View-Oriented Parallel Programming and its Performance Evaluation on Multicore Architectures

Kai-Cheung Leung (梁啓昌)

a thesis submitted for the degree of

Doctor of Philosophy

at the University of Otago, Dunedin,

New Zealand.

17th August, 2013

Abstract

Shared-memory multicore architectures have become pervasive, and there is a pressing need for parallel programming models to facilitate both performance and convenience. However, most existing shared-memory programming models are tedious for programming and are prone to errors such as data race, which are difficult to debug.

To solve this problem, this thesis proposes a data race prevention scheme in the View-Oriented Parallel Programming (VOPP) paradigm. VOPP was proposed for distributed shared memory systems. It is adapted to shared-memory multicore architectures in this thesis. VOPP is a shared-memory data-centric parallel programming model, which uses views to bundle mutual exclusion with data access. In VOPP, programmers partition the shared memory into "views", which are non-overlapping sets of shared data objects. The data race prevention scheme proposed for VOPP can prevent data race through the memory protection mechanism while keeping the extra overhead low.

To improve the programmability of VOPP, this thesis proposes an automatic view access management scheme where a view is automatically acquired upon its first access, and automatically released when no longer needed, thus relieving programmers from arranging locks to protect critical sections.

To further improve performance and programmability, this thesis proposes the View-Oriented Transactional Memory (VOTM) system, which uses Restricted Admission Control (RAC) to manage the number of processes holding each view according to its contention. In VOTM, RAC can restrict the number of processes holding the view when its contention is high, and in

extreme cases, RAC can fall back to the locking mode, in order to avoid abort overheads of transactions. On the other hand, RAC allows unlimited concurrent access to other low-contention views to maximize concurrency, just as in transactional memory. Therefore, VOTM has the merits of both the locking mechanism and the transactional memory (TM) and integrated them nicely through RAC.

This thesis has also provided a theoretical analysis for RAC to investigate factors that indicate performance gain by restricting admission to a view, including disproportionately large portion of time spent in aborted transactions due to high contention and excessive TM mechanism overheads. Experimental results demonstrate that in many cases, RAC correctly responds to these situations by restricting admission to a view, thus improves the performance.

Apart from the improvements of programmability in VOPP, this thesis has done extensive experiments on two multicore architectures, a 16-core machine and a 64-core machine. Experimental results demonstrate that VOPP can provide a data race free environment with low overheads on multicore architectures, and VOTM outperforms both traditional transactional memory models and lock-based models in most benchmark applications.

Acknowledgements

There have been many people who have helped me throughout my PhD candidature. The completion of this thesis would not have been possible without the support and guidance of Associate Professor Zhiyi Huang, my primary supervisor. Throughout my time at the Department of Computer Science, Professor Huang helped me to grow as both a researcher, and more importantly, as an all-rounded person.

I am also very grateful to Dr Yawen Chen, who analyzed works carried in this thesis with a fresh pair of eyes at the statistician's point of view, and collaborated with Professor Huang and me to develop the RAC theoretical model.

I would also like to thank Dr Richard O'Keefe (my co-supervisor), Dr David Eyers, Dr Haibo Zhang and Cameron Kerr for training my programming skills and knowledge in operating systems, which are crucial in both this thesis as well as my future career. In addition, I would like to thank my postgraduate advisors Professor Mike Atkinson and Associate Professor Alistair Knott, and our Head of Department Associate Professor Brendan McCane for their academic and career guidance.

During my time in this department, I was very fortunate to have Jason, Manish, Qihang, Ayesha, Yan, Adeel, Bhaskar, Sheetal, Faisal, Jay, Umair, Nabeel, Reece, Hamza, Lynn, Azam, Maryam, Jie, Nick and all the people at the System Lab for all the advice, fun and great time shared.

Finally, I would like to acknowledge the love and support of my friends and family, and particularly my parents while I pursued my extended education. I cannot thank you all enough.

Contents

List of Tables

List of Figures

List of Publications

Part of this manuscript have already appeared as:

Journal Papers

1. Huang, Z. and Leung, K. Performance Evaluation of View-Oriented Transactional Memory. To appear in *Parallel Computing*.

2. Leung, K., Chen, Y., and Huang, Z. (2013). Restricted Admission Control in View-Oriented Transactional Memory. *The Journal of Supercomputing, 63(2), 348–366.*

3. Leung, K., Huang, Z., Huang Q., and Werstein, P. (2010). Data Race: Tame the Beast. *The Journal of Supercomputing, 51(3), 258–278.*

Conference Papers

1. Leung, K., Chen, Y., and Huang, Z. (2012). When and how VOTM can improve performance in contention situations. In *The Fifth International Workshop on Parallel Programming Models and Systems Software for High-end Computing, in Proceedings of the 41st International Conference on Parallel Processing.*

2. Leung, K. and Huang, Z. (2011). View-Oriented Transactional Memory. In *The Fourth International Workshop on Parallel Programming Models and Systems Software for High-end Computing, in Proceedings of the 40th International Conference on Parallel Processing.*

3. Leung, K. and Huang, Z. (2010). Maotai 3.0: Automatic Detection of View Access in VOPP. In *Proceedings of the 11th International Conference on Parallel and Distributed Computing, Applications and Technologies.*

4. Mair, J., Leung, K., and Huang, Z. (2010). Metrics and task scheduling policies for energy saving in multicore computers. In *Proceedings of the 2010 11th*

IEEE/ACM International Conference on Grid Computing, Brussels, Belgium, October 25–29, 2010, 266–273. IEEE.

5. Leung, K., Huang, Z., Huang, Q., and Werstein, P. (2009). Maotai 2.0: Data Race Prevention in View-Oriented Parallel Programming. In *Proceedings of the 10th International Conference on Parallel and Distributed Computing, Applications and Technologies, 263–271. IEEE Computer Society.*

Chapter 1

Introduction

Parallel programming has become pervasive with the advent of multicore and chip-multithreading (CMT) technologies [57, 90]. These technologies allow multiple processors to be packed into a chip, and multiple multicore chips often share memory and cache. Apart from servers and workstations, multicore CPUs also become prevalent in consumer devices, including laptops as well as mobile devices such as tablets and smartphones [87]. However, to utilize the benefits of multicore architectures, there is a pressing need for parallel programming models to facilitate both performance and convenience.

Currently, there are two camps of parallel programming models – message passing and shared memory. In the message passing paradigm, memory is private to each process, and processes share data by sending and receiving messages. Examples of this paradigm include PVM [36], MPI [3], Occam [16] and the Akka Framework [42] as well as modern agent-oriented models such as Erlang [4] and Charm++ [55].

Message passing is an efficient means of communication between different computing nodes in distributed environments such as cluster and grid where memory is not physically shared across computers. It allows programmers to finely tune the algorithm using the low-level message passing mechanism. Since there are no shared variables between processes, message passing models are data race free. However, models like MPI force the communication mechanism into the application algorithm, which makes data sharing tedious due to message passing, especially when the number of processes becomes large. Since programmers are forced to manually code the inter-process communication protocol, errors in the communication protocol can easily lead to *communication deadlock*, which can be difficult to debug [54, 68].

On the other hand, message passing is not the most efficient way for data sharing

in multicore architectures with shared memory. In these increasingly prevalent architectures, message passing between processes residing on the same multicore machine results in unnecessary memory copying, which can be eliminated when shared memory is used for communication in the multicore architectures [108].

Due to these reasons, the scope of this thesis will focus on the issues such as data race and concurrency control in shared memory parallel programming models which share data directly through physical memory of the multicore architectures.

1.1 Concurrency Control in Shared Memory

In shared memory programming models, variables could be shared among processes. However if multiple processes access a variable at the same time, and at least one of the processes writes to the variable, such a so-called data race condition can incur unexpected results [47].

For example, although incrementation is only a single statement:

```
x++;
```

If two processes increment the variable `x` concurrently, programmers would expect that at the end, `x` would be incremented by 2. However, the incrementation is not atomic. It is in fact carried out by the following three instructions:

```
load x R;
add 1 R;
store R x;
```

Since the two processes increment the variable `x` concurrently, the above instructions from the two processes may interleave in an order as shown in Figure 1.1, where P2 reads `x` into its register `R`, before P1 updates `x`. However, after P1 updates `x`, the register `R` of P2 still has the *old value* of `x` as 0, which it increments. When P2 writes its `R` to `x`, it tramples the result calculated by P1, which results in a data race bug. The final result of `x` is 1 in this scenario, rather than the expected 2.

```
x = 0;

   P1                  P2
load x R;                          // P1's register R = 0  x = 0
                    load x R;      // P2's register R = 0  x = 0
add 1 R;                           // P1's register R = 1  x = 0
store R x;                         // x = P1's register R (= 1)
                    add 1 R;       // P2's register R = 1  x = 1
                    store R x;     // x = P2's register R (= 1)
```

Figure 1.1: Code snippet demonstrating data race bug in two non-atomic operations not protected by a critical section

The Sequential Consistency (SC) model requires that "the result of any execution is the same as if the operations of all the processes were executed in some (global) sequential order, and the operations of each individual process appear in this sequence in the order determined by its (own) program" [60]. In the above case, "x++;" is supposed to be an atomic operation, and SC requires that the effect of the parallel execution be consistent with the result of a sequential execution of all operations. If the operations of "x++;" were to be executed sequentially, the result of Figure 1.1 would be 2 regardless of which incrementation is executed first.

To guarantee the sequential order of the operations on the same variable or data object, a critical section (also known as "atomic section" in transactional memory models to be described shortly) is needed to make sure an operation be performed atomically. The size of a critical section can range from a single statement, as in the above example, to a large code block that may take a significant portion of program execution time.

To implement the critical sections and guarantee their atomicity, traditionally there are two approaches: locking and transactional memory (TM).

1.1.1 Locking

Traditionally locking [59, 77] is used for concurrency control, where multiple processes [1] have to access a shared data object in an exclusive way. Locking is used in many shared memory programming models, including Java [37], C# [38], Python [93], Ruby [98], Scala [73, 91], Pthreads [72], Ada [17], Cilk [96], as well as modern partitioned global address space (PGAS) models such as UPC [30] and Chapel [21]. Atomic

[1] In the rest of the thesis, we use "process" to mean both process and thread for simplicity since they are identical in terms of concurrency control.

access to a shared object is achieved through a locking mechanism. This lock-based concurrency control is generally regarded as a pessimistic approach [97] where conflicts are prevented before they are allowed to happen. Even though locking is an effective mutual exclusive mechanism for concurrency control, it could result in the deadlock problem if multiple objects are locked in different orders by multiple processes, as shown in Figure 1.2.

```
P1                          P2
acquire_lock(1);
                            acquire_lock(2);
acquire_lock(2);
                            acquire_lock(1);
/* here P1 holding lock 1 waits for lock 2
   which is held by P2, but P2 will not
   release lock 2 until it gets lock 1
   ----> DEADLOCK */
  .....                     ......

  release_lock(2);          release_lock(1);
  release_lock(1);          release_lock(2);
```

Figure 1.2: Code snippet demonstrating a deadlock situation resulting from processes acquiring locks in different orders

Moreover, apart from the deadlock problem, fine-grained locks are often tedious for programming, while coarse-grained locks often suffer from poor performance due to lack of concurrency, especially when a large portion of program execution is spent in critical sections. For example, the Ruppert's Algorithm [85] has a central lock that protects a critical section which takes most of the execution time of the program. A total redesign of the algorithm would be required to break the central lock into multiple fine-grained locks to extract concurrency, which would require expert knowledge in parallel programming.

In addition, when a lock is acquired very frequently, which is common in applications with fine-grained locks, the lock itself can become a cache contention hotspot, especially on hardware with a large number of cores. This locking hotspot can severely impact the performance and scalability of an application [47].

1.1.2 Transactional Memory

To avoid the deadlock problem as well as to increase concurrency, Transactional Memory (TM) [46, 66] was proposed for shared-memory programming. In TM, atomic

access to shared objects is achieved through transactions. All processes can freely enter a transaction, access the shared objects, and commit the accesses at the end of the transaction. If there are conflicts of access among processes, one or more transactions will be aborted and rolled back. TM will undo the effects of the rolled-back transactions and restart them from the beginning. This transaction based concurrency control is labelled as an optimistic approach [9, 58] where it is assumed nothing will go wrong, and if it does go wrong deal with it later.

In terms of performance, both lock-based and TM-based approaches have their own merits in different situations. When access conflicts are rare, the TM-based approach has little roll-back overhead and encourages high concurrency since multiple processes can access different parts of the shared data simultaneously. In this situation, however, the lock-based approach has little concurrency due to the sequential access to the shared data, which results in low performance. To increase concurrency and performance, the programmer has to break the shared data into finer parts and use a different lock for each part. This solution using fine-grained locks often complicates the already-complex parallel programs and could incur deadlocks.

On the other hand, when access conflicts are frequent, the TM-based approach could have staggering roll-back overheads and is not scalable due to a large number of aborts of transactions. In the worst case, transactions can abort each other, and result in livelocks [19, 63, 99]. In such a situation, it is more effective to use the pessimistic lock-based approach to avoid the excessive operational overheads of transactions.

1.2 Motivation

This thesis investigates the issues of data race and concurrency control under the View-Oriented Parallel Programming (VOPP) paradigm [49, 52, 108].

VOPP is a novel data-centric model that bundles mutual exclusion and data access together. In VOPP, shared data is partitioned into non-overlapping *views*. The grain (size) and content of a view are decided by the programmer as part of the programming task, which is as easy as declaring a shared data structure or allocating a block of memory space. Each view can be dynamically created, merged, and destroyed. The most important property for views is that they do not intersect with each other. Before a view is accessed (read or written), it must be acquired; after the access of a view, it must be released. In this way, programmers only need to consider which data needed to access atomically, acquire the view as needed and leave the underlying system to

control the concurrency and grant access to the view.

Data-centric models like VOPP are safer for parallel programming. Traditionally concurrency control in parallel programming is code-centric, where lock primitives are used to demarcate critical code sections. Mistakes in the demarcation of critical sections using locks can result in problems like data race. In contrast, data-centric models [20] are only concerned with which shared object is used and thus lock it when it is being used. Since the shared object to be locked is known, the locking process could be done automatically by the underlying system.

For instance, since VOPP has the information of the views such as size and location, it becomes possible to automatically detect accesses to them. In this way, the programmability of VOPP can be improved. For example, a view can be automatically acquired upon its first access, and later released when the control flow of the execution leaves the scope of the view acquisition. This kind of extension of VOPP would relieve programmers from manually acquiring/releasing the views, and avoid mistakes such as missing acquiring/releasing primitives in the program and their related issues such as data race and deadlock.

Moreover, since access control of each view is independent from each other, it also becomes possible to individually control access of each view according to its own contention to maximize the performance. For example, if a view has very high contention, it can switch to the locking mode and only allow one process accessing it to stem the contention overhead, while another view with low contention can allow concurrent access by multiple processes without limitations, like traditional TM models, to maximize concurrency.

This thesis will build upon the VOPP paradigm to investigate its strengths and issues as a data-centric model, such as data race freedom, concurrency control, and performance evaluation on multicore architectures.

1.3 Contributions of this Thesis

The VOPP paradigm was originally proposed for distributed shared memory systems [50–52]. Its features as a data-centric model have never been explored on shared-memory multicore architectures.

This thesis first proposes a data race prevention scheme for VOPP, which can prevent data race from occurring in the first place. The efficiency of the scheme is evaluated experimentally against other parallel programming models. A shared-memory parallel

programming system, called Maotai 2.0, is implemented for VOPP on multicore architectures. Maotai 2.0 has enhanced VOPP with advanced features such as deadlock avoidance and producer/consumer view.

To further improve the programmability of VOPP, this thesis proposes and implements a scheme for automatic detection of view access, which no longer requires programmers to use explicit view acquire/release primitives. A view is automatically acquired when it is first accessed and released when the execution flow leaves the scope of the view acquisition. The thesis shows the automatic detection scheme improves the programming convenience of VOPP, and its programmability is similar to transactional memory models in many cases. The parallel programming system with automatic view detection is implemented and codenamed Maotai 3.0. Performance results show that Maotai 3.0 has superior performance over transactional memory models like TL-2 0.9.6 [26].

Then this thesis proposes a novel View-Oriented Transactional Memory (VOTM) paradigm that seamlessly integrates the merits of locking and TM into the same programming model. VOTM is designed based on the generic principle of VOPP. This data-centric model bundles concurrency control and data access together and therefore relieves the programmer from controlling concurrent data access directly with either locks or transactions. When a shared object (i.e. a view) is to be accessed, the programmer just simply uses acquire_view to inform the system that the corresponding view is going to be accessed. It is up to the system to decide whether the locking mechanism should be adopted or a transaction should be started for the concurrent access of the shared data.

This thesis also proposes an original Restricted Admission Control (RAC) scheme for VOTM that can dynamically adjust the number of processes allowed to access the same view. With the RAC scheme, a view in VOTM is restricted to be accessed by a limited number of processes Q (called admission quota) whose value ranges from 1 to the maximum number of processes (N). If Q is 1, the processes access the set of data objects sequentially as in the lock-based approach. If Q equals N, the RAC scheme behaves like the conventional TM systems where any process is allowed to start a transaction to access the data objects of the view. However, if Q is greater than 1 but smaller than N, only Q processes are allowed to access the data objects concurrently through transactions. If there are already Q processes accessing the data objects inside uncommitted transactions, other processes are excluded from accessing the set of data objects and have to wait until some existing transactions commit. In

addition, RAC can flexibly adjust Q at runtime according to the contention situation, e.g., the number of transactional aborts, to achieve optimal performance. This thesis proposes a theoretical model for RAC to measure the contention levels and decide when Q should be adjusted to achieve optimal performance for TM applications. As far as we know, this is the first time that a theoretical analysis is applied to model admission control of transactions. The RAC model is evaluated with microbenchmarks and show the model can correctly decide if Q should be adjusted at various contention levels. The experiment shows that this theoretical model is general enough to help measure the contentions for various TM systems.

1.4 Thesis Structure

The rest of the thesis is organized as follows.

Chapter 2 presents the data race prevention scheme for VOPP. The scheme is implemented using the memory protection mechanism (mprotect()) to guard against improper access of shared data. Experimental evaluation is carried out to show the performance of the scheme and its limitations.

Chapter 3 extends the VOPP paradigm with automatic view access semantics to further improve programmability. It also provides an experimental evaluation on the performance and programmability of this scheme.

Chapter 4 discusses the novel VOTM system, and presents a performance evaluation on VOTM, lock-based, and TM-based systems. It also gives an analysis on the benefits of VOTM over both lock-based and TM-based models.

Chapter 5 provides a theoretical analysis of the RAC algorithm used for concurrency control in VOTM. It gives discussion on how the contention among transactions should be modelled, in order to restrict the number of processes concurrently accessing the same view by setting the proper admission quota to the optimal value.

Chapter 6 discusses related work on other data-centric programming models and concurrency control mechanisms.

Finally, Chapter 7 concludes the thesis and sheds light on potential future work.

Chapter 2

Taming the Data Race

As mentioned in Introduction, data race is an important problem in shared memory programming models. A software bug caused by data race is often called *Heisenbug* [75] because it often disappears when one attempts to find it. There have been many studies on debugging data races. Some perform a post-mortem analysis based on program execution traces [22, 31, 44, 70, 71], while others perform on-the-fly analysis during program execution [8, 28, 69, 88]. Among modern shared-memory parallel programming models [23, 72, 74, 80], only Cilk++ [23] provides a data race detector called Cilkscreen [8, 23, 56].

Even though race detectors can help debug some data races, they often have the following problems:

- Race detectors are often expensive to run, both in terms of computation and memory space. For example, Cilkscreen can take up to 30 times the normal execution time of the debugged program to run and the memory footprint can be "several times" the memory footprint of the original application [23].

- Race detectors can only detect data races for one given input of a program. If data races do not occur when the program is run with a given input, this does not imply the program is data race free. The reason is that a different input may result in threads being executed in different order, and the resultant interaction may cause data races.

- To a novice programmer, race detectors can be difficult to use. For example, Cilkscreen gives a detailed trace of memory addresses and their associated function names and line numbers, which can be very scary and confusing to inexperienced programmers. In addition, this trace is of little help to programmers about

the dynamic nature of the data races, e.g. when and how the data races happen.

Instead of data race detection, this chapter proposes a data race prevention scheme in VOPP, which can prevent data races from occurring in the first place. It also presents an implementation of VOPP, Maotai 2.0, with a data race prevention scheme on multicore architectures.

The rest of this chapter is organized as follows: Section 2.1 describes a data race prevention scheme that can eliminate data races in VOPP. Section 2.2 presents the performance evaluation of Maotai 2.0 against other lock-based models including Cilk, OpenMP and Pthreads. Section 2.3 briefly introduces the advanced features of Maotai 2.0 for improving programmability and performance. Finally, Section 2.4 concludes this chapter.

2.1 Data Race Prevention

In VOPP, shared data is defined through views. Unlike most shared memory parallel programming models, variables are private to a process by default in VOPP. Shared objects must be *explicitly* defined as "views".

Views can be created, destroyed, merged, or resized, but a process must acquire a view (read-only or read-write) before accessing it and must release it after finishing with the view. The current VOPP implementation adopts the Single-Writer Multiple-Reader (SWMR) as its concurrency control model. At any given time, a view can either be read/written by one process or allow read-only access to multiple processes. In the current implementation, a view uses a contiguous memory space to store shared variables. Below is a simple example of VOPP in C.

```
typedef struct {int a[ARRAY_SIZE];
                int result;} Foo;
Foo *ptr;
if (0 == Vpp_proc_id) {
  /* master allocates view 0 with
     type SWV, which is a shared object
     with "Foo" type */
  Vpp_alloc_view(0, sizeof(Foo), SWV);
}
Vpp_barrier();
   ...
ptr = Vpp_acquire_view(0);
ptr->result += do_work(ptr->a);
Vpp_release_view(0);
```

Figure 2.1: An example of VOPP code

As illustrated in Figure 2.1, if a data structure should be shared by multiple processes, a view has to be created for it with Vpp_alloc_view. For exclusive access to the view, the view type is SWV, which means "Single Writer View". However, VOPP also provides other advanced views to enhance the programmability and flexibility (refer to Section 2.3).

If a process wants access to a view, the view must be acquired with Vpp_acquire_view (or Vpp_acquire_Rview for read-only access). The view must be released with Vpp_release_view after accessing it.

A summary of the VOPP API is shown below:

int Vpp_alloc_view(int vid, size_t size, view_type type)

Creates a view with ID vid, and size size. If vid is a negative value, the system will allocate a free view ID. The view type type can be SWV (single writer view), MWV (multiple-writer view) or PCV (producer-consumer view). MWV and PCV will be discussed in detail in Section 2.3. This function will return the view ID of the allocated view. On failure, this function will return -1.

void Vpp_free_view(int vid)

Frees the view vid.

void *Vpp_acquire_view(int vid)

Acquires read-write access to the view vid and return its base address. Upon failure, NULL will be returned.

void *Vpp_acquire_Rview(int vid)

Acquires read-only access to the view vid and return its base address. Upon failure, NULL will be returned.

void Vpp_release_view(int vid)

Release access to the view vid.

void Vpp_barrier()

Block the process until all processes reach this barrier.

2.1.1 Implementation

In the data race prevention scheme, data races are prevented by a memory protection mechanism available in most UNIX systems. All views are initially protected from access using system calls such as mprotect(). mprotect() can deny access to a page, or allows read-only access to a page, or allows read/write access to a page. This mechanism is used to prevent a view from illegal accesses. Only after a view is acquired is a process allowed to access the memory pages of the view via mprotect(). When a view is released, the process is again denied access to the view.

If a process accesses a view before Vpp_acquire_view or after Vpp_release_view, the pages of the view would not have the necessary access permission and thus a segmentation fault will occur. The system will handle the fault, send a warning message to the programmer that a view is accessed without acquisition, and quit the program execution.

In this way, a view can either be written to by one process or read by multiple processes at a time. Programmers do not need to worry about the data race bugs. If a view is accessed by calling Vpp_acquire_view, mutual exclusion of the view access is automatically done by the system. If a view is accessed without view acquisition, a segmentation fault will occur, and the system will alert the programmer about which view is accessed without acquisition. The programmer can easily fix the bug by inserting Vpp_acquire_view and Vpp_release_view into the faulted code section.

The extra cost of this data race prevention scheme is the overhead of the memory protection. In the VOPP implementation Maotai 2.0, this cost is very

low. On a Sun T2000 Server equipped with a 1GHz UltraSPARC T1 processor [94], micro-benchmarking results demonstrate that the overhead of memory protection added to the view primitives is generally very low (around 2-3μs). The exception is Vpp_acquire_view, requiring up to 35μs extra, which covers the essential overhead of the memory protection mechanism(see Table 2.1). Note that Vpp_acquire_Rview and Vpp_release_Rview means acquiring and releasing views as read-only.

Table 2.1: Breakdown of view primitive costs (in μs)

Primitive	no prot	prot	cost
Vpp_acquire_view()	3.14	39.08	35.94
Vpp_acquire_Rview()	3.60	6.32	2.72
Vpp_release_view()	1.91	4.54	2.63
Vpp_release_Rview()	1.99	4.64	2.65

However, in application benchmarks, this overhead does not cause noticeable difference in application speedup. Table 2.2 shows the speedups (at 32 processes) of different applications with and without memory protection in Maotai 2.0. This experiment has six benchmark applications: Successive Over-Relaxation (SOR), Gaussian Elimination (GE), Integer Sort (IS), Neural Network (NN), Mandelbrot, and Mergesort, which typically represent a wide variety of parallel applications. For details of these applications, refer to Section 2.2. As we can see from Table 2.2, in all 32-process benchmark cases, the difference is around 0.5%. Therefore, the overhead introduced by data race prevention is trivial.

Table 2.2: Effects of memory protection on benchmark application speedups with 32 processes

Application	no prot	prot
SOR	16.82	16.77
GE	22.41	22.36
IS	16.51	16.47
NN	16.98	16.92
Mandelbrot[a]	7.61	7.60
Mergesort	12.52	12.50

[a]speedup with eight processes

One issue about the implementation is that memory protection such as mprotect is

page-based. Therefore, in order to protect view data properly, memory space allocated to a view is aligned by pages. This can result in memory space wastage. Table 2.3 shows the requested and actual sizes of the memory space allocated by VOPP in the benchmark applications. The page size is 8kB and 32 processes are used when the data of the table are collected. From this table, it can be seen that some applications like GE and Mandelbrot, which have many views that do not exactly fit a page, have a higher proportion of memory wastage (up to 51%), though other applications have less than 7% wastage. However, this memory wastage is much smaller than the memory footprint of race detectors, which can be "several times" the memory footprint of the original applications.

Table 2.3: Requested vs actual VOPP shared size (in Kbytes) in different applications

Algorithm	Requested	Actual	Wasted	Percent wasted
SOR	4,097,024	4,194,304	97,280	2.32
GE	64,016,004	98,328,576	34,312,572	34.9
IS	4,194,304	4,194,304	0	0
NN	271,612	294,912	23,300	7.90
Mandelbrot	2,000,000	4,096,000	2,096,000	51.2
Mergesort	1,600,001,280	1,600,274,432	273,152	0.0171

Fortunately, with architectural support of variable-size pages [15, 104], this memory wastage can be greatly reduced.

2.2 Performance Evaluation with Other Models

This section compares the performance of Maotai 2.0 with other modern shared memory parallel programming models OpenMP and Cilk. These models are evaluated in benchmark applications from the SPLASH-2 benchmark suite [105] including Successive Over-Relaxation (SOR), Integer Sort (IS), Gaussian Elimination (GE), Neural Network (NN) and Mandelbrot, as well as Mergesort adapted from the Cilk-5.4.6 benchmark [96].The SPLASH-2 benchmark suite represent different classes of parallel algorithms commonly found in real-life applications. The experiments are carried out on a Sun T2000 server with an UltraSPARC T1 processor and 16GB memory. The UltraSPARC T1 has eight cores, each of which is clocked at 1GHz and supports four hardware threads. In total, the UltraSPARC T1 processor supports up to 32 hardware threads [94]. Linux kernel 2.6.24-sparc64-smp and the compiler gcc-4.4 are used during benchmarking. The benchmark applications are implemented on Maotai 2.0,

Cilk-5.4.6 [96], and OpenMP 3.0 [74], respectively. All programs are compiled with the optimization flag "-O2". In each case, speedup is measured against the serial implementation of the benchmark algorithm. The elapsed time calculated in each case excludes initialization and finalization costs, because they are one-off and are difficult to measure within the program in models that involve source-translation, such as Cilk and OpenMP. Instead, startup and finalization times for each model are measured separately. Runtime of functions that are irrelevant to the original application, such as generation of random sequences and result-verification, are also excluded.

Successive Over-relaxation (SOR) is a multiple-iteration algorithm where each element is updated by the values of the neighbouring elements from the last iteration. In this experiment, the implementation is adapted from [108]. Matrix size is set to 8000×4000 and 40 iterations are performed.

The Integer Sort (IS) algorithm used in this experiment is based on the NPB version [101]. This is a counting-sort algorithm. In this experiment, the problem size is 2^{26} integers with a B_{max} of 2^{15} and 40 repetitions are performed.

The Gaussian Elimination (GE) implementation from [108] is used in this experiment and the matrix size is set to 4000×4000.

The parallel Neural Network (NN) algorithm is based on [78]. This algorithm trains a back-propagation neural network in parallel using a training data set. In this experiment, the size of the neural network is set to $9 \times 40 \times 1$ and the number of epochs is set to 200.

The Mandelbrot algorithm is embarrassingly-parallel. However, the workload of pixels is extremely uneven, and thus requires a load-balancing mechanism to prevent process starvation [39, 103]. In this experiment, the size of the screen is set to 500×500, the maximum number of iterations is set to 500 and each pixel is calculated 5000 times. The maximum number of processes / threads is set to eight for this experiment because hyperthreading relies on memory latency. Since this application has very few memory accesses, there is little speedup when more processes / threads than the number of CPU cores are used (The UltraSparc T1 has eight cores).

The parallel Mergesort algorithm is recursive [61, 96] and is implemented verbatim in Cilk and OpenMP to test performance of the newly-available task-parallelism feature in OpenMP [7]. The array consists of 200 million integers. This algorithm is converted to the iterative version for VOPP. The iterative version requires the number of processes to be a power of 2. This version first divides the array equally between the processes and each process sorts its own subarray. Then the merge procedure largely models the

recursive version of the parallel merge algorithm.

Since the UltraSPARC T1 has only one floating-point unit, all floating-point calculations in the above algorithms are converted to integer calculation to avoid the bottleneck at the floating-point unit. Removal of floating point calculations is done in all implementations and does not affect the scalability of the algorithm or the fairness of the comparison.

2.2.1 Experimental Results

The experimental results are illustrated with speedup curves. Speedup curves on Maotai 2.0, Cilk, and OpenMP are given for each application. In the discussion below, n refers to the number of processes / threads.

Speedup is calculated by:

$$speedup = \frac{time_{serial\_implementation}}{time_{parallel\_implementation}} \tag{2.1}$$

To ensure fair comparison, the same serial implementation of each benchmark application is used as a baseline for calculating speedups of all parallel programming models.

For SOR (Figure 2.2), Maotai 2.0 has the best performance. At $n = 32$, Maotai 2.0 is 13.6% better than Cilk and 17.9% better than OpenMP.

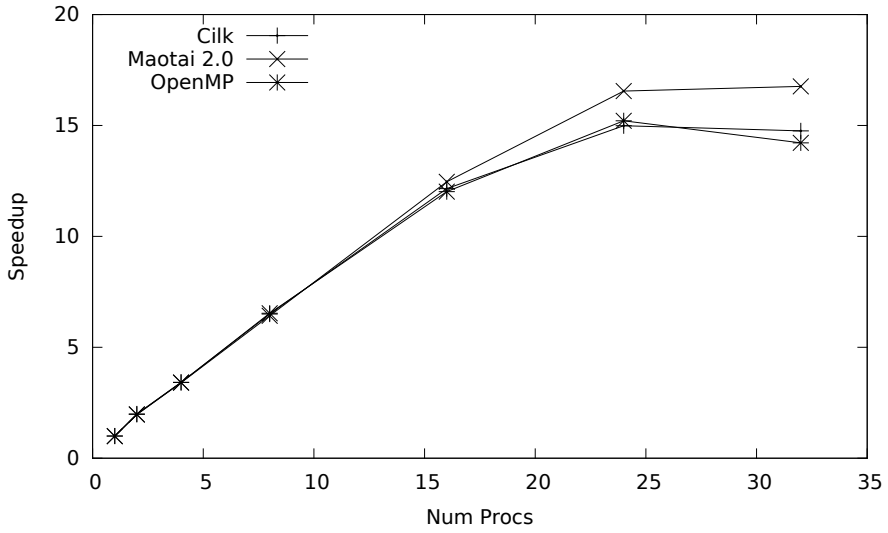

Figure 2.2: Speedup of SOR

For GE (Figure 2.3), Maotai 2.0 again has the highest speedup. At $n = 32$, Maotai 2.0 is 7.4% better than Cilk and 33% better than OpenMP.

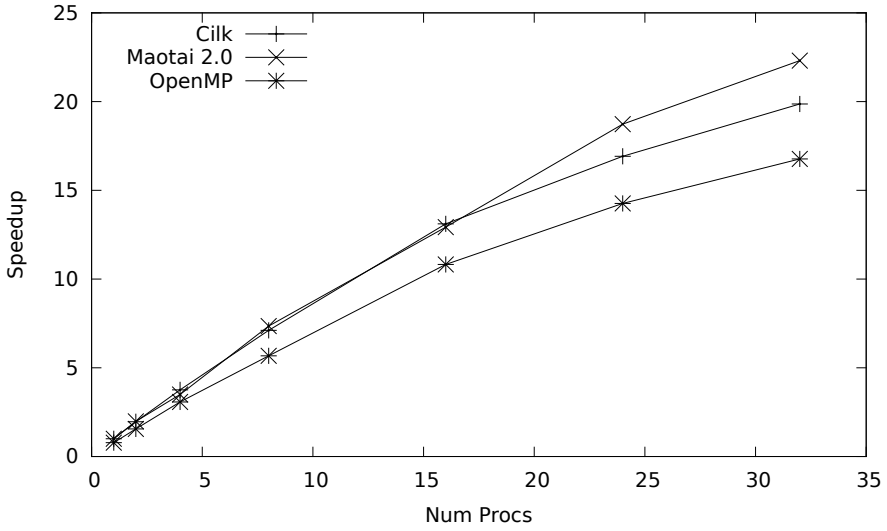

Figure 2.3: Speedup of GE

In IS (Figure 2.4), there are less variations in speedups in different models. However at $n = 32$, Maotai 2.0 is 5% faster than Cilk and 15% faster than OpenMP.

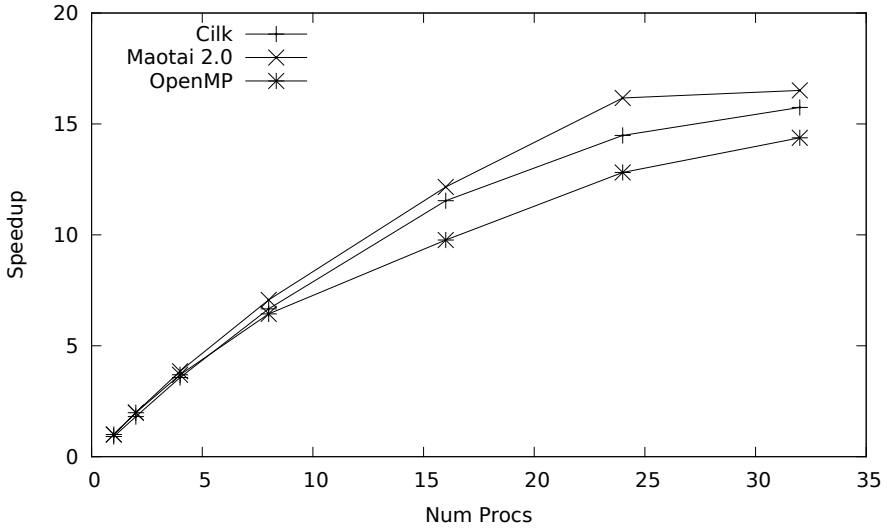

Figure 2.4: Speedup of IS

In NN (Figure 2.5), all models have similar speedups. Maotai 2.0 is 3.1% faster than OpenMP, but it is 1.8% slower than Cilk.

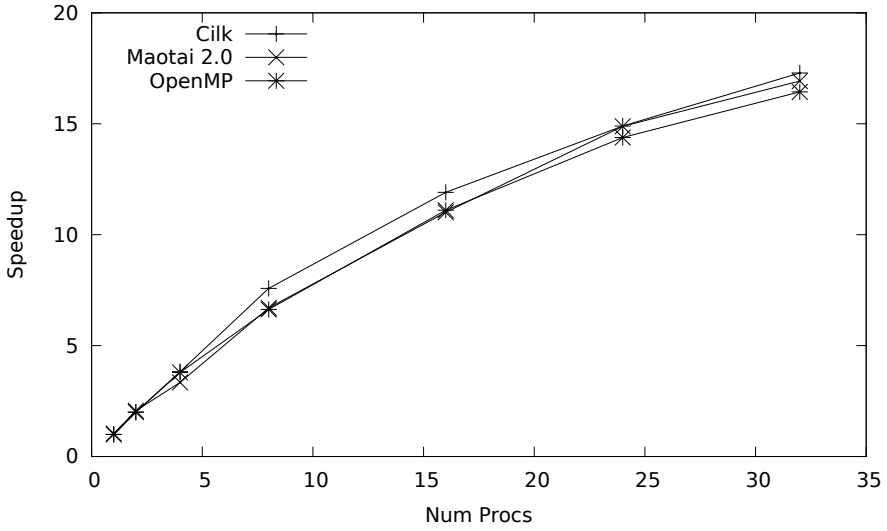

Figure 2.5: Speedup of NN

In Mandelbrot (Figure 2.6), there are relatively little differences between speedups of different models. At $n = 8$, Maotai 2.0 is 0.8% faster than Cilk and 7.2% faster than OpenMP.

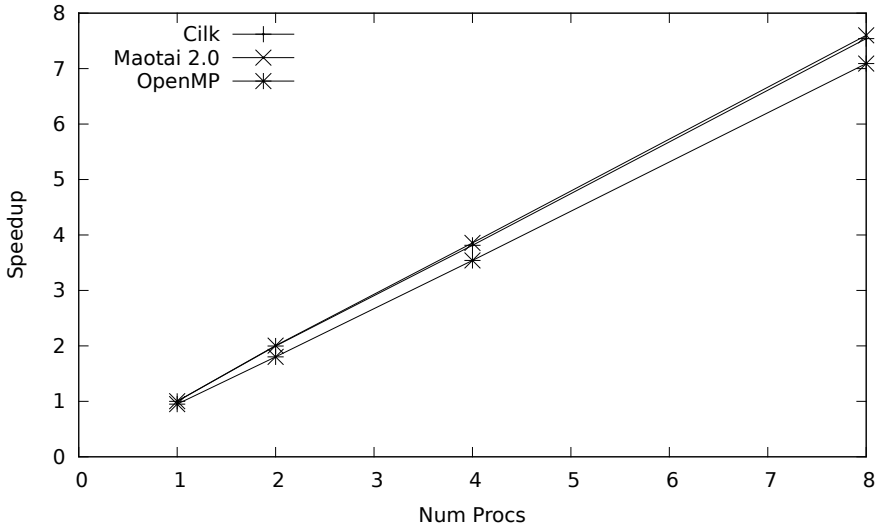

Figure 2.6: Speedup of Mandelbrot

For Mergesort, Figure 2.7 shows speedup of Maotai 2.0 is relatively slower. This issue will be addressed in Section 2.2.2.

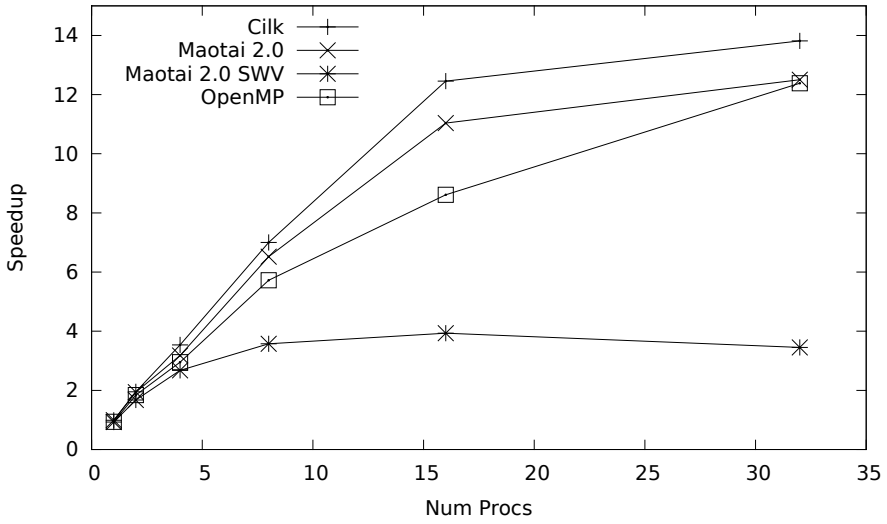

Figure 2.7: Speedup of Mergesort

Note that, in the above collected results, the standard deviations of the elapsed time at $n = 32$ for Maotai 2.0 and Cilk cases are less than 0.1s, but the standard deviations of the elapsed time for OpenMP are between 0.2 to 0.5s, which may be due to the random nature of the OpenMP task scheduler.

Table 2.4 presents the startup and finalization time of each system. As expected, startup and finalization costs for thread-based models including Cilk and OpenMP are lower than process-based system like Maotai 2.0.

Table 2.4: Combined startup and finalization time (in *ms*) for different number of processes/threads on a Sun T2000 server

	1	2	4	8	16	24	32
Cilk	2	2	2	2	2	2	2
OpenMP	2	2	2	2	2	2	2
Maotai 2.0	9	10	11	13	15	19	22
Serial	2						

All thread-based models have the same combined startup and finalization time as the serial version regardless of the number of threads. Maotai 2.0 has a startup/finalization cost of 9ms (at $n = 1$) and the cost grows to 22ms at $n = 32$, almost linear to the number of processes. Despite Maotai 2.0 having a larger startup/finalization overhead, the 22ms is still negligible compared to the time consumed in $n = 32$ cases, which is at least 10 seconds. Also the startup/finalization

time in Maotai 2.0 is only a one time event, therefore, this overhead should have negligible effect on the speedup curves.

2.2.2 Discussion

The following is an analysis on why Maotai 2.0 performs better or worse than other systems.

The producer/consumer view (PCV) in Maotai 2.0 enhances both programmability and performance of SOR and GE. In SOR, PCV is used to pass boundary rows to neighbour processes, thus allowing the natural expression of the message-passing relationship without the use of barrier, which would hold up irrelevant processes. Apart from programmability, the resultant performance gain is reflected in Figure 2.9, where the PCV VOPP version is 11.2% faster than the barrier-based SOR version. More detailed reason regarding why PCV is more efficient than barrier will be explained in Section 2.3.2.

Similarly in GE, PCV is used to *broadcast* the pivot row and the swap index, which improves programmability by mimicking the broadcasting semantics in the parallel algorithm. Also the removal of barriers by PCV improves the VOPP performance by 4.2% (Figure 2.10). Time is saved by replacing lock and barrier primitives with a PCV primitive.

Multiple-Program Multiple-Data (MPMD) models such as Cilk/Cilk++ and OpenMP do *not* have barriers because in this case, the parallel calculation part is conveniently expressed by parallel for-loop (or in case of Cilk, spawn recursive task decomposition threads and sync at end of parallel calculation) and the pivot part is run serially. Synchronization is implicit in the parallel for-loop construct, where tasks are forked at the beginning of the loop and joined at the end of the loop, therefore these fork-join actions are essentially barriers and have the similar overhead to the barriers in VOPP. In multiple-iterative cases such as GE and SOR, the cumulative task scheduling and synchronization overheads can be considerable. Therefore, the Maotai model would be more suited for these problems.

Mandelbrot is an embarrassingly-parallel algorithm. This application demonstrates the slight performance advantage of the VOPP single-program multiple-data (SPMD) model, in which a task queue is used to balance workload in the program, instead of using general runtime schedulers as in OpenMP and Cilk. This result has also demonstrated that the implementation of the system queue is efficient.

In IS, the performance advantage seen in Maotai 2.0 over other models can be

attributed to the split of global keyden array into N views, where N is the number of processes. In the global keyden construction step, each process updates *all* global keyden parts in the round-robin fashion, starting from the $proc\_id^{th}$ part. Here, the SWMR view access pattern removes the need for barriers for preventing data race due to multiple processes updating an element simultaneously. This removal of barriers can contribute to the performance gain by the VOPP program.

In NN, since multiple items are updated by multiple processes at the end of the iteration, barriers are still used in the VOPP program. Therefore, it has the same synchronization overhead of other models. However the performance of Maotai 2.0 is still comparable to other models, which shows that being data race free has little impact on performance.

However, the SWMR model in VOPP does have its limitations in cases where the access pattern changes in every iteration. In those cases, view data must be copied to a local buffer of a process, where the process works on the data. After the data is processed, the view is acquired again by the process and the results copied back to the view. In applications tested in this chapter, Mergesort is such an example. In Mergesort, the resultant excessive memory-copying renders the implementation unscalable (Refer to VOPP-SWV in Figure 2.7). For this application, VOPP does trade off some programming convenience and performance for data race prevention. However, Maotai 2.0 has provided a Multiple Writer View (MWV) to offer the programming convenience for experienced programmers. A MWV is a view that can be accessed at *different locations* simultaneously by multiple processes. Therefore, it is up to the programmer to make sure there is no data race in a MWV. In contrast to other programming models, the data races of a MWV are *confined* inside the view should they occur. This alternative MWV implementation allows multiple processes to work directly on the view and avoid memory copying. With MWV, the speedup of Mergesort in Maotai 2.0 is comparable to other shared-memory models. Figure 2.7 shows, at $n = 32$, Maotai 2.0 (refer to VOPP-MWV) is 1% faster than OpenMP, but 9% slower than Cilk.

Cilk performs very well in cases like Mergesort and NN. This can be attributed to its recursive task decomposition that ensures cache locality [61].

The parallel for-loop in OpenMP allows easy specification of data-parallelism. However, it would introduce a task-scheduling cost, especially when the workload is fixed and no load-balancing is required. The lower speedups of GE, SOR and NN of OpenMP can be attributed to this parallel for-loop overhead. Although Cilk++ cannot be benchmarked in this experiment because it does not support sparc64-smp, its equivalent

construct cilk_for can also have the similar task-scheduling overhead.

2.3 Advanced Features in Maotai 2.0

In addition to data race prevention, Maotai 2.0 also offers primitives for acquiring multiple views in order to avoid deadlocks, producer/consumer views, and system queues to enhance programmability and performance. These features are discussed below.

2.3.1 Deadlock Avoidance

Similar to data race, deadlock is another pain that can happen easily but is difficult to debug in shared-memory parallel programming. In VOPP, deadlock can happen if views are acquired in a nested way and different processes acquire them in different orders.

To avoid deadlocks due to acquiring multiple views in different orders, Maotai 2.0 offers primitives for acquiring multiple views. Programmers can list all views to be acquired with these primitives which will acquire the views in a specific, same order. In this way, there is no chance for deadlocks to happen.

An example illustrating the use of the primitives for acquiring multiple views is shown in Figure 2.3.1

```
/* acquire access to both view 0 and 1 */
Vpp_acquire_multiviews(0, &ptr0, 1, &ptr1);
ptr0->result += compute0(ptr0->a, ptr1->a);
ptr1->result += compute1(ptr1->a, ptr0->a);
Vpp_release_view(); /* release all views */
```

Figure 2.8: Code snippet showing how multiple views are acquired together in VOPP

In the above example, the process acquires both view 0 and 1 with Vpp_acquire_multiviews which puts the view base addresses into ptr0 and ptr1. Finally the process releases both views with Vpp_release_view.

Note that the above solution cannot eliminate deadlocks from VOPP programs as the data race prevention scheme does data races. There are two reasons: first, the

programmer may choose not to use Vpp_acquire_multiviews for nested view acquisition; second, even if the programmer would like to use the primitive, it is difficult to know which views to acquire in advance in some programs where inner views can only be decided after the outer views are processed.

Nevertheless, the above primitives provide an avenue for novice programmers to avoid unnecessary deadlocks.

2.3.2 Producer/Consumer View

A Producer/Consumer View (PCV) is provided to allow direct expression of producer/consumer relationships in parallel algorithms. Traditionally barriers are used to synchronize the producers and the consumers in shared memory parallel programming. Barriers are expensive because they make all processes wait, which causes unnecessary waiting in applications where producer/consumer processes can be individually synchronized. Moreover, the cost of barriers would increase with increasing number of processes. With the introduction of PCV, programming with producer/consumer problem is more straightforward and thus increases programmability. Additionally PCV can avoid expensive synchronization overhead since a consumer process only synchronizes with its producer process.

PCV is implemented as a queue. The producer enqueues a new version of the view by acquiring the view, producing the data, and finally releasing the view. The consumer dequeues a version of the view by acquiring read-only access to the view. After it finishes with the view, it releases the view whose buffer may be recycled by the producer.

In this experiment, the SOR and GE benchmark applications demonstrate that PCVs give a better speedup than barrier based implementations. Figure 2.9 and 2.10 shows the speedup difference between applications using barriers and those using PCVs.

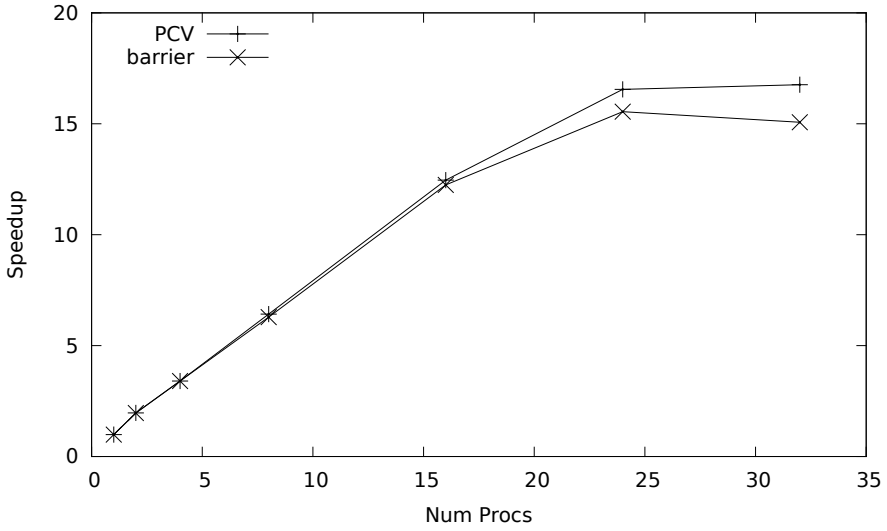

Figure 2.9: Speedup of SOR in VOPP

Figure 2.9 shows the speedup of SOR which uses PCV to improve its performance. Compared with its barrier implementation, the improvement of speedup is 11.2% at 32 processes.

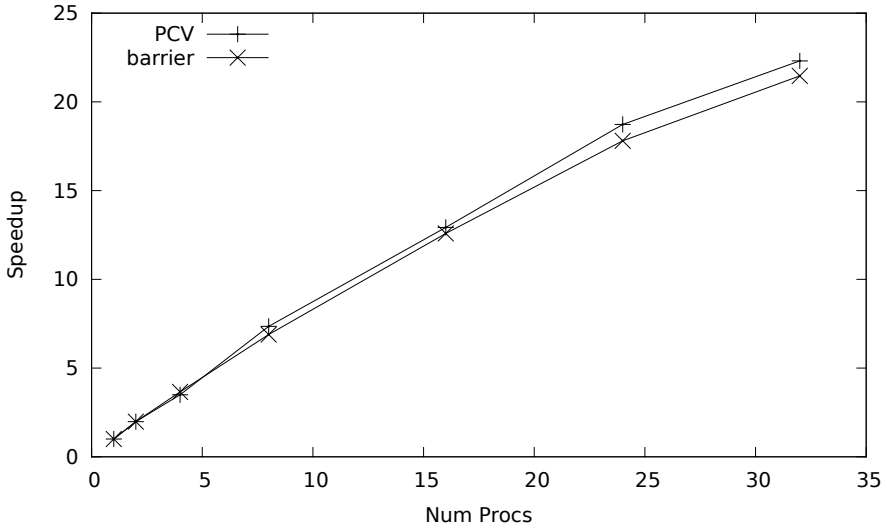

Figure 2.10: Speedup of GE in VOPP

Figure 2.10 shows the speedup of GE which uses PCV to improve its performance. Compared with its barrier implementation, the improvement of speedup is 4.2% at 32 processes.

2.3.3 System Queues

System queues are provided in Maotai 2.0 to store view IDs. This facility allows easy implementations of task queues. Task queues are good for load balancing parallel applications (e.g. Mandelbrot and tree search algorithms), where the data for each job or node can be put in a view and its ID is simply enqueued in a system queue for other processes to work on.

In Maotai 2.0, the enqueue and dequeue calls are efficient. In a microbenchmark test on a Sun T2000 server, an enqueue call only takes $2.65\mu s$ and a dequeue call takes $2.56\mu s$.

2.4 Concluding Remarks

The data race prevention scheme based on views is shown to be efficient and adds little extra overhead to parallel programming systems. Though there is some memory wastage due to page alignment in the implementation, architectural support for variable-size pages will significantly reduce the wastage. Even with a fixed page size, view constructs are useful to remove data races. Moreover, "big data" applications tend to have huge data sets, which means the size of a view will become much larger and can easily surpass several pages. This trend will make the memory wastage proportionally small.

With the advanced features in Maotai 2.0, the performance and programmability of VOPP are enhanced. Though strict SWV views are rigid for some applications like Mergesort, Maotai 2.0 offers MWV views to allow programmers to fall back to traditional shared memory programming, with the risk of data races that are confined in a single MWV view.

Performance results demonstrate that Maotai 2.0 is very competent among modern parallel programming models, especially with the unique data race prevention scheme.

To further improve the programmability of VOPP, the next chapter will propose an automatic detection scheme for view access, which will free the programmers from manually acquiring/releasing views.

Chapter 3

Automatic Detection of View Access

As discussed in Chapter 2, Maotai 2.0 provides a data race free parallel programming model based on VOPP. However, in Maotai 2.0, views must be explicitly acquired before access and released after access. It is often troublesome to manage view acquiring/releasing constructs.

For example, Figure 3.1 shows a serial version of a list traversal program and its Maotai 2.0 version.

```
/* serial version */
typedef struct Node_rec Node;
struct Node_rec {
  Node *next;
  Elem elem;
};

Node *list_search(Elem elem, Node *list) {
  while (NULL != list) {
    if (elem == list->elem) {
       return list;
    }
    list = list->next;
  }
  return NULL;
}
```

```
/* Maotai 2.0 */
typedef struct Node_rec Node;
struct Node_rec {
  Node *next;
  Elem elem;
};

Node *list_search(Elem elem, int vid) {
  Node *list = Vpp_acquire_view(vid);

  while (NULL != list) {
    if (elem == list->elem) {
      Vpp_release_view();
      return list;
    }
    list = list->next;
  }
  Vpp_release_view();
  return NULL;
}
```

Figure 3.1: Code snippets comparing serial and Maotai 2.0 implementations of the list traversal function

In the above list traversal in Maotai 2.0, the view is first acquired by Vpp_acquire_view, which returns the base address of the list, then the while loop traverses the list until the element elem is found. When elem is found (within the while loop), the view must be manually released by Vpp_release_view before returning the current element. If the element cannot be found, then the view will be released and the search function will return NULL. The code in the while loop is prone to error, because it is easy to forget to release the view before calling return within the while loop. If that happens, the next process that is acquiring the view will wait forever.

To solve this problem, this chapter proposes a scheme for automatic detection of view access which has greatly improved the programming interface of VOPP (refer to Section 3.1 for details), as VOPP no longer requires programmers to use explicit view acquire/release constructs. In this scheme, a view is automatically acquired when first accessed and released when leaving the scope of the view acquisition. The automatic view access detection scheme is shown to improve the programming convenience of VOPP, the programmability of which is similar to transactional memory models in many cases.

This chapter presents the parallel programming system, Maotai 3.0, which is based on the VOPP paradigm with the automatic detection scheme that supports both C and C++. Performance results show that the cost for the automatic detection is relatively small, and Maotai 3.0 has superior performance over transactional memory models like TL-2 0.9.6 [26], which provides a similar programming interface as Maotai 3.0.

The rest of the chapter is organized as follows. Section 3.1 describes the automatic detection scheme, the language constructs, and implementation details of Maotai 3.0. Section 3.2 compares the programmability of Maotai 3.0 with transactional memory models. Section 3.3 covers experimental results and performance evaluation. Finally, Section 3.4 concludes this chapter.

3.1 The Programming Model and Implementation Details

The introduction of automatic detection of view access helps remove the explicit view acquiring and releasing, and thus greatly simplifies the programming interface to shared data access in Maotai 3.0. It reduces extra code instrumentation needed to parallelize existing serial code. This section will discuss the language constructs and their semantics used in Maotai 3.0, as well as the implementation details of the

automatic detection scheme.

3.1.1 Automatic Detection of View Access

In this scheme, a view is automatically acquired when its memory is first accessed. Then the view is automatically released when control leaves the *scope of view acquisition*.

The scope of view acquisition is often the function that first accesses the view. During the execution of such a function, the executing process acquires the view when it is first accessed, and automatically releases the view when the function returns. Applications in this experiment shows that, in most cases, this functional scope of view acquisition is the intention of the programmer. Below is an example that shows at what time a view, called foo, is acquired and released automatically in the function func.

```
VPP void func(void) {
  Foo *foo = Vpp_alloc_view(sizeof(foo[0]), SWV);

  .....
  foo->index = 5;            /* view foo acquired */
  printf("%d\n", foo->val);
  ....
  ....
}                            /* view foo released */
```

Figure 3.2: A simple example illustrating when a view is automatically acquired and released under Maotai 3.0

Bearing in mind the above scope of view acquisition, the list traversal code becomes as follows.

```
/* Maotai 3.0 */

typedef struct Node_rec Node;

struct Node_rec {
  Node *next;
  Elem elem;
};

VPP list *list_search(Elem elem, Node *list) {
  while (NULL != list) {
    if (elem == list->elem) { /* view acquired */
       return list;          /* view released */
    }
    list = list->next;
  }
  return NULL;             /* view released */
}
```

Figure 3.3: List traversal in Maotai 3.0

In the list traversal code above, there is very little code changes compared with the original serial code. The only changes to the serial code are adding the keyword VPP as an attribute of the function list_search().

Compared with Maotai 2.0, the programmers do not need to keep track of view IDs and the acquire/release statements.

The keyword VPP is used to declare that a function will have effect on the scope of view acquisition. When a VPP function returns, it will automatically release all views acquired during the execution of the function, including those views acquired in the callee functions. Also the subsequent callee functions have access to the views once they are acquired.

For example, in the example code below, func1 acquires view 1 and then calls func2. func2 inherits the acquisition of view 1 throughout its scope (Figure 3.5).

```
/* a third-party non-VPP library function */
void func3() {        /*  inherit v1 and v2  */
  ptr_3->val = 0;     /*  acquire v3,
  ........                  but v3 belongs to
                            immediate VPP ancestor (func2())
                            and will only be released
                            at the end of func2() */
}

VPP void func2(Object *ptr_1) { /* inherit v1 */
  ptr_1->done = 1;
  ....
  ptr_2->index =      /*  acquire v2  */
    ptr_1->index + 1;
  ....
  func3();            /*  func3 inherits v1 and v2 */
  ....
  ....
}                     /*  release v2, v3 */

VPP void func1() {
  ptr_1->index = 0;   /*  acquire v1  */
  ....
  func2(ptr_1);    /* func2 inherits v1 */
  ....
}                     /*  release v1  */
```

Figure 3.4: Code snippet illustrating the inheritance of views acquired by VPP functions

Similarly, func2 acquires view 2 and calls func3. func3 inherits the acquisition of view 1 and 2 throughout its execution.

However, since func3 does *not* have the VPP attribute, it has no effect on the scope of view acquisition. Therefore, view 3 that is acquired during the execution of func3 will *not* be released when func3 returns. The scope of view acquisition for view 3 is

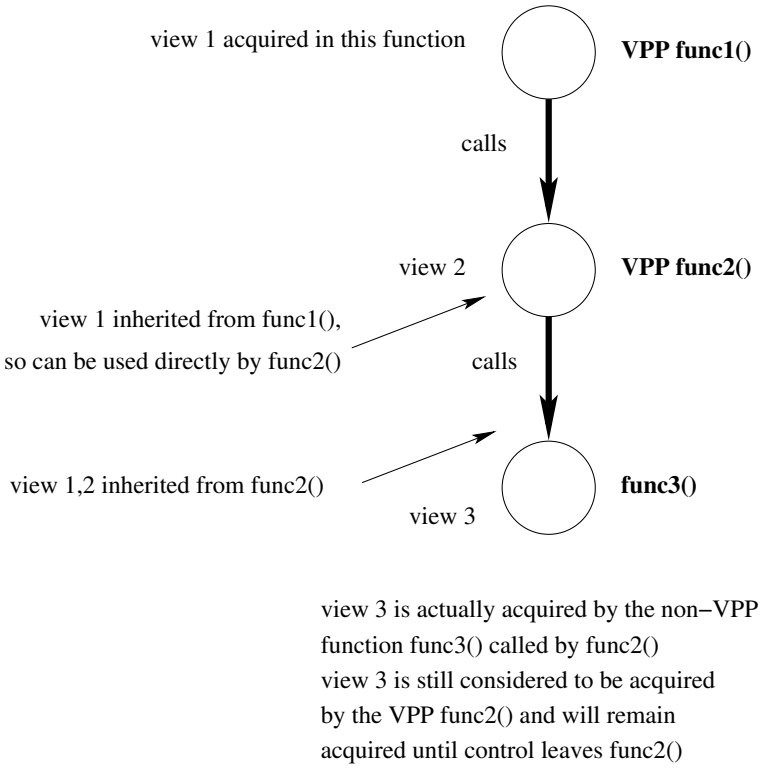

Figure 3.5: Inheritance of views acquired by VPP functions

func2, the immediate VPP ancestor of func3, which will release view 3 as well as view 2 when it returns.

From above, it can also be seen that the scheme supports recursive calls where a new view is acquired in every recursive call, because views acquired by the callee function still belong to the callee function, and is released when control leaves the callee function even if it is the same function as the caller function.

In summary, if a view is acquired in a VPP function, it will be automatically released when control leaves the scope of the VPP function. However, if a view is acquired in a non-VPP function, then the view will belong to the immediate VPP ancestor of the non-VPP function, and will be held until control leaves the scope of the *immediate VPP ancestor function.*

3.1.2 View Scope Construct

The automatic view access detection model described above works well in most cases. However, in some cases, a view acquired by a callee function is actually intended to be held until the end of the *current* function. Even though this can be achieved by making the callee function a non-VPP function, this restricts the use of the callee

function as a VPP function. The following example illustrates this problem in more details.

```
VPP void bar(char *shared_str) {
  ......
  shared_str[0] = 'a'; /* view shared_str acquired */
    .
    .
    .
}                      /* view shared_str released */

VPP void foo(char *shared_str) {
  .....
  bar(shared_str);   /* view shared_str acquired and
                        released by bar, but
  ....                  actually intended to be
  ....                  acquired until end of foo() */
  ....
  ....
  str[1] = str[0] + 1;  /* RW access to view
  ....                     shared_str reacquired */
  ....
}                       /* shared_str released */
```

Figure 3.6: An example illustrating how views acquired by a callee function can be unwittingly released

In the example above, foo() intends to hold the view shared_str during its execution, though it is acquired during the execution of bar(). However, since bar() is a VPP function, and shared_str is acquired during the execution of bar(), under the automatic detection scheme, shared_str will be released when bar() returns. Therefore, shared_str will be re-acquired when it is accessed again in foo(). In this situation, the view acquisition of shared_str is unwittingly fragmented. Though there is no data race involved in this situation, it may affect the atomicity of the operation on shared_str intended by the programmer.

To address this problem, the view scope construct:

VPP_View(access_type, pointer_to_view,...) {...}

is proposed to allow views to be automatically acquired at the beginning of the declared scope, where access_type can either be VPP_RO or VPP_RW, which stand for read-only access and read-write access respectively. Programmers can use view scope constructs to manually define the scope of view acquisition. Any views automatically acquired within a view scope construct, including those acquired in non-VPP callee functions, will be released when control leaves the declared view scope.

In summary, the view scope construct works according to the following rules:

1. A view scope construct acquires the listed views at the beginning of the scope according to the listed order.

2. Within the scope, accesses of other unacquired views are still automatically detected and acquired at their first access.

3. All views acquired within the view scope, including those acquired by the non-VPP callee functions, will be released automatically when control leaves the view scope construct.

With the view scope construct, the above example can be written as below to achieve the programmer's intended scope of view acquisition.

```
VPP void bar(char *shared_str) { /* view shared_str inherited*/
  ......
  shared_str[0] = 'a';
  ......
}

VPP void foo(char *shared_str) {
  .....
  VPP_View(VPP_RW, shared_str) { /* view shared_str acquired */
    bar(shared_str);                        |
    ....                                    |
    str[1] = str[0] + 1;                    |
    ....                                    |
  }                           /* view shared_str no longer needed, and is released at
                                 the end of VPP_View scope instead of the end of foo()
  /* more calculations.... */     in order to maximize the concurrency of view accesses */
  ........
}
```

Figure 3.7: A code snippet illustrating the use of a view scope to specify when a view is acquired and released

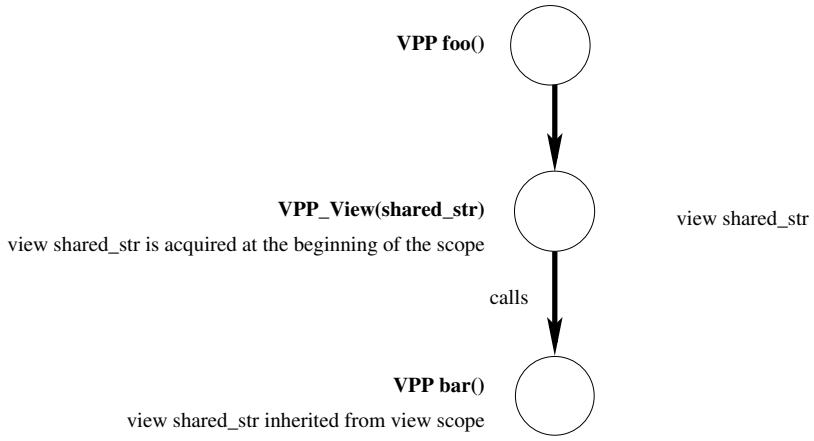

Figure 3.8: Inheritance of views acquired by VPP functions and view scopes

In the example above, a view scope construct is used to acquire the view shared_str in the caller foo(), as intended by the programmer, and the callee bar() inherits the view shared_str from foo() (refer to Figure 3.8 for the inheritance of views).

In addition, view scope constructs allow programmers to define exactly when views are acquired and released, so views can be released as soon as they are not needed. In this way, views are not unnecessarily held until the end of the function and thus do not unnecessarily hinder concurrent accesses of views.

3.1.3 Deadlock Free Mode

The automatic view access detection scheme acquire views automatically as they are first accessed, therefore views may be acquired in different orders by different processes. As a result, the compiler cannot guarantee that views are acquired in a consistent order, deadlock remains possible with the automatic detection scheme, as shown in Figure 3.9.

In the example above, foo and bar are separate views. Process 0 (P0) acquires foo and process 1 (P1) acquires bar first, then P0 tries to acquire bar, but bar is already held by P1, so P0 blocks. At the same time, P1 tries to acquire foo, which is held by P0 (also blocked). As a result, deadlock occurs. This scenario can happen when inexperienced programmers fail to ensure that views are accessed in a consistent order.

To prevent deadlocks, deadlock-free mode is offered in Maotai 3.0. The deadlock-free mode can be specified during initialization of a VOPP session and is effective for the whole VOPP session. A VOPP program starts with serial execution. When par-

```
Process 0                  Process 1
{                          {
  /* foo acquired */
  foo->index++;
                             /* bar acquired */
                             bar->index++;
  /* acquire bar,
     but held by P1 */
  bar->val = foo->index;
                             /* acquire foo,
                                which is held by P0 */
                             /* DEADLOCK */
                             foo->next = bar->val;
}                          }
```

Figure 3.9: A deadlock between two processes accessing views in different orders

allel processing is desired, a VOPP session is started with Vpp_session() which creates multiple processes to execute the same function in parallel. When a VOPP session is finished, the program reverts to serial execution, but can start another VOPP session anytime later.

The following rules are applied in the deadlock-free mode to avoid deadlocks:

- Automatic detection of view accesses is disabled.
- All views must be *explicitly* listed in the VPP_View(access_type, ptr_to_view,...) {...} view scope construct. The system will acquire the listed views at the beginning of the view scope construct in the same system-determined order to prevent deadlocks.
- Access to unacquired views will result in termination of the program and an error message will be printed to notify the user where the violation occurs, as in Maotai 2.0.
- Nesting of view scope constructs and recursive call of VPP functions are forbidden.

3.1.4 The Maotai 3.0 API

This section will describe the Maotai 3.0 API, which can be used on both C and C++ code. The following constructs are related to program flow control:

int Vpp_nprocs the number of processes.

int Vpp_proc_id the ID of the current process.

void Vpp_startup(int nprocs) initializes VOPP with nprocs processes. VOPP API can be used after calling this function.

void Vpp_exit() performs the VOPP cleanup. VOPP API cannot be used after calling this function.

void Vpp_session(void *VPP_func, void *args, bool deadlock_free) starts a parallel session by executing VPP_func with all processes in parallel. VPP_func must be a VPP function. When deadlock_free is set to true, the parallel session is deadlock free and all views must be explicitly acquired by the view scope construct VPP_View.

The following constructs are related to view creation and destruction:

view_type type of view, can be SWV, MWV or PCV

void *Vpp_alloc_view(size_t size, view_type type) allocates a view with the specified size and type. Returns a pointer to the allocated memory upon success, and NULL upon failure.

void *Vpp_alloc_block(void *ptr, size_t size) allocates a block with the specified size to the view pointed by ptr. A call on this function is considered as a view access. Returns a pointer to the allocated block upon success, and NULL upon failure.

void Vpp_free_block(void *ptr) frees the block pointed by ptr. A call on this function is considered as a view access.

void Vpp_free_view(void *ptr) frees the view pointed by ptr.

The following constructs are related to the automatic view acquisition control:

VPP <function signature> declares the function as a VPP function.

VPP_View(access, void *ptr1,...) { **...** } declares a view scope, where all listed views are acquired at the beginning of the scope and released at the end of the scope. Here, pointers pointing to each view to be acquired are listed, together with the access requested, which can be one of read-only (VPP_RO) or read-write (VPP_RW). View scopes can only be declared in VPP functions.

3.1.5 Implementation Details and Overheads

In the automatic detection of view access scheme, virtual memory protection (such as mprotect()) is used to detect view access. Initially a view (consisting of a number of pages) is protected against any access. When it is accessed, a page fault will occur and the page fault handler will be invoked to process the fault.

A view is automatically identified and acquired when its memory is first accessed. In this implementation, each view keeps track of a list of memory regions it owns in the metadata, so when a memory access triggers a page fault, the page fault handler can determine which view owns the faulting address, and subsequently acquires the view.

The view is automatically released when control leaves the scope of view acquisition, as defined by either the control scope of the VPP function or the view scope construct. In this implementation, the private metadata of each process keeps a stack that records views acquired by each VPP function and each view scope in the call stack, therefore when control leaves the scope of VPP function or a view scope, the system can determine which views should be released. In this way, this implementation allows recursive function calls, as each new call on a VPP function simply makes a new entry on the metadata stack.

For the same reason, this implementation also supports object-oriented languages such as C++. In C++, one method may call another via dynamic dispatch, and therefore the identity of the callee method (which may or may not be a VPP method) can only be determined at runtime. However, the above implementation allows tracking of views acquired by each VPP function dynamically, and if the callee method happens to be a non-VPP method, the view acquired in the non-VPP callee method scope will be owned by the caller VPP method. Therefore, the dynamic dispatch mechanism in object-oriented languages such as C++ does not interfere with the Maotai 3.0 automatic view acquisition mechanism in any way.

Like Maotai 2.0, view acquisition is lock-based and is implemented using Pthreads rwlock [72], which is based on futex [34].

Therefore the overheads of Maotai 3.0 include:

- futex lock overhead
- virtual memory protection overhead
- view identification overhead (for calculating the view identity at runtime from the accessed memory address)
- page fault handler overhead (only for automatic detection of view access)

The lock-based view acquisition itself incurs the futex lock overhead and virtual memory protection overhead, the automatic detection of view access incurs the rest of the listed overheads.

To examine these overheads, a microbenchmark is run on a Dell PowerEdge R905 server with four AMD Opteron 8380 quad-core processors running at 800MHz. Overhead is measured for:

- basic pthread_rwlock operations (pthread_rwlock)
- explicit view acquire without runtime protection (still requires view identification mechanism) (no_prot)
- explicit view acquire with runtime protection (manual)
- automatic view access detection (automatic)

To amortize measurement errors, results are collected by first measuring the execution time of 100,000 sequentially-executed identical operations and then calculating the average execution time of one operation. The results are presented in Table 3.1.

Table 3.1: Breakdown of view primitive costs (in μs)

Primitive	pthread_rwlock	no_prot	manual	automatic
a_v()	0.08	0.94	4.15	15.24
a_rv()	0.08	0.92	4.14	15.17
r_v()	0.08	0.08	2.74	N/A
r_rv()	0.08	0.08	2.73	N/A

Note: "a" stands for acquire; "r" stands for release; "v" stands for view and "rv" stands for read-only view. In the pthread_rwlock test, a_v() stands for pthread_rwlock_wrlock(); a_rv() stands for pthread_rwlock_rdlock(); r_v() stands for releasing the wrlock and r_rv() stands for releasing the rdlock.

The above results show that automatic detection mechanism does incur runtime computation overheads for view identification, virtual memory protection and the page fault handler. In automatic detection mode, it takes $15\mu s$ to acquire a view, whereas it only takes $80ns$ to acquire a pthread_rwlock. The automatic detection overhead of $15\mu s$ is small enough for most applications, as shown in performance comparison between Maotai 2.0 (which has no automatic detection) and Maotai 3.0 in Section 3.3. However, this overhead would make applications requiring fine-grain view partition and frequent view accesses unscalable. Also, due to the page-based memory protection mechanism, all view allocations must be page-aligned, which may waste memory space.

In the future, to reduce the runtime overheads and the waste of memory space, compiler support of VOPP will be investigated to allow compile-time tracking of view allocation and access, so that data race free feature in VOPP can be partially implemented at compile time.

3.2 Programmability of Maotai 3.0 and Transactional Memory Models

As mentioned earlier in this chapter, automatic detection of view access improves programmability of Maotai by eliminating programming errors in Maotai 2.0 arising from forgetting to release acquired views, especially when control leaves the scope not at the end of the scope (e.g. by keywords such as break or return) as illustrated in the list search example:

```
typedef struct Node_rec Node;

struct Node_rec {
  Node *next;
  Elem elem;
};

Node *list_search(Elem elem, int vid) {
  Node *list =
     Vpp_acquire_view(vid);

  while (NULL != list) {
    if (elem == list->elem) {
      Vpp_release_view();     /* the list will be held forever
                                 by this process if forgotten
                                 to be released */
      return list;
    }
    list = list->next;
  }
  Vpp_release_view();
  return NULL;
}
```

Figure 3.10: List traversal in Maotai 2.0

However, programming errors arising from forgetting to release views are eliminated by automatic detection of view access in Maotai 3.0 (its code snippet is shown below), since views are automatically acquired and released by the runtime system.

```
typedef struct Node_rec Node;

struct Node_rec {
  Node *next;
  Elem elem;
};

VPP list *list_search(Elem elem, Node *list) {
  while (NULL != list) {
    if (elem == list->elem) { /* view acquired */
       return list;           /* view released */
    }
    list = list->next;
  }
  return NULL;              /* view released */
}
```

Figure 3.11: List traversal in Maotai 3.0

Comparing the above two code snippets, it can be seen that the lines-of-code (LOC) of the Maotai 3.0 version is around 20% fewer than the Maotai 2.0 version.

Moreover, as seen in the code snippets, converting a serial program to Maotai 3.0 requires very little code instrumentation, apart from tagging some functions with the keyword **VPP**. If programmers want to optimize the program performance, they can easily fine-tune the program by using the view scope construct to control how long a view is held.

However, in Transactional Memory (TM) models, programmers must *manually* instrument all code that access shared data to put them into atomic constructs. For example, for the same list traversal example, TM models often have the following code snippet:

```
typedef struct Node_rec Node;

struct Node_rec {
  Node *next;
  Elem elem;
};

/* search a list in shared memory */
list *list_search(Elem elem, Node *list) {
  atomic {
    while (NULL != list) {
      if (elem == list->elem) {
         return list;
      }
      list = list->next;
    }
  }
  return NULL;
}
```

Figure 3.12: List traversal in TM

The above list traversal code (which accesses a shared list) must be included in an atomic construct. Failure to put code that access the shared data into an atomic construct can result in data race bugs.

In contrast, Maotai 3.0 is always *data race free*. Suboptimal programming only compromises performance by holding views longer than necessary, but does not cause data races in Maotai 3.0. Violation of safe view accesses can be detected by the system. Therefore, Maotai 3.0 is safer than TM models.

While TM does not suffer from deadlocks, Maotai 3.0 can avoid deadlocks by using the deadlock free mode.

3.3 Performance Evaluation and Discussion

In this section, the performance of Maotai 3.0 is compared with Maotai 2.0 [64] and the software transactional memory system TL-2 version 0.9.6 [26]. Benchmark applications evaluated in this experiment include Mergesort, Raytrace, Barnes-Hut, Parallel Neural Network (PNN), Binary-tree (BT), Linked-List (LL) and Travelling Salesman Problem (TSP), representing different classes of applications. The experiments are carried out on a Dell PowerEdge R905 server with four AMD Opteron 8380 quad-core processors running at 800MHz and 16GB DDR2 memory. Linux kernel 2.6.31 and the compiler gcc-4.4 are used during benchmarking.

All programs are compiled with the optimization flag "-O2". In each case, speedup is measured against the serial implementation of the benchmark algorithm. The elapsed time calculated in each case includes initialization and finalization costs. However, runtime of functions that are irrelevant to the original application, such as generation of random input sequences and result-verification, are excluded.

The experimental results are illustrated with speedup curves. For each application, the speedup curves of Maotai 2.0, Maotai 3.0 and TL-2 are shown. In the discussion below, N refers to the number of processes.

To ensure fair comparison, the same serial implementation of each benchmark application is used as a baseline for calculating speedups of all parallel programming platforms. Each run is repeated for 10 times and the geometric mean is used.

3.3.1 Maotai 3.0 Outperforms TL-2 in High-Contention Cases TSP, LL and BT

The Travelling-Salesman Problem (TSP) algorithm [1] uses the branch-and-bound depth-limited search approach to identify the shortest path solution. The 33-city case ftv33.atsp from TSPLIB95 [81] is used.

In this algorithm, the priority queue (storing partially-evaluated tours) is the shared object. First, the master process pushes the root tour into the priority queue. Then, in a loop, each process pops a tour. If the tour is small, it will be evaluated serially; otherwise, sub-tours will be created and pushed into the priority queue.

In the TL-2 implementation, the shared priority queue is pushed and popped by transactions. High contention of the priority queue results in the poor speedup of 7.03 in TL-2, as shown in Figure 3.13.

In both Maotai 2.0 and 3.0 implementations, the priority queue is allocated as

a view. The speedup of Maotai 3.0 is 12.92, which is 84% better than the TL-2 implementation, as shown in Figure 3.13. However, Maotai 3.0 is only 3% slower than Maotai 2.0, which has a speedup of 13.28. This small overhead can be attributed to the automatic detection of view accesses.

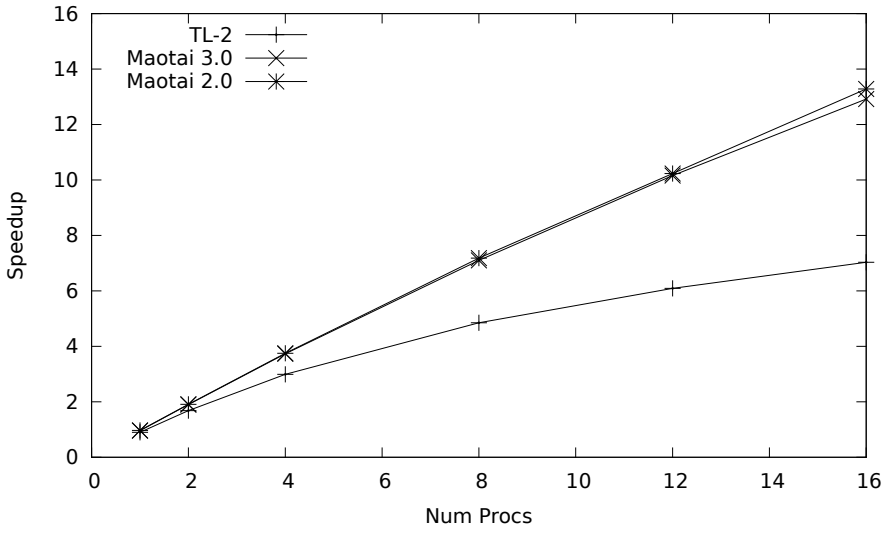

Figure 3.13: Speedup of TSP

Linked-list (LL) inserts nodes in an ascending-ordered singly-linked list, and deletes the nodes afterwards.

In both Maotai implementations, the entire linked-list is allocated as a SWV, while in the TL-2 implementation, naturally each insertion/deletion is put into a transaction. Size of the linked-list is set to 4096.

At $N = 16$, speedup of Maotai 3.0 is 13.59, which is 26% better than TL-2 (10.79) as shown in Figure 3.14. Maotai 3.0 is only 4% slower than Maotai 2.0.

Binary Tree (BT) constructs a binary tree in parallel and uses a task queue for load balancing. When a node is explored, a small amount of dummy work is done, then based on the id of the node, it works out whether the node has a left child and/or right child. The left child id is $curr.id \times 2$ and right child id is $curr.id \times 2 + 1$. Left children are always evaluated immediately and right children are pushed into the task queue for future evaluation. Idle processes pop unexplored nodes from the task queue, until the entire tree is explored. In Maotai 2.0 and 3.0 implementations, the task queue is allocated as a SWV, whereas in the TL-2 implementation, the task queue is accessed by short transactions. The depth of the tree is set to 21.

At $N = 16$, speedup of Maotai 3.0 is 35% better than TL-2 as shown in Figure 3.15. This is another case that lock-based implementations performs better than

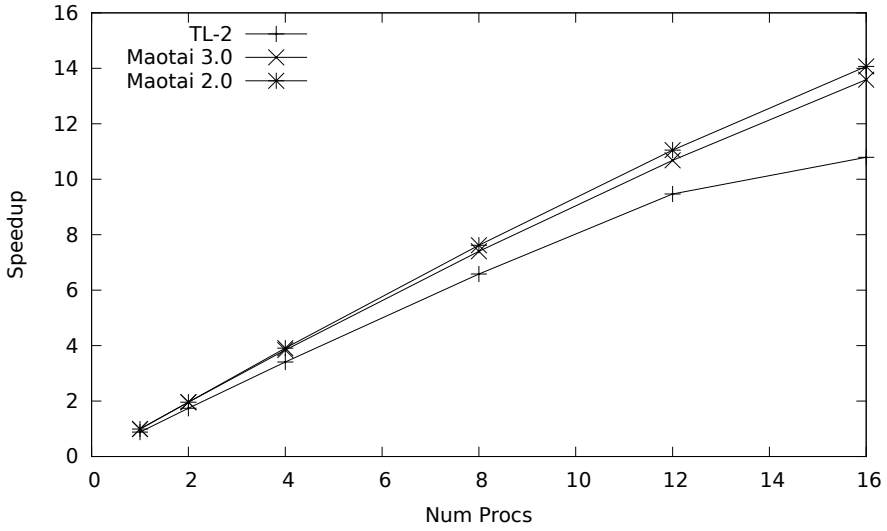

Figure 3.14: Speedup of LL

transactional memory. Again, speedup of Maotai 3.0 is only 3% worst than Maotai 2.0.

The above applications show that TL-2 is inferior to Maotai 3.0 in terms of performance. The slight performance drop of Maotai 3.0 against Maotai 2.0 in the above applications can be attributed to the automatic detection overhead described in Section 3.1.5, as these applications have ten thousands of automatic view acquisitions throughout their executions.

3.3.2 PNN - Multiple Iteration Algorithm Updating a Shared Array

Parallel Neural Network (PNN) [79, 101] trains a back-propagation neural network in parallel using a training data set. In this experiment, the size of neural network is set to $9 \times 40 \times 1$, and the number of epochs is set to 400.

At $N = 16$, speedup of Maotai 3.0 is 50% better than TL-2 as shown in Figure 3.16. In TL-2, there is a shared array with size of 4800, which all processes need to *increment* each element in this array at the end of each iteration. A short transaction would need to increment *each* element. This arrangement results in millions of transactions. Since the overhead of the transaction itself (start and commit) is not negligible, the sheer number of transactions has unnecessarily compromised the performance of TL-2. It is not possible to simply cover the entire array incrementation with a single transaction, because if one element aborts, the entire array operation will be aborted and redone,

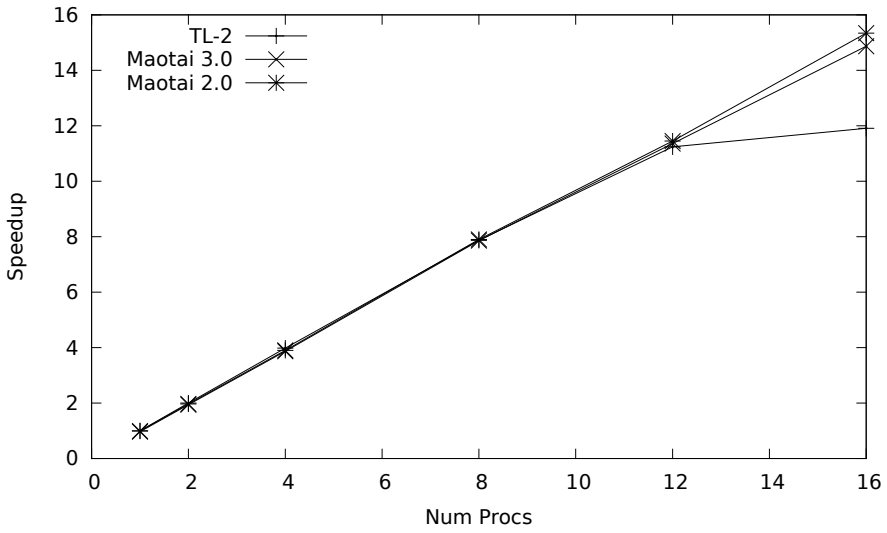

Figure 3.15: Speedup of BT

which would make performance worse.

However in Maotai 2.0 and 3.0, the entire array is allocated as a single-writer view (SWV), which removes unnecessary overheads from the TL-2 implementation and does not complicate the programmability since there is only one lock in the application and thus no deadlock issue in this case. As a result, there are only 25000 view acquires. At $N = 16$, speedup of Maotai 3.0 is only 4% slower than Maotai 2.0.

3.3.3 Barnes-Hut, Raytrace and Mergesort - Low to Moderate Contention Cases Shows Very Little Overhead in Maotai 3.0 Automatic View Access Detection

Barnes-Hut [105] is a multiple-iteration algorithm where in each iteration, the master process constructs an octree for all particles in the model space based on their current location and mass. Then the force acting on each particle is calculated using the octree. Based on the force, acceleration, velocity and position of the particle, the position of the particle at the next iteration is also calculated. In this experiment, the number of bodies is set to 32768 and the number of iterations is set to 160. Due to the complexity of parallelizing the octree construction and its relatively small share of the workload, the octree construction is not parallelized. However, the workload of force calculation on each particle is unpredictable; therefore a work queue is implemented for load balancing. In the TL-2 implementation, accessing the work queue (i.e. incrementing the index) is handled by transactions; therefore Barnes-Hut is a short transaction

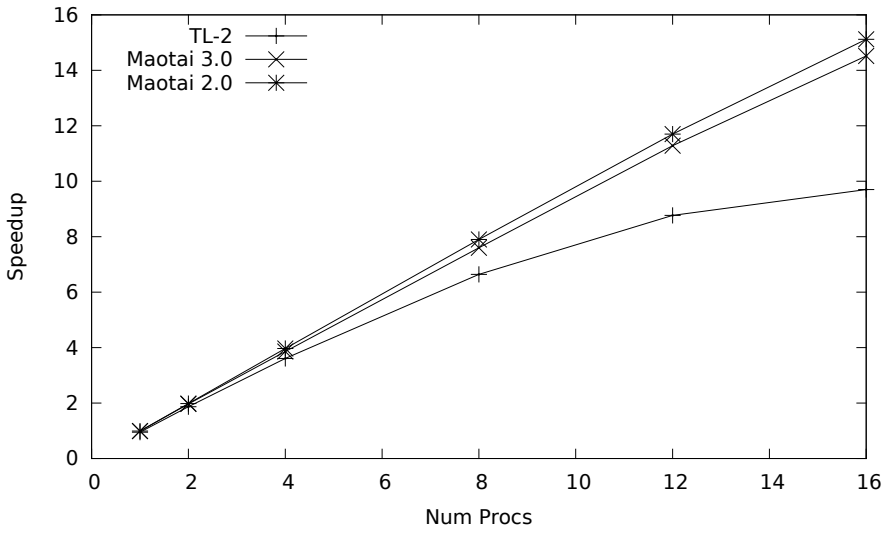

Figure 3.16: Speedup of PNN

case. In Maotai 3.0, the work queue index is implemented as a SWV.

Raytrace [105] is an embarrassingly-parallel case with uneven workload, so a task queue is used for load balancing (a row of pixels is a unit). The input file car.env is used, and anti-aliasing level is set to 400.

The Mergesort algorithm sorts a 1,000,000,000-element array. The Maotai 3.0 implementation comes from [64], and the TL-2 implementation is derived from the Pthreads implementation [64]. This is a barrier-based case, and transactions are not needed in the TL-2 implementation.

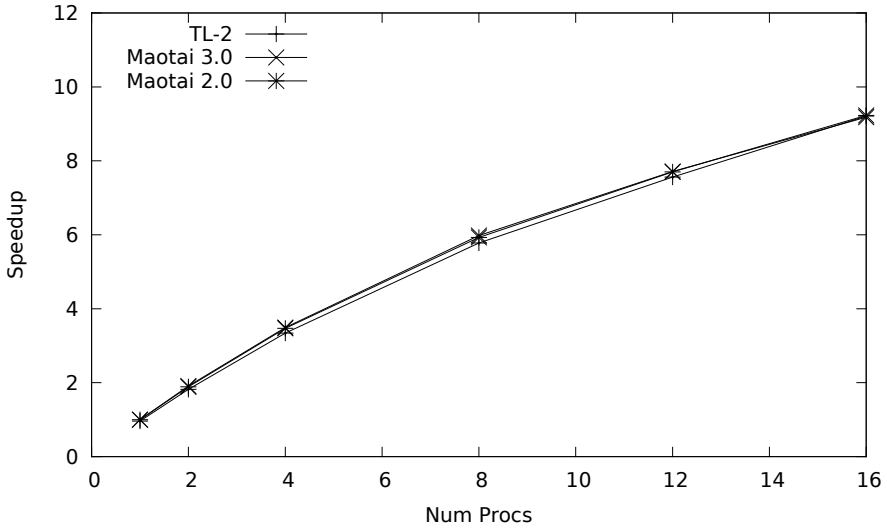

Figure 3.17: Speedup of Barnes-Hut

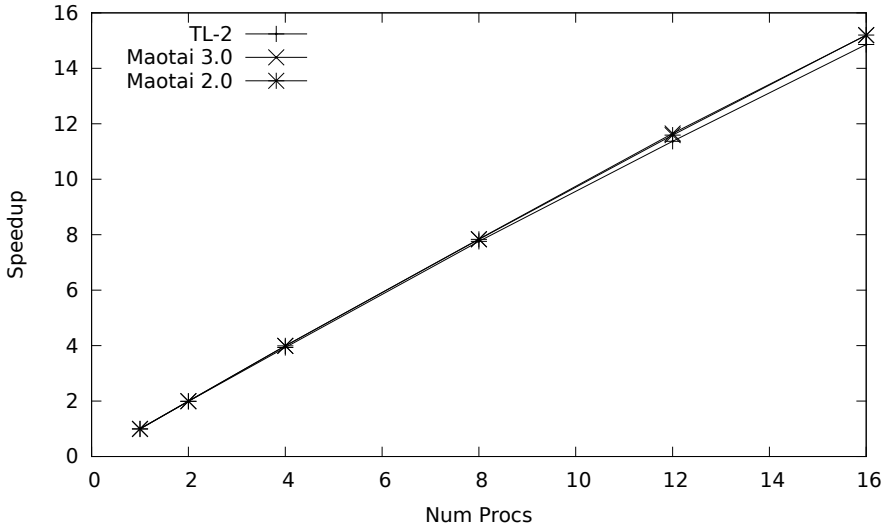

Figure 3.18: Speedup of Raytrace

In Barnes-Hut (Figure 3.17), Raytrace (Figure 3.18) and Mergesort (Figure 3.19), speedup of Maotai 2.0, Maotai 3.0 and TL-2 are nearly identical, except at $n = 16$, TL-2 is 2% worse than Maotai 2.0 and Maotai 3.0 in Raytrace. (Figure 3.19). The same performance of Maotai 3.0 and 2.0 demonstrates the extra overhead of automatic detection of view access in Maotai 3.0 is relatively trivial when the view acquisition is not frequent.

3.4 Concluding Remarks

The performance evaluation between Maotai 3.0 and TL-2 0.9.6 demonstrates that both performance and programmability of Maotai 3.0 surpass TM systems. Comparison between Maotai 3.0 and Maotai 2.0 also demonstrates that in applications evaluated in this chapter, automatic detection overhead is relatively low. Even in high-contention cases such as TSP, LL, BT and PNN, performance penalty is under 4%.

Although the current automatic detection system has small runtime overheads in applications shown in this chapter, other applications with fine-grained, frequent view accesses will not scale easily, as the overheads of runtime detection mechanisms such as mprotect() would become prohibitive. To eliminate the runtime overheads, a higher-level language for VOPP should be investigated as a future work, that offers view management, garbage collection and safe pointer manipulation, so that data race can be partially detected at *compile time*, reducing the runtime overheads in the current

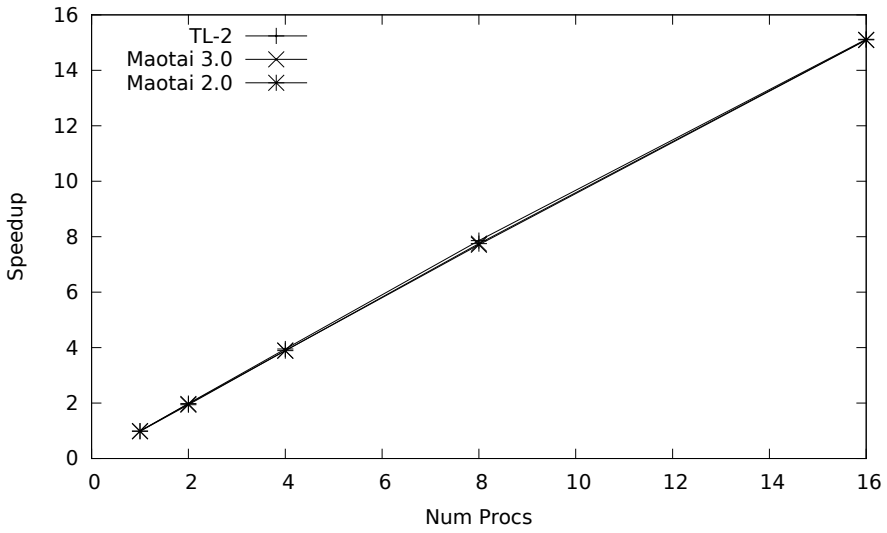

Figure 3.19: Speedup of Mergesort

Maotai 3.0 system, as compiler support and language development of VOPP are outside the scope of this thesis.

Maotai 3.0 is safer than TM systems because the data-centric nature of Maotai 3.0 ensures that it is data race free. Though the automatic detection scheme does not guarantee that there is no deadlock, deadlock can be avoided by using the VOPP sessions with deadlock-free mode offered in Maotai 3.0. To further address the deadlock problem, the next chapter will present the View-Oriented Transactional Memory (VOTM) scheme, where deadlock-prone shared data that can be accessed together atomically are placed in the same view, where each view access is essentially a transaction.

Chapter 4

View-Oriented Transactional Memory

Previous chapters illustrated that, in the VOPP paradigm, an application only needs to acquire the views as needed (automatic or manual), and leaves it for the system to grant access to the view. If locking is used as the view access control mechanism as in Maotai 2.0 and Maotai 3.0, then concurrency will be decreased in coarse-grained views. To improve concurrency, programmers will need to use fine-grain views, which are tedious to program, and can be prone to deadlocks [83].

To address the above concurrency issue, this chapter proposes a novel programming paradigm View-Oriented Transactional Memory (VOTM), that seamlessly integrates the locking mechanism and transactional memory into the same programming model. In VOTM, access to each view is individually controlled by the Restricted Admission Control (RAC) scheme that can dynamically decide the number of processes allowed to acquire the view (known as the admission quota Q) according to the contention of the view. If the contention is low RAC will allow unlimited access to the view to maximize concurrency using TM, but if the contention is high, RAC will decrease the quota Q to decrease contention and improve progress, and in the extreme case, RAC can fall back to the lock-based mode by setting Q to 1. In each view, RAC can flexibly adjust Q of a view according to its contention situation. Therefore a view with high contention will have its access restricted to a low value to decrease contention and improve progress, while RAC will leave access to other views with low contention unrestricted to maximize concurrency. In this way, RAC improves performance of VOTM by maximizing both the progress and concurrency of the view access.

The rest of this chapter is organized as follows: Section 4.1 will present the details

of the VOTM system and the RAC scheme. Section 4.2 will give an overview of TM algorithms. Section 4.3 will describe the details of the VOTM implementation. Section 4.4 will show experimental results and performance evaluation and Section 4.5 concludes this chapter.

4.1 The VOTM Programming Model

VOTM is based on the philosophy of shared memory partitioning. Since different shared data can have different access patterns and contention levels, VOTM allows groups of shared objects that are not required to be accessed atomically to be put into different views, so that concurrency control on each view can be separately optimized using the RAC scheme (refer to Section 4.1.2 for more details).

This optimization cannot be achieved by traditional transactional memory without grouping data objects into views. For example, in VOTM a tree structure with thousands of nodes can be put into one view, and a hash table can be put into another view if they are not required to be accessed atomically at the same time in an application. Suppose the tree in the application has high contention, but the hash table has low contention. The RAC scheme in VOTM would quickly restrict the access to the tree to relieve its contention, without restricting the number of processes accessing the hash table. In this way, the system would continue to allow maximal concurrent access to the hash table, though the access to the highly-contentious tree is restricted. Therefore, by putting the tree and the hashtable in different views, their accesses are separately optimized, which cannot be achieved by traditional transactional memory.

In addition, the RAC scheme also allows seamless integration of the locking mechanism and transactional memory into VOTM. RAC can dynamically control the admission quota Q of a view, or alternatively, Q can be manually specified during view allocation. When Q is greater than 1, a transaction starts when the process is admitted to the view. However, when Q is equal to 1, it is equivalent to the lock mechanism, thus eliminating all transactional overheads. In this way, programmers only need to partition shared data into views according to the access patterns, but leave concurrency control to RAC.

4.1.1 The VOTM Programming Interface

Figure 4.1 shows a C example to explain how to create a view for a linked list in VOTM. In the example, create_view() creates a view vid for the linked list, and

malloc_block() allocates a memory block from the view for the list head.

```
typedef struct Node_rec Node;

struct Node_rec {
  Node *next;
  Elem val;
};

typedef struct List_rec {
  Node *head;
} List;

List *ll_init(int vid) {
  List *result;
  create_view(vid, size, 0);
  result = malloc_block(vid, sizeof(result[0]));
  acquire_view(vid);
  result->head = NULL;
  release_view(vid);
  return result;
}
```

Figure 4.1: Code snippet of list initialization in VOTM

In VOTM, programmers can either let RAC to dynamically control access to the view by specifying a value smaller than 1 to the third argument of create_view(). Alternatively, if the contention of the view is known to the programmer, the admission quota Q of the view can be statically set via the third argument.

A VOTM code snippet for list insertion is shown in Figure 4.2. The parameter *node* of the function points to a node that is a memory block belonging to the view of the linked list. Compared with the sequential version of the code snippet, the only extra code is the view primitives, acquire_view() and release_view(), that demarcate view access.

```
void ll_insert(List *list, Node *node, int vid) {
  Node *curr;
  Node *next;

  acquire_view(vid);

  if (list->head->val >= node->val) {
    /* insert node at head */
    node->next = list->head;
    list->head = node;
  } else {
    /* find the right place */
    curr=list->head;
    while (NULL != (next = curr->next) &&
           next->val < node->val) {
      curr = curr->next;
    }
    /* now insert */
    node->next = next;
    curr->next = node;
  }
  release_view(vid);
}
```

Figure 4.2: Code snippet of list insertion in VOTM

Deadlock can be avoided in VOTM if view acquisitions are not nested. If two views need to be acquired in a nested way, they can often be either put into the same view initially or merged together dynamically. If views are carefully partitioned, nested view acquisitions are rarely needed in real applications. When nested view acquisitions are needed, they can often be resolved in VOTM by merging the involved views into one view.

Below is a summary of the VOTM API:

int create_view(int vid, size_t size, int q)

Creates a view with ID vid and size size, and returns the view ID. If vid is set to

a negative value, the view ID will be allocated by the system. q is the maximum number of processes admitted to this view. If q is less than 1, admission quota of this view will be dynamically managed by RAC.

void *malloc_block(int vid, size_t size)

Allocates a memory block with the specified size for the view vid. Returns the base address of the allocated block.

void free_block(int vid, void *ptr)

Frees the memory block pointed by ptr from the view vid.

void destroy_view(int vid)

Destroys the view vid.

void sbrk_view(int vid, size_t size)

Increments the memory space of the view vid by size.

void acquire_view(int vid)

Acquires read-write access to the view vid.

void acquire_Rview(int vid)

Acquires read-only access to the view vid.

void release_view(int vid)

Releases access to the view vid.

4.1.2 Restricted Admission Control (RAC) Scheme

The RAC scheme is implemented for every view. Each view consists of memory blocks that may store an entire linked list, tree or graph. Each view has an admission quota Q that restricts the maximum number of processes accessing the view concurrently. Before a view is accessed, the primitive acquire_view is used. If Q equals 1, acquire_view is equivalent to a lock acquisition. In this case, the lock mechanism is used instead of the transaction mechanism to avoid transactional overheads. If Q is greater than 1, acquire_view will either start a new transaction or wait according to the following RAC scheme.

Suppose a view has an admission quota Q and the current number of processes concurrently accessing the view is P, when the view is acquired through acquire_view, RAC follows the steps below:

- Compare P with Q. If P is smaller than Q, increase P by 1, start a new transaction, and return with success.
- If P equals Q, the calling process is blocked until P becomes smaller than Q.

When the view is released through release_view, RAC executes the following steps:

- Try to commit the transaction. If the commit fails, abort and roll back the transaction, decrease P by 1, and reacquire the view as shown above.
- If the commit succeeds, decrease P by 1, and then return with success.

Furthermore, RAC can dynamically adjust the admission quota Q in the following way according to the contention situation.

The admission quota Q of each view is initialized as the maximum number of processes N if it is not set statically at view creation time. RAC regularly checks the contention situation of the view. The current RAC algorithm estimates the contention situation by the number of aborts as well as the number of successfully committed transactions that are related to the view. If the number of aborts is high, the contention is usually high. However, if the number of successful transactions is larger than the number of aborts, then the contention may be not high enough to affect the overall progress of the computation, even though the number of aborts may be high in such a situation. Therefore, the ratio between the number of aborts and the number of successful transactions ($\frac{aborts}{successful\_tx}$) is used to reflect the severity of the contention situation.

If this abort/success ratio is larger than MAX (currently set to 8.0), the view is considered as highly contentious. When this happens, RAC will relieve the contention of the view by halving the admission quota Q of the view. Then, the number of aborts and the number of successful transactions will be reset in the view. This process can be repeated periodically until Q reaches 1, in which case the concurrency control is switched to the lock-based approach. Then the transaction mechanism is no longer used to access the view, Q will stay at 1 and the abort/success ratio for the view concerned is no longer checked.

Conversely, when Q is greater than 1 and the abort/success ratio is smaller than MIN (currently set to $\frac{1}{8}$), the view is considered as having low contention. Then, RAC will increase concurrency by doubling Q. When Q is changed, the numbers of aborts and successful transactions of the view will be reset. This process will repeat periodically until Q reaches N.

To eliminate cache flushing overheads on incrementing the shared counters in the RAC metadata of the view, when it is clear that access restriction to the view is unnecessary because of following low contention condition:

- 20000 transactions are executed since Q is set to N, and
- the abort/success ratio $< MIN$

The RAC mechanism will be disabled.

After the RAC mechanism of the view is disabled, access to the view will no longer be restricted until the contention situation of the view changes, which is detected when a process encounters a large number of consecutive aborts.

The choices of *MAX* and *MIN* are currently empirical. Different TM algorithms may favor different values. For example, the encounter-time locking TM algorithm used in TinySTM aborts potentially-conflicting transactions early to reduce wasted computation. Under the same contention situation, this would result in higher abort/success ratio than other TM algorithms such as commit-time locking used in TL-2. Therefore, the same genuine high contention case will have higher abort/success ratio for TinySTM than for TL-2. Optimal *MAX* and *MIN* settings are dependent on the underlying transactional memory system. Automatic adjustment of these values is an interesting issue for further research.

Frequent check of the abort/success ratio is costly since a spinlock is used for multiple processes to access the numbers, which would significantly increase the overhead of RAC. Therefore, the periodic check is only triggered under the condition when the sum of aborts and successful transactions is a multiple of 5000. It is observed that, checking under this condition is frequent enough in most cases, because if the contention is high, the number of aborts will rise quickly to trigger the check.

4.1.3 Origin of Performance Gain in VOTM

The origin of performance gain in VOTM is very different from TM systems that use either in-transaction conflict resolution algorithms and/or transaction scheduling algorithms. In-transaction conflict resolution algorithms [26, 40, 89] only detect conflicts and control contentions during the execution of transactions and on their own still allow any processes to freely enter transactions. Transaction scheduling algorithms [2, 29, 106] prevent conflicts by serializing transactions or limiting the number of concurrent transactions. These algorithms treat the entire TM with the same

scheduling decision. However, it is not reasonable to restrict access to a low-contention shared object due to another shared object that has high contention, a situation that could happen on these algorithms. In VOTM, transactional memory is divided into views where shared objects that will be accessed together in a transaction are grouped into the same view. In this way, restricting access to a view with high contention does not affect access to a view with low contention, which enables more concurrency. In VOTM, RAC is used as the transactional scheduling algorithm for each view, but any in-transaction conflict resolution algorithms can be applied in each view. Regardless of the choice of the underlying in-transaction conflict resolution algorithm, there are always cases where the number of aborts becomes very high. Here RAC can reduce contention by limiting the admission of processes to the view, and improve progress.

Section 4.3 will discuss the details of the VOTM implementation; and Section 4.4 will show that VOTM with RAC can reduce the number of aborts, and therefore reduce contention and increase throughput, by controlling the admission to each view.

4.2 Overview of Transactional Memory Algorithms

As mentioned in the Introduction chapter, the TM model improves concurrency by allowing multiple processes to enter an atomic section. A transaction begins when a process enters an atomic section, and the transaction ends when control leaves the atomic section. If there are no conflicts between transactions, then concurrency is achieved. If there is a conflict between transactions, the TM system will resolve the conflict by aborting one or more transactions. Aborted transactions will be rolled back, and restarted. Therefore transactions in TM is like transactions in a database system: A transaction must either complete successfully, or if aborted, must undo all changes it makes and revert to the state at the beginning of the transaction.

There are two ways TM mechanisms can handle read/write of variables during transactions – undo-log and redo-log. In both ways, each transaction keeps a read-set and a write-set to record its read and write activities respectively.

In the undo-log system, the transaction will directly write to the actual memory location, but also add an entry to the write-set in its undo-log, with the original value of that memory location. If there is no conflict, then nothing else needs to be done when the transaction commits. However when the transaction aborts, it needs to play the undo-log and undo all the changes it makes. As mentioned in [33] and [99], the abort mechanism for undo-log systems can be complicated, as the rollback mechanism must

handle the situation where multiple transactions have written to the same location, and the correct version of the value must be restored to the location. This also makes the abort mechanism expensive in undo-log systems, especially when the contention of the application is high.

In the redo-log system, all reads and writes in a transaction are tentatively recorded in the redo-log. When the transaction commits successfully, the redo-log is replayed to write the changes to the actual memory locations. Conversely, when the transaction aborts, there is no need to undo changes to the shared memory, as nothing has been written to the actual locations in the shared memory.

In TM systems, generally there are two approaches to detect the conflict – encounter-time locking (ETL) and commit-time locking (CTL).

In ETL, a transaction (T1) will lock a location `x` upon its first write. Other transactions attempting to access `x` will detect the conflict, and self-abort. ETL aims at detecting potential conflicts early, to reduce the time wasted by transactions that will be ultimately aborted, especially in high-contention situations. However the aborted transaction (T2) can restart while the originally-winning transaction (T1) in the previous conflict is still in progress. If the restarted T2 then accesses another location `y` first and `y` is later accessed by T1, then T2 can abort T1. Therefore as illustrated in Figure 4.3, it is possible for a pair of transactions to abort each other in a vicious cycle, and results in a livelock. However, RAC can effectively prevent livelocks, because in high contention situations, RAC will quickly restrict the admission quota Q to a low value, such as 1, to ensure progress. ETL can use either undo-log or redo-log. Undo-log can be slightly more efficient in low contention cases, as the log needs not to be replayed when the transaction successfully commits. However if a transaction aborts, the undo-log needs to be replayed to undo the changes. This mechanism is complicated and has a high overhead. Therefore in high contention cases, redo-log should be used. Later in this section, we will go into the details of TinySTM [33], which is an ETL TM algorithm.

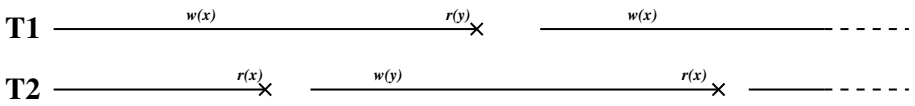

Figure 4.3: Transactions T1 and T2 livelock in ETL

However in CTL, a transaction will lock the locations it writes to at *commit time*. As in most TM algorithms, transactions in CTL algorithms also keep a read-set and a

write-set to record its speculative reads and writes respectively. Then during commit, all locations in the write-set are locked, and the read-set is validated. If both are successful, then the new values in the write-set will be written to the actual location, and the locks are released. Since CTL writes the changes to the actual shared memory locations after the transaction successfully commits, redo-log must be used.

When a transaction detects a conflict, aborts and restarts, the other transaction in the conflict would have already committed. Therefore it is impossible for a restarting transaction to abort the originally-winning transaction. As a result, CTL algorithms are livelock-free. However, since the written locations are locked at commit time, the time between the occurrence and detection of the conflict can be potentially long, as illustrated in Figure 4.4. Therefore considerable time could be wasted in ultimately-doomed transactions. However in advanced CTL algorithms such as NOrec [24], optimization techniques such as read-set validations during read operations can make conflict detection earlier, and therefore reduce the time wasted. Details of the NOrec algorithm will be discussed later in this section.

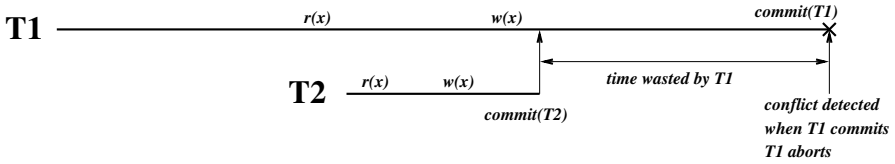

Figure 4.4: **Time wasted by ultimately-doomed transactions in CTL** In this case, both transactions T1 and T2 access x. T2 commits first, and its changes to x is published upon its commit. However T1 only finds out the conflict when it attempts to commit. As a result, the time wasted by the aborted transaction T1 is substantial (between the conflict at the time T2 commits and when T1 discovers the conflict at its commit time)

4.2.1 The TinySTM Algorithm

As mentioned above, the TinySTM algorithm [33] is an encounter-time locking algorithm based on the Lazy Snapshot Algorithm [82]. TinySTM uses a global clock (GC) to maintain the serial order of transactions and it uses a word-granularity lock array in the global metadata to track lock ownership. Here, a location in the shared memory is mapped to a lock in the global lock array through a hash function. Each lock is word-sized, and its least significant bit indicates whether the lock is acquired

by a transaction. When a lock not acquired, the remaining bits represents the version number of the lock; otherwise when the lock is acquired, the remaining bits point to the write-set entry of the owner transaction, thus allow lookup of the updated value of the location, and whether the transaction owns this lock, in constant time, without the need of traversing the write-set. Here are the global metadata used in TinySTM:

```
typedef struct versioned_lock_t {
  bit lock;
  uint63_t version;  // if locked, represents a pointer to owner write-set
} versioned_lock_t;  // total size of lock is 64 bits (one word)

versioned_lock_t locks[];

uint64_t GC;
```

Figure 4.5: TinySTM metadata

In addition, each transaction privately keeps a read-set and a write-set, like most TM algorithms. The redo-log configuration is chosen in this experiment because it has a lower overhead in high-contention situations.

When a transaction begins, or restarts, it first samples the global clock and takes this value as the version number of the transaction.

```
void TxStart(TxDesc tx) {
  tx.start_time = GC;
}
```

Figure 4.6: TinySTM TxStart() pseudocode

As mentioned above, TinySTM [33] is encounter-time locking, which means that the location is locked by a transaction upon its first write. When a transaction tries to write to a location, it checks whether the location is locked. If it is locked and if the transaction owns the lock, then the transaction just updates the value in the write-set, otherwise if the location is owned by other transactions, the transaction will abort. If the location is not locked, the transaction will acquire the lock of the location, and add an entry to its write-set. The pseudocode for the write operation is as follows:

```
void TxWrite(TxDesc tx, off_t addr, word val) {
  versioned_lock_t l;

  tx.writeSet.insert(addr, val);

  while (true) {
    l = locks[addr];

    if (l is not locked and l.version <= tx.start_time) {
      if (atomic_acquire_lock(locks[addr]) fails) {
        TxAbort(tx);
      }
      return;
    } else if (tx already owns l) {
      return;
    } else if (l locked by other transactions)
      TxAbort(tx);
    } else {
      // l.version > tx.start_time, need to extend snapshot
      newts = GC;
      validate(tx);
      tx.start_time.newts;
    }
  }
}
```

Figure 4.7: TinySTM TxWrite() pseudocode

When reading a location, the transaction must verify that the lock is not owned by other transactions. If the transaction already owns the lock, then it simply reads the value from the write-set entry. Once a value has been read, the transaction will check whether it can be used to construct a consistent snapshot (as in the lazy-snapshot algorithm [82]). The validity range of the snapshot consists of a range of versions from the beginning up to the current start_time of the transaction. If the lock version is outside the current validity range, then the transaction will try to "extend" its

consistent snapshot (in its read-set). Here the read-set will be validated by checking whether the value of each entry in the read-set is modified. If any entries in the read-set are updated by other transactions, the validation will fail and the transaction will abort. If the validation is successful, then the validity range of the transaction will be extended to cover the version number of the lock (by setting tx.start_time to the current GC). Therefore read-only transactions are efficient, as the transaction incrementally construct a snapshot that is valid at all time. As a result, no validations are necessary at commit-time. The pseudocode for the read operation is as follows:

```
word TxRead(TxDesc tx, off_t addr) {
  word result;
  versioned_lock_t l;

  if (addr in tx.writeSet) {
    return tx.writeSet.getValue(addr);
  }

  while (true) {
    l = locks[addr];
    result = *addr;
    if (l is locked by other transactions) {
      TxAbort(tx);
    } else {
      // unlocked
      if (l.version > tx.start_time) {
        //need to extend snapshot
        if (false == validate(tx)) TxAbort(tx);
      } else {
        tx.readSet.insert(addr);
        return result;
      }
    }
  }
}
```

Figure 4.8: TinySTM TxRead() pseudocode

When a transaction commits, it first validates its read-set. The transaction will abort if validation fails, otherwise the transaction will write the changes in its write-set to the actual memory location, increments the global clock and acquires the global clock as its unique end_time through an add-and-fetch atomic operation. Finally, the transaction atomically set the version of all locks it holds to end_time and release the locks. Pseudocodes for the commit, abort and validation operations are shown in Figure 4.9, Figure 4.10 and Figure 4.11 respectively.

```
void TxCommit(TxDesc tx) {
  if (tx.writeSet is empty) {
    tx.readSet.reset();
    return;
  }

  validate(tx);
  tx.writeLog.run();  // run the redo log
  tx.end_time = atomic_add_and_fetch(GC, 1); // increment GC
  foreach (lock in tx.writeLog) {
    atomically set version to tx.end_time and release lock
  }
  tx.writeSet.reset();
  tx.readSet.reset();
}
```

Figure 4.9: TinySTM TxCommit() pseudocode

```
void TxAbort (TxDesc tx) {
  tx.writeSet.reset();
  tx.readSet.reset();
}
```

Figure 4.10: TinySTM TxAbort() pseudocode

```
void validate(TxDesc tx) {
  foreach (entry in tx.readSet) {
    if (entry.lock.version > tx.start_time ||
        entry.lock is locked by other transactions)
      TxAbort(tx);
  }
}
```

Figure 4.11: Pseudocode of the read-set validation algorithm of TinySTM

4.2.2 NOrec

NOrec [24] is a word-granularity CTL TM algorithm. To reduce the overhead of the TM mechanism, NOrec does not have a global lock array. Instead, NOrec uses *value-based validation* to detect conflict. Therefore an entry of the read-set and write-set must record both the value and the address of the memory location. The only global metadata is the global versioned lock (GC), which has a lock-bit, and the rest of the bits represents the version number. Like all CTL algorithms, redo-log is used in NOrec, because all changes are globally-visible at commit time.

```
typedef struct versioned_lock_t {
  bit lock;
  uint63_t version;
} versioned_lock_t;  // total size of lock is 64 bits (one word)

versioned_lock_t GC;
```

Figure 4.12: NOrec metadata

```
void TxBegin(TxDesc tx) {
  tx->start_time = GC.version;
}
```

Figure 4.13: NOrec TxBegin() pseudocode

Write operation is cheap in NOrec, as the only operation is inserting the write operation into the write-set. No validations, or sampling of the global versioned lock, are required.

```
void TxWrite(TxDesc tx, off_t addr, word val) {
  tx.writeSet.insert(addr, val);
}
```

Figure 4.14: NOrec TxWrite() pseudocode

As mentioned earlier in this section, the time wasted in ultimated-doomed transactions can be considerable in CTL algorithms due to the commit-time locking policy. However NOrec reduces this wasted time by validating the read-set when the read operation TxRead() finds out that the start_time of the transaction is smaller than GC. So a potential conflict will be detected at the next read, thus greatly reducing the time wasted by ultimately doomed transactions. Unlike TinySTM, NOrec does not have a lock array that stores the versioning information of each location, this necessitates the re-validation of the read-set whenever the GC is incremented since the last read, rather than only when reading a value that is updated since the last read-set validation. Therefore in NOrec, transactions can have very frequent read-set validations, causing high contention on the global versioned lock, as read-set validation samples the global versioned lock. This contention can affect scalability of NOrec. Here is the pseudocode of the read operation:

```
word TxRead(TxDesc tx, off_t addr) {
  word result;

  if (addr in tx.writeSet) {
    return tx.writeSet.getValue(addr);
  }

  result = *addr;

  while (tx.start_time < global_lock) {
    tx.start_time = validate(tx);
    if (validate() failed) {
      TxAbort(tx);
    }
    result = *addr;
  }

  tx.readSet.insert(result, addr);
  return result;
}
```

Figure 4.15: NOrec TxRead() pseudocode

Like TinySTM, a read-only transaction does not need to validate the read-set when committing, because it already has a valid read-set. However an updating transaction needs to acquire the global versioned lock GC. Then a validation is required if GC is larger than the start_time of the transaction. Once this is done, GC will be atomically incremented and released. Finally, changes in the write-set will be written to the actual memory location, and both the read-set and write-set of the transaction will be reset. Here are the pseudocode for the commit, abort and validation operations in NOrec:

```
void TxCommit(TxDesc tx) {
  if (tx.writeSet is empty) {
    tx.readSet.reset();
    return;
  }

  while (GC > tx.start_time or acquire(GC) fails) {
    if (validation(tx) fails) {
      TxAbort(tx);
    }
  }

  tx.writeLog.run();  // run the redo log

  atomically increment GC and release GC;

  tx.writeSet.reset();
  tx.readSet.reset();
}
```

Figure 4.16: NOrec TxCommit() pseudocode

```
void TxAbort (TxDesc tx) {
  tx.writeSet.reset();
  tx.readSet.reset();
}
```

Figure 4.17: NOrec TxAbort() pseudocode

```
uint64_t validate(TxDesc tx) {
  versioned_lock_t curr_version;

  while (true) {
    curr_version = GC;

    if (curr_version.locked) continue;

    foreach (entry in tx.readSet) {
      if (entry.value != *(entry.addr)) {
        return failed;
      }
    }

    if (curr_version == GC) return curr_version.version;
  }
}
```

Figure 4.18: Pseudocode of the read-set validation algorithm of NOrec

4.3 Implementation

The VOTM system implementation is based on the software transactional memory system TinySTM [33], a word-granularity timestamp-based TM system based on the C language. In this VOTM implementation, TinySTM is configured as a redo-log-based TM system with encounter-time locking.

In VOTM, access to each view can be controlled independently so that a view with high contention will not affect concurrency of other views that may have low contention. Experimental results in the next section demonstrate that using multiple views in this way improves performance.

Similar to TinySTM and many other software TM systems, in the current implementation, the memory accesses in VOTM have to be explicitly labelled with primitives such as Tx_read and Tx_write. However, these primitives can be removed with compiler support or hardware TM systems [25].

Since encounter-time locking is used in VOTM, the transaction first writing to a

location commonly accessed by other transactions wins (as opposed to TL-2, which uses commit-time locking instead). However, no matter what conflict detection policy is used, short transactions can easily abort a long transaction and computation done by the long transaction will be wasted. This situation will be further explained in Section 4.4.1.

4.4 Performance Evaluation

This experiment compares the performance of VOTM with the software transactional memory system TinySTM version 1.0.0 [33] and the lock-based approach which uses Pthreads mutexes. These systems will be evaluated in benchmark applications including Bayes, Intruder, Genome, Labyrinth, Vacation and SSCA2 from the STAMP transactional memory benchmark suite version 0.9.10 [18], which represents real-life applications with long critical sections that abolish concurrency of lock-based systems such as the previous Maotai implementations. In addition, Travelling Salesman Problem (TSP) from the SPLASH-2 benchmark suite [105], which has many memory-intensive short critical sections is also evaluated. These benchmarks represent different classes of applications. The experiments are carried out on a Dell PowerEdge R905 server with four AMD Opteron 8380 quad-core processors running with 800MHz and 16GB DDR2 memory. Linux kernel 2.6.32 and the compiler gcc-4.4 are used during benchmarking.

All programs are compiled with the optimization flag -O2 because it is more stable than -O3. The runtime calculated in each case includes initialization and finalization costs. However, the runtime of functions that are irrelevant to the original application, such as generation of random input sequences and result-verification, is excluded.

Intruder is a memory-intensive TM application. In this application, a dictionary is used to store partial results, and jobs are handled by a centralized task queue. In the VOTM version, the task queue and the dictionary are allocated in separate views. Default parameters "-a10 -l128 -n262144 -s1" are used for the application.

In Bayes, the shared net is accessed by long transactions with high contention, whereas access to the task queue is short and does not take computation time. Since the net is never accessed together atomically with the task queue, they are allocated in separate views. Default parameters "-v32 -r4096 -n10 -p40 -i2 -e8 -s1" are used.

Genome is a gene-sequence alignment algorithm which has multiple shared hash tables with low contention and two shared arrays with higher contention. Shared data

structures include an input hash table as well as an array of hash tables containing intermediate fragments plus two arrays tracking prefixes and suffixes. In the VOTM version, a view is used to host all shared data structures. In the pure lock-based version, the hashtable, prefix array, and suffix array are each protected by a lock. The VOTM version could protect each bucket in each hash table with a lock, but this would be too tedious and change the original algorithm drastically. Default parameters "-g16384 -s64 -n16777216" are used.

Both Labyrinth and Vacation have long transactions with little contention. Labyrinth finds the shortest path between pairs of starting and ending points in a maze, which is implemented as a shared grid. The shared grid is accessed with long transactions with low contention. The input file "random-x512-y512-z7-n512.txt" is used. The shared grid is allocated as a view in the VOTM version. Since access to the grid cannot be divided without a complete rewrite of the algorithm, the pure lock-based version simply uses lock to protect access to the grid.

Vacation simulates the operation of a travel agency manager. Each transaction consists a set of operations including adding/removing reservations. The transaction succeeds only if all operations succeed; otherwise, it will abort and restart. Transactions are long and with a moderately high memory accesses, but with low contention. Since all shared data can be accessed together atomically, they must be put into a single view in the VOTM version. Also for the same reason, the critical section cannot be broken down in the pure lock-based version; therefore, a single lock is used to protect the critical section. Default parameters "-n4 -q60 -u90 -r1048576 -t4194304" are used.

SSCA2 has high number of very short transactions with low contention; therefore, it serves as a test case testing overheads for starting and ending transactions. SSCA2 operates on a large, directed and weighted multigraph. Kernel 1 in this application is used in STAMP, which constructs the graph data structure in parallel using adjacency arrays and auxiliary arrays. Similar to Labyrinth, the graph in SSCA2 is put in a single view in the VOTM version. In SSCA2, operation on each graph node is done by a very short transaction that takes little computation time. Contention is very low in SSCA2 because the large number of graph node means concurrent updates on the same adjacency list is rare. However there are many transactions in this application. Default parameters "-s20 -i1.0 -u1.0 -l3 -p3" are used.

The Travelling-Salesman Problem (TSP) algorithm have short transactions with very high contention. Transactions in this algorithm are memory intensive but does not have computational work; therefore, only a small portion of execution time is spent in

transactions. The algorithm uses the branch-and-bound depth-limited search approach. The 33-city case ftv33.atsp from TSPLIB95 [81] is used. In this algorithm, the priority queue (storing partially-evaluated tours) is the shared object, and is allocated in a view. Since access to this view is short but contentious, a VOTM version with the Q manually set to 1 is also implemented to test the benefit of manual Q optimization against the VOTM version with dynamic Q adjusted by RAC.

Kmeans and Yada from the STAMP benchmark are excluded from this experiment because, in Kmeans, the incrementation of each element in the shared array is atomic, so atomic operations should be used instead of TM. The Yada application crashes frequently whenever it runs with multiple processes, and when it does not crash, parallelization shows little performance gain, if any, because all computation time is spent in transactions with extremely high contention.

Table 4.1: Application runtime (s) at $N = 16$

Application	VOTM	TinySTM	Lock-based
TSP $Q = 1$	52.23	194.73	52.23
Intruder	43.05	127.70	100.62
Bayes	11.15	19.51	30.72
Genome	4.93	5.91	37.48
Labyrinth	35.60	35.08	331.28
Vacation	14.84	14.1	61.88
SSCA2	8.80	8.77	56.28

Table 4.2: Number of transactions and aborts at $N = 16$

Application	#transactions	VOTM	TinySTM
TSP $Q = 1$	3,925,092	0	4,150,852,440
Intruder	23,428,141	10,986,905	1,238,254,062
Bayes	1,751	4,591	522,972
Genome	2,472,907	83,273	64,595,381
Labyrinth	1056	196	202
Vacation	4,194,304	1,443	1,059
SSCA2	22,362,292	62	64

From Table 4.1, it can be seen that VOTM has superior performance over TinySTM in high contention applications. In TSP and Intruder, VOTM is 270% and 200% faster than TinySTM respectively. In Bayes and Genome, VOTM is also 75% and 20% faster than TinySTM, respectively.

In the above applications, RAC successfully prevents speedup degradation by restricting the number of processes admitted to a view. In TSP, RAC eliminates aborts in VOTM altogether, and for the rest of the applications shown in Table 4.2, RAC cuts the number of aborts in VOTM by up to 100 times. The reasons RAC improves performance of VOTM will be discussed in detail in Section 4.4.1.

In low contention applications, such as Labyrinth and Vacation with long transactions and SSCA2 with a high number of very short transactions, the runtime of VOTM and TinySTM are similar.

At low contention, RAC will allow admission of all processes in order to maximize concurrency, and will thus behave like traditional TM. The runtimes of VOTM and TinySTM for these applications shown in Table 4.1 are similar, suggesting that VOTM has little extra overhead.

However, the pure lock-based version in general has poor performance because the applications Intruder, Bayes, Genome, Labyrinth and Vacation have coarse-grained critical sections occupying the majority of the execution time, thus eliminating concurrency. To make them work with fine-grained locking, a total design of the parallel algorithm is necessary, which requires expert knowledge in parallel programming.

Although in SSCA2, the time spent in critical sections is very short, the sheer number of acquires of the same lock (22 million acquires) in the pure lock-based version makes the lock a hot-spot. The resultant CPU cache coherence overhead makes the pure lock-based version unscalable.

Table 4.3: Performance of TSP at $N = 16$

	VOTM (dynamic Q)	VOTM ($Q = 1$)	TinySTM	Lock-based
time(s)	71.54	52.23	194.73	52.23
#aborts	15,658,595	0	4,150,852,440	0

The application TSP has a shared priority queue with high contention. Therefore, TinySTM is not scalable. Since access to the priority queue is known to be very short

but memory-intensive, VOTM benefits from manually setting Q to 1 to avoid transactional overheads. As shown in Table 4.3, VOTM with $Q = 1$ has a 27% performance gain over VOTM with dynamic Q. By manually setting $Q = 1$, the performance of VOTM now matches the lock-based version, because locking is more effective to protect highly contentious shared data such as this priority queue.

The above results show the VOTM system has the performance advantage over TM in high contention situations and allows the performance benefit of fine-grained locking through the optimization of admission quota.

4.4.1 How RAC Improves Performance

RAC improves performance in two ways. The first way is through removing the transactional overhead by switching to lock-based mechanism when the admission quota Q equals 1.

To investigate this transactional overhead, microbenchmarks of transactions with 0, 1, 10, 100, 1000, 10000 and 100000 read and write operations are performed. Each read/write operation is performed in a *separate* location to examine the real cost of read- and write-set maintenance. To amortize measurement errors, we have collected the results by first measuring the execution time of 100,000 sequentially-executed identical transactions and then calculating the average execution time of one transaction. The results are presented in Table 4.4.

Table 4.4: Overhead of transactions with different sizes

no. of r/w	0	1	10	100	1000	10000	100000
time(μs)	0.21	0.35	1.30	10.65	109.47	1216.22	14425.03

From Table 4.4, it can be seen that the cost of starting and ending a transaction itself is not trivial ($0.21\mu s$ per empty transaction), and for a long transaction with 100,000 reads and 100,000 writes, the overhead can be up to $14ms$ per transaction. Therefore, transactions are expensive.

To avoid the expenses in transactional memory, RAC drops the transactional memory mechanism when the admission quota of a view becomes 1.

The second way that RAC improves performance is through reducing the number of aborts by decreasing Q. As the application is run, the RAC algorithm adjusts Q according to the abort/success ratio. Q will eventually settle at the value where

access speedup saturates (i.e. the number of processes where maximum concurrency is reached).

After the speedup of accessing the view is saturated, RAC prevents speedup degradation by restricting admission to the view to Q processes to prevent extra processes from increasing contention and conflicts. This is very important in real-life situations, as it can be difficult to determine in advance the number of processes needed to saturate access speedup if the access patterns are dynamic and bursty.

In order to demonstrate the effect of RAC in terms of restricted admission, we use Bayes in this part of the experiment. Here the number of running processes (N) is fixed to 16 and the admission quota (Q) is fixed to 1, 2, 4, 8 and 16 respectively. The $Q = 16$ case is equivalent to no restriction of admissions, but the $Q = 1$ case still uses transactions (tx) in order to show only the effect of admission control. However, result of a $Q = 1$ case run without transactions (no tx) is also shown to demonstrate transactional overheads.

Table 4.5: Runtime and number of aborts of Bayes at different Q

	1(no tx)	1(tx)	2	4	8	16
time(s)	27.51	28.34	23.53	12.42	9.4	12.54
#aborts	0	0	337	1143	3422	536384

From Table 4.5, it can be seen that Bayes performs the best at $Q = 8$. When Q is smaller, the performance is not good due to lack of concurrency, though the number of aborts is small. However, when Q is larger, the performance gets worse due to high contention. Therefore, RAC is very useful for adjusting Q to the optimal value. Differences between $Q = 1$ cases with and without using transactional mechanisms reflect transactional overheads.

Figure 4.19 shows a scenario explaining theoretically why RAC can improve performance with restricted admission. As mentioned earlier, in TinySTM, a late-coming short transaction can easily abort a long transaction that has been running for a long time if the short transaction locks an object first. The time between the entry of the long transaction and the short transaction will be wasted. RAC can reduce the likelihood of this situation by restricting the number of processes acquiring the view.

In Figure 4.19, the long transaction T1 conflicts with the short transaction T3, although T3 starts much later than T1, T3 locks the variable a first. T1 finds out the

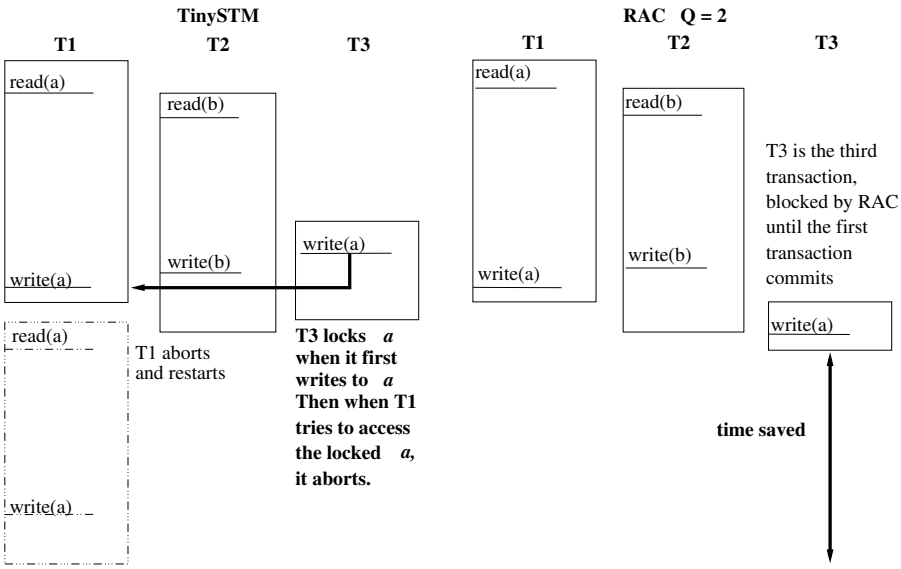

Figure 4.19: RAC implementation over TinySTM - RAC blocks T3 and prevents it from aborting T1 in high contention

conflict when it tries to write to the variable a, then it aborts and restarts. However, if Q is set to 2 by RAC, T3 is the third transaction to begin, so it is blocked until the first transaction (T1) commits, which prevents it from conflicting and aborting T1.

The above results and analysis have demonstrated the advantage of RAC that it can dynamically adjust the admission quota Q to enhance performance, keeping the best balance between concurrency and contention.

4.4.2 View Partitioning Improves Performance

To investigate benefits of view partitioning over traditional TM with transaction scheduling, performance of VOTM and "TinySTM + RAC" is compared using the applications Intruder and Bayes. "TinySTM + RAC" is a system that implements the transaction scheduling algorithms like RAC for the entire TM.

Table 4.6: Performance of VOTM and TinySTM + RAC at $N = 16$

	Application	VOTM	TinySTM + RAC
time(s)	Bayes	11.15	11.97
	Intruder	43.05	59.50
#aborts	Bayes	4591	4587
	Intruder	10986905	10337777

Table 4.6 shows that for Intruder and Bayes where their VOTM versions have multiple views, VOTM outperforms "TinySTM + RAC" by 38% and 7% respectively. In these applications, both VOTM and "TinySTM + RAC" experience similar contention.

In both applications, a view with high contention is often accessed at the same time as another view with low contention. For example, in Intruder, a process can dequeue a task from the task list (with low contention) while another process can access the dictionary, which has high contention, and therefore access is restricted by RAC in VOTM. Similarly in Bayes, the task list and the highly-contended net are allocated in separate views. By placing the task list and the high contention data, such as the dictionary and the net in separate views in VOTM, the restriction placed on access to the dictionary and the net will *not* affect access to the task list and reduce concurrency. However, in "TinySTM + RAC", the entire shared memory is restricted to access under the same admission quota. Therefore, access of all data structures in the shared memory, including the task list with little contention, will be restricted as a result of contention in the dictionary and the net. That is, the concurrency of processes accessing the task list will be unnecessarily affected in "TinySTM + RAC". As shown in the above results, the memory partitioning philosophy of VOTM resolves this problem and therefore has superior performance over transactional memory with transaction scheduling algorithms like "TinySTM + RAC".

4.5 Concluding Remarks

VOTM allows shared data with different access patterns to be allocated in different views, and then let RAC optimize access to each view independently according to the contention level of each view. Therefore, processes accessing a view with low contention will not be hindered by restrictions placed on another view with high contention.

With RAC, VOTM seamlessly integrates locking and transactional memory into one programming paradigm. It can take advantage of the merits of both the pessimistic (locking) and the optimistic (TM) approaches to concurrency control. Programmers do not need to worry about concurrency control of the view, because concurrency control is left to the system (RAC) to decide whether a locking mechanism or a transactional mechanism should be used based on the contention situation of the view.

RAC can improve performance of VOTM regardless of which underlying TM algorithm is used. In any TM algorithms, there will be situations where the contention become very high (number of aborts becomes much larger than the number of transactions), and in these situations, RAC will step in and restrict admission to the view to control contention, thereby reducing works wasted by aborted transactions and improving progress. Experimental results show that RAC has superior performance to both TM and the lock-based approach because of the ability of RAC controlling admission and switching between TM and locking, whereas traditional TM has a performance issue when the contention is high and lock-based approach only works well in fine-grained locking but poorly in coarse-grained locking. Therefore, through the definition/creation of different views in TM, VOTM offers better performance than traditional transactional memory and better convenience (and sometimes better performance) than lock-based programming. We believe this new programming paradigm will bridge the gap between TM and lock-based programming, and thus will bring more vitality to the research of TM.

One issue with RAC is blocking of processes by RAC when Q is smaller than N. This blocking seems to violate the lock-free or obstruction-free feature of TM systems [41]. Even though this feature is arguably necessary [32], RAC can quickly resolve this kind of blocking when the contention becomes low and thus Q is increased up to N, as long as Q does not become 1. If necessary, RAC can completely avoid blocking by using transactions even when Q equals 1, though it will lose some performance gain. In this way, if the system discovers that processes are blocked for too long, the blocking can be easily lifted by increasing Q. Actually, in normal situations, the blocking in RAC is not worse than the live-locking in TM when transactions abort each other without progress under high contention.

Another issue with the current VOTM system is the possibility of deadlock during nested view acquisition. However, in most of the cases, nested view acquisition is not necessary, as shared data that can be accessed together atomically should be allocated in the same view. For example, in VOTM, all nodes in a tree will be allocated into the

same view, thus nested view acquisition for individual nodes of the tree is unnecessary.

The adaptive RAC algorithm presented determines contention level by the abort/success ratio. However given the same contention level, the abort/success ratio would be different across different TM algorithms. For example, commit-time locking (CTL) TM algorithms would be expected to have a much lower abort/success ratio than ETL algorithms, as transactions would be aborted at a much later time. Therefore parameters such as MIN and MAX for the abort/success ratio would be specific to the underlying TM algorithm, and would be tedious to optimize RAC manually for each new TM algorithm.

To overcome this problem, the next chapter proposes a new version of the RAC algorithm that determines contention according to the total time spent in aborted transactions and successful transactions. In this way, the RAC algorithm will become TM-algorithm-neutral, as regardless of which TM algorithm is used, if the time spent in aborted transactions is much higher than the time spent in committed transactions, the contention is high, and it is therefore worthwhile to cut the time wasted in aborted transactions by reducing Q.

Chapter 5

Improvements on the RAC Algorithm

The last chapter proposed the VOTM system that uses RAC to control admission to each view, and demonstrated performance and programmability improvements of VOTM over both traditional TM and lock models. Unfortunately, in the VOTM implementation shown in the last chapter, the contention situation was evaluated empirically. Some empirical thresholds were used to decide if the contention is too high and the admission quota Q should be adjusted to reduce the contention level. These empirical thresholds may work well for one TM system (e.g. TinySTM [33] using encounter time locking), but may not be suitable to measure the contention situation of other TM systems. Therefore, different threshold values have to be decided in an ad-hoc way for different TM systems. As far as we know, there has been no theoretical model to guide the selection of the thresholds for various TM systems.

To address the above issue, this chapter proposes a theoretical model for RAC. Based on this model, the system is able to decide the contention level of TM systems in a systematic way and to adjust the admission quota Q to an optimal value for any TM systems. This model works better especially when the contentions are dynamic and/or bursty. Based on this model, VOTM is extensively evaluated with microbenchmarks and real applications, and different transactional memory implementations are used to investigate in which cases and in which ways VOTM can improve performance.

The rest of this chapter is organized as follows: Section 5.1 presents the theoretical model of RAC. Section 5.2 discusses the details of the VOTM implementation. Section 5.3 describes the experimental design. Section 5.4 presents the initial results of RAC. Section 5.5 refines the RAC theoretical model by taking TM overheads into

account. Section 5.6 presents experimental results of the refined RAC model and Section 5.7 concludes the chapter.

5.1 The Restricted Admission Control Theoretical Model

RAC vs. Conventional TM

Consider a set of *transactions* $S_T = \{T_1, ..., T_n\}$, which access the same view and are executed by N threads. The *duration* of transaction $T_i (1 \leq i \leq n)$ is denoted by t_i and refers to the time period that T_i is executed from start to commit without conflicts and interruptions. For simplicity of the analysis, it is assumed that, during the execution of T_i, the expected number of aborts is c_i and the average time spent by an aborted transaction is d_i, where $c_i, d_i \geq 0$. Therefore, the expected execution time for T_i is $c_i d_i + t_i$ in conventional TM that has no admission control of transactions.

makespan is defined as the total time needed to perform all transactions. Suppose that N threads are continuously executing the transactions, then the best possible *makespan* for S_T in conventional TM, denoted by $makespan_{TM}(S_T)$, can be calculated as

$$makespan_{TM}(S_T) = \frac{\sum_{i=1}^{n} c_i d_i + t_i}{N} \tag{5.1}$$

In RAC, Q transactions are allowed to be executed at any given time, where $1 \leq Q \leq N$. The expected execution time for T_i is $\frac{Q-1}{N-1} \times c_i d_i + t_i$, which can be proven as follows.

Suppose T_i aborts due to the conflict of shared memory location s accessed by $T_{i'}$ in conventional TM. However, in RAC, if T_i is allowed to access s at a given time, the probability that $T_{i'}$ is also allowed to access s is $\frac{Q-1}{N-1}$, because RAC allows only Q threads accessing s at any given time. So, the probability that T_i has 1 abort due to the conflict with $T_{i'}$ is $\frac{Q-1}{N-1}$. According to the binomial distribution, the probability that T_i has k aborts ($k \in \{0, 1, ..., c_i\}$) is $p(k) = \binom{c_i}{k}(\frac{Q-1}{N-1})^k(\frac{N-Q}{N-1})^{c_i-k}$. Therefore, the expected execution time for T_i in RAC is $\sum_{k=1}^{c_i}(kd_i + t_i)p(k) = \sum_{k=1}^{c_i} kp(k)d_i + \sum_{k=1}^{c_i} p(k)t_i = \frac{Q-1}{N-1} \times c_i d_i + t_i$. (By the binomial distribution, $\sum_{k=1}^{c_i} kp(k) = \frac{Q-1}{N-1} \times c_i$ and $\sum_{k=1}^{c_i} p(k) = 1$)

Suppose the Q threads are continuously executing the transactions in RAC, then

the *makespan* for S_T in RAC, denoted by $makespan_{RAC}(S_T)$, is

$$makespan_{RAC}(S_T) = \frac{\sum_{i=1}^{n} \frac{Q-1}{N-1} \times c_i d_i + t_i}{Q} \tag{5.2}$$

Therefore, the difference of $makespan_{RAC}(S_T)$ and $makespan_{TM}(S_T)$, denoted by Δ, can be obtained by Equation 5.1 and 5.2 as follows.

$$\begin{aligned}
\Delta &= makespan_{RAC}(S_T) - makespan_{TM}(S_T) \\
&= \frac{\sum_{i=1}^{n} \frac{Q-1}{N-1} \times c_i d_i + t_i}{Q} - \frac{\sum_{i=1}^{n} c_i d_i + t_i}{N} \\
&= \frac{1}{N-1}(\frac{1}{N} - \frac{1}{Q})(\sum_{i=1}^{n} c_i d_i - \sum_{i=1}^{n} t_i(N-1))
\end{aligned} \tag{5.3}$$

Let $\delta = \frac{\sum_{i=1}^{n} c_i d_i}{\sum_{i=1}^{n} t_i(N-1)}$. It can be derived from Equation 5.3 that

(a) if $\delta > 1$, then $\Delta < 0$ and $makespan_{RAC}(S_T) < makespan_{TM}(S_T)$. That is, RAC outperforms conventional TM and the performance improvement is $|\Delta|$ when $\delta > 1$ (i.e., $\sum_{i=1}^{n} c_i d_i > \sum_{i=1}^{n} t_i(N-1)$). From this condition, it can be seen that RAC works especially well for transactions with high contention (c_i can be considered as the number of conflicts experienced by T_i), which will be verified in the experimental results.

(b) If $\delta \leq 1$, then $\Delta \geq 0$ and $makespan_{RAC}(S_T) \geq makespan_{TM}(S_T)$. That is, when $\delta \leq 1$, RAC should set Q to N. When Q equals to N, $\Delta = 0$ and RAC works the same as the conventional TM.

RAC with Q' Threads vs Q Threads

Similar to the deduction of Equation 5.3, the difference between makespans of RAC using Q' (new) threads ($makespan_{RAC}(S_T, Q')$) and Q (previous) threads ($makespan_{RAC}(S_T, Q)$) is

$$\begin{aligned}
&makespan_{RAC}(S_T, Q') - makespan_{RAC}(S_T, Q) \\
&= \frac{1}{Q-1}(\frac{1}{Q} - \frac{1}{Q'})(\sum_{i=1}^{n} c_i(Q) \times d_i(Q) - \sum_{i=1}^{n} t_i \times (Q-1))
\end{aligned} \tag{5.4}$$

where $c_i(Q)$ and $d_i(Q)$ are the expected number of aborts and the average time spent by an abort of T_i when using Q threads in RAC.

Let $\delta(Q) = \frac{\sum_{i=1}^{n} c_i(Q) \times d_i(Q)}{\sum_{i=1}^{n} t_i \times (Q-1)}$. It can be derived from Equation 5.4 that

(a) if $\delta(Q) > 1$ and $Q' < Q$, then $makespan_{RAC}(S_T, Q) > makespan_{RAC}(S_T, Q')$. That is, if $\delta(Q) > 1$, RAC should decrease Q to reduce the execution time of the concurrent transactions.

(b) if $\delta(Q) < 1$ and $Q' > Q$, then $makespan_{RAC}(S_T, Q) > makespan_{RAC}(S_T, Q')$. Therefore, to reduce the execution time of the concurrent transactions, RAC should increase Q.

In summary, we have the following observation[1]:

Observation 1. *If $\delta(Q)$ is larger than 1, Q should be decreased; if $\delta(Q)$ is smaller than 1, Q should be increased, in order to reduce the makespan of S_T in RAC.*

In the RAC implementation, $\sum_{i=1}^{n} c_i(Q) \times d_i(Q)$ is estimated with the total CPU cycles spent in aborted transactions, and $\sum_{i=1}^{n} t_i$ is estimated with the total CPU cycles spent in successful transactions. Therefore, $\delta(Q)$ is estimated with Equation 5.5 in RAC:

$$\delta(Q) = \frac{CPUcycles_{aborted\_tx}}{CPUcycles_{successful\_tx} \times (Q-1)} \tag{5.5}$$

RAC in Multiple Views vs Single View

This section analyzes the potential gain of performance in multiple-views scenario where RAC can separately control admission to each view according to its contention. It is compared with the scenario where RAC controls access to the entire transactional memory.

Assume the set of transactions $S_T = \{T_1, ..., T_n\}$ can be divided into two non-intersecting subsets $S_T^1 = \{T_1^1, ..., T_n^1\}$ and $S_T^2 = \{T_1^2, ..., T_n^2\}$, where transactions in S_T^1 access data in $Object_1$, and transactions in S_T^2 access data in $Object_2$. So, if $\delta^1 = \frac{\sum_{i=1}^{n} c_i^1 d_i^1}{\sum_{i=1}^{n} t_i^1 (N-1)} > 1$ (high contention), $\delta^2 = \frac{\sum_{i=1}^{n} c_i^2 d_i^2}{\sum_{i=1}^{n} t_i^2 (N-1)} \leq 1$ (low contention), and $Q^1 \leq Q \leq Q^2$, then the *makespan* of putting $Object_1$ and $Object_2$ into separate views with independent RAC $makespan_{MV-RAC}((S_T^1, Q^1), (S_T^2, Q^2))$ should be smaller than the *makespan* of a single view with RAC $makespan_{RAC}(S_T, Q)$:

$$\begin{aligned} &makespan_{MV-RAC}((S_T^1, Q^1), (S_T^2, Q^2)) \\ &\quad \leq makespan_{RAC}(S_T, Q) \end{aligned} \tag{5.6}$$

[1]The RAC theoretical model is jointly developed with Dr Yawen Chen and Associate Professor Zhiyi Huang. This model has been published in [62] and Dr Yawen Chen provided the mathematical deductions of Equations 5.1, 5.2, 5.3, 5.4 and 5.6

The proof of Equation (5.6) is as follows.

$$
\begin{aligned}
&makespan_{RAC}(S_T, Q) \\
&= \frac{\sum_{i=1}^{n} \frac{Q-1}{N-1} \times c_i d_i + t_i}{Q} \\
&= \frac{\sum_{i=1}^{n} c_i d_i}{N-1} + \frac{1}{Q} \times (\sum_{i=1}^{n} t_i - \frac{\sum_{i=1}^{n} c_i d_i}{N-1}) \\
&= \frac{\sum_{i=1}^{n} c_i^1 d_i^1}{N-1} + \frac{1}{Q} \times (\sum_{i=1}^{n} t_i^1 - \frac{\sum_{i=1}^{n} c_i^1 d_i^1}{N-1}) \\
&+ \frac{\sum_{i=1}^{n} c_i^2 d_i^2}{N-1} + \frac{1}{Q} \times (\sum_{i=1}^{n} t_i^2 - \frac{\sum_{i=1}^{n} c_i^2 d_i^2}{N-1}) \\
&= makespan_{RAC}(S_T^1, Q) + makespan_{RAC}(S_T^2, Q)
\end{aligned} \tag{5.7}
$$

Suppose view 1 has high contention,
i.e., $\delta^1 = \frac{\sum_{i=1}^{n} c_i^1 d_i^1}{\sum_{i=1}^{n} t_i^1 (N-1)} > 1$, and $Q^1 \leq Q$. Then,

$$
makespan_{RAC}(S_T^1, Q^1) \leq makespan_{RAC}(S_T^1, Q) \tag{5.8}
$$

Suppose view 2 has low contention,
i.e., $\delta^2 = \frac{\sum_{i=1}^{n} c_i^2 d_i^2}{\sum_{i=1}^{n} t_i^2 (N-1)} \leq 1$, and $Q \leq Q^2$. Then,

$$
makespan_{RAC}(S_T^2, Q^2) \leq makespan_{RAC}(S_T^2, Q) \tag{5.9}
$$

Therefore, we have

$$
\begin{aligned}
&makespan_{RAC}(S_T^1, Q^1) + makespan_{RAC}(S_T^2, Q^2) \\
&\leq makespan_{RAC}(S_T^1, Q) + makespan_{RAC}(S_T^2, Q)
\end{aligned} \tag{5.10}
$$

Since the *makespan* of the multiple-view RAC is:

$$
\begin{aligned}
&makespan_{MV-RAC}((S_T^1, Q^1), (S_T^2, Q^2)) \\
&= makespan_{RAC}(S_T^1, Q^1) + makespan_{RAC}(S_T^2, Q^2)
\end{aligned} \tag{5.11}
$$

and the *makespan* of the single view RAC is:

$$
\begin{aligned}
&makespan_{RAC}(S_T, Q) \\
&= makespan_{RAC}(S_T^1, Q) + makespan_{RAC}(S_T^2, Q)
\end{aligned} \tag{5.12}
$$

From Equation (5.10), we have:

$$\begin{aligned} &makespan_{MV-RAC}((S_T^1, Q^1), (S_T^2, Q^2)) \\ &\leq makespan_{RAC}(S_T, Q) \end{aligned} \tag{5.13}$$

Now we have this observation:

Observation 2. *If there are two shared objects, which are not required to be accessed together in the same transaction, and the first object has high contention ($\delta(Q)$ is larger than 1) but the second object has low contention ($\delta(Q)$ is smaller than 1), then the two objects should be put into separate views to reduce the makespan of RAC.*

The experimental section will examine the multiple-view RAC model with different applications, and the performance of RAC over different TM implementations.

5.2 Implementation

In this chapter, the implementation of VOTM is based on RSTM-7.0 [67], a C++-based modular software transactional memory system where TM algorithms such as the encounter-time locking algorithm *OrecEagerRedo* and the commit-time locking algorithm *NOrec* [24] are implemented as plug-ins and can be chosen easily by reconfiguration.

The reason for basing the VOTM implementation on TinySTM in the last chapter was that when the experiments in the last chapter were carried out (also published in [63]), RSTM was not available to x86_64 architectures, which the machine used in the experiment was based on.

In this chapter, switching to the RSTM-VOTM implementation allows easy comparison with multiple TM algorithms without extra coding. Moreover, since the RSTM base code is object-oriented, it becomes possible to refactor the RSTM base code so that each view has its own separate TM metadata, and therefore each view is now genuinely a separate TM system. This arrangement allows reduction of TM metadata contention by partitioning the application shared data into multiple views, and prevention of false conflicts between accesses to different views, whereas in the TinySTM-VOTM implementation in the last chapter, the entire TM uses a single set of TM metadata, and therefore the TinySTM-VOTM implementation would not get these benefits in view partitioning.

In RSTM-VOTM, access to each view is separately controlled by its own RAC mechanism as explained in the last chapter. As mentioned previously, $\delta(Q)$ in Observations 1 and 2 is estimated with Equation 5.5. `rdtsc()` is used to measure the CPU cycles spent in aborted transactions and successful transactions. The model shown in the last section states that if $\delta(Q)$ of the view is larger than 1, it has high contention and its admission quota Q should be decreased; and if $\delta(Q)$ of the view is lower than 1, it has low contention and its admission quota Q should be increased. However, to prevent the overheads of unnecessarily frequent adjustments of Q, a critical zone with two values MAX and MIN is used. When $\delta(Q) > MAX$, Q will be decreased; and Q will be increased when $\delta(Q) < MIN$. MAX and MIN are set to 1.1 and 0.5 respectively. To avoid the hotspot problem of accessing the counter P of RAC by up to 64 processes, the counter P is implemented as a scalable counter similar to the sloppy counter in [13].

5.3 Experimental Design

The first objective of this experiment is to examine the performance gain of VOTM using RAC. This experiment uses various TM applications including Vacation, SSCA2, Labyrinth and Intruder from the STAMP-0.9.10 benchmark suite [18], Eigenbench microbenchmarks [48], and MultiRBTree (which is derived from the red-black tree microbenchmark in RSTM-7.0). For this objective, only one view is used in these applications so that the performance gain of RAC can be separated from the performance gain of using multiple views.

The second objective of this experiment is to examine the performance gain of using multiple views. Intruder, Eigenbench, and MultiRBTree are evaluated where multiple views are applicable. For these applications, both the single-view version and the multi-view version are implemented. In the single-view version, all shared objects are put into the same single view, while in the multi-view version shared objects are placed in separate views. As mentioned before, partitioning shared objects into separate views can improve performance by allowing RAC to optimally adjust the admission quota Q of each view individually according to its contention level. In this way, the system can restrict access to a high contention view, without affecting the concurrency of transactions that access other low-contention views. In the rest of this paper, a multi-view configuration for an application is denoted by its number of views used, e.g. "1-view", "2-view", "4-view" and "8-view".

Furthermore, to examine how well VOTM can interact with different TM algo-

rithms, this work has implemented two versions of VOTM:

VOTM-OrecEagerRedo
: is based on the encounter-time locking TM algorithm "OrecEagerRedo" (similar to TinySTM [33]), which is implemented in RSTM-7.0 [67].

VOTM-NOrec
: is based on the commit-time locking TM algorithm "NOrec" [24] which is also from RSTM, as described in Chapter 4.

The STAMP applications Intruder, Vacation, SSCA2 and Labyrinth are discussed in detail in Chapter 4, and the rest of this section will describe the multiple-view applications Eigenbench and MultiRBTree.

5.3.1 Eigenbench

Eigenbench [48] can generate transactions using orthogonal parameters, and allows a better understanding of the behaviour of a TM system by adjusting the parameters.

For example, contention in Eigenbench is controlled by adjusting the size of hot_array and the number of read and write accesses to the hot_array. High contention can be caused by a large number of read-write accesses to a relatively small length of hot_array. The shared mild_array is also accessed by transactions, but each process has its own subarray and therefore access to mild_array will not cause conflicts, but will increase transaction size and rollback overheads.

Long transactions can be generated by adjusting one or more of the following features:

- reading/writing to a large range of locations in shared memory;
- many repeated accesses to the same locations in shared memory;
- frequent access to local memory;
- long transactions (using NOPs).

In Eigenbench, a transaction is modelled by a sequence of reads/writes to the shared memory, interleaved with accesses to local memory and computation (represented by NOPs). There are also accesses to local memory and computations outside transactions in Eigenbench.

In this experiment, the Eigenbench program is modified to have two views, each view has its own hot_array, mild_array and parameters concerning the number of read/write accesses and the number of NOPs in each transaction that access the view.

The modified Eigenbench program will execute a number of iterations, which is the total number of transactions specified for each view. Each iteration accesses one of the two views randomly, followed by the activities outside transactions. The pseudocode outlining the modified Eigenbench application is shown in Figure 5.1, and parameters used in Eigenbench are shown in Table 5.1.

In the "2-view" version, each process executes 25000 transactions that access view 1 (the high contention view) and 25000 transactions that access view 2 (the low contention view), with the accesses interleaved randomly. View 1 is set to be accessed by long transactions with high contention, with each transaction accessing many elements in a small hot_array; whereas view 2 is accessed by long transactions with low contention.

Table 5.1: Eigenbench parameters for the 2-view version

View	1	2
N	64	
loops	25k	25k
A1	256	16k
A2	16k	16k
A3	8k	8k
R1	80	10
W1	20	10
R2	10	10
W2	10	10
R3i	0	5
W3i	0	1
NOPi	0	20
R3o	0	
W3o	0	
NOPo	0	

In the "1-view" version, each process executes 50000 transactions. In each iteration, after the view is acquired, the transaction can access either object 1 (with high contention) or object 2 (with low contention). Accesses to object 1 and 2 have the same access patterns as view 1 and 2 in the "2-view" version.

```
struct View_data {
    /* shared array where conflict occurs, accessed in tx */
    shared word hot_array[A1];

    /* shared array where each process accesses its own subarray, so does not
       cause conflict, but still needs rollback should tx be aborted */
    shared word mild_array[A2];

    /* private to each process, can be accessed either inside or outside tx.
       if accessed inside tx and tx aborted then needs to roll back changed */
    word cold_array[A3];
};

View_data views[2];

each process:

for loops:
do
    acquire view 1 or 2 randomly
    in acquired view:
    perform
       r1 reads and w1 writes to the shared hot_array,and
       r2 reads and r2 writes to the shared mild_array in *random order*
       each access touches a random element (word) in the shared hot_array, or
       in the dedicated subarray within the shared mild_array

       between two accesses to shared arrays, there will also be r3i reads
       and w3i writes to the private cold array, and NOPi instructions
    release view

    /* activities outside transactions:
    perform r3o reads and w3o writes to the private array
    perform NOPo instructions
done
```

Figure 5.1: Pseudocode of the modified Eigenbench application

5.3.2 MultiRBTree

MultiRBTree has eight red-black trees. Each transaction will randomly search, insert, or delete a value in a tree chosen randomly. Therefore, this application only needs to guarantee the atomicity of the operations on a single tree and thus each tree can be placed into a separate view.

However, to examine the potential of the benefit of multiple views, the trees are divided into 1, 2, 4 and 8 views. Their performance results can show how much benefit can be exploited by pure view partitioning without RAC.

In this microbenchmark, each tree has 1,000,000 elements, and each process executes 500,000 transactions. The ratio of the search operation is 34%, and the ratio for both insertion and deletion is 33%.

5.4 Experimental Results

In this experiment, all tests are carried out on a Dell PowerEdge R815 server, which has four AMD Opteron 6276 16-core processors running at 2.3GHz, and thus has a total of 64 cores with a total 64GB DDR3 RAM. Linux kernel 3.2 and the compiler gcc-4.4 are used during the experiment. All programs are compiled with the optimization flag -O3. Time spent in transactions are measured with `rdtsc()` and data cache misses are measured with the Performance Monitoring Counters (PMCs) [107].

For each application, this experiment evaluates the performance gain by restricting the admission quota Q, and whether the RAC algorithm can correctly identify this optimal Q.

The rest of this section will show the results of our two VOTM implementations: VOTM-OrecEagerRedo and VOTM-NOrec.

5.4.1 Performance of VOTM-OrecEagerRedo

Figure 5.2 shows the performance of VOTM compared with the traditional TM using the OrecEagerRedo algorithm. The applications on VOTM all use a single view with the RAC mechanism controlling concurrency.

For the rest of this chapter, "VOTM-OPT" denotes the performance of VOTM with Q set to the optimal value; "VOTM-RAC" denotes the performance of VOTM with Q determined by the RAC algorithm; "cputx" denotes the total number of CPU cycles spent in successful transactions and "cpuabort" denotes the total number of

CPU cycles spent in aborted transactions in the application.

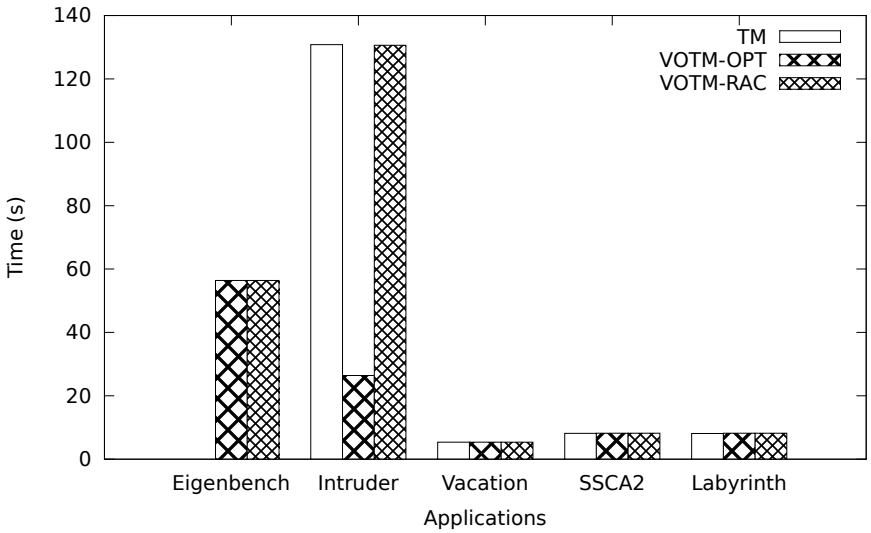

Figure 5.2: Single-view applications in VOTM-OrecEagerRedo (Eigenbench on TM is not shown due to livelock)

Table 5.2: Single-view applications with VOTM-OrecEagerRedo

Application	version	time(s)	cputx	cpuabort	$\delta(Q)$	Q	#cachemiss
Eigenbench	TM	livelock					
	VOTM-OPT	56.4	148G	96G	0.65	2	2.83G
	VOTM-RAC	56.4	148G	96G	0.65	2	2.83G
Intruder	TM	130.8	1.12T	14.9T	0.21	64	20.8G
	VOTM-OPT	26.4	411G	363G	0.05	16	4.45G
	VOTM-RAC	130.9	1.12T	14.3T	0.20	64	21.2G
Vacation	TM	5.38	685G	65.4G	0.002	64	3.61G
	VOTM-OPT	5.38	687G	65.2G	0.002	64	3.60G
	VOTM-RAC	5.38	687G	65.2G	0.002	64	3.60G
SSCA2	TM	8.17	763.2G	229M	0	64	2.29G
	VOTM-OPT	8.19	760G	225M	0	64	230G
	VOTM-RAC	8.19	760G	225M	0	64	230G
Labyrinth	TM	8.11	315G	590G	0.03	64	6.61G
	VOTM-OPT	8.21	318G	603G	0.03	64	669G
	VOTM-RAC	8.21	318G	603G	0.03	64	669G

For Eigenbench, Figure 5.2 shows livelock is incurred on the traditional TM, but VOTM-OrecEagerRedo can prevent the livelock with RAC by setting Q to the optimal value of 2, as illustrated in Table 5.2.

In Vacation, SSCA2 and Labyrinth, the contention is low as illustrated by the very low $\delta(Q)$ in Table 5.2, so RAC does not need to restrict admission and Q is correctly left at 64. In these applications, the runtime of VOTM is only slightly (1-3%) longer than that of TM, which shows that VOTM has very low overhead.

For Intruder, although the optimal Q should be set to 16 (as indicated by VOTM-OPT), which would have a 400% performance improvement over TM, VOTM-RAC fails to restrict Q to improve performance, as its low $\delta(Q)$ indicates low contention.

5.4.2 Performance of VOTM-NOrec

Figure 5.3 shows the performance of VOTM compared with the traditional TM using the NOrec algorithm. The applications on VOTM all use a single view with the RAC mechanism to control concurrency.

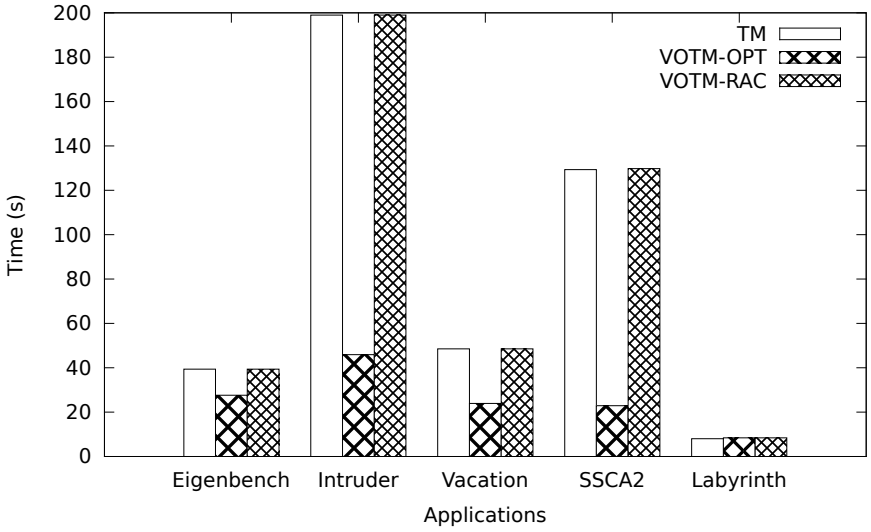

Figure 5.3: Single-view applications in VOTM-NOrec

Table 5.3: Single-view applications in VOTM-NOrec

Application	version	time(s)	cputx	cpuabort	$\delta(Q)$	Q	#cachemiss
Eigenbench	TM	39.4	274G	4.0T	0.23	64	6.18G
	VOTM-OPT	27.6	123G	319G	0.37	8	1.69G
	VOTM-RAC	39.4	275G	4.0T	0.23	64	6.19G
Intruder	TM	199	10.8T	17.4T	0.03	64	28.2G
	VOTM-OPT	45.9	644G	82G	0.02	8	3.56G
	VOTM-RAC	199	10.8T	17.3T	0.03	64	28.2G
Vacation	TM	48.5	6.54T	9.46G	0	64	24.5G
	VOTM-OPT	23.9	786G	50M	0	16	5.87G
	VOTM-RAC	48.5	6.53T	9.37G	0	64	24.5G
SSCA2	TM	129.3	15.7T	1.94G	0	64	4.45G
	VOTM-OPT	22.9	82.4G	267K	0	4	2.18G
	VOTM-RAC	129.8	15.7T	2.0G	0	64	4.44G
Labyrinth	TM	8.03	312G	594G	0.03	64	6.67G
	VOTM-OPT	8.39	320G	611G	0.03	64	6.72G
	VOTM-RAC	8.39	320G	611G	0.03	64	6.72G

In Labyrinth, $\delta(Q)$ indicates that the contention is low, so the RAC algorithm correctly leaves Q to 64 (Table 5.3). The runtime of VOTM-RAC is only 2% slower than the runtime of TM, which indicates that VOTM has very low overheads.

From Table 5.3 shows that optimal performance in Eigenbench, Intruder, Vacation and SSCA2, could be achieved by setting Q to 8, 8, 16 and 4 respectively, By setting Q to the optimal value, VOTM-OPT has a performance gain over TM by 50%, 400%, 100% and 500% respectively, as shown in Figure 5.3. However, since the $\delta(Q)$ of these applications are very low (< 0.00001), the RAC algorithm does not restrict Q, and therefore VOTM-RAC does not enjoy this performance gain. In these cases, when Q is restricted in VOTM-OPT, total times spent in both successful and aborted transactions (cputx and cpuabort) decreases, and as a result, $\delta(Q)$ remains as a small value (Table 5.3).

This finding indicates that as Q increases in these memory-intensive applications, TM overheads in both successful and aborted transactions increase. These overheads can be explained by the read-set re-validation in NOrec, which becomes very frequent when Q is large. The frequent read-set re-validation then causes considerable cache

hotspot on its global clock, and hence the dramatic increase of CPU cycles and cache misses spent in the overheads, as seen in Vacation, where restricting Q to 16 cuts the number of cache misses to $\frac{1}{5}$ and results in 100% performance gain. Therefore restricting Q can cut this overhead and improve performance. However the current RAC model shown in the last section assumes negligible TM mechanism overhead, and therefore fails to take the overhead into account. The next section will extend the RAC model to take TM overhead into account.

5.5 Refinements on the RAC Model

The RAC model presented in Section 5.1 assumes that TM mechanism overhead (such as overhead incurred by TxWrite() and TxRead()) is negligible. However in memory-intensive applications, TM mechanism overhead arisen from cache contention of TM metadata and/or read-set re-validation can be considerable as the admission quota Q increases, and can easily outweigh the benefits of concurrency in TM. Therefore in order to determine the optimal admission quota Q, the RAC model must take TM mechanism overhead into consideration. This section will investigate how the RAC theoretical model should be extended to take TM overhead into account.

When a memory location is accessed by TM, this access will be logged by the TM mechanism. Therefore the log maintenance overhead will be present, *even when transactions are run serially.* This overhead will form the static component of the TM mechanism overhead, as it is independent from the number of transactions concurrently run. At low Q, the static component dominates the TM mechanism overhead. For example, if the cumulative time of all transactions (calculated by adding transactional runtime of all processes together) is $16s$, and the overhead is $14s$, then the *user_time* will be 2s. Suppose the current Q is 4, then it will take $4s$ to run the transactions. However if these transactions were to be run serially, *without the transactional mechanism*, then it will only take $2s$ to run the transactions, faster than running concurrently at $Q = 4$. Therefore, in cases where the static component dominates, if the overhead is larger than *user_time* $\times Q$, running the transactions serially without TM mechanism would have better performance than running concurrently with admission quota set to the current Q.

One could argue that if the static component were to be the only component in the TM mechanism overheads, then by raising Q to a very high value, such as 32, we would eventually get a performance gain, as the transactions will theoretically take 0.5s to

run. However as explained below, the TM mechanism overhead will be dramatically increased by the dynamic component at higher Q, and therefore raising Q in this case will worsen the performance.

When transactions are run concurrently, transactions will have extra overheads from read-set re-validations, and cache contention of TM metadata (such as the global clock). These overheads form the dynamic component of the TM mechanism overhead, which increases dramatically as Q increases.

In Figure 5.4, the VOTM-NOrec version of Vacation has a static overhead of approximately 150×10^9 CPU cycles, which dominates the TM mechanism overhead at $1 <= Q <= 4$. Then as Q increases further, the dynamic component drives the overall TM mechanism overhead up rapidly. When the dynamic component of the TM mechanism overheads is high, restricting Q can often improve performance by cutting this overhead. Empirical observation on different applications suggests that at $Q >= 8$, performance can be improved by reducing Q when the TM mechanism overhead is larger than $user\_time \times 8$.

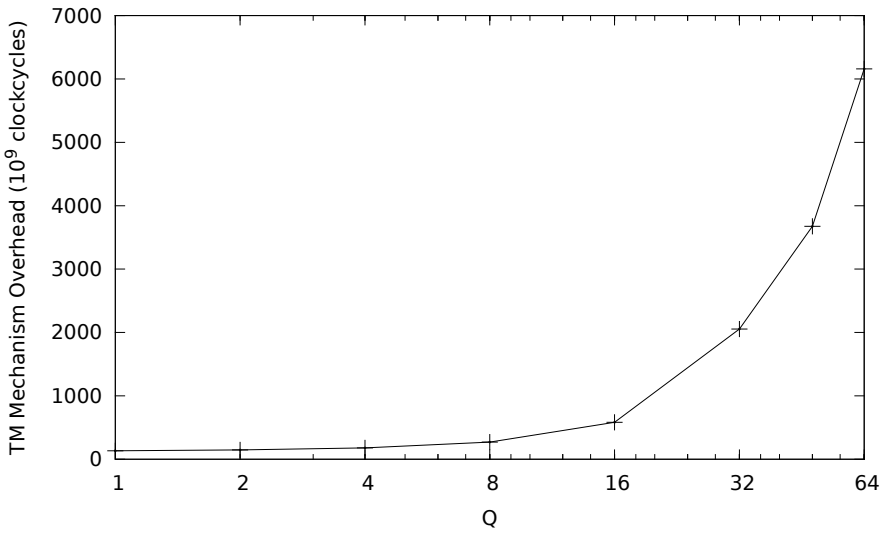

Figure 5.4: TM mechanism overhead of Vacation in VOTM-NOrec

Based on the above analysis, we use the following *overhead_score* to enhance the RAC model:

$$overhead\_score(Q) = \begin{cases} \frac{overhead}{user\_time \times Q} & , Q < 8 \\ \frac{overhead}{user\_time \times 8} & , Q >= 8 \end{cases} \tag{5.14}$$

where

$$overhead = overhead_{successful\_tx} + overhead_{aborted\_tx} \tag{5.15}$$

and

$$user\_time = user\_time_{successful\_tx} + user\_time_{aborted\_tx} \tag{5.16}$$

In the RAC implementation, *user_time* is calculated by subtracting the total TM mechanism overhead from the total time spent in successful and aborted transactions.

In real-life TM applications, there will be TM mechanism overhead, as well as time wasted in aborted transactions. Therefore the RAC model is now extended to take both factors into the account:

$$score(Q) = \delta(Q) + overhead\_score(Q) \tag{5.17}$$

In the extended RAC model, $score(Q) > 1$ would suggest that Q should be reduced to reduce contention and/or TM mechanism overhead; and $score(Q) < 1$ would suggest that Q should be increased to increase concurrency. However, in the implementation, the critical zone *MIN* and *MAX* of 0.5 and 1.1 respectively will still be used to avoid overheads of excessively frequent adjustment of Q.

The next section will show the accuracy of the updated RAC mechanism in determining the optimal Q in different applications, and evaluate the benefits of partitioning shared data into multiple views.

5.6 Experimental Results of the Refined RAC Model

In this section, the accuracy of the refined RAC model in determining the optimal admission quota Q will be tested on both VOTM-OrecEagerRedo and VOTM-NOrec.

5.6.1 Performance of VOTM-OrecEagerRedo

Figure 5.5 shows the performance of VOTM with the updated RAC algorithm (VOTM-RAC) compared with VOTM with Q set to the optimal value (VOTM-OPT) and traditional TM on single-view applications using the OrecEagerRedo algorithm.

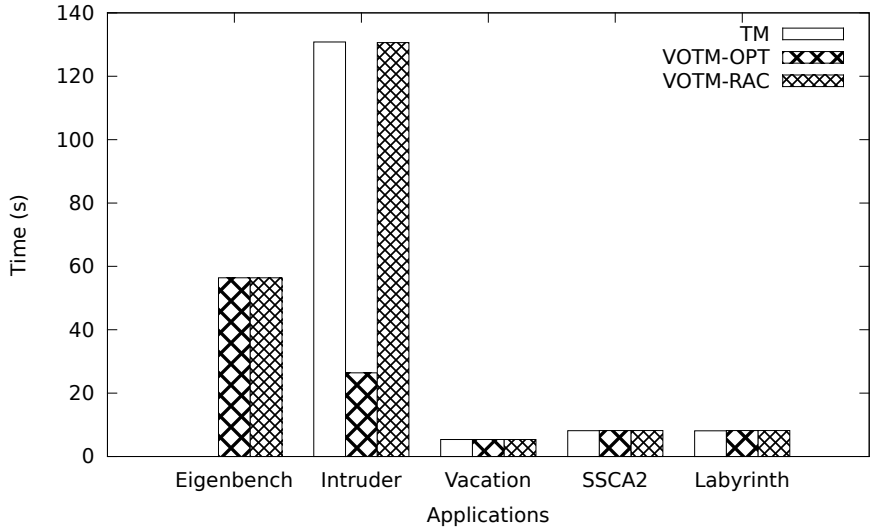

Figure 5.5: Single-view applications in VOTM-OrecEagerRedo (Eigenbench on TM is not shown due to livelock)

Table 5.4: Single-view applications with VOTM-OrecEagerRedo

Application	version	time(s)	cputx	cpuabort	$\delta(Q)$	overhead	$overhead\_score(Q)$	$score(Q)$	Q	#cachemiss
Eigenbench	TM	livelock								
	VOTM-OPT	56.4	148G	96G	0.65	68.9G	0.20	0.85	2	2.83G
	VOTM-RAC	56.4	148G	96G	0.65	68.9G	0.20	0.85	2	2.83G
Intruder	TM	130.8	1.12T	14.9T	0.21	7.53T	0.11	0.32	64	20.8G
	VOTM-OPT	26.4	411G	363G	0.05	358G	0.11	0.16	16	4.45G
	VOTM-RAC	130.9	1.12T	14.3T	0.20	7.21T	0.11	0.31	64	21.2G
Vacation	TM	5.38	685G	65.4G	0.002	372G	0.12	0.12	64	3.61G
	VOTM-OPT	5.38	687G	65.2G	0.002	373G	0.12	0.12	64	3.60G
	VOTM-RAC	5.38	687G	65.2G	0.002	373G	0.12	0.12	64	3.60G
SSCA2	TM	8.17	763.2G	229M	0	511G	0.25	0.25	64	2.29G
	VOTM-OPT	8.19	760G	225M	0	507G	0.25	0.25	64	2.30G
	VOTM-RAC	8.19	760G	225M	0	507G	0.25	0.25	64	2.30G
Labyrinth	TM	8.11	315G	590G	0.03	633M	0.0001	0.03	64	6.61G
	VOTM-OPT	8.21	318G	603G	0.03	633M	0.0001	0.03	64	6.69G
	VOTM-RAC	8.21	318G	603G	0.03	633M	0.0001	0.03	64	6.69G

Table 5.4 shows that RAC prevents livelock on Eigenbench by setting Q to 2, which is shown by VOTM-OPT to be the optimal value. In Vacation, SSCA2 and Labyrinth, the values of $score(Q)$ are low, so VOTM-RAC correctly leaves Q as 64, which gives the optimal performance.

However in Intruder, although the optimal Q is 8 at VOTM-OPT, RAC fails to settle Q to this value, because at $Q = 64$, $score(Q) < 1$, as both $\delta(Q)$ and

overhead_score(Q) are very small. The small value of *overhead_score*(Q) (0.11) at $Q = 64$ indicates that transactional overhead (7.53T) only takes a small portion of the total time in aborted and successful transactions (16T). Therefore the transactional overhead is actually low. The other source of overhead could be the cache misses on the user space in the application itself. In Intruder, there is a frequently-accessed shared array, where accesses to neighbouring elements can easily cause false cacheline conflicts. This hypothesis is supported with the results where VOTM-OPT cuts the number of cache misses to $\frac{1}{5}$ by setting Q to 8 despite very low $\delta(Q)$ and *overhead_score*(Q) values. The effects of cacheline and memory bandwidth contention caused by user space application on RAC need to be thoroughly studied as a future work.

The following part of this experiment will examine how multiple views can further help improve performance. Figure 5.6 shows the performance of Eigenbench and Intruder based on their 1-view and 2-view versions. To show the pure benefit of view partitioning, Figure 5.6 also shows the performance of the two applications running with 1-view and 2-view without RAC, which are denoted by 1-view-nr and 2-view-nr respectively.

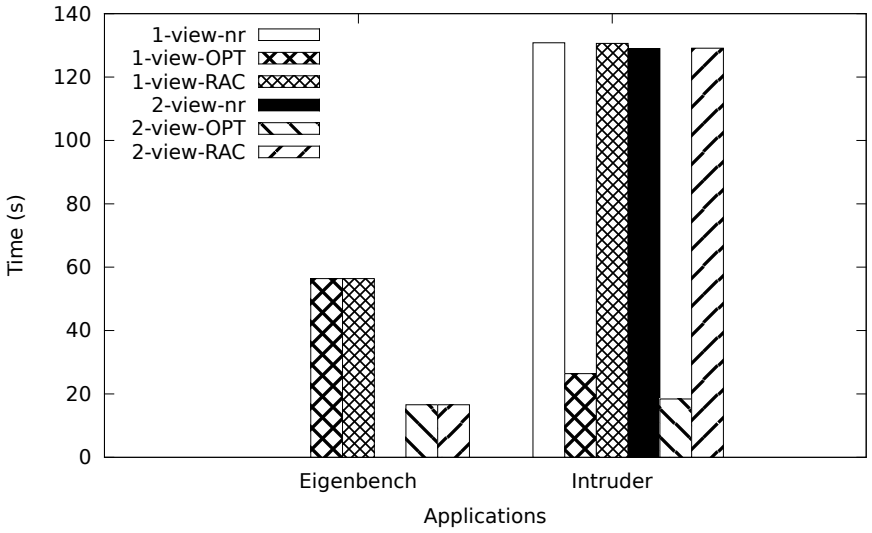

Figure 5.6: Two-view applications on VOTM-OrecEagerRedo. For Eigenbench, its 1-view-nr and 2-view-nr versions have livelock.

Table 5.5: Eigenbench in VOTM-OrecEagerRedo

version	time(s)	Q_1	Q_2	#cachemiss	$cputx_1$	$cpuabort_1$	$\delta(Q_1)$	$overhead_1$	$overhead\_score(Q_1)$	$score(Q_1)$
1-view-nr	livelock									
1-view-OPT	56.4	2	N/A	2.83G	148G	96G	0.65	68.9G	0.20	0.85
1-view-RAC	56.4	2	N/A	2.83G	148G	96G	0.65	68.9G	0.20	0.85
2-view-nr	livelock									
2-view-OPT	16.6	1	64	843M	31.2G	0	N/A	0	N/A	N/A
2-view-RAC	16.6	1	64	843M	31.2G	0	N/A	0	N/A	N/A

version	$cputx_2$	$cpuabort_2$	$\delta(Q_2)$	$overhead_2$	$overhead\_score(Q_2)$	$score(Q_2)$
2-view-nr	livelock					
2-view-OPT	74.2G	2.57G	0.0005	15.5G	0.032	0.032
2-view-OPT	74.2G	2.57G	0.0005	15.5G	0.032	0.032

For Eigenbench, both 1-view-nr and 2-view-nr have livelock which is prevented by RAC in 1-view and 2-view. For the 1-view-RAC version, RAC correctly settles Q to 2 to contain the contention. For the 2-view-RAC version, the quota Q of the first view is set to 1 due to its high contention, but the Q of the second view is set to 64 due to its low contention. Obviously, due to the separate concurrency control for each view in the 2-view-RAC version, 2-view-RAC has a further 200% performance gain over 1-view-RAC. In both versions, Table 5.5 shows that RAC correctly predicts the optimal value of Q in all views.

Table 5.6: Intruder with VOTM-OrecEagerRedo

version	time(s)	Q_1	Q_2	#cachemiss	$cputx_1$	$cpuabort_1$	$\delta(Q_1)$	$overhead_1$	$overhead\_score(Q_1)$	$score(Q_1)$
1-view-nr	130.8	64	N/A	20.8G	1.12T	14.9T	0.21	7.53T	0.11	0.32
1-view-OPT	26.4	16	N/A	4.45G	411G	363G	0.05	358G	0.11	0.16
1-view-RAC	130.9	64	N/A	21.2G	1.12T	14.3T	0.20	7.21T	0.11	0.31
2-view-nr	129.0	64	64	17.5G	299G	15.4T	7.13T	0.82	0.10	0.92
2-view-OPT	18.4	8	64	5.30G	829.1G	26.3G	0.30	26.4G	0.23	0.52
2-view-RAC	129.1	64	64	17.9G	288G	14.4T	0.79	6.76T	0.11	0.90

version	$cputx_2$	$cpuabort_2$	$\delta(Q_2)$	$overhead_2$	$overhead\_score(Q_2)$	$score(Q_2)$
2-view-nr	995G	237G	0.004	699G	0.16	0.17
2-view-OPT	484G	349G	0.01	493G	0.18	0.19
2-view-RAC	967G	241G	0.004	693G	0.17	0.17

Similarly, for Intruder, Table 5.6 shows splitting the shared data into two views allows separate concurrency control in each view. For Intruder, 2-view-OPT has a 50% improvement over 1-view-OPT, as 2-view-OPT can separately set the Q of the two views to 4 and 64 respectively. In 1-view-OPT, the contention of the entire shared memory is treated as a whole view, and thus Q can only be set to a value between 4 and 64, which is 16.

In 2-view-RAC, RAC fails to restrict Q_1 (the Q of view 1) to improve performance,

as the $score(Q_1)$ is not sufficiently high (0.90 at $Q_1 = 64$) to drive Q down. However, RAC has correctly left Q_2 to 64, as $score(Q_2)$ is much smaller than 1 (0.17 at $Q_2 = 64$).

From the performance results of 1-view-nr and 2-view-nr for Intruder in Figure 5.6, there is not much performance gain by simply splitting the shared data without using RAC. 2-view-nr is only 0.3% better than 1-view-nr.

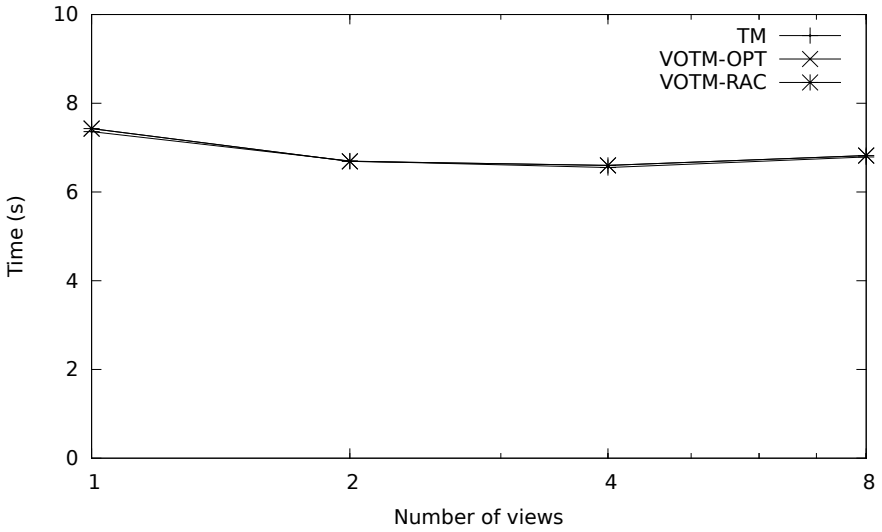

Figure 5.7: MultiRBTree in VOTM-OrecEagerRedo

Figure 5.7 shows the performance results of MultiRBTree using multiple views. This experiment uses 1, 2, 4, 8 views in the application to show potential of using multiple views. In all cases, the $score(Q)$ of all views are low (< 0.4), therefore RAC correctly leaves Q as 64.

As shown in Figure 5.7, partitioning shared data can have a slight improvement on the performance of MultiRBTree up to 4 views. This performance improvement is due to the reduction of transactional aborts according to Table 5.7. Since each view has its own TM metadata, the lock array in the TM metadata of a view will not be shared with other views, and thus it eliminates some aborts caused by false conflicts. Although in all cases, $\delta(Q)$ of all views are very low ($<= 0.0002$) and time wasted inside aborted transactions would be small, the extra overhead of restarting aborted transactions in false conflicts in 1-view cases can explain the performance difference between 1-view and 4-view cases.

However, when the trees are partitioned into eight views such as 8-view-RAC and 8-view-nr in Figure 5.7, the performance slightly deteriorates. The reason is that, when the number of views is increasing, the size of the TM metadata (lock array,

Table 5.7: MultiRBTree in VOTM-OrecEagerRedo

version	#tx	#abort	#cachemiss
1-view-nr	32m	388k	4.02G
1-view-OPT	32m	366k	4.05G
1-view-RAC	32m	366k	4.05G
2-view-nr	32m	303k	4.16G
2-view-OPT	32m	292k	4.22G
2-view-RAC	32m	292k	4.22G
4-view-nr	32m	145k	4.29G
4-view-OPT	32m	146k	4.35G
4-view-RAC	32m	146k	4.35G
8-view-nr	32m	138k	4.46G
8-view-OPT	32m	119k	4.53G
8-view-RAC	32m	119k	4.53G

global clock, etc) is increasing as well. The size of TM metadata for each view is up to 100MB. Therefore, if a process accesses a large number of views during its execution, it will also access the metadata of all views, which will cover a large memory footprint. As a result, cache misses due to frequent cacheline refills are increasing as shown in Table 5.7, which leads to the slight performance degradation. It is possible that this slight performance degradation can be fixed by reducing the size of metadata for each view using optimization techniques.

Since the contention is low in MultiRBTree, Q is set to 64 for all views. Therefore, there is not much performance benefit from using RAC, which is shown in Figure 5.7 by the similar performance between N-view and N-view-nr, e.g. 4-view and 4-view-nr. One the other hand, it shows the little extra overhead of VOTM.

5.6.2 Performance of NOrec

Figure 5.8 shows the performance of VOTM with the updated RAC algorithm (VOTM-RAC) compared with VOTM with Q set to the optimal value (VOTM-OPT) and traditional TM on single-view applications using the NOrec algorithm.

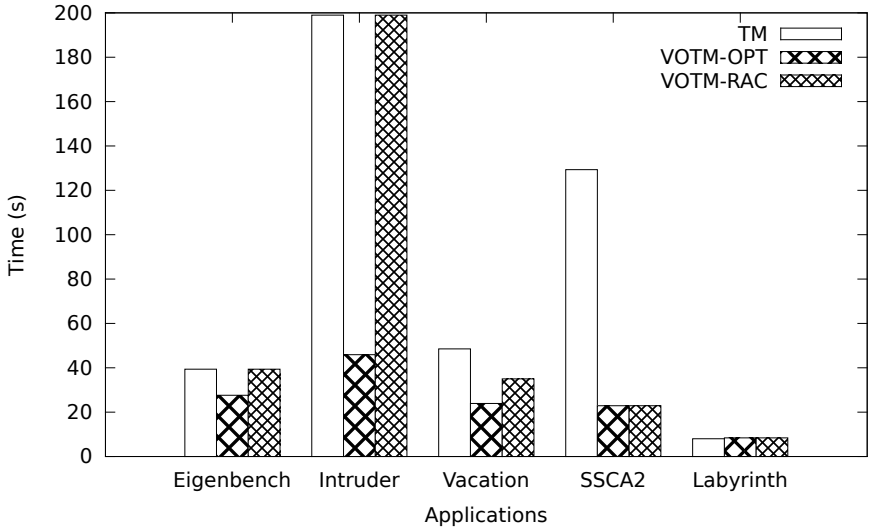

Figure 5.8: Single-view applications in VOTM-NOrec

Table 5.8: Single-view applications in VOTM-NOrec

Application	version	time(s)	cputx	cpuabort	$\delta(Q)$	overhead	$overhead\_score(Q)$	$score(Q)$	Q	#cachemiss
Eigenbench	TM	39.4	274G	4.0T	0.23	1.78T	0.09	0.32	64	6.18G
	VOTM-OPT	27.6	123G	319G	0.37	155G	0.07	0.44	8	1.69G
	VOTM-RAC	39.4	275G	4.0T	0.23	175G	0.09	0.32	64	6.19G
Intruder	TM	199	10.8T	17.4T	0.03	18.3T	0.23	0.26	64	28.2G
	VOTM-OPT	45.9	644G	82G	0.02	538G	0.35	0.37	8	3.56G
	VOTM-RAC	199	10.8T	17.3T	0.03	18.3	0.24	0.27	64	28.2G
Vacation	TM	48.5	6.54T	9.46G	0	6.17T	2.02	2.02	64	24.5G
	VOTM-OPT	23.9	786G	503M	0	583G	0.36	0.36	16	5.87G
	VOTM-RAC	35.0	2.33T	403M	0	2.06T	0.94	0.94	32	10.9G
SSCA2	TM	129.3	15.7T	1.94G	0	15.6T	16.8	16.8	64	4.45G
	VOTM-OPT	22.9	82.4G	267K	0	54.6G	0.50	0.50	4	2.18G
	VOTM-RAC	22.9	82.4G	267K	0	54.6G	0.50	0.50	4	2.18G
Labyrinth	TM	8.03	312G	594G	0.03	135M	0	0.03	64	6.67G
	VOTM-OPT	8.39	320G	611G	0.03	139M	0	0.03	64	6.72G
	VOTM-RAC	8.39	320G	611G	0.03	139M	0	0.03	64	6.72G

In SSCA2, VOTM-RAC correctly sets Q to the optimal value of 4 and has a 500% performance gain over TM. In this application, although the application contention is low, as shown in the very low $\delta(Q)$, the TM overhead is high, as indicated by the high *overhead_score* of 16.8 at $Q = 64$ (Table 5.8). Therefore the $score(Q)$ is very high (16.8), and RAC decreases Q until $score(Q)$ is within the critical range (0.5 - 1.1).

Similarly in Vacation, the high $score(Q)$ (2.02, at $Q = 64$ at TM) driven by the high TM overhead also causes RAC to decrease Q to 32, and this gives a 50% performance

gain over TM. Once Q reaches 32 in VOTM-RAC, *score*(Q) becomes 0.94, which is smaller than 1, and therefore RAC does not decrease Q further. However, the actual optimal Q is 16, as shown in VOTM-OPT, which gives further performance improvement. By taking TM overheads into account, RAC can now improve the performance of SSCA2 and Vacation over TM.

In Labyrinth, since both contention ($\delta(Q)$) and TM overhead (*overhead_score*) are low, the low *score*(Q) correctly guides RAC to leave Q as 64. Since the runtime of VOTM is only 5% longer than the runtime of TM, the overhead of VOTM is very small.

In Eigenbench and Intruder, RAC fails to set Q to the optimal value of 8 as shown in VOTM-OPT. In both applications, both contention and the TM mechanism overhead are low, therefore the *score*(Q) are low (0.32 and 0.26 respectively at $Q = 64$). As a result, RAC does not decrease Q. Similar to Intruder, Eigenbench also have shared arrays which are prone to false cache conflicts, and in addition, large memory movements in arrays can also cause memory bandwidth bottleneck in some hardware architectures. These factors need to be further investigated to establish how they interplay with the overhead of the application.

The following part of the experiment will examine how multiple views can further improve performance in VOTM-NOrec.

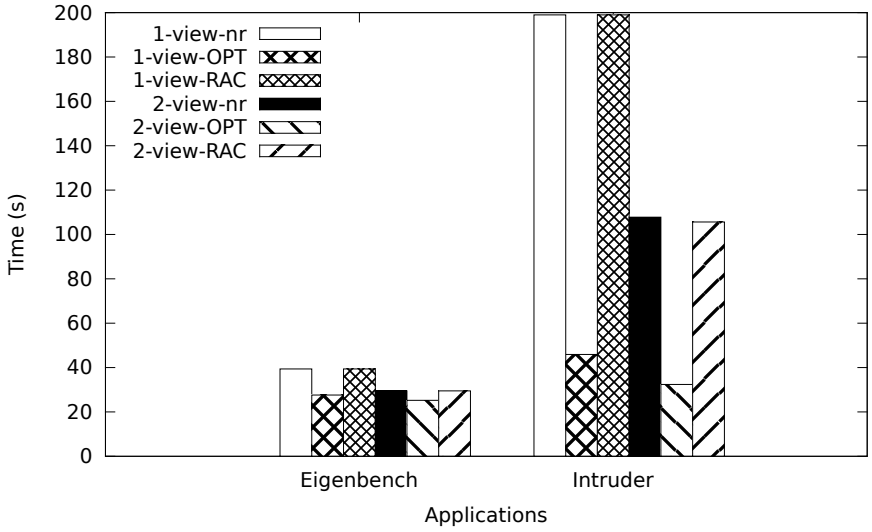

Figure 5.9: Two-view applications in VOTM-NOrec

Table 5.9: Eigenbench in VOTM-NOrec

version	time(s)	Q_1	Q_2	#cachemiss	$cputx_1$	$cpuabort_1$	$\delta(Q_1)$	$overhead_1$	$overhead\_score(Q_1)$	$score(Q_1)$
1-view-nr	39.4	64	N/A	6.18G	274G	4.0T	0.23	1.78T	0.09	0.32
1-view-OPT	27.6	8	N/A	1.69G	123G	319G	0.37	155G	0.07	0.44
1-view-RAC	39.4	64	N/A	6.19G	275G	4.0T	0.23	175G	0.09	0.32
2-view-nr	29.5	64	64	5.91	93.6G	3.07T	0.52	762G	0.04	0.56
2-view-OPT	25.2	8	16	1.73	54.3G	330G	0.87	124G	0.06	0.93
2-view-RAC	29.5	64	64	6.01	94.6G	3.08T	0.52	768G	0.04	0.56

version	$cputx_2$	$cpuabort_2$	$\delta(Q_2)$	$overhead_2$	$overhead\_score(Q_2)$	$score(Q_2)$
2-view-nr	59.4G	88.8M	0	10.1G	0.026	0.026
2-view-OPT	57.6G	114M	0.0001	9.57G	0.025	0.025
2-view-RAC	59.0G	90.8M	0	10.0G	0.026	0.026

Table 5.10: Intruder with VOTM-NOrec

version	time(s)	Q_1	Q_2	#cachemiss	$cputx_1$	$cpuabort_1$	$\delta(Q_1)$	$overhead_1$	$overhead\_score(Q_1)$	$score(Q_1)$
1-view-nr	199	64	N/A	28.2G	10.8T	17.4T	0.03	18.3T	0.23	0.26
1-view-OPT	45.9	8	N/A	3.56G	644G	82G	0.02	538G	0.35	0.37
1-view-RAC	199	64	N/A	28.2G	10.8T	17.3T	0.03	18.3	0.24	0.27
2-view-nr	107.8	64	64	17.2G	43.9G	8.18G	0.003	21.6G	0.089	0.092
2-view-OPT	32.4	8	8	2.87G	20.1G	6.58G	0.05	11.4G	0.09	0.14
2-view-RAC	105.6	64	16	10.4G	44.0G	8.19G	0.003	21.9G	0.089	0.092

version	$cputx_2$	$cpuabort_2$	$\delta(Q_2)$	$overhead_2$	$overhead\_score(Q_2)$	$score(Q_2)$
2-view-nr	11.9T	3.37T	0.005	13.9T	1.32	1.33
2-view-OPT	399G	35.4G	0.01	271G	0.21	0.22
2-view-RAC	465G	97.0G	0.014	331G	0.18	0.19

For Eigenbench and Intruder, Figure 5.9 shows that 2-view-nr outperforms 1-view-nr by 50% and 90% respectively on VOTM-NOrec. This improvement could be attributed to the reduction of the TM metadata contention, as each view is essentially a separate TM system with its own metadata, and splitting into two views effectively halves the contention in each TM system. This reduction of TM metadata contention is shown by a reduction of cache misses in 2-view-nr in both applications (by 10% and 40% respectively, shown in Tables 5.9 and 5.10).

In Eigenbench, 2-view-OPT shows that the optimal Q for view 1 and 2 are 8 and 16 respectively, and it has a further 20% performance gain over 2-view-nr. However, as both $score(Q_1)$ and $score(Q_2)$ are low, RAC does not restrict either Q_1 or Q_2 on 2-view-RAC, and thus 2-view-RAC cannot have this performance gain (Table 5.9).

In Intruder, 2-view-OPT shows that the optimal Q for view 1 and 2 are 8, which results in a 200% performance gain over 2-view-nr, and cuts the number of data cache misses to 1/8. However in 2-view-RAC, it is only the high TM overhead in view 2 ($overhead\_score(Q_2)$) which makes $score(Q_2)$ (1.33 at $Q_2 = 64$) high enough for

the RAC algorithm to drive Q_2 down to 16. Therefore 2-view-RAC only has 2% performance gain over 2-view-nr.

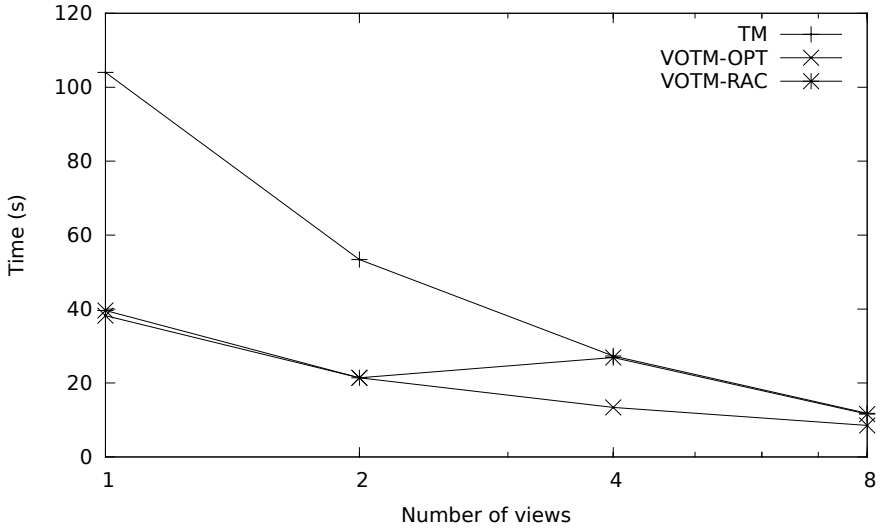

Figure 5.10: MultiRBTree in VOTM-NOrec

Table 5.11: MultiRBTree in VOTM-NOrec (In all RAC cases, all views settled to the same Q)

version	time(s)	Q	#cachemiss
1-view-nr	104.0	64	11.6G
1-view-OPT	38.2	8	3.61G
1-view-RAC	39.6	16	4.72G
2-view-nr	53.4	64	7.04G
2-view-OPT	21.4	16	3.67G
2-view-RAC	21.4	16	3.67G
4-view-nr	27.3	64	4.91G
4-view-OPT	13.4	32	3.81G
4-view-RAC	27.4	64	4.91G
8-view-nr	11.8	64	3.87G
8-view-OPT	8.51	32	3.50G
8-view-RAC	11.8	64	3.89G

Figure 5.10 shows that partitioning shared data alone can improve performance of

MultiRBTree, where 8-view-nr outperforms 1-view-nr by 8 times. This performance improvement is largely due to splitting the global clock contention in multiple view versions, as Table 5.11 shows that 8-view-nr cuts the number of data cache misses to 1/3.

In 1-view-RAC and 2-view-RAC, the high $score(Q)$ drive the Q down to 16, which improves the performance over the -nr versions by 170% and 100% respectively, as it further cuts the number of data cache misses to 1/3 and 1/2 respectively. However in 1-view-RAC, performance could have been further improved by setting Q to 8.

In 4-view and 8-view, the reduction of global clock contention by view partitioning reduces TM overhead to an extent that $score(Q)$ no longer triggers RAC to restrict Q, and therefore both 4-view-RAC and 8-view-RAC leave Q of all views to 64. However, results from Table 5.11 show that performance can be further improved by restricting Q to 32.

The above results are consistent with the effects of relieving TM metadata contention in NOrec by splitting shared data into multiple views and lowering the access quota Q to control the validation-related contention on the global clock in NOrec.

5.7 Concluding Remarks

The above experiment has extensively shown the performance of VOTM using two implementations: VOTM-OrecEagerRedo and VOTM-NOrec. Results in this experiment have demonstrated that the performance gain of VOTM can originate from three sources: contention control using RAC, fine-grained contention control using multiple views, and contention reduction of metadata.

First, performance gain can be greatly achieved by the contention control and TM overheads reduction of RAC. For example, the Eigenbench microbenchmark illustrates that in highly-contentious situations, RAC can greatly improve its performance on VOTM-OrecEagerRedo by preventing livelocks, as explained in Section 5.2. For other applications such as Vacation and Labyrinth, when the number of processes N is raised to a sufficiently high level (such as 4096), one may also see increased contention that causes RAC to restrict the admission quota Q to reduce the contention. However, at the time of my PhD candidature, multicore machines do not have this high number of cores, and therefore this hypothesis could not be tested.

In addition, applications such as SSCA2, Vacation and MultiRBTree show that in memory intensive applications, RAC can also improve performance on VOTM-NOrec

by cutting TM overheads through restricting the admission quota Q.

Second, performance gain can be achieved by fine-grained contention control in multiple views. For example, the 2-view versions of Eigenbench can treat the different contention situations of the two views and RAC can set an optimal Q individually to each view. Therefore, restricting access to a high-contention view does not affect the access to the low-contention view.

Third, performance gain can be achieved by reducing the contention accessing the TM metadata. As explained in Section 5.2, NOrec performs worse for memory-intensive applications such as Intruder due to the high contention on the global clock. By splitting shared objects into multiple views, the contention on the global clock is reduced since each view is essentially a separate TM system that has its own global clock. If the contention on the global clock within a view is still too high, Q can be reduced to a proper value to further relieve the contention. Most applications on VOTM-NOrec have demonstrated this performance gain.

Even though VOTM has these sources of performance gain, dynamic adjustment of the quota Q according to contention levels remains a challenging issue. In applications such as Eigenbench (NOrec only) and Intruder (both TM systems), the contention appears to be low due to the low $\delta(Q)$ score, and the measured TM overheads is also low, yet results show that restricting the admission quota Q further improves the performance. This observation could not be explained by the current RAC model, and it could be due to other factors related to application-side memory overheads, including limitations of the memory bus bandwidth.

The philosophy of RAC is that there are factors that cause performance deterioration when a large number of processes access a view concurrently, and if any one of these factors exist, then the admission quota, Q, should be restricted to improve performance. Currently, the RAC model identifies application contention and TM metadata overheads as two of the factors that justify restriction of the admission quota Q. However, to further refine the RAC model, other possible factors that *justify restriction of the admission quota* such as effects of memory bandwidth, overheads of both the application and TM mechanism on concurrency will be investigated as a future work. As an emerging interesting work, VOTM implementations over Hardware Transactional Memory architectures such as Intel Haswell [14] will also be investigated, as the overheads in the Haswell HTM will probably be different from the overheads in the current STM systems which VOTM is based on, and will certainly play a role in how RAC performs.

Chapter 6

Related Work

First, this chapter presents existing parallel programming models and examines how these models deal with data races. Then this chapter explores modern concurrency control mechanisms for TM systems that optimize the concurrency control according to the contention situation of the application. Finally, this chapter gives an overview of non-blocking algorithms, which allow high concurrency on accessing shared data structures.

6.1 Programming Models

As illustrated in the Introduction, code-centric shared-memory models are prone to data race, as these models require programmers to manually arrange locks to protect atomic access of shared data. To address this problem, data-centric programming models, such as Deterministic Parallel Java, and Colorama, aim at preventing data race, rather than detecting data race after it occurs. Then this section examines these data-centric models in detail, and discusses the deterministic multithreading model Dthreads, which guarantees determinism even when there are conflicts of data accesses between processes.

6.1.1 Deterministic Parallel Java

Deterministic Parallel Java (DPJ) [10] is a data-centric shared-memory model aiming at ensuring determinism in parallel code. Based on extra information supplied through annotations, this model determines at compile-time whether it is possible for tasks to have conflicting shared data access. If the compiler is sure that it is impossible to have conflicting data access between two threads, then these tasks are allowed to

run in parallel; otherwise, they will be run sequentially. Therefore DPJ is data-race free. Since locks are not used to protect shared data at runtime, DPJ is also deadlock free.

DPJ lets programmers define regions within a class, and each region in the object must be accessed atomically. Then methods that may be run concurrently must explicitly declare its "effects" (i.e. which regions it reads from and write to), and the compiler will ensure the correctness of this annotation. This extra information helps the compiler to determine potential access conflicts between two tasks, and thus allows more concurrency. If a region is accessed by more than one task, and at least one task writes to the region, then these tasks have conflicting access, and will not be allowed to be executed concurrently. A code snippet demonstrating concurrent access of two regions is shown in Figure 6.1.

```
class Pair {
  region Fst, Snd;
  int fst in Fst;
  int snd in Snd;

  void setFst(int fst) writes Fst {
    this.fst = fst;
  }

  void setSnd(int snd) writes Snd {
    this.snd = snd;
  }

  void setBoth(int fst, int snd) {
    cobegin {
      setFst(fst); /* writes Fst */
      setSnd(snd); /* writes Snd */
    }
  }
}
```

Figure 6.1: Code snippet of a concurrent Pair class in DPJ [11]

```
class Tree<region P> {
  region L, R;
  int data in P;
  Tree<P:L> left in P:L;
  Tree<P:R> right in P:R;

  int increment() writes P:* {
    ++data; /* writes P */
    cobegin {
      /* writes P:L:* */
      if (left != NULL) left.increment();
      /* writes P:R:* */
      if (right != NULL) right.increment();
    }
  }
}
```

Figure 6.2: Code snippet of a concurrent binary search tree in DPJ using the region path list [11]

In Figure 6.1, regions are declared in the class, and each data field is assigned to a region. Then the shared data access pattern of setFst() and setSnd() are declared. Finally in setBoth(), cobegin spawns each statement inside its block as a separate task if the compiler can determine that these tasks within the cobegin block will not conflict; otherwise, these statements will be sequentially executed. All tasks must return before control leaves the cobegin block.

In recursive data structures such as lists and trees, DPJ allows recursive region subdivision using the region path list system.

In the binary search tree example shown in Figure 6.2, data belongs to the parent region P. Then the left child and the right child are put into the region path lists P:L and P:R respectively. Here, P:L denotes that L is a sub-region of P. The method increment() is annotated with writes P:*, which means it will write to the entire region P, including its sub-regions. In this case, since the compiler can satisfy that left.increment() and right.increment() access disjoint regions, so it will execute these calls concurrently. As increment() will make recursive concurrent calls in the cobegin block, it can achieve

considerable concurrency in this case.

However, in other cases such as node insertion in a tree or a graph, where access to the next node depends on result of current node (only known at runtime) and access pattern cannot be decided at compile time, execution will still fall back to sequential access and performance will be affected. Since this approach depends on compiler support, complex, fine-grained applications like list traversal may not be fully parallelized.

To address this problem, the paper of [12] extends the DPJ to allow explicitly-defined non-determinism. Here are the rules:

- A region that may be accessed non-deterministically must be marked with the keyword atomic, and similarly, a method that accesses such a region must also mark its effect on that region with atomic.
- All code sections that access atomic regions must also be annotated with an atomic block.
- cobegin_nd block, a non-deterministic version of the cobegin block, is used to concurrently execute atomic blocks that may conflict with each other.

In a cobegin_nd block, statements will only be executed concurrently if it can be satisfied that only atomic statements access common atomic regions, and non-atomic statements only access non-disjoint non-atomic regions. If this is not satisfied, the statements will be executed sequentially. In the DPJ models, atomic statements are run as transactions.

In Figure 6.3, the cobegin_nd block in MyApp::work() executes two red-black-tree insertions. Both insertion statements are marked with "atomic". The RBtree::insert() method called by these statements also declares that it has "atomic write" access to the red black tree. Since there are no non-atomic regions that are accessed by both insertion statements, the cobegin_nd block will execute both insertion statements concurrently as transactions.

```
class RBTree<atomic region P> {
  atomic region L, R;
  int data in P;
  Tree<P:L> left in P:L;
  Tree<P:R> right in P:R;
  void balance() atomic writes P:* {
    ....
  }
  void insert(int val) atomic writes P:* {
    if (val < data) {
      if (left == NULL) {
        left = new Tree<P:L>(val);
      } else {
        left.insert(val);
      }
      balance();
    } else if (val > data) {
      if (right == NULL) {
        right = new Tree<P:R>(val);
      } else {
        right.insert(val);
      }
      balance();
    }
  }
}
class MyApp {
  atomic region R;
  RBTree<R> tree in R;
  void work() {
    cobegin_nd {
      atomic tree.insert(100);
      atomic tree.insert(150);
    }
  }
}
```

Figure 6.3: Code snippet showing two tasks executed concurrently in a cobegin_nd block in DPJ

This approach provides extra concurrency. However, in many cases, it is still too restrictive for applications where tasks only access shared memory for a short time, because regions are effectively acquired throughout the entire lifetime of a task. In the case of an atomic statement (task), the transaction would also last for the entire runtime of the task, whereas in VOPP, a view will only be held for as long as necessary.

6.1.2 Colorama

Colorama is a data-centric shared memory model [20]. In Colorama, shared objects are explicitly defined as "colours", as opposed to views in VOPP, and multiple blocks of shared data can be allocated with the same colour. Under the Colorama scheme, access to data owned by colours is automatically acquired and released:

- A colour is acquired when its memory is first accessed.
- A colour is released when control leaves the scope of the colour acquisition.

A code snippet demonstrating how Colorama automatically determines the scope of a critical section is shown in Figure 6.4.

```
void foo () {
  color(A, 1024, red);  // allocate color red
  ...
  A = ......  <<<<<
  ...         ColorID_A critical section
  ...         <<<<<
}
```

Figure 6.4: Automatic critical section inference in Colorama

Like the automatic view access detection in VOPP, the automatic colour-acquiring semantics improves programmability of Colorama by relieving programmers from manually acquiring and releasing colours. However as mentioned in Chapter 3, the automatic view access detection model in VOPP also allows fine-grain control of when views are acquired and released by the view scope construct, which allows programmers to specify exactly when views are acquired and released, whereas in Colorama, a colour is always held until the end of the function scope of where the colour is acquired. Therefore the colour can be held for longer than necessary. This can affect the concurrency of the application.

In addition, Colorama would require hardware support in the form of special hardware instructions, as well as OS support in the form of system calls. Therefore the portability of Colorama is limited, whereas Maotai 3.0 can run on any architectures.

6.1.3 Dthreads

Dthreads [65] is a C / C++-based deterministic multithreading model designed as a drop-in replacement of Pthreads [72]. In Dthreads, the effects of each thread are only published at synchronization time (e.g. reaching a barrier). Therefore this approach eliminates read-write data races, because a read operation will always read the version at the last synchronization. For write-write races between threads (i.e. two threads writing to the same shared memory location), this model enforces a determined order (e.g. based on thread ID) to decide which thread will win. Therefore the end-result would still be deterministic, even in the write-write race case. This determinism makes debugging much easier, because although the result can be wrong, it is reproducible when the application is re-run in a debugger.

Like the distributed shared memory system TreadMarks [1] and Maotai 2.0, Dthreads uses the runtime mprotect() page-based system to detect access to shared data. The first write to a page by a process will trigger a page fault, and the fault handler will create a twin page to store the subsequent changes of this page in a diff. Then at synchronization time, the system will apply the diffs from all processes in a pre-determined order. Therefore if diffs from two processes have conflicts, it is pre-determined who is the winner. However, the mprotect() and the diff mechanism overhead would be considerable in memory-intensive applications, where threads access a wide range of locations (i.e. many pages), thus making these applications difficult to scale.

In Dthreads, deadlocks are eliminated by converting all locks into a global lock. This is done by requiring the thread to hold the global token when it holds any locks. However, this approach will adversely affect the application performance, as it does not allow disjoint access parallelism. Moreover, the increased contention on the global token will easily create cache contention in fine-grain applications.

6.2 Concurrency Control Models in Modern Transactional Memory Systems

As discussed in Chapter 1, TM optimistically allows concurrency in critical sections and only resolves conflicts afterwards by aborting some transactions in the conflict, but the overhead of aborted transactions is staggering when the contention is high. To address this concurrency problem, there are a number of adaptive approaches to optimize the concurrency control mechanism according to the contention: in-transactional conflict resolution, transactional scheduling and adaptive locks. This section will also discuss work on adaptive transactional memory.

6.2.1 In-Transactional Conflict Resolution

In-transaction conflict resolution aims to resolve conflicts effectively to reduce wasted work of aborted transactions. All in-transaction conflict resolution algorithms, including both encounter-time locking (DSTM [45, 89], SXM [40], TLRW [27] and McRT-STM [86]), commit-time locking (TL-2 [26] and NOrec [24]) algorithms, as well as algorithms such as Relaxed Concurrency Control [6] that attempt to resolve conflicts by converting memory accesses of conflicting transactions to a serializable schedule, resolve conflicts *within* a transaction only *after* these conflicts have been detected, but threads are still freely admitted into transactions. Therefore, aborts cannot be stemmed in high contention and work is still wasted by transactions that eventually abort, as shown in experimental results presented earlier in Chapter 4.

6.2.2 Transactional Scheduling

Transactional scheduling can control the admission of transactions when contention is high. It can prevent conflicts before they occur, therefore reducing wasted work on aborted transactions. For example, transaction scheduling algorithms such as [106] use a thread-local contention score. When a thread experiences high contention, it queues the starting transaction to a central scheduler, which will execute queued transactions serially. A similar approach is adopted in [29], except when a thread experiences high contention it uses a heuristic approach that predicts read and write sets of the starting transaction using read and write sets of previous transactions of the threads. If any address in the predicted read and write sets is being written by any other currently executing transactions, then the starting transaction will be queued to be executed

serially. Otherwise, the transaction executes immediately. This algorithm relies on heuristic prediction of what will be read/written in the starting transactions. The admission control algorithm in [2] also adopts a similar approach.

This family of transactional scheduling algorithms works orthogonally with the in-transaction conflict resolution algorithms mentioned above. They use empirical thresholds to decide contention level.

All transactional scheduling algorithms described above treat the entire TM with the same scheduling decision. Therefore, access to objects of low contention can be unreasonably restricted due to the high contention of other objects in TM. Also the statistics collected for the entire TM are not as accurate as those collected in a per-view basis.

6.2.3 Adaptive Locks

The speculative lock elision (SLE)-based model [84] was proposed to avoid unnecessary exclusive accesses in lock-based programs. An elidable lock can be acquired "speculatively" (using TM) or "non-speculatively" (using mutex). At any time, an elidable lock can be acquired speculatively by multiple threads, but only one thread can hold an elidable lock non-speculatively at any time.

The adaptive lock model in [100] has a similar approach, except a thread trying to acquire the lock in mutex mode must wait until all existing threads holding the lock in the transaction mode to finish.

Like VOTM, both SLE and adaptive lock models have separate access control on each elidable lock, to ensure restrictions placed by the system on locks with high contention will not unnecessarily affect concurrency of accessing other elidable locks with low contention. These models either allow all threads to hold the elidable lock in speculative mode, or only allow exclusive access to one thread during non-speculative (mutex) mode. However, as shown in Chapter 5, there are some cases where the optimal admission quota of a lock/view is actually between 1 and N. Therefore, the RAC scheme can achieve a superior performance by finding out the optimal admission quota to achieve the optimal concurrency rather than only choosing between the two extremes – exclusive access to one thread or admitting all threads.

6.2.4 Adaptive Transactional Memory

There are also TM systems, such as [102] by the RSTM group from the University of Rochester, that choose a TM algorithm at runtime according to the access pattern and contention situation of the transactional memory. These adaptive TM systems use machine learning methods such as decision trees and neural network to learn from a training set of microbenchmarks and TM algorithms, to create an executable adaptive policy. Then, when a real application is run, "profiles" are taken at some pre-defined events such as thread creation/destruction and consecutive aborts. These profiles are subsequently used to compare with the adaptive policies to select the best TM algorithm on the fly. For example, when the contention increases, the system can switch to a more pessimistic algorithm.

Adaptive TM is orthogonal to VOTM. It can be adopted by VOTM, where different views can have different access patterns, and therefore have different optimal TM algorithms. This area will be investigated as a future work.

6.3 Non-Blocking Algorithms

Non-blocking algorithms provide an efficient way to avoid critical sections and allow multiple processes to make progress without blocking each other. There are three classes of non-blocking algorithms: obstruction-free, lock-free and wait-free. Wait-free algorithm is a subset of lock-free algorithm, which is in turn a subset of obstruction-free algorithm. An algorithm is obstruction-free if at any point after which the operation executes *in isolation*, it finishes in a finite number of steps. All TM algorithms belong to this class. Obstruction-freedom only requires any partially completed operations can be aborted and rolled back. However this does not eliminate the possibility of livelocks, which happen if operations abort each other, as illustrated in encounter-time locking TM algorithms in Chapter 4. An algorithm is lock-free if it guarantees that infinitely often *some* operations finish in a finite number of steps [35, 47]. Lock-free algorithms admit the possibility of some threads to starve, but guarantee that *some* operations in the application will complete in finite time. This is different from the lock-based approach, where a process will block until the contending location is released by other processes. In addition to all requirements of lock-freedom, wait-free algorithms also guarantee that *all operations* must finish in a finite number of steps [47].

Fault tolerance is another advantage of lock-free and wait-free algorithms. When a process terminates while holding a shared object, other processes can grab the object

and continue, and in wait-free algorithms, other processes will "help" the terminated process to complete its tasks, thus ensuring the progress. However in locking algorithms, if a process terminates while holding a lock, then other processes waiting for the lock will be blocked forever.

Currently, most architectures, including x86 and amd64, have hardware support of compare-and-swap (CAS) on a single word. CAS allows efficient single-word atomic updates, but it cannot detect the case where a word is concurrently changed to a new value and then restored to the original value. This problem is known as the "ABA problem". For this purpose, another primitive load-linked/store conditional (LL/SC) is used for algorithms that requires atomic updates on a word that is safe from the ABA problem [47]. However, only a limited number of architectures provide hardware LL/SC support, such as Alpha, PowerPC, MIPS and ARM. Multiple-CAS (MCAS) allows compare-and-swap of multiple locations. If all listed locations are not updated by other processes, then MCAS can atomically update all listed locations with their new values, otherwise the values of the locations will not be changed. MCAS is very handy for atomically accessing a shared data structure, such as a tree, where multiple locations need to be atomically updated. Although none of the commercially-available architectures have hardware MCAS support, there are software-based MCAS implementations such as [5, 43, 95] that utilize the hardware CAS.

In operations that atomically update a shared data structure, lock-free algorithms often enjoy superior performance over both lock-based and TM-based approaches, because lock-free algorithms enjoy disjoint access parallelism, which means operations that access disjoint data structures are *not* serialized, whereas in many TM algorithms such as TL2 [26], TinySTM [33] and NOrec [24], transactions are serialized by the global clock, which causes considerable cache contention when the number of transactions are high. Also in lock-based models, if the lock assignment is not sufficiently fine-grained, operations can also be unnecessarily serialized by the lock.

However, there are some disadvantages with lock-free and wait-free algorithms. First, a large amount of CPU time can be wasted in lock-free algorithms if the contention is very high, as lock-free algorithms must retry failed operations, and wait-free algorithms must do repetitive work by "helping" other threads. Second, CAS, LL/SC and MCAS cannot be directly used to replace long atomic sections in TM, as these primitives only atomically update a set of locations, whereas atomic sections may include other computation work. Also, designing lock-free data structures and algorithms requires expert knowledge in concurrent programming. It is tedious to tailor-made

lock-free data structures and algorithms for each application, and implementation of these algorithms are prone to errors, that are difficult to debug. Therefore lock-free algorithms are generally reserved for niche high-performance applications and libraries.

Chapter 7

Conclusions and Future Work

This thesis has implemented a data race prevention scheme and an automatic view access detection scheme over the VOPP paradigm, which effectively prevents data race and improves programmability of VOPP by relieving programmers from acquiring and releasing views. Experimental results demonstrated that in most applications, these schemes improve programmability of VOPP with very little overhead, and outperform shared memory models such as OpenMP as well as traditional TM systems.

However, the view access detection in the current Maotai 3.0 implementation is runtime-based, which would introduce substantial overheads from the memory protection system and the interrupt handler in fine-grained view accesses. To reduce these overheads, compile-time support on automatic view access detection would be an interesting future work.

This thesis has also proposed the novel View-Oriented Transactional Memory (VOTM) system which seamlessly integrates the merits of the locking mechanism and TM into the same programming model. VOTM allows concurrency control of each view to be individually optimized by the RAC scheme according to its own contention.

In addition, VOTM substantially improves the programmability of VOPP. Since VOTM allows concurrent access to a view, programmers are no longer required to perform fine-grained partitioning to extract concurrency, which would be tedious and prone to errors such as deadlock. For example, a tree can be put into a view, rather than partitioning each node into a separate view. In this way, excessive overheads of frequently acquiring and releasing fine-grained views can be mitigated. To enhance performance, programmers only need to put data that will not be accessed together atomically into separate views. For example, a dictionary with high contention can be put in a different view from a graph with low contention if they are not accessed

together atomically, therefore admission restriction on the dictionary will *not* hinder concurrency of processes that access the graph. In this way, concurrency control of each view can be individually optimized according to its access pattern.

In this thesis, a novel RAC scheme for VOTM has been proposed to dynamically adjust the admission quota of each view according to its contention. A theoretical model of RAC has been proposed to estimate the optimal admission quota according to the contention of the view and the TM overhead. Performance evaluations have shown that, in most cases, this RAC theoretical model can improve performance by restricting admission when the contention is high or when the TM metadata overhead is excessive.

Experimental results have shown that the performance gain of VOTM comes from:

Reduction of contention and TM overhead In high contention applications such as Eigenbench, RAC can prevent livelocks by quickly cutting the admission quota to reduce contention and ensure progress. On the other hand, when memory-intensive applications incur excessively high TM mechanism overhead, RAC can also reduce this overhead by cutting the admission quota as illustrated by the VOTM-NOrec versions of Vacation and SSCA2. As a result, VOTM-NOrec has a performance gain of 50% and 500% on Vacation and SSCA2 respectively over TM.

View partitioning allows individual optimization of admission to each view If two shared objects with different access patterns are placed into separate views, then RAC can restrict admission of the high contention view to reduce its contention, while giving unlimited access to the low contention view to maximize its concurrency. In applications such as the OrecEagerRedo version of Eigenbench, view partitioning allows RAC to set the optimal admission quota for each view, and therefore outperforms the single-view version by 200%, where RAC can only settle the overall admission quota in between the shared objects.

Reduction of TM metadata contention through view partitioning Since each view is essentially a separate TM system with its own metadata, partitioning shared memory into multiple views will also split contention in the TM metadata. The VOTM-NOrec version of multiple-view applications such as Eigenbench, Intruder and MultiRBTree highlight this performance gain by splitting data into multiple views alone, even when RAC is not used to control admission to each view.

However, performance evaluations also pointed out that in some memory-intensive cases, such as Intruder, the application itself also incurs memory overhead, that is *not* accounted for by the current RAC model, and consequently, RAC failed to determine the optimal admission quota. This overhead could be due to cacheline contention of the application data, or memory bandwidth limitations in the hardware architecture. As the number of cores in modern multicore architecture increases, these overheads will become predominant. Therefore, the impact of the memory overheads on RAC, from both the application and the TM mechanism, needs to be investigated in detail as a future work. Moreover, in addition to admission control, I will also investigate the potential benefits of applying adaptive TM algorithms (as shown in the RSTM paper [102]) in each view of VOTM to optimize performance in multiple-view applications where the optimal TM algorithm of each view is different.

In the current VOTM implementation, nested view acquisition is forbidden. This would limit the composability of VOTM, because after acquiring a view, the VOTM application may call other third-party libraries that access other views. To address this composability problem, I will investigate the automatic view partitioning scheme that automatically partition shared data into views and/or merge views at runtime according to the data access pattern. In this way, whenever two views are accessed together, these two views can be merged into a single view. In addition, when two shared objects in the same view have different access patterns, but are never accessed together, then the system can split them into separate views to allow fine-grained control of their accesses according to their own contention, thus further improve the performance. Moreover, the automatic view partitioning scheme will also relieve programmers from manually partitioning shared data into different views, thus substantially improve the programmability of VOTM.

In addition, architectures with hardware transactional memory (HTM) support, such as the IBM BlueGene/Q [53] and Intel Haswell [14] are beginning to emerge in the commercial market. Currently, these HTM architectures support HTM only at the cache level within a CPU. Since the HTM metadata is stored in the CPU cache, it can only be accessed by cores within the same CPU socket. Therefore HTM can only be supported between the cores on the same CPU socket, but TM between different CPU sockets still requires software transactional memory (STM).

Since the data owned by a view can be determined by the system due to the data-centric philosophy of VOTM, VOTM can further optimize performance by scheduling processes that access the same view onto the same CPU socket, thus avoiding the STM

overhead, and the data transfer overhead between sockets. This will be investigated as a future work once the Intel Haswell processor becomes commercially available.

Similarly, the benefits of VOTM can also be realized in distributed STM systems. In distributed STM systems, the network overhead between nodes is relatively high [92], despite the advent of high speed network architectures such as InfiniBand [76]. Therefore, performance can be greatly enhanced if processes that frequently access the same view can be scheduled onto the same computer to avoid the network overhead. Implementation and performance analysis of VOTM for distributed multi-core architectures would be an interesting future work as well.

In conclusion, this thesis has clearly demonstrated that VOPP can provide a data race free environment on shared-memory multicore architectures with little overhead, and VOTM outperforms both traditional transactional memory models and lock-based models in most benchmark applications.

References

[1] Amza, C., Cox, A., Dwarkadas, S., Keleher, P., Lu, H., Rajamony, R., Yu, W., and Zwaenepoel, W. (1996). Treadmarks: shared memory computing on networks of workstations. *Computer*, *29*(2), 18–28.

[2] Ansari, M., Kotselidis, C., Jarvis, K., Luján, M., and Watson, I. (2007). Adaptive concurrency control for transactional memory. Technical report, University of Manchester.

[3] Argonne National Laboratory and The MPI Consortium (1997). MPI-2: Extensions to the message-passing interface. Technical report, Argonne National Laboratory.

[4] Armstrong, J. (2007). *Programming Erlang, Software for a Concurrent World*. The Pragmatic Programmers.

[5] Attiya, H. and Hillel, E. (2008). Highly-concurrent multi-word synchronization. In *Proceedings of the 9th international conference on Distributed computing and networking*, ICDCN'08, Berlin, Heidelberg, 112–123. Springer-Verlag.

[6] Aydonat, U. and Abdelrahman, T. S. (2012). Relaxed concurrency control in software transactional memory. *IEEE Transactions on Parallel and Distributed Systems*, *23*, 1312–1325.

[7] Ayguadé, E., Copty, N., Duran, A., Hoeflinger, J., Lin, Y., Massaioli, F., Teruel, X., Unnikrishnan, P., and Zhang, G. (2009). The design of OpenMP tasks. *IEEE Transactions on Parallel And Distributed Systems*, *20*(3), 404–418.

[8] Bender, M. A., Fineman, J. T., Gilbert, S., and Leiserson, C. E. (2004). On-the-fly maintenance of series-parallel relationships in fork-join multithreaded programs. In *Proceedings of the 16th ACM Symposium on Parallelism in Algorithms and Architectures*, SPAA'04.

[9] Bernstein, P. and Goodman, N. (1981). Concurrency control in distributed database systems. *ACM Computer Survey*, *13*(2), 185–221.

[10] Bocchino, R. L., Adve, V. S., Adve, S. V., and Snir, M. (2009). Parallel programming must be deterministic by default. In *First USENIX Workshop on Hot Topics in Parallelism*.

[11] Bocchino, R. L., Adve, V. S., Dig, D., Adve, S., Heumann, S., Komuravelli, R., Overbey, J., Simmons, P., Sung, H., and Vakilian, M. (2009). A type and effect system for Deterministic Parallel Java. In *The 24th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications*, Number UIUCDCS-R-2009-3032.

[12] Bocchino, Jr., R. L., Heumann, S., Honarmand, N., Adve, S. V., Adve, V. S., Welc, A., and Shpeisman, T. (2011). Safe nondeterminism in a deterministic-by-default parallel language. In *Proceedings of the 38th annual ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, POPL '11, New York, NY, USA, 535–548. ACM.

[13] Boyd-Wickizer, S., Clements, A. T., Mao, Y., Pesterev, A., Kaashoek, M. F., Morris, R., and Zeldovich, N. (2010). An analysis of Linux scalability to many cores. In *Proceedings of the 9th USENIX conference on Operating systems design and implementation*, OSDI'10, Berkeley, CA, USA, 1–8. USENIX Association.

[14] Bright, P. (2012). Transactional memory going mainstream with Intel Haswell. *Ars Technica*.

[15] Burcea, M., Steffan, J. G., and Amza, C. (2008). The potential for variable-granularity access tracking for optimistic parallelism. In *The ACM SIGPLAN Workshop on Memory Systems Performance and Correctness (MSPC'08)*.

[16] Burns, A. (1987). *Programming in Occam 2*. Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc.

[17] Burns, A. and Wellings, A. (2007). *Concurrent and Real-Time Programming in Ada* (3rd ed.). New York, NY, USA: Cambridge University Press.

[18] Cao Minh, C., Chung, J., Kozyrakis, C., and Olukotun, K. (2008). STAMP: Stanford transactional applications for multi-processing. In *Proceedings of The IEEE International Symposium on Workload Characterization*.

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

[20] Ceze, L., Montesinos, P., von Praun, C., and Torrellas, J. (2007). Colorama: Architectural support for data-centric synchronization. In *Proceedings of the 13th International Symposium on High-Performance Computer Architecture*, 133–134.

[21] Chamberlain, B., Callahan, D., and Zima, H. (2007). Parallel programmability and the chapel language. *Int. J. High Perform. Comput. Appl.*, *21*(3), 291–312.

[22] Choi, J.-D., Miller, B. P., and Netzer, R. H. B. (1991). Techniques for debugging parallel programs with flowback analysis. *ACM Transactions on Programming Languages and Systems*, *13*, 491–530.

[23] Cilk Arts Inc. (2008). *Cilk++ User Guide*. Cilk Arts Inc.

[24] Dalessandro, L., Spear, M. F., and Scott, M. L. (2010). NOrec: streamlining STM by abolishing ownership records. In *Proceedings of the 15th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, New York, NY, USA, 67–78. ACM.

[25] Dice, D., Lev, Y., Moir, M., and Nussbaum, D. (2009). Early experience with a commercial hardware transactional memory implementation. In *Proceeding of the 14th International Conference on Architectural Support for Programming Languages and Operating Systems*, 157–168. ACM.

[26] Dice, D., Shalev, O., and Shavit, N. (2006). Transactional locking II. In *Proceedings of the 20th International Symposium on Distributed Computing*.

[27] Dice, D. and Shavit, N. (2010). TLRW: return of the read-write lock. In *Proceedings of the 22nd ACM Symposium on Parallelism in Algorithms and Architectures*, SPAA '10, New York, NY, USA, 284–293. ACM.

[28] Dining, A. and Schonberg, E. (1991). An empirical comparison of monitoring algorithms for access anomaly detection. In *The 2nd ACM SIGPLANS Symposium on Principles & Practice of Parallel Programming (PPoPP)*, 1–10.

[29] Dragojević, A., Guerraoui, R., Singh, A. V., and Singh, V. (2009). Preventing versus curing: avoiding conflicts in transactional memories. In *Proceedings of the*

28th ACM Symposium on Principles of Distributed Computing, New York, NY, USA, 7–16. ACM.

[30] El-Ghazawi, T., Carlson, W., Sterling, T., and Velick, K. *UPC : distributed shared memory programming*. El-Ghazawi, Tarek and Carlson, William and Sterling, Thomas and Velick, Katherine.

[31] Emrath, P. A., Ghosh, S., and Padua, D. A. (1991). Event synchronization analysis for debugging parallel programs. In *Supercomputing'91*, 580–588.

[32] Ennals, R. (2006). Software transactional memory should not be obstruction-free. Technical report, Intel Corporation.

[33] Felber, P., Fetzer, C., and Riegel, T. (2008). Dynamic performance tuning of word-based software transactional memory. In *Proceedings of the 13th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, New York, NY, USA, 237–246. ACM.

[34] Franke, H. and Russell, R. (2002). Fuss, futexes and furlocks: Fast userlevel locking in Linux. In *Ottawa Linux Symposium*.

[35] Fraser, K. (2004). Practical lock-freedom. Technical Report UCAM-CL-TR-579, University of Cambridge, Computer Laboratory.

[36] Geist, A., Beguelin, A., Dongarra, J., Manchek, R., and Sunderam, V. (1994). *PVM: Parallel Virtual Machine - A Users' Guide and Tutorial for Networked Parallel Computing*. MIT Press.

[37] Goetz, B. (2007). Java theory and practice: Stick a fork in it. Technical report, IBM Corporation.

[38] Griffiths, I. (2012). *Programming C# 5.0*. O'Reilly.

[39] Gropp, W., Lusk, E., and Thakur, R. (1999). *Using MPI : Portable Parallel Programming with the Message-Passing Interface* (2nd ed.). MIT Press.

[40] Guerraoui, R., Herlihy, M., and Pochon, B. (2005). Polymorphic contention management. In *Proceedings of the 19th International Symposium on Distributed Computing*, 26–29. LNCS, Springer.

[41] Guerraoui, R. and Kapalka, M. (2008). On obstruction-free transactions. In *Proceedings of the 20th ACM Symposium on Parallelism in Algorithms and Architectures*.

[42] Gupta, M. K. (2012). *Akka Essentials*. Packt Publishing.

[43] Harris, T. L., Fraser, K., and Pratt, I. A. (2002). A practical multi-word compare-and-swap operation. In *Proceedings of the 16th International Symposium on Distributed Computing*, 265–279. Springer-Verlag.

[44] Helmbold, D. P., McDowell, C. E., and Wang, J.-Z. (1990). Analyzing traces with anonymous synchronization. In *The 19th International Conference on Parallel Processing (ICPP'90)*, 1170–1177.

[45] Herlihy, M., Luchangco, V., Moir, M., and Scherer, III, W. N. (2003). Software transactional memory for dynamic-sized data structures. In *Proceedings of the 22nd annual symposium on Principles of Distributed Computing*, New York, NY, USA, 92–101. ACM.

[46] Herlihy, M. and Moss, J. E. B. (1993). Transactional memory: architectural support for lock-free data structures. *SIGARCH Computer Architecture News*, *21*, 289–300.

[47] Herlihy, M. and Shavit, N. (2008). *The Art of Multiprocessor Programming*. Morgan Kaufmann.

[48] Hong, S., Oguntebi, T., Casper, J., Bronson, N., Kozyrakis, C., and Olukotun, K. (2010). Eigenbench: A simple exploration tool for orthogonal TM characteristics. In *Proceedings of the IEEE International Symposium on Workload Characterization (IISWC'10)*, Washington, DC, USA, 1–11. IEEE Computer Society.

[49] Huang, Z. and Chen, W. (2007). Revisit of View-Oriented Parallel Programming. In *Proceedings of the 7th IEEE International Symposium on Cluster Computing and the Grid*, 801–806. IEEE Computer Society.

[50] Huang, Z., Chen, W., Purvis, M., and Zheng, W. (2006). VODCA: View-oriented, distributed, cluster-based approach to parallel computing. In *The 2006 International Workshop on DSM (DSM2006), In (CD-ROM) Proc. of the IEEE/ACM Symposium on Cluster Computing and Grid 2006 (CCGrid06)*, Singapore. IEEE Computer Society.

[51] Huang, Z., Cranefield, S., Purvis, M., and Sun, C. (2001). View-based consistency and its implementation. In *Proceedings of the First International Symposium on*

Cluster Computing and the Grid, CCGRID '01, Washington, DC, USA, 74–. IEEE Computer Society.

[52] Huang, Z., Purvis, M., and Werstein, P. (2005). Performance evaluation of View-Oriented Parallel Programming. In *Proceedings of the 34th International Conference on Parallel Processing*, Oslo, 251–258. IEEE Computer Society.

[53] IBM Corporation (2012). IBM Blue Gene/Q supercomputer delivers petascale computing for high-performance computing applications. Technical report, IBM Corporation.

[54] Jájá, J. (1992). *Introduction to Parallel Algorithms*. Addison-Wesley.

[55] Kale, L. (2012). *The Charm++ Programming Language Manual*. University of Illinois at Urbana-Champaign.

[56] Karunaratna, T. C. (2005). Nondeterminator-3: A provably good data-race detector that runs in parallel. Master's thesis, Department of Electrical Engineering and Computer Science, Massachusetts Institute of Technology.

[57] Kirk, D. B. and Hwu, W.-M. (2013). *Programming Massively Parallel Processors* (2nd ed.). San Francisco, Califonia, USA: Morgan Kaufmann.

[58] Kung, H. and Robinson, J. (1981). On the optimistic methods for concurrency control. *ACM Transactions on Database Systems*, *6*(2), 213–226.

[59] Lamport, L. (1974). A new solution of Dijkstra's concurrent programming problem. *Commun. ACM*, *17*(8), 453–455.

[60] Lamport, L. (1979). How to make a multiprocessor computer that correctly executes multiprocess programs. *IEEE Transactions on Computers*, *28*(9), 690–691.

[61] Leiserson, C. E. (1998). A minicourse in multithreaded programming. Technical report, MIT Laboratory for Computer Science.

[62] Leung, K., Chen, Y., and Huang, Z. (2012). When and how VOTM can improve performance in contention situations. In *The Fifth International Workshop on Parallel Programming Models and Systems Software for High-end Computing, in Proceedings of the 41st International Conference on Parallel Processing.*

[63] Leung, K. and Huang, Z. (2011). View-Oriented Transactional Memory. In *The Fourth International Workshop on Parallel Programming Models and Systems Software for High-end Computing, in Proceedings of the 40th International Conference on Parallel Processing*.

[64] Leung, K., Huang, Z., Huang, Q., and Werstein, P. (2009). Maotai 2.0: Data race prevention in View-Oriented Parallel Programming. In *Proceedings of the 10th International Conference on Parallel and Distributed Computing, Applications and Technologies*, 263–271. IEEE Computer Society.

[65] Liu, T., Curtsinger, C., and Berger, E. D. (2011). Dthreads: efficient deterministic multithreading. In *Proceedings of the Twenty-Third ACM Symposium on Operating Systems Principles*, SOSP '11, New York, NY, USA, 327–336. ACM.

[66] Lomet, D. B. (1977). Process structuring, synchronization, and recovery using atomic actions. In *ACM Conference on Language Design for Reliable Software*, 128–137.

[67] Marathe, V. J., Spear, M. F., Heriot, C., Acharya, A., Eisenstat, D., Scherer III, W. N., and Scott, M. L. (2006). Lowering the overhead of nonblocking software transactional memory. In *The First ACM SIGPLAN Workshop on Languages, Compilers, and Hardware Support for Transactional Computing*.

[68] Mattson, T. G. and Sanders (2005). *Patterns for Parallel Programming*. Addison-Wesley.

[69] Mellor-Crummey, J. (1991). On-the-fly detection of data races for programs with nested fork-join parallelism. In *Supercomputing'91*, 24–33.

[70] Miller, B. P. and Choi, J.-D. (1988). A mechanism for efficient debugging of parallel programs. In *Proceedings of the SIGPLAN '88 conference on Programming language design and implementation*, Volume 23, 135–144.

[71] Netzer, R. H. and Ghosh, S. (1992). Efficient race condition detection for shared-memory programs with post/wait synchronization. In *The 21st International Conference on Parallel Processing (ICPP'92)*.

[72] Nichols, B., Buttlar, D., and Farrell, J. P. (1996). *Pthreads Programming*. O'Reilly.

[73] Odersky, M., Spoon, L., and Venners, B. (2008). *Programming in Scala: A Comprehensive Step-by-step Guide* (1st ed.). USA: Artima Incorporation.

[74] OpenMP Architecture Review Board (2008). *OpenMP Application Program Interface Version 3.0*. OpenMP Architecture Review Board.

[75] Oppenheimer, D., Ganapathi, A., and Patterson, D. A. (2003). Why do internet services fail, and what can be done about it? In *Proceedings of the 4th conference on USENIX Symposium on Internet Technologies and Systems - Volume 4*, USITS'03, Berkeley, CA, USA, 1–1. USENIX Association.

[76] Pentakalos, O. (2002). *An Introduction to the InfiniBand Architecture*. O'Reilly.

[77] Peterson, G. (1981). Myths about the mutual exclusion problem. *Information Processing Letters*, *12*(3), 115–116.

[78] Pethick, M., Liddle, M., Werstein, P., and Huang, Z. (2003a). Parallelization of a backpropagation neural network on a cluster computer. In *International Conference on Parallel and Distributed Computing and Systems (PDCS 2003)*.

[79] Pethick, M., Liddle, M., Werstein, P., and Huang, Z. (2003b). Parallelization of a backpropagation neural network on a cluster computer. In *Proceedings of the 15th IASTED International Conference on Parallel and Distributed Computing and Systems*.

[80] Reinders, J. (2007). *Intel Threading Building Blocks : Outfitting C++ for Multi-core Processor Parallelism*. O'Reilly.

[81] Reinelt, G. (1995). TSPLIB95. Technical report, Institut für Angewandte Mathematik, Universität Heidelberg.

[82] Riegel, T., Felber, P., and Fetzer, C. (2006). A lazy snapshot algorithm with eager validation. In *20th International Symposium on Distributed Computing*.

[83] Rossbach, C. J., Hofmann, O. S., and Witchel, E. (2010). Is transactional programming actually easier? In *Proceedings of the 15th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPoPP '10, New York, NY, USA, 47–56. ACM.

[84] Roy, A., Hand, S., and Harris, T. (2009). A runtime system for software lock elision. In *Proceedings of the 4th ACM European Conference on Computer Systems*, New York, NY, USA, 261–274. ACM.

[85] Ruppert, J. (1995). A delaunay refinement algorithm for quality 2-dimensional mesh generation. *J. Algorithms*, *18*(3), 548–585.

[86] Saha, B., Adl-Tabatabai, A.-R., Hudson, R. L., Minh, C. C., and Hertzberg, B. (2006). McRT-STM: a high performance software transactional memory system for a multi-core runtime. In *Proceedings of the eleventh ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, New York, NY, USA, 187–197. ACM.

[87] Sakr, S. (2012). ARM's eight-core Mali GPUs promise "dramatic" boost to mobile graphics. *Engadget*.

[88] Savage, S., Burrows, M., Nelson, G., Sobalvarro, P., and Anderson, T. (1997). Eraser: A dynamic race detector for multithreaded programs. In *The 16th ACM Symposium on Operating Systems Principles (SOSP'97)*.

[89] Scherer, III, W. N. and Scott, M. L. (2005). Advanced contention management for dynamic software transactional memory. In M. K. Aguilera and J. Aspnes (Eds.), *Proceedings of the Twenty-Fourth Annual ACM Symposium on Principles of Distributed Computing*, 240–248. ACM.

[90] Spracklen, L. and Abraham, S. G. (2005). Chip multithreading: Opportunities and challenges. In *Proc. of Inter. Symp. on High-Performance Computer Architecture*, 248–252.

[91] Subramaniam, V. (2011). *Programming Concurrency on the JVM: Mastering Synchronization, STM, and Actors*. The Pragmatic Programmers.

[92] Suganuma, T., Koseki, A., Ishizaki, K., Ueda, Y., Mizuno, K., Silva, D., Komatsu, H., and Nakatani, T. (2011). Distributed and fault-tolerant execution framework for transaction processing. In *Proceedings of the 4th Annual International Conference on Systems and Storage*, SYSTOR '11, New York, NY, USA, 2:1–2:12. ACM.

[93] Summerfield, M. (2008). *Programming in Python 3: A Complete Introduction to the Python Language* (1st ed.). Addison-Wesley Professional.

[94] Sun Microsystems (2006). *OpenSPARC T1 Microarchitecture Specification*. Sun Microsystems.

[95] Sundell, H. (2009). Wait-free multi-word compare-and-swap using greedy helping and grabbing. In H. R. Arabnia (Ed.), *Proceedings of the International Conference on Parallel and Distributed Processing Techniques and Applications, PDPTA 2009, Las Vegas, Nevada, USA, July 13-17, 2009, 2 Volumes*, 494–500. CSREA Press.

[96] Supercomputing Technologies Group, MIT Laboratory for Computer Science (1998). *Cilk 5.4.6 Reference Manual*. Supercomputing Technologies Group, MIT Laboratory for Computer Science.

[97] Tanenbaum, A. and Steen, M. (2002). *Distributed Systems: Principles and Paradigms, Chapter 5*. Prentice Hall.

[98] Thomas, D., Fowler, C., and Hunt, A. (2009). *Programming Ruby 1.9: The Pragmatic Programmers' Guide* (3rd ed.). Pragmatic Bookshelf.

[99] Tim Harris, J. R. L. and Rajwar, R. (2010). *Transactional Memory* (2nd ed.). Synthesis Lectures on Computer Architecture. Morgan and Claypool.

[100] Usui, T., Behrends, R., Evans, J., and Smaragdakis, Y. (2009). Adaptive locks: Combining transactions and locks for efficient concurrency. In *Proceedings of the 18th International Conference on Parallel Architecture and Compilation Techniques*, Washington, DC, USA. IEEE Computer Society.

[101] van der Wijngaart, R. F. and Frumkin, M. (2002). NAS grid benchmarks version 1.0. Technical Report NAS-02-005, NASA Advanced Supercomputing Division, NASA Ames Research Center.

[102] Wang, Q., Kulkarni, S., Cavazos, J., and Spear, M. (2012). A transactional memory with automatic performance tuning. *ACM Trans. Archit. Code Optim.*, *8*(4), 54:1–54:23.

[103] Wilkinson, B. and Allen, M. (2005). *Parallel Programming Techniques and Applications Using Networked Workstations and Parallel Computers* (2nd ed.). Prentice Hall.

[104] Witchel, E., Cates, J., and Asanović, K. (2002). Mondrian memory protection. In *ASPLOS-X 2002*.

[105] Woo, S. C., Ohara, M., Torrie, E., Singh, J. P., and Gupta, A. (1995). The SPLASH-2 programs: Characterization and methodological considerations. In *Pro-*

ceedings of the 22nd Annual International Symposium on Computer Architecture, 24–36.

[106] Yoo, R. M. and Lee, H.-H. S. (2008). Adaptive transaction scheduling for transactional memory systems. In *Proceedings of the Twentieth Annual Symposium on Parallelism in Algorithms and Architectures*, New York, NY, USA, 169–178. ACM.

[107] Zamani, R. and Afsahi, A. (2012). A study of hardware performance monitoring counter selection in power modeling of computing systems. In *Proceedings of the 4th International Green Computing Conference*.

[108] Zhang, J., Huang, Z., Chen, W., Huang, Q., and Zheng, W. (2008). Maotai: View-Oriented Parallel Programming on CMT processors. In *Proceedings of the 37th International Conference on Parallel Processing*, 636–643.

Appendix A

Contents of the Source Code CD-ROM

The source code of all VOPP and VOTM implementations from Chapter 2–5, together with benchmark applications for each implementation, are included in the CD-ROM attached in this thesis. All implementations are tested on amd64-based shared memory multicore machines running on Linux 2.6.32 or later. To compile the source code, the following software are required:

- gcc and g++ compilers version 4.4 or later
- GNU Makefile system
- CMake 2.6 or later

The contents of the CD-ROM are as follow:

/Maotai-2.0	The Maotai 2.0 implementation presented in Chapter 2
/Maotai-3.0	The Maotai 3.0 implementation presented in Chapter 3
/TinySTM-VOTM	The TinySTM-based VOTM implementation presented in Chapter 4
/RSTM-VOTM	The RSTM-based VOTM implementation presented in Chapter 5