chip multiprocessors (CMP). The advent of CMPs introduces new challenges in both architecture and software. New programming models have to be developed in order to harness the full potential of CMPs and new hardware also needs to be designed to further accelerate the speedup. Transactional memory (TM) has been a promising programming paradigm to simplify parallel programming and delivers performance comparable to the state of the art conventional lock based techniques.

Software transactional memory (STM) makes transactional memory feasible without the need of designing new hardware. One of the state of the art STM implementations is TL2, which uses a global clock as a timestamp in order to provide coherency for shared memory locations. Although an effective method, the contention over the shared global clock causes an extra overhead for the system and floods the interconnection network. Also, TL2 uses a direct mapped lock table which in turn may lead to several memory locations being mapped to the same lock entry. This causes false abort and impedes the performance of the Transactional Memory system.

In this thesis, we presented Transactional Read Tracking (TRT) which is a distributed method and does not have the deficiencies of TL2. TRT tracks transactional read and write operations and aborts transactions if they conflict over shared memory locations. Therefore, TRT manages to remove the burden of global clock and decreases the rate of unnecessary aborts compared to TL2. We improved performance of TRT by introducing two dynamic techniques which alternates between TRT and GV4. The first dynamic technique relies on number of read-write conflicts and the second one exploits perceptron predictors. We showed that the PGVT is superior to the RGVT.

Furthermore, we introduced the HW-SAL. In particular, we proposed a method to reduce the false conflicts that impede the performance of TL2. We proposed moving the lock table into the hardware and increase the associativity of the table, reducing the likelihood of false aborts. The HW-SAL was successful and improved performance of SW-SAL significantly.

5.1 Future Work

This thesis has shown that providing hardware support for Software Transactional Memory systems can provide performance improvements. In particular, moving the lock table to hardware allows the system to exploit higher associativity; this leads to reduction of false conflicts and performance improvement.

The same technique could be applied to other parts that are bottlenecks in STMs. In particular, the global clock is one of the main bottlenecks in TL2. The number of cores in CMPs will increase in future. If the global clock is implemented in hardware, the traffic generated in memory hierarchy will be reduced. Also, the operations to read and modify the global clock will be performed much faster, since it resides in hardware.

The other extension to hardware support for STM would be implementation of thread local clock (TLC) [13] in hardware. The problem with software implementation of TLC is that different threads are not updated with the latest value of the clock when they start their execution. This results in false aborts and degrades performance. In hardware implementation, a thread can broadcast a message to other processors indicating its current local clock value. As such, other threads update their local copies of remote clocks. This prevents false aborts in TLC and improves performance.

Appendix A

TRT Implementation Detail

This appendix has source code for RGVT and PGVT.

A.1 Adaptive TRT

```
void
TxStart (Thread* Self, sigjmp_buf* envPtr, int* ROFlag, uint32_t pc_addr)
{
  PROF_STM_START_BEGIN();

  if(Self->startTimeSaved == 0)
  {
    gettimeofday(&(Self->startTime), NULL);
    Self->startTimeSaved = 1;
  }

  if ( Self->UniqID == 0 )
   {
    Self->sc_pc_addr = pc_addr;
    sc_tmp_lookup = lookup_sat_count(Self, pc_addr);

    if(sc_tmp_lookup == 1)
     {
      Self->sc_pred_conflict = 1;
      Self->sc_num_pred_abort++;
     }
   }

  ASSERT(Self->Mode == TIDLE || Self->Mode == TABORTED);
  txReset(Self);

  if (Adaptive)
  {

   if ( Self->UniqID == 0 )
    {
     if(rwConflict > GV4_THRESHOLD)
      {
       if(tMode != GV4)//should change validation technique
          {
           pthread_rwlock_wrlock( &(rw_sync_lock));
           tMode = GV4;
           pthread_rwlock_unlock( &(rw_sync_lock) );
```

```
                Self->GV4_switches++;
              }
          }
        else//pass, TRC
          {
           if(tMode != TRC)//should change validation technique
              {
               pthread_rwlock_wrlock( &(rw_sync_lock));
               tMode = TRC;
               pthread_rwlock_unlock( &(rw_sync_lock) );
               Self->TRC_switches++;
              }
          }
      }
    }


    if ( Self->UniqID != 0 )
     pthread_rwlock_rdlock( &(rw_sync_lock) );

    if(tMode == GV4)
     Self->GV4_mode++;

    if(tMode == TRC)
     Self->TRC_mode++;

    if (tMode == GV4)
       Self->rv = GVRead(Self);
    ASSERT((Self->rv & LOCKBIT) == 0);
    MEMBARLDLD();

    Self->Mode = TTXN;
    Self->ROFlag = ROFlag;
    Self->IsRO = ROFlag ? *ROFlag : 0;
    Self->envPtr= envPtr;

    ASSERT(Self->LocalUndo.put == Self->LocalUndo.List);
    ASSERT(Self->wrSet.put == Self->wrSet.List);

    Self->Starts++;

   PROF_STM_START_END();
}


int TxCommit (Thread* Self)
```

```
  {
    PROF_STM_COMMIT_BEGIN();

    ASSERT(Self->Mode == TTXN);

    /* Fast-path: Optional optimization for pure-readers */
  # ifdef TL2_OPTIM_HASHLOG
    if (Self->wrSet.numEntry == 0)
  # else /* !TL2_OPTIM_HASHLOG*/
    if (Self->wrSet.put == Self->wrSet.List)
  # endif /* !TL2_OPTIM_HASHLOG*/
    {
      if (tMode == TRC)
        RevertRdSetChanges(Self);


      /* Given TL2 the read-set is already known to be coherent. */
      txCommitReset(Self);
      tmalloc_clear(Self->allocPtr);
      tmalloc_releaseAllForward(Self->freePtr, &txSterilize);
  # if defined(TL2_OPTIM_HASHLOG) && defined(TL2_RESIZE_HASHLOG)
      if (Self->wrSet.numLog > HASHLOG_INIT_NUM_LOG) {
        /*
         * If we are read-only, reduce the number of logs so less time
         * iterating over logs only to find that they are empty.
         */
        Self->wrSet.numLog--;
      }
  # endif
      PROF_STM_COMMIT_END();
      PROF_STM_SUCCESS();
      gettimeofday(&(Self->stopTime), NULL);
      Self->startTimeSaved = 0;
      Self->localTxTime    +=    (EHSAN_TIMER_DIFF_SECONDS(Self->startTime,    Self-
>stopTime));
      if ( Self->UniqID != 0 )
       pthread_rwlock_unlock( &(rw_sync_lock) );
       return 1;
    }

    if (TryFastUpdate(Self)) {
      txCommitReset(Self);
      tmalloc_clear(Self->allocPtr);
      tmalloc_releaseAllForward(Self->freePtr, &txSterilize);
  #if defined(TL2_OPTIM_HASHLOG) && defined(TL2_RESIZE_HASHLOG)
      if (Self->wrSet.numLog > HASHLOG_INIT_NUM_LOG &&
```

```
        Self->wrSet.numEntry < (HASHLOG_INIT_NUM_LOG *
HASHLOG_RESIZE_RATIO))
        {
          /*
           * Current hash log is too large. Reduce the number of logs so less
           * time is spent iterating over logs only to find that they are empty.
           */
          Self->wrSet.numLog--;
        }
  #endif
        PROF_STM_COMMIT_END();
        PROF_STM_SUCCESS();
        gettimeofday(&(Self->stopTime), NULL);
        Self->startTimeSaved = 0;
        Self->localTxTime += (EHSAN_TIMER_DIFF_SECONDS(Self->startTime, Self-
>stopTime));
        if ( Self->UniqID != 0 )
          pthread_rwlock_unlock( &(rw_sync_lock) );
        return 1;
      }

      PROF_STM_COMMIT_END();
      TxAbort(Self);
      ASSERT(0);
      return 0;
  }

  void
  TxAbort (Thread* Self)
  {

      PROF_STM_ABORT_BEGIN();
      Self->Mode = TABORTED;

  #ifdef TL2_EAGER
      WriteBackReverse(&Self->wrSet);
      RestoreLocks(Self);
  #else /* !TL2_EAGER */
      if (Self->HoldsLocks) {
        RestoreLocks(Self);
      }
  #endif /* !TL2_EAGER */

      /* Clean up after an abort. Restore any modified locals */
      if (Self->LocalUndo.put != Self->LocalUndo.List) {
        WriteBackReverse(&Self->LocalUndo);
```

```
    }

    Self->Retries++;
    Self->Aborts++;

    /*
     * Beware: back-off is useful for highly contended environments
     * where N threads shows negative scalability over 1 thread.
     * Extreme back-off restricts parallelism and, in the extreme,
     * is tantamount to allowing the N parallel threads to run serially
     * 1 at-a-time in succession.
     *
     * Consider: make the back-off duration a function of:
     * - a random #
     * - the # of previous retries
     * - the size of the previous read-set
     * - the size of the previous write-set
     *
     * Consider using true CSMA-CD MAC style random exponential backoff
     */

  #ifndef TL2_NOCM
    if (Self->Retries > 3) { /* TUNABLE */
      backoff(Self, Self->Retries);
    }
  #endif

    if ( Self->UniqID != 0 )
     pthread_rwlock_unlock( &(rw_sync_lock) );

    tmalloc_releaseAllReverse(Self->allocPtr, NULL);
    tmalloc_clear(Self->freePtr);

    gettimeofday(&(Self->stopTime), NULL);
    Self->startTimeSaved = 0;
    Self->localTxTime    +=    (EHSAN_TIMER_DIFF_SECONDS(Self->startTime,    Self-
>stopTime));

    PROF_STM_ABORT_END();
    SIGLONGJMP(*Self->envPtr, 1);
    ASSERT(0);
  }

  __INLINE__ long
  TryFastUpdate (Thread* Self)
  {
```

```
# ifdef TL2_OPTIM_HASHLOG
  long numLog = Self->wrSet.numLog;
  Log* logs = Self->wrSet.logs;
  Log* end = logs + numLog;
  Log* wr;
# endif /* !TL2_OPTIM_HASHLOG */

# ifndef TL2_EAGER
#  ifdef TL2_OPTIM_HASHLOG
  Log* wr;
#  else /* !TL2_OPTIM_HASHLOG */
  Log* const wr = &Self->wrSet;
#  endif /* !TL2_OPTIM_HASHLOG */
  Log* const rd = &Self->rdSet;
  long ctr;
# endif /* !TL2_EAGER */

  ASSERT(Self->Mode == TTXN);

  /*
   * Optional optimization -- pre-validate the read-set.
   *
   * Consider: Call ReadSetCoherent() before grabbing write-locks.
   * Validate that the set of values we've fetched from pure READ objects
   * remain coherent.  This avoids the situation where a doomed transaction
   * grabs write locks and impedes or causes other potentially successful
   * transactions to spin or abort.
   *
   * A smarter tactic might be to only call ReadSetCoherent() when
   * Self->Retries > NN
   */

# if 0
  if (!ReadSetCoherent(Self)) {
    return 0;
  }
# endif

# ifndef TL2_EAGER

  /*
   * Consider: if the write-set is long or Self->Retries is high we
   * could run a pre-pass and sort the write-locks by LockFor address.
   * We could either use a separate LockRecord list (sorted) or
   * link the write-set entries via SortedNext
   */
```

```
  /*
   * Lock-acquisition phase ...
   *
   * CONSIDER: While iterating over the locks that cover the write-set
   * track the maximum observed version# in maxv.
   * In GV4:   wv = GVComputeWV(); ASSERT wv > maxv
   * In GV5|6: wv = GVComputeWV(); if (maxv >= wv) wv = maxv + 2
   * This is strictly an optimization.
   * maxv isn't required for algorithmic correctness
   */
  Self->HoldsLocks = 1;
  ctr = 1000; /* Spin budget - TUNABLE */
  vwLock maxv = 0;
  AVPair* p;
#   ifdef TL2_OPTIM_HASHLOG
  for (wr = logs; wr != end; wr++)
#   endif /* TL2_OPTIM_HASHLOG*/
  {
    AVPair* const End = wr->put;
    for (p = wr->List; p != End; p = p->Next) {
      volatile vwLock* const LockFor = p->LockFor;
      vwLock cv;
      ASSERT(p->Addr != NULL);
      ASSERT(p->LockFor != NULL);
      ASSERT(p->Held == 0);
      ASSERT(p->Owner == Self);
      /* Consider prefetching only when Self->Retries == 0 */
      prefetchw(LockFor);
      cv = LDLOCK(LockFor);
      if ((cv & LOCKBIT) && ((AVPair*)(cv ^ LOCKBIT))->Owner == Self) {
        /* CCM: revalidate read because could be a hash collision */
        if (FindFirst(rd, LockFor) != NULL) {
          if (((AVPair*)(cv ^ LOCKBIT))->rdv > Self->rv) {
            Self->abv = cv;
            return 0;
          }
        }
        /* Already locked by an earlier iteration. */
        continue;
      }

      /* SIGTM does not maintain a read set */
      if (FindFirst(rd, LockFor) != NULL) {
        /*
         * READ-WRITE stripe
```

```
         */
        if ((cv & LOCKBIT) == 0 &&
           cv <= Self->rv &&
           UNS(CAS(LockFor, cv, (UNS(p)|UNS(LOCKBIT)))) == UNS(cv))
        {
           if (cv > maxv) {
              maxv = cv;
           }
           p->rdv  = cv;
           p->Held = 1;
           continue;
        }
        /*
         * The stripe is either locked or the previously observed read-
         * version changed.  We must abort. Spinning makes little sense.
         * In theory we could spin if the read-version is the same but
         * the lock is held in the faint hope that the owner might
         * abort and revert the lock
         */
        Self->abv = cv;
        return 0;
      } else
      {
        /*
         * WRITE-ONLY stripe
         * Note that we already have a fresh copy of *LockFor in cv.
         * If we find a write-set element locked then we can either
         * spin or try to find something useful to do, such as :
         * A. Validate the read-set by calling ReadSetCoherent()
         *    We can abort earlier if the transaction is doomed.
         * B. optimistically proceed to the next element in the write-set.
         *    Skip the current locked element and advance to the
         *    next write-set element, later retrying the skipped elements
         */
#     ifdef TL2_NOCM
        /* wkbaek: no spinning in NOCM mode */
        long c = 0;
#     else /* !TL2_NOCM */
        long c = ctr;
#     endif /* !TL2_NOCM */
        for (;;) {
          cv = LDLOCK(LockFor);
          /* CCM: for SIGTM, this IF and its true path need to be "atomic" */
          if ((cv & LOCKBIT) == 0 &&
             UNS(CAS(LockFor, cv, (UNS(p)|UNS(LOCKBIT)))) == UNS(cv))
          {
```

```
                if (cv > maxv) {
                   maxv = cv;
                }
                p->rdv  = cv; /* save so we can restore or increment */
                p->Held = 1;
                break;
             }
             if (--c < 0) {
                /* Will fall through to TxAbort */
                return 0;
             }
             /*
              * Consider: while spinning we might validate
              * the read-set by calling ReadSetCoherent()
              */
             PAUSE();
          }
       } /* write-only stripe */
    } /* foreach (entry in write-set) */
  }

# endif /* !TL2_EAGER */

  if ( tMode == GV4 && !ReadSetCoherent(Self)) {
     return 0;
  }

  /*
   * CCM: for SIGTM, the read filter would have triggered an abort already
   * if the read-set was not consistent.
   */

  /*
   * We are now committed - this txn is successful.
   */

# ifndef TL2_EAGER
#   ifdef TL2_OPTIM_HASHLOG
  for (wr = logs; wr != end; wr++)
#   endif /* TL2_OPTIM_HASHLOG*/
  {
     WriteBackForward(wr); /* write-back the deferred stores */
  }
# endif /* !TL2_EAGER */
  MEMBARSTST(); /* Ensure the above stores are visible  */
  DropLocks(Self); /* Release locks and increment the version */
```

```
    /*
     * Ensure that all the prior STs have drained before starting the next
     * txn.  We want to avoid the scenario where STs from "this" txn
     * languish in the write-buffer and inadvertently satisfy LDs in
     * a subsequent txn via look-aside into the write-buffer
     */
    MEMBARSTLD();

    return 1; /* success */
}

intptr_t
TxLoad (Thread* Self, volatile intptr_t* Addr)
{
    PROF_STM_READ_BEGIN();

    intptr_t Valu = 0;

# ifdef TL2_STATS
    Self->TxLD++;
# endif

    ASSERT(Self->Mode == TTXN);

    byte GV4_Accepted = 1;
    volatile vwLock* LockFor = PSLOCK(Addr);
    vwLock cv = LDLOCK(LockFor);
    vwLock rdv = cv & ~LOCKBIT;

    if (tMode == GV4 && (cv & LOCKBIT) == 0 && Self->rv < rdv)
        GV4_Accepted = 0;

    if (tMode == GV4) {
        MEMBARLDLD();
        Valu = LDNF(Addr);
        MEMBARLDLD();
    }

    if ((GV4_Accepted && LDLOCK(LockFor) == rdv) ||
        ((cv & LOCKBIT) && (((AVPair*)rdv)->Owner == Self)))
    {
        if (!Self->IsRO) {
            if (!TrackLoad(Self, LockFor, Addr, cv)) {
                PROF_STM_READ_END();
                TxAbort(Self);
```

```
        }
      }
      if (tMode == TRC) {
        MEMBARLDLD();
        Valu = LDNF(Addr);
        MEMBARLDLD();
      }
      PROF_STM_READ_END();
      return Valu;
    }

    /*
     * The location is either currently locked or has been updated since this
     * txn started.  In the latter case if the read-set is otherwise empty we
     * could simply re-load Self->rv = _GCLOCK and try again.  If the location
     * is locked it's fairly likely that the owner will release the lock by
     * writing a versioned write-lock value that is > Self->rv, so spinning
     * provides little profit.
     */

    Self->abv = rdv;
    PROF_STM_READ_END();
    TxAbort(Self);
    ASSERT(0);

    return 0;
}

void
TxStore (Thread* Self, volatile intptr_t* addr, intptr_t valu)
{
    PROF_STM_WRITE_BEGIN();

    ASSERT(Self->Mode == TTXN);
    if (Self->IsRO) {
      *(Self->ROFlag) = 0;
      PROF_STM_WRITE_END();
      TxAbort(Self);
      ASSERT(0);
    }

# ifdef TL2_STATS
    Self->TxST++;
# endif

    /*
```

```
 * Try to acquire the lock. If we are not the owner, we spin a
 * bit before deciding to abort. If we acquire the lock, we
 * set the lockbit in the lockword and use a temporary AVPair* value.
 * Later we update it with a the actual AVPair*.
 */

volatile vwLock* LockFor = PSLOCK(addr);
vwLock cv = LDLOCK(LockFor);

if ((cv & LOCKBIT) && (((AVPair*)(cv ^ LOCKBIT))->Owner == Self)) {
   /*
    * We own this lock; update cv with the correct value for RecordStore.
    */
   if (tMode == TRC)
      ((AVPair*)(cv ^ LOCKBIT))->duplicateEntry = 1;
   else
      cv = ((AVPair*)(cv ^ LOCKBIT))->rdv;
} else if (tMode == GV4) {
   long c = 100; /* TUNABLE */
   AVPair* p = &(Self->tmpLockEntry);
   for (;;) {
      if ((cv & LOCKBIT) == 0)
      {
         if (UNS(CAS(LockFor, cv, (UNS(p)|UNS(LOCKBIT)))) == UNS(cv))
         {
            break;
         }
      }
      cv = LDLOCK(LockFor);
      if (--c < 0) {
         PROF_STM_WRITE_END();
         TxAbort(Self);
         ASSERT(0);
      }
   }
} else if (tMode == TRC) {
   long c = 20; /* TUNABLE */
   AVPair* p = &(Self->tmpLockEntry);
   for (;;) {
      if ((cv & LOCKBIT) == 0)
      {
         //If the previous mode was GV4, the lock must be set to 0 first.
         if ((cv & GVMASK) && UNS(CAS(LockFor, cv, 0)) != UNS(cv)) {
            cv = LDLOCK(LockFor);
            if (--c < 0) {
               PROF_STM_WRITE_END();
```

```
                TxAbort(Self);
                ASSERT(0);
              }
              continue;
            }
            if ((cv & ~Self->rdMask) == 0
              && UNS(CAS(LockFor, cv, (UNS(p)|UNS(LOCKBIT)))) == UNS(cv))
            {
              break;
            }
            else if ((cv & ~Self->rdMask) && c == 0)
            {
             if(Self->UniqID == 0)
                  rwConflict++;
            }
          }
          cv = LDLOCK(LockFor);
          if (--c < 0) {
            PROF_STM_WRITE_END();
            TxAbort(Self);
            ASSERT(0);
          }
        }
      }

      Log* wr = &Self->wrSet;
      if(tMode == GV4 && cv == 0)
      {
        cv = GVRead(Self);
        if (cv > Self->maxv) {
          Self->maxv = cv;
        }
      }

      AVPair* e = RecordStore(wr, addr, *addr, LockFor, cv);

      *LockFor = UNS(e) | UNS(LOCKBIT);

      *addr = valu;

      PROF_STM_WRITE_END();
    }
```

A.2 PGVT

```
__INLINE__ int
perceptron_dir_lookup (Thread* Self, unsigned int address) {
```

```
  int i, output, *w;
  unsigned long long int mask;
  struct perceptron_node *p;
  struct perceptron_state *u;

  u = &(Self->perceptron_state_buf);//[perceptron_state_buf_ctr++]);
  p = percept_link_list_find(Self, address);
  if(p == NULL)
    p = percept_link_list_addNode(Self, address);

  if(p->pc == 0)
    p->pc = address;

  w = &p->weights[0];

  output = *w++;

  for (mask=1,i=0; i<PERCEPTRON_HISTORY; i++,mask<<=1,w++) {
    if (Self->spec_global_history & mask)
      output += *w;
    else
      output += -*w;
  }

  u->output = output;
  u->perc = p;
  u->history = Self->spec_global_history;
  u->prediction = output > 0;//output >= 0;
  u->dummy_counter = u->prediction ? 3 : 0;

  Self->spec_global_history <<= 1;
  Self->spec_global_history |= u->prediction;
  return u->prediction;//u;
}


__INLINE__ void
perceptron_update (Thread* Self, int taken) {
  int i, y,*w;
  unsigned long long int mask, history;
  struct perceptron_state *u;

  u = &(Self->perceptron_state_buf);

  Self->global_history <<= 1;
  Self->global_history |= taken;
```

```
  if (u->prediction != taken) Self->spec_global_history = Self->global_history;

  if (u->output > THETA)
    y = 1;
  else if (u->output < -THETA)
    y = 0;
  else
    y = 2;

  if (y == 1 && taken) return;
  if (y == 0 && !taken) return;

  w = &u->perc->weights[0];

  if (taken)
    (*w)++;
  else
    (*w)--;
  if (*w > MAX_WEIGHT) *w = MAX_WEIGHT;
  if (*w < MIN_WEIGHT) *w = MIN_WEIGHT;

  w++;
  history = u->history;

  for (mask=1,i=0; i<PERCEPTRON_HISTORY; i++,mask<<=1,w++) {
    if (!!(history & mask) == taken)
      {
        (*w)++;
        if (*w > MAX_WEIGHT)
          *w = MAX_WEIGHT;
      } else
      {
        (*w)--;
        if (*w < MIN_WEIGHT)
          *w = MIN_WEIGHT;
      }
    }
}
```

Appendix B

SAL Implementation Details

In this appendix, different implementation aspects of SAL are presented. The source code for simulation in Gem5 is provided in this appendix.

B.1 HW-SAL

```
typedef struct _lockNode{
   uint64_t lock;
   uint64_t addr;
   Tick time;
   int idx;
} lockNode;
 int SimpleThread::SALReadLockEntry(uint64_t addr);
 uint64_t SimpleThread::SALIdxtoContent(int idx);
 void SimpleThread::SALSetLockEntry(uint64_t value);
 uint64_t SimpleThread::SALCAS(uint64_t value);
```

B.2 HW-SAL

In this section, we present the implementation detail of HW-SAL and the source code of HW-SAL in Gem5.

B.2.1 New Instructions defined in HW-SAL.

```
#define XACTION_READLOCKENTRY (save, save2, idx, value) \
  {\
  asm volatile ("mov $1, %0" : "=r"(save)::"memory");\
  asm volatile ("mov $2, %0" : "=r"(save2)::"memory");\
  asm volatile ("mov %0, $2" :: "r"(idx) : "memory");\
  asm volatile (".long (((0x02) << 26) | ((2) << 21) | ((0) << 16) | ((0x4) << 12) | ((14) << 5) |
(1))":::"$1");\
  asm volatile ("mov $1, %0" : "=r"(value)::"memory");\
  asm volatile ("mov %0, $1" :: "r"(save):"memory");\
  asm volatile ("mov %0, $2" :: "r"(save2):"memory");\
  }

#define XACTION_ACQUIRELOCK (save, save2, idx, addr) \
  {\
  asm volatile ("mov $1, %0" : "=r"(save)::"memory");\
  asm volatile ("mov $2, %0" : "=r"(save2)::"memory");\
  asm volatile ("mov %0, $2" :: "r"(addr) : "memory");\
  asm volatile (".long (((0x02) << 26) | ((2) << 21) | ((0) << 16) | ((0x4) << 12) | ((13) << 5) |
(1))":::"$1");\
  asm volatile ("mov $1, %0" : "=r"(idx)::"memory");\
  asm volatile ("mov %0, $1" :: "r"(save):"memory");\
  asm volatile ("mov %0, $2" :: "r"(save2):"memory");\
```

```
}

#define XACTION_ WRITETOLOCK (save, save2, value) \
{\
asm volatile ("mov $1, %0" : "=r"(save)::"memory");\
asm volatile ("mov %0, $1" :: "r"(value) : "memory");\
asm volatile (".long (((0x02) << 26) | ((1) << 21) | ((0) << 16) | ((0x3) << 12) | ((14) << 5))");\
asm volatile ("mov %0, $1" :: "r"(save):"memory");\
}

#define XACTION_SALCAS (save, save2, old_value, value_ret)
{\
asm volatile ("mov $1, %0" : "=r"(save)::"memory");\
asm volatile ("mov $2, %0" : "=r"(save2)::"memory");\
asm volatile ("mov %0, $2" :: "r"(old_value) : "memory");\
asm volatile (".long (((0x02) << 26) | ((2) << 21) | ((0) << 16) | ((0x4) << 12) | ((15) << 5) |
(1))":::"$1");\
asm volatile ("mov $1, %0" : "=r"(value_ret)::"memory");\
asm volatile ("mov %0, $1" :: "r"(save):"memory");\
asm volatile ("mov %0, $2" :: "r"(save2):"memory");\
}
```

B.2.2 HW-SAL in Gem5

```
void XactionStart()
{
 xaction_num_start++;
 thread->XactionStart();

 tick_xaction_begin = curTick();
}

void XactionCommit()
{
 xaction_num_commit++;
 thread->XactionCommit();
 xaction_cycles += tickToCycles(curTick() - tick_xaction_begin);
}

void XactionAbort()
{
 xaction_num_abort++;

 thread->XactionAbort();
 xaction_cycles += tickToCycles(curTick() - tick_xaction_begin);
}

void setXactionActiveFlag(short mode)
{
```

```
  thread->setXactionActiveFlag(mode);
}

int SALAddrtoIdx(uint64_t addr)
{
  int id;

  id = thread->SALAddrtoIdx(addr);

  if(thread->readFalseConf())
    xaction_num_false_conflict++;

  return id;
}

uint64_t SALIdxtoContent(int idx)
{
  return thread->SALIdxtoContent(idx);
}

void SALSetId(int idx)
{
  return thread->SALSetId(idx);
}
void SALSetLockEntry(uint64_t value)
{
  return thread->SALSetLockEntry(value);
}

void SALSetCASNewVal(uint64_t value)
{
  return thread->SALSetCASNewVal(value);
}

uint64_t SALCAS(uint64_t value)
{
  return thread->SALCAS(value);
}

void initialize_lock()
{
int i, j;

for(i=0; i<MAX_TABSZ; i++)
  for(j=0; j<MAX_ASSOCIATIVITY; j++)
    {
```

```
        lockArray[i][j].lock = 0;
        lockArray[i][j].addr = 0;
        lockArray[i][j].time = 0;
      }

  for(i=0; i<_TABSZ; i++)
    for(j=0; j<ASSOCIATIVITY; j++)
      {
        lockArray[i][j].idx = i*ASSOCIATIVITY + j;
      }

 SALIdx = 0;
 CASNewVal = 0;
  return;
}

int SALAddrtoIdx(uint64_t addr)
{
 unsigned int row;
 unsigned int j;

 lockNode *unaccessed_node = NULL;
 lockNode *unlocked_node = NULL;
 lockNode *lru_node = NULL;

 false_conflict = 0;

 for (j=0; j<ASSOCIATIVITY; j++)
   {
    if(lockArray[row][j].addr == addr)
      {
        lockArray[row][j].addr = addr;
        lockArray[row][j].time = curTick();
        return lockArray[row][j].idx;
      }
  if(lru_node==NULL)
      lru_node = &(lockArray[row][j]);
  else
      if(lru_node->time > lockArray[row][j].time)
        lru_node = &(lockArray[row][j]);

   if( (lockArray[row][j].lock & ~LOCKBIT)==0 )
      unaccessed_node = &(lockArray[row][j]);

   if( (lockArray[row][j].lock & LOCKBIT)==0 )
      unlocked_node = &(lockArray[row][j]);
```

```
    }

   if(unaccessed_node != NULL)
    {
     unaccessed_node->addr = addr;
     unaccessed_node->time = curTick();
     return  unaccessed_node->idx;//unaccessed_node;
    }

  if(unlocked_node != NULL)
    {
     unlocked_node->addr = addr;
     unlocked_node->time = curTick();
     return unlocked_node->idx;//unlocked_node;
    }

  assert(lru_node != NULL);

  lru_node->addr = addr;
  lru_node->time = curTick();//_GCLOCK;
  return lru_node->idx;//lru_node;

}

uint64_t SALIdxtoContent(int idx)
{
  int row, col;

  row = (idx / ASSOCIATIVITY);
  col = (idx % ASSOCIATIVITY);

  return lockArray[row][col].lock;
}
```

Chapter 6 Bibliography

[1] J. L. Hennessy and D. Patterson, Computer Organization and Design: A Quantitative Approach, Morgan Kauffman Publications, 2007.

[2] D. Patterson and J. L. Hennessy, Computer Organization and Design: The Hardware/Software Interface, morgan kaufmann Publishers.

[3] Y. Tsividis, Operation and modeling of the MOS transistor, McGraw-Hill, 1999.

[4] N. S. Kim, T. M. Austin, D. Baauw, T. N. Mudge, K. Flautner, J. S. Hu, M. J. Irwin, M. T. Kandemir and V. Narayanan, "Leakage current: Moore's law meets static power," in *IEEE Computer, Vol. 36, Issue 12, pp. 68-75*, December 2003.

[5] S. Behling, The POWER4 Processor, Introduction and Tuning Guide, 2001.

[6] "Intel Has Double Vision: First Multi-Core Silicon Production Begins," Intel press room, 2005.

[7] "AMD Opteron™ Processor Power and Thermal Data Sheet".

[8] M. Herlihy and N. Shavit, The Art of Multiprocessor Programming, Morgan Kaufmann Publications, 2008.

[9] M. Herlihy and J. E. B. Moss, "Transactional Memory: Architectural support for lock-free data structures.," in *Twentieth Annual International Symposium on Computer Architecture*, 1993.

[10] M. Lupon, G. Magklis and A. Gonzalez, "FASTM: A log-based hardware transactional memory with fast abort recovery," in *Proceedings of the 18th International Conference on Parallel Architectures and Compilation Techniques*, September 2009.

[11] M. Lupon, G. Magklis and A. Gonzalez, "A Dynamically Adaptable Hardware Transactional Memory," in *43rd Annual IEEE/ACM International Symposium on Microarchitecture*, Atlanta, Georgia, USA, December 2010.

[12] D. Dice, O. Shalev and N. Shavit, "Transactional locking ii," *DISC,* pp. LNCS, vol. 4167, pp. 194–208. Springer, Heidelberg (2006), 2006.

[13] H. Avni and N. Shavit, "Maintaining Consistent Transactional States Without a Global Clock," in *Proceedings of 15th International Colloquium on Structural Information and Communication Complexity. pp. 131-140. Springer-Verlag Lecture Notes in Computer Science volume 5058*, 2008.

[14] P. Damron, A. Fedorova, Y. Lev, V. Luchangco, M. Moir and D. Nussbaum, "Hybrid transactional memory," in *Proceedings of the 12th International Conference on Architectural Support for Programming Languages and Operating Systems, pp. 336-346*, 2006.

[15] M. Herlihy, "A Methodology for Implementing Highly Concurrent Data Objects," *ACM Transactions on Programming Languages and Systems (TOPLAS),* vol. 15, no. 5, pp. 745 - 770 , 1993 .

[16] H. Massalin and C. Pu, "A lock-free multiprocessor OS kernel," Technical report CUCS-005--91, Computer Science Department, Columbia University,, 1991.

[17] M. Herlihy, V. Luchangco and M. Moir, "Obstruction-Free Synchronization: Double-Ended Queues as an Example," in *Proceedings of the 23rd International Conference on Distributed Computing Systems, Page 522* , 2003.

[18] D. E. Culler, J. P. Singh and A. Gupta, Parallel Computer Architecture: a Hardware/Software Approach, Morgan Kaufmann Publishers, 1998.

[19] I. Corporation, Intel® 64 and IA-32 Architectures Software Developer's Manual, Intel Corporation, March 2013.

[20] S. I. Inc., The SPARC Architecture Manual, SPARC International Inc..

[21] C. C. Corporation, Alpha Architecture Handbook, Compaq Computer Corporation.

[22] V. Haug and J. Indest, "RS/6000 7044 Model 270, Technical Overview and Introduction," IBM corporation, February, 2000.

[23] S. Tomic, C. Perfumo, C. Kulkarni, A. Armejach, A. Cristal, O. Unsal, T. Harris and M. Valero, "EazyHTM: Eager-lazy hardware transactional memory," in *MICRO '09:Proceedings of the 2009 42nd IEEE/ACM International Symposium on Microarchitecture*, 2009.

[24] K. E. Moore, J. Bobba, M. J. Moravan, M. D. Hill and D. A. Wood, "LogTM: Log-based Transactional," in *Proceedings of the 12th International Symposium on HighPerformance Computer Architecture*, February 2006.

[25] L. Yen, J. Bobba, M. M. Marty, K. E. Moore, H. Volos, M. D. Hill, M. M. Swift and D. A. Wood, "LogTMSE: Decoupling Hardware Transactional Memory from Caches," in *Proceedings of the 13th International Symposium on High Performance Computer Architecture*, February 2007.

[26] N. Shavit and D. Touitou, "Software Transactional Memory," in *Proceedings of the 14th Annual ACM Symposium on Principles of Distributed Computing (PODC'95), pages 204-213*, August 1995.

[27] M. Moir, "Transparent Support for Wait-Free Transactions," in *Distributed Algorithms, 11th International Workshop, volume 1320 of Lecture Notes in Computer Science, pages 305?319. Springer-Verlag,*, September 1997.

[28] A. Israeli and L. Rappoport, "Disjoint-access-parallel Implementations of Strong Shared Memory Primitives," In Proceedings of the 13nd Annual ACM Symposium on Principles of Distributed Computing (PODC '94), 151-160., 1994.

[29] M. Herlihy, V. Luchangco, M. Moir and W. Scherer, "Software Transactional Memory for Dynamic-Sized Data Structures," in *Proceedings of the 22nd Annual ACM Symposium on Principles of Distributed Computing (PODC '03), pages 92-101*, 2003.

[30] M. F. Spear, V. J. Marathe, W. N. S. III and M. L. Scott, "Conflict Detection and Validation Strategies for Software Transactional Memory," in *Proceedings of the 20th International Symposium on Distributed Computing (DISC), pages 179–193*, September 2006.

[31] S. Mannarswamy and R. Govindarajan, "Making STMs Cache Friendly with Compiler Transformations," in *Proceedings of the 20th International Parallel Architectures and Compilation Techniques (PACT 2011)*, Galveston Island, TX, October 2011.

[32] T. Riegel, C. Fetzer and P. Felber, "Time-based transactional memory with scalable time bases," in *SPAA '07: Proceedings of the nineteenth annual ACM symposium on Parallel algorithms and architectures, pages 221–228*, New York, NY, USA, 2007.

[33] E. Atoofian, "Set Associative Lock in Software Transactional Memory," in *Proceedings of 2nd Workshop on Applications for Multi and Many Core Processors*, 2011.

[34] C. S. Ananian, K. Asanovic, B. C. Kuszmaul, C. E. Leiserson and S. Lie, "Unbounded Transactional Memory," in *Proceedings of the 11th Intl Symp on High-Performance Computer Architecture*, February 2005.

[35] L. Hammond, V. Wong, M. Chen, B. D. Carlstrom, J. D. Davis, B. Hertzberg, M. K. Prabhu, H. Wijaya, C. Kozyrakis and a. K. Olukotun, "Transactional Memory Coherence and Consistency," in *Proceedings of the 31st International Symposium on Computer Architecture*, June 2004.

[36] H. Chafi, J. Casper, B. D. Carlstrom, A. McDonald, C. C. Minh, W. Baek, C. Kozyrakis and K. Olukotun, "A Scalable, Non-blocking Approach to Transactional Memory," in *HPCA, pages 97-108,*, 2007.

[37] B. Saha, A.-R. Adl-Tabatabai and a. Q. Jacobson, "Architectural Support for Software Transactional Memory," in *Proceedings of the 39th Annual IEEE/ACM International Symposium on Microarchitecture*, December 2006.

[38] L. Ceze, J. Tuck, C. Cascaval and J. Torrellas, "Bulk Disambiguation of Speculative Threads in Multiprocessors," in *Proceedings of the 33th International Symposium on Computer Architecture*, June 2006.

[39] I. N. Room, IBM Unveils zEnterprise EC12, a Highly Secure System for Cloud Computing and Enterprise Data, 2012.

[40] I. Doboš, P. Hamid, G. Laumay, F. Nogal, V. R. Jr, A. Spahni, H. Wijngaard, W. Fries, O. Lascu, S. S. Ng, F. Packheiser, K. Singh, E. Ufacik and Z. Zhang, IBM zEnterprise EC12 Technical Introduction, IBM Redbooks, December 2012.

[41] I. N. Release, "IBM Announces Supercomputer to Propel Sciences Forward," IBM, ARMONK, New York, 2011.

[42] A. Wang, M. Gaudet, P. Wu, J. N. Amaral, M. Ohmacht, C. Barton, R. Silvera and M. Michael, "Evaluation of Blue Gene/Q Hardware Support for Transactional Memories," in *Proceedings of 21st International Conference on Parallel Architectures and Compiler Techniques*, Minneapolis, Minnesota USA, 2012.

[43] A. Dragojevic, R. Guerraoui and M. Kapalka, "Stretching Transactional Memory," in *Proceedings of the ACM SIGPLAN conference on Programming language design and implementation*, 2009.

[44] C. C. Minh, J. Chung, C. Kozyrakis and K. Olukotun, "STAMP: Stanford Transactional Applications for Multi-processing," in *IISWC '08: Proceedings of The IEEE International Symposium on Workload Characterization*, September 2008.

[45] Y. Lev, V. Luchangco, V. Marathe, M. Moir, D. Nussbaum and M. Olszewski, "Anatomy of a scalable software transactional memory," in *TRANSACT '09: 4th Workshop on Transactional Computing*, February 2009.

[46] F. Rosenblatt, Principles of Neurodynamics: Perceptrons and the Theory of Brain Mechanisms, Spartan, 1962.

[47] D. Jimenez and C. Lin, "Dynamic branch prediction with perceptrons," in *HPCA '01, pages 197–206*, 2001.

[48] N. Binkert, B. Beckmann, G. Black, S. K. Reinhardt, A. Saidi, A. Basu, J. Hestness, D. R. Hower, T. Krishna, S. Sardashti, R. Sen, K. Sewell, M. Shoaib, N. Vaish, M. D. Hill and D. A. Wood, "The gem5 simulator," in *ACM SIGARCH Computer Architecture News*, 2011 .

[49] W. Litwin, "Linear Hashing : A New Tool for File and Table Addressing," in *IEEE*, 1980.

[50] R. Pagh and F. F. Rodler, "Cuckoo Hashing," in *Algorithms — ESA 2001. Lecture Notes in Computer Science. 2161. pp. 121–133*, December 2003.

[51] N. L. Binkert, R. G. Dreslinski, L. R. Hsu, K. T. Lim, A. G. Saidi and S. K. Reinhardt, "The m5 simulator: Modeling networked systems," in *IEEE Micro, 26(4):52–60*, 2006.

[52] M. M. K. Martin, D. J. Sorin, B. M. Beckmann, M. R. Marty, M. Xu, A. R. Alameldeen, K. E. Moore, M. D. Hill and a. D. A. Wood, "Multifacet's General Execution-driven Multiprocessor Simulator (GEMS) Toolset," in *ACM SIGARCH Computer Architecture News, Vol. 33, No. 4*, September 2005.

[53] J. Bobba, K. E. Moore, L. Yen, H. Volos, M. D. Hill, M. M. Swift and D. A. Wood, "Performance pathologies in hardware transactional memory," in *Proceedings of the 34th Annual International Symposium on Computer Architecture*, 2007.

[54] C. Cascaval, C. Blundell, M. Michael, H. W. Cain, P. Wu, S. Chiras and S. Chatterjee, "Software transactional memory: Why is it only a research toy?," in *Queue, 6(5):46–58*, 2008.

[55] W. N. S. III and M. L. Scott, "Advanced contention management for dynamic software transactional memory," in *Proceedings of the 24th ACM Symp on Principles of Distributed Computing*, July 2005.

[56] S. Kumar, M. Chu, C. J. Hughes, P. Kundu and A. Nguyen, "Hybrid Transactional Memory," in *Proceedings of the ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, March 2006.

[57] A. Shriraman, S. Dwarkadas and M. L. Scott, "Flexible Decoupled Transactional Memory Support," in *Proceedings ofthe 35th International Symposium on Computer Architecture*, June 2008.

[58] D. Knuth, "An almost linear recurrence," in *Fib. Quart., 4:117–128*, 1966.

[59] E. W. Dijkstra, "Cooperating Sequential Processes," in *Technical Report EWD-123*, Technical University, Eindhoven, 1965.

[60] D. B. Lomet, "Process Structuring, Synchronization, and Recovery Using Atomic Actions," in *In Proceedings of an ACM conference on Language design for reliable software, pages 128–137*, New York, NY, USA, 1977.

[61] J. Bartlett, "Assembly language for Power Architecture," www.ibm.com, October 2006.

[62] J. M. Mellor-Crummey and M. L. Scott, Algorithms for Scalable Synchronization on SharedMemory Multiprocessors, ACM Transaction on Computer Systems, February 1991.