# ON CONTENTION MANAGEMENT FOR DATA ACCESSES IN PARALLEL AND DISTRIBUTED SYSTEMS

A Thesis
Presented to
The Academic Faculty

by

Xiao Yu

In Partial Fulfillment
of the Requirements for the Degree
Doctor of Philosophy in the
School of Electrical and Computer Engeering

Georgia Institute of Technology
May 2015

# ON CONTENTION MANAGEMENT FOR DATA ACCESSES IN PARALLEL AND DISTRIBUTED SYSTEMS

Approved by:

Professor Sudhakar Yalamanchili,
Committee Chair
School of Electrical and Computer Engeering
*Georgia Institute of Technology*

Professor George Riley
School of Electrical and Computer Engeering
*Georgia Institute of Technology*

Professor Sudhakar Yalamanchili,
Advisor
School of Electrical and Computer Engeering
*Georgia Institute of Technology*

Professor Linda Wills
School of Electrical and Computer Engeering
*Georgia Institute of Technology*

Professor Bo Hong
School of Electrical and Computer Engeering
*Georgia Institute of Technology*

Professor Karsten Schwan
Department of Computer Science
*Georgia Institute of Technology*

Date Approved: 11 December 2014

*Dedicated to*

*My parents, who support me without doubt,*

*My teachers, who inspire me by their personal examples, and*

*My friends, who contribute to my countless enjoyable moments.*

# ACKNOWLEDGEMENTS

I would like to use this opportunity to express my gratitude to the many people whom, without their support, this thesis would not have been possible.

I would like to thank Dr. Bo Hong. Dr. Hong granted me the opportunity to a joyful journey of doctoral study. He introduced me to the richness and depth of parallel computing. I am very grateful to Dr. Hong for his unwavering trust and belief in me, for sharing with me his knowledge and experience, for untiringly guiding me on the right trail and for his patience in allowing me to explore. I would also like to thank Dr. Sudhakar Yalamanchili who is extremely kind to provide me sparkling ideas and broaden my view on the thesis topics, and spent days and nights to critique on my thesis.

I would like to thank my dissertation committee members: Dr. George Riley, Dr. Karsten Schwan, and Dr. Linda Wills for taking time to serve on my thesis committee and give insightful suggestions and comments on my research.

I want to give my thanks to my fellow lab members. I would like to thank Zhengyu He who collaborated with me on my first several years of research and provided me numerous advices. I would like to thank Weiming Shi, Jiadong Wu for their kind assistance and suggestions on my research works. I also want to thank all of my friends who make my doctoral study a most pleasant experience.

Last but not least, I owe my Ph.D. degree to my dearest family: my father Pengnian Yu and my mother Huili Zhang. They sacrifice many things to give me a wonderful life; they provide me courage and strength when I face challenges; they believe in me and support me with no doubt. I give my deepest gratitude to them.

# TABLE OF CONTENTS

# LIST OF TABLES

# LIST OF FIGURES

# SUMMARY

Data access is an essential part of any program, and is especially critical to the performance of parallel computing systems. The objective of this work is to investigate factors that affect data access parallelism in parallel computing systems, and design/evaluate methods to improve such parallelism - and thereby improving the performance of corresponding parallel systems. We focus on data access contention and network resource contention in representative parallel and distributed systems, including transactional memory system, Geo-replicated transactional systems and MapReduce systems. These systems represent two widely-adopted abstractions for parallel data accesses: transaction-based and distributed-system-based. In this thesis, we present methods to analyze and mitigate the two contention issues.

We first study the data contention problem in transactional memory systems. In particular, we present a queueing-based model to evaluate the impact of data contention with respect to various system configurations and workload parameters. We further propose a profiling-based adaptive contention management approach to choose an optimal policy across different benchmarks and system platforms. We further develop several analytical models to study the design of transactional systems when they are Geo-replicated.

For the network resource contention issue, we focus on data accesses in distributed systems and study opportunities to improve upon the current state-of-art MapReduce systems. We extend the system to better support map task locality for dual-map-input applications. We also study a strategy that groups input blocks within a few racks to balance the locality of map and reduce tasks. Experiments show that both

mechanisms significantly reduce off-rack data communication and thus alleviate the resource contention on top-rack switch and reduce job execution time.

In this thesis, we show that both the data contention and the network resource contention issues are key to the performance of transactional and distributed data access abstraction and our mechanisms to estimate and mitigate such problems are effective. We expect our approaches to provide useful insight on future development and research for similar data access abstractions and distributed systems.

# CHAPTER I

# INTRODUCTION

Parallel and distributed computing systems have become indispensable for our modern information infrastructure. This trend is driven by the ever-growing availability of computing resources thanks to the decreasing prices of hardware and the cloud technology which provides computing resources as a service. Given the potential abundance in available computation power, it is essential to fully exploit the parallelism in applications to effectively harness such power. Data access is a critical part in many applications. After all, programs rely on data accesses (read and write) to feed computation. Accordingly, parallelism in data access is one of the most significant aspect in parallel and distributed computer systems. In this dissertation, we study some of the parallelism problems of data access in representative modern parallel and distributed systems.

Data access is the collection of read or write operations in programs. Various layers of abstraction for data access exist in hardware and software stacks to provide an easy-to-use interface and hide the details and intricacies for the upper-layer users. For example, in many languages such as C and Java, built-in read/write functions are provided for various storage devices such as disk and network; the clients call these functions without having to know the internal details such as operating system pages, disk head movement or network protocol. In the context of parallel and distributed systems, two types of abstractions are widely-used and extensively studied:

- Transaction-based Abstraction, which ensures operations (especially data access) inside a transaction comply to a (sub)set of properties: Atomicity, Consistency, Isolation and Durability(ACID). Read, write, and computation are allowed within a transaction. Transactions interact with the system state through reading and writing on the abstraction of shared data. The set of properties provide intuitive requirements on how a transaction interacts with the rest of the system so that upper-level users can logically reason about the program. For example, the intermediate modification of shared data inside a transaction should not be visible to other transactions; concurrent transactions modifying the same set of shared data should appears in some sequential order for the user. With such abstraction, the programmability of many parallel applications can be significantly improved: the programmer only need to locally consider the share-data access and mark the corresponding code inside a transaction; the system underneath the abstraction level ensures the correctness of concurrent access.

- Distributed-system-based Abstraction, which grants users illusion of data access no different than local storage while hides the actuality of a cluster of storage devices. With the growing size of applications, it is often not possible to hold the application data inside a single device. On the other hand, granting data access to multiple distributed storages brings out many complex and tedious works that upper-level users are not willing to care about. These includes maintaining meta data, providing fault tolerance and availability, and last but not least guaranteeing throughput for concurrent data access. Examples of such abstraction include distributed file systems (DFS) [50], distributed shared memory cache [76] and shuffling services in MapReduce paradigm [21]. Using all of these abstractions, upper-level users just utilize some read/write functions (such as in DFS) or iterator objects (such as in shuffling service in MapReduce)

and do not need to care about the details such as how data is streamed from different remote locations.

Note that the two abstraction types are orthogonal, i.e., a distributed-system-based abstraction can also maintain some ACID properties. To this extend, we can categorize several popular data access abstractions into three classes: transactional, distributed and transactional-distributed.

The data access abstraction maintains the functionality required by the upper user levels, e.g., the transactional properties and the distributed access through network. Furthermore, in the context of parallel systems, it takes the responsibility of exploiting the parallelism of concurrent data access as well. A poorly-designed implementation of such abstraction can greatly hinder the performance of user applications as the performance of a parallel program is limited by its sequential part, according to Amdahl's law. We identify two causes that degrades the parallelism in concurrent data access corresponding to the two abstractions mentioned above:

- Data Contention. Such degradation to parallelism is inherited in the transaction-based abstraction because the isolation property (the 'I' in ACID) often requires serializability for transaction execution: the outcome of concurrent transactions is equivalent to some sequential execution. When two transactions access the same set of shared data, there may not exist a concurrent execution that satisfies the serializability and therefore parallelism is limited. This can be illustrated by a simple example: both $T1$ and $T2$ first read $A$ and then writing $A$. Any interleave of the four operations other than the cases that the write of $T1$ happens before the read of $T2$ or the write of $T2$ happens before the read of $T1$ violates serializablity. Because such degradation is caused by multiple transactions having conflicted data access, we name it data contention.

- Network Resource Contention. The parallelism in the distributed data access

abstraction is best exploited by having the nodes accessing their local data. However, this is not always the case and cross-node transfers are often required by applications. Parallelism in such systems is therefore limited by the network resource available, i.e., performance degradation is expected when multiple streams of data access compete for a scarce network resource. Such contention is especially severe when the network architecture is a tree hierarchical topology: a cluster consists of many racks connecting to a top switch while each rack is filled with leaf compute nodes. Under such circumstances, the top-rack switch bandwidth could be a major factor limiting the overall system capability.

To study impact of data contention and network resource contention on modern parallel and distributed systems, we selected three representative systems that attracted a lot of interests in the research community in recent years, namely, transactional memory (TM), Geo-replicated transactional data store and MapReduce/Hadoop systems.

The target architecture for TM is multi-core processors with shared memory. TM is a promising replacement for locks to ease the programmability of multi-threaded applications. By adopting the transaction-based abstraction, users wrap their code in transactions and calls the TM data access functions (*tm_read* and *tm_write*) and the underlying system takes care of the parallelism issue. This liberates the programmer from using the traditional synchronization primitives (locks, barriers, etc.), which are notorious difficult to program and debug, and vulnerable to failures and faults.

Geo-replicated transactional data store, as the name suggests, places multiple replicas across several data centers in different regions to ensure a fault tolerance level that can survive data center break down. Such system adopts both transactional and distributed data access abstraction. A major difference between such system and traditional transactional systems (including TM) is the impact of the distributed-system-based abstraction, that is, the large latency of remote replica data

access because of the long distance among data centers. Such a distinct characteristic suggests different (than traditional transactional systems) contention management designs are needed.

Both transactional memory and Geo-replicated transactional data store adopts the transactional data access abstraction, in which case the major cause of parallelism degradation is data contention. Understanding the impact of data contention and finding suitable contention management strategies are thus essential to improve the performance of such systems.

The MapReduce programming model, in contrast to the above two transactional systems, eliminates the transactional data access abstraction. Applications in MapReduce runs two functions in order: map and reduce. Data access in such systems includes map input, shuffle between map and reduce and reduce output. Users define the map and reduce function; and the system provides distributed-system-based abstractions for the three phases of data access and handles the implementation detail in the background. Both data access in map input and reduce output is wrapped in a distributed file system abstraction while the system provide an iterator interface abstracting away the process of collecting data from all maps for the shuffle stage. Because of the prevalent demand of distributed data access, the scarcity of network resource and the data-intensive nature of the MapReduce applications, network resource contention becomes the major focus in improving the parallelism of MapReduce systems.

## *1.1 Problem Statement*

Given the above observation of two widely-used data access abstractions (i.e., transaction-based abstraction and distributed-system-based abstraction) and two common cause of parallelism degradation (i.e., data contention and network resource contention), this dissertation seeks to study the data access patterns and related issues in three

representative systems, i.e., transactional memory, Geo-replicated transactional data store and MapReduce systems.

Our contributions established in the dissertation aim to provide insights into the following two problems:

- What is the impact of data and network resource contention on modern parallel and distributed systems.
- What are the effective contention management methods to alleviate the mentioned two types of contentions.

## *1.2 Solution Summary*

The dissertation makes the following contributions:

- An analytical study on the impact of data contention on TM. An queueing-theory-based analytical model is adopted to evaluate the performance of TM [122].
- An adaptive approach to choose the optimal contention management policy for TM. The approach selects the best policy during runtime based on the performance history of the workload such that the system always maintains a high throughput [53].
- An analytical study on the impact of data contention and various design options to manage both transactional and distributed-system-based abstraction in Geo-replicated transactional data store [125].
- A framework to extend the MapReduce implementation to support dual-map-input applications and reduce the network resource contention for such applications [123].

- An extensive study on the impact of a grouping-block strategy to reduce the network resource contention on both map and reduce input in MapReduce systems [124].

## *1.3 Structure of the Dissertation*

The remainder of the dissertation is organized as follows: In Chapter 2, we study the data contention problem in transactional memory systems. We provide a queueing-based approach to model the impact of data contention and behavior of the system in Section 2.2. Furthermore, we introduce an adaptive contention management strategy in Section 2.3. In Chapter 3, we present a set of analytical approaches to study the impact of data contention under various design options for the Geo-replicated transactional data store. Chapter 4 discusses the network resource contention problem in MapReduce system. The framework to reduce the network contention for dual-map-input application is illustrated in Section 4.2. Section 4.3 discusses the grouping-block strategy for both map and reduce data access. Finally, in Chapter 5, we conclude the dissertation.

# CHAPTER II

# DATA CONTENTION ON TRANSACTIONAL MEMORY

Transactional memory (TM) [47] has emerged as a promising paradigm for parallel programming. It is expected to improve programmability over the traditional lock-based concurrency control mechanisms, which are known to have various issues such as vulnerability to failures and faults, and the likelihood of deadlocks. With the rapid trend shifting towards multi-core and multi-processor computing systems, intensive research efforts are being dedicated to the investigation of TM.

TM-based programs wrap the codes of critical sections inside a transaction and calls the read/write interface of the transaction-based abstraction provided by the TM implementation to access data. The implementation ensures the atomicity and isolation of transactions. Read/write operations and intermediate results may be buffered and checked to detect conflicting data access. When two transactions have conflicted data access, at least one of them is aborted and restart again by the underlying system. When a transaction aborts because of a conflict, all computations performed so far is wasted. Restart transactions can be aborted again, resulting in further waster. In a word, the performance of the program is limited by the probability of data contention and how data contention is handled under the abstraction (i.e., contention detection and resolution).

The objective of this chapter is two-folded. Firstly, We provide a theoretical model that can reveal the impact of key parameters on data contention and system performance. These key parameters include the length of transactions, transaction arrival rate, number of check points (the time points a transaction validate its read/write set), and the computing cost of transactions. Furthermore, we present an adaptive

strategy for contention management such that dynamically selecting the best policy becomes possible under various system configurations and workload specifications.

The rest of the chapter is organized as follows: Section 2.1 summarizes the background and related works. Section 2.2 introduces our analytical model for TM systems. Section 2.3 presents our adaptive contention management strategy. Section 2.4 concludes the chapter.

## *2.1 Background and Related Work*

The idea of providing hardware support for transactions originated in [65] and has since been explored in [1, 129, 121, 9]. Software-only transactional memory has recently been the focus of intensive research, and support for practical implementations is growing [88, 91, 79, 99, 48, 49, 74, 80, 57]. Schemes that mix hardware and software have also been explored in [101, 93, 16].

The three key aspects for TM designs are (1) conflict detection, (2) version management, and (3) conflict resolution [12]. **Conflict detection** decides when to examine the read/write-sets to detect conflicts, and the two popular design choices are *eager* or *lazy*. The eager option (e.g., in TinySTM [32]) attempts to detect conflict for every memory access. The lazy option (e.g., in TL2 [23]) may delay the detection to the commit phase, which has been demonstrated to be able to avoid certain conflicts [12]. **Version management** handles the storage policy for permanent and transient data copies. Similarly, the policy can be either *eager* or *lazy*. In TM systems with eager version management (e.g., TinySTM in write-through mode), new data will replace the old data in the memory and the old data will be logged. In TM systems with lazy version management (e.g., TinySTM in write-back mode and TL2), on the contrary, the old data is kept in place while the new data is logged. **Conflict resolution** means the actions to be taken when a transaction encounters a conflict. Available options are abort-self, abort-other, and back-off.

### 2.1.1 Summary of TM Systems

The first TM and HTM system was proposed by Herlihy and Moss [58]. The system utilized the existing cache and cache coherence protocol in hardware to support the transactional data access for critical sections in program. Hammond et al. [44] proposed to fundamentally change the definition of memory consistency and accordingly a entirely new hardware architecture to support transactional data access. VTM proposed by Rajwar et al. [92] broke the limitation of on-chip resources and stored transactional state information in the virtual address space which enabled the transactions to survive context switches. Log-TM [81], proposed by Moore et al., wrote new value in-place and logged old values in the main memory, unlike LTM and TCC which buffered all the intermediate results until the commit time. Yen et al. [121] further improved Log-TM by decoupling caches from HTM systems which saved hardware resources and provided convenience for virtualization. Tomic et al. [115] proposed EazyHTM combining eager conflict detection and lazy conflict resolution which gained higher throughput for the system.

The first software transactional memory system was proposed by Shavit et al. [98] with a limitation that all the input and output of a transaction to be known in advance. A dynamic STM (DSTM) was proposed by Herlihy et al. [56] which accessed memory at an object granularity. Ennals et al. [29] argued that non-blocking transactions are unnecessary and therefore proposed a design called encounter-time-locking (ETL) which attempted to gain ownership at data access; such design is closely related to eager conflict detection. On the contrary, TL2 proposed by Dice et al. [23] used a commit-time-locking (CTL) strategy that acquired locks at commit time. RSTM proposed by Marathe et al. [75] was another object-based STM system which equipped multiple types of contention managers. SwissTM proposed by Dragojevic et al. [25] contained both object- and word-based implementations; SwissTM also differentiated write-after-write (WAW) conflicts from read-after-write (RAW) conflicts.

HyTM seeks to utilize the hardware architecture to support basic contention management, such as conflict detection, to maintain performance while using software support the rest of operations, such as restart/abort transaction, to reduce hardware cost. Kuman et al. [68] started from the STM side and proposed to add additional hardware to accelerate the logging operation of STM. On the contrary, Damron et al. [20] built their system from the HTM side and fell back to STM when the transaction size exceeded the limits of hardware. Saha et al. [95] proposed to extend the instruction-set architecture to provide architectural support for STM systems. RTM proposed by Shriraman et al. [102] introduced an alert-on-update [106] architecture for shared memory programming. Baugh et al. [10] designed a system where software and hardware transactions can be executed concurrently.

### 2.1.2 Related Work on Performance Modeling of TM

Most of the previous studies evaluated the performance of TM systems through experiment based on either simulation and actual executions. Such an empirical evaluation method provided very useful insight to TM studies. Quantifying the execution of TM-based programs through an analytical model, on the other hand, is another approach to study such systems. Heindl [55] described the transactional memory system as a series of conflict detection and resolution (CDR) points where each transaction needs to access certain shared data elements that may conflict with others. The study in [55] used a simplified conflict model where two transactions accessing the same data would conflict regardless of whether they overlap in time or not. Moreover, this model studied the expected number of retries a transaction needs to perform before it commits, which was not a direct measure of execution speed. He [51] built another model to predict the mean transaction completion time of the transactions. The model in [51], however, assumed that all concurrent transactions always conflict regardless of whether the data sets overlap or not. Porter et. al. [89] developed a tool called

Syncchar which modeled the workload performance of TM. This model statically estimated $D_n$, the expected number of pair-wise conflicts assuming all $n$ transactions execute simultaneously, and assumed that transactional execution of $n$ threads would be slowed down by $D_n$ times. This model, however, did not take into consideration that conflicts are dynamic and transactions with conflicting read/write-sets may not execute simultaneously.

### 2.1.3 Related Work on Contention Management Policies

Contention management (CM), i.e., the detection and resolution of conflicts, is the major focus for TM systems. Many static resolution policies have been proposed in various systems [23, 32, 75]. Guerraoui et al. [42] proposed a framework called polymorphic contention management that allowed CM to be changed on-the-fly. There were also attempts to automatically choose a CM policy from two CM policies, such as SwissTM [26]. In SwissTM, they analyzed the suitable cases for two CM (`Suicide` and `Timestamp`), and based on the experiments, a fixed threshold of transaction size is set to decide which CM to use. Heber [54] implemented an adaptive algorithm that could automatically switch to serialize transactions when the contention level is high. They demonstrated through experiments that this adaptive method could effectively reduce the abort rate of the STM system. For adaptive contention management, Frank et al. [34] proposed a "reinforced-learning"" scheme on DSTM that uses a separate thread to profile the CMs (i.e., throughput) and poll to choose the best CM during the last execution period. The interval between selection was tuned and fixed to one second in [34]. This scheme achieved adaptation among CMs by profiling target workload at run-time. However, this strategy did not solve the problem on how to properly choose the length of adaptation interval.

## 2.2 Performance Modeling of Transactional Memory

In this section we present our analytical approach to describe the impact of various key parameters on data contention and performance of a TM system.

One of the fundamental aspects of TM design is conflict detection and resolution [12]. When we say two transactions conflict, it implies that (1) the two transactions share some data; and (2) the two transactions accessed the shared data during an overlapped time period. (For simplicity, we do not distinguish read and write accesses in our model.) These are the spatial and temporal conditions of a conflict. In our model, we quantify both aspects statistically and analyze the mean transaction completion time. Our model also takes other factors into account, which include the processing capability of the system, the rate at which transactions are issued, the processing capability demanded by the individual transactions, the overhead of implementation, etc.

Our model is based on queuing theory and the Markov Chain [40]. The system is described by the states that represent the number of active threads and transactions. Transaction arrival, commit, and abort are the events that trigger state transition. By quantifying the transition rates, we calculate the mean number of active transactions in the system, and subsequently obtain the mean transaction completion time.

Our model is validated through extensive experiments using both the widely used STAMP benchmark suite [15] and a specially designed set of micro benchmarks. To demonstrate the effectiveness of the model, we further explore the impact of two design issues using the model: (1) the frequency of conflict detection and resolution, and (2) the categorization and impact of implementation overhead. Our results show that there exists an optimal frequency to perform conflict detection and resolution, and it is directly linked to the contention level of the system. As for the impact of overhead, we observe that the most significant type of overhead is related to the number of threads, and reducing this part will significantly accelerate the execution

speeds.

Our study is expected to improve the existing models of TM [55, 51, 89] by considering both the spatial and temporal aspects of conflicts. Our model can be used to analytically evaluate the performance of TM systems without using simulation or actual executions. We expect our model to provide useful information for TM researchers and programmers to improve their TM systems.

The rest of the section is organized as follows: Section 2.2.1 provides an abstract of the computer platforms. Section 2.2.2 abstracts transactional memory systems for the model. Section 2.2.3 presents our model analysis. Section 2.2.4 presents validation experiments and our exploration on the impact of input parameters.

### 2.2.1 Abstraction of the Target Computing Platform

We assume that the target multi-processor platform consists of multiple processors that access a shared memory. Both Symmetric Multiple Processor (SMP) and the multi-core processors are example of such platforms.

We characterize the platform using its processing capability. We assume each thread receives a share of this capability in proportion to the thread's demand. Formally, if we let $c$ denote the processing capability of the platform, $r_i$ denote the demand of thread $i$', and $n$ denote the total number of threads, the actual share of computing capability $c_i$ allocated to thread $i$ is given by

$$c_i = \begin{cases} r_i & \text{if } \sum_j r_j \leq c \\ \dfrac{r_i}{\sum_j r_j} c & \text{otherwise} \end{cases} \tag{1}$$

This represents the typical resource allocation scheme in a multi-processor system. For example, CPU time slices are often evenly allocated to threads using a round-robin policy (threads in real systems may have different priorities, but for the transactional memory workload, it is reasonable to assume that all the threads have the same priority); the memory is shared by the processors where the interconnect between the

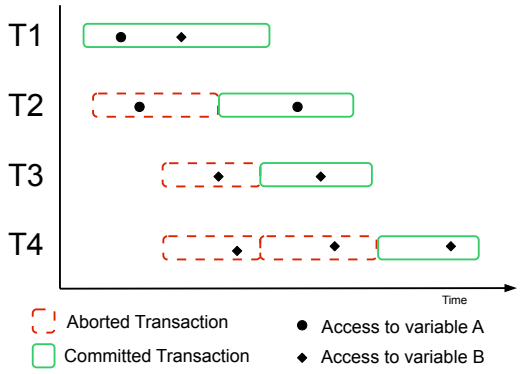



**Figure 1:** Illustration of temporal and spatial conflicts during the execution of TM-based Program

memory and processors often features fair arbitration; and for simultaneous multi-threading (SMT) architectures, the functional units in a processor are often allocated according to the needs of the threads. For such systems, it is reasonable to assume that each thread will get what it demands for when the accumulated demand is less than the system capability; and when the accumulated demand exceeds the system capability, the threads will share the processing capability in proportion to their demands.

The above abstraction also applies when the target architecture is augmented with hardware TM (HTM) support (conflict detection, rollback mechanisms, etc.). This is because HTM is typically supported on a per core/processor basis, which can be assumed to shared by the threads through time-slicing. When an HTM thread needs to access the shared memory, the memory request will be sent through the interconnect that typically features fair arbitration.

### 2.2.2 Abstraction of TM-based Programs

Figure 1 illustrates a sample execution time line of TM. Note that two transactions will conflict only if they overlap in time and access the same shared data. Without loss of generality, we assume that threads $T1$, $T2$, $T3$, and $T4$ start their transactions in the illustrated order. Thus threads $T2$, $T3$ and $T4$ have temporal conflict with $T1$.

As $T1$ successfully commits its transaction, $T2$, $T3$, and $T4$ all abort (because they accessed the same shared variables as $T1$) at certain point. The exact time of the abort depends on the specific conflict detection scheme of that thread. $T2$ detects the conflict earlier and also restarts earlier. $T2$ and $T3$ do not access the same shared variables so both will commit even though their executions overlap in time. $T4$ aborts twice, first due to its conflict with $T1$, and then due to its conflict with $T3$.

In our model, we study the following scenario abstracted from the above example:

- *Threads and Execution Modes*: We consider the scenario where the number of threads is fixed throughout the execution of the program. A thread may start a new transaction if it is not executing a transaction. We do not consider nested transaction. Therefore, if a thread is already executing a transaction, it does not start a new transaction until the current one is completed. We postpone nested transactions for our future studies.

- *Arrival of Transactions*: We assume that during any short period of time $\Delta t$, the probability that a thread starts a new transaction is $\lambda_0 \Delta t$, if the thread is not executing a transaction already. The longer the time period is, the higher the probability that a transaction may start, with a simple linear relation between the length of the time period and the probability. This abstracts the typical behavior of programs: more progress will be made in longer period of time. For a particular program that is already coded, the locations of the transactions are most likely fixed. However, from a system's point of view, it is reasonable to assume such randomness, especially when multiple threads execute different transactions. In addition, the actual execution of a program is also affected by random external events such as interrupts, which further adds to the randomness of the execution.

- *Service of Transactions*: Similar to the arrival of transactions, we assume a

simple linear relation between the length of the time period and the probability of commit. If a transaction has already started, the probability that it commits during a $\Delta t$ time period is $\mu\Delta t$ if it does not conflict with other threads. In case of conflict, the transaction may abort. This scenario is discussed below.

- *Conflicts Between Transactions*: The conflicts between the transactions are modeled through the collision of their data sets and the overlap in execution time, namely, spatial conflict and temporal conflict. To balance complexity and accuracy, we do not differentiate read and write data sets and assume that the collection of all the transactions access $D$ units of data, where each transaction needs to access $d$ out of the $D$ units of data. The per thread $d$ data elements are uniformly distributed within the union of the $D$ data elements. The probability that two transactions conflict is therefore $1 - \frac{\binom{D-d}{d}}{\binom{D}{d}}$ if the execution of the two transaction overlaps in time (note that $\frac{\binom{D-d}{d}}{\binom{D}{d}}$ is the probability of no conflict). In Figure 1, $T2$ and $T3$ both have spatial conflict with $T1$, but do not conflict with each other. $T4$ has spatial conflict with $T1$ and $T3$.

- *Conflict Detection*: We assume that threads may use either eager or lazy conflict detection. With eager detection, a transaction is able to detect conflict before it reaches the commit point. On the other hand, a transaction featuring lazy conflict detection would not be aware of any conflict until it reaches the commit point. Using either an eager or a lazy strategy, a conflict detection mechanism needs to check whether the read/write set of one transaction has been modified by another transaction. A variety of design options exist. A typical design choice is to piggyback the conflict check with regular read-/write operations within the transaction. Certain amount of overhead may be associated with the piggybacking, especially for software transactional memory. When the detection mechanisms tries to detect a conflict (and subsequently to

resolve it), we call it a conflict detection/resolution (CDR) point in the program. The lazy and eager strategies are distinguished by the number of CDR points in our model. Lazy strategy has only one CDR point (the commit point). Eager strategy has multiple CDR points (thus conflicts may be detected before a transaction committs). We assume that $k$ CDR points are evenly distributed in a transaction. The probability that a transaction hits a CDR point during $\Delta t$ time is therefore $k$ times larger than it hits the commit point. Similar to the arrival and service of transactions, we assume that the probability of a transaction reaching a CDR point in $\Delta t$ time is $k\mu\Delta t$. If transactions on average passes $E_k$ out of $k$ CDR points before detecting a conflict, the probability of a transaction reaching a *conflicting* CDR point in $\Delta t$ time is $\frac{k}{E_k}\mu\Delta t$.

- *Conflict Resolution*: We assume that a thread aborts and restarts upon detection of a conflict. This is one of the widely adopted resolution schemes. Other conflict resolution strategies have also been proposed in the literature, such as back-off [100], de-schedule the conflicting thread [11], or keep executing the conflicting thread speculatively to delay the resolution. De-scheduling is effectively equivalent to a very long back-off time. Speculation requires the hardware or the software to buffer the immediate results that would otherwise conflict with existing transactions. The benefit of speculation is still being investigated by the research community. In our model, we focus on the abort-and-restart scheme. Note that with this scheme, when two transactions conflict, the transaction that first reaches its commit point is given a higher priority to win the conflict. We plan to extend our study to the other schemes in the future.

- *Implementation overhead of TM systems*: TM systems will instrument the original multi-threaded program at either the software or the hardware level. The instrumentation requires more resources for execution and will slow down the

program, where the extent of the slow down is directly affected by the quality of the implementation. We approximates those overhead and includes them into our model using an experimental and profiling method instead of analytically (see section 2.2.4). We model the overhead as the three additive categorizes: (1) initialization overhead when starting a transaction such as lock initialization; (2) thread overhead, which increases when adding more threads; and (3) CDR overhead, which is proportional to the CDR points. In actual TM systems, the CRD overhead may be caused by multiple factors:

- `Ownership Search`: When accessing an object, a transaction needs to identify the ownership of the object. For example, under lazy detection strategy, read object need to check if it is written by the same transaction previously ; write object need to obtain the identity of the owner when being locked. The overhead of those search operations depend on the data structure of read/write log. Various data structures have been proposed in different TM systems to speed up some of those operations, e.g. TinySTM [32] proposed to store the pointer of the write entry with the lock of an address so that write lock ownership can be obtained in O(1) time.
- `Version Clock`: Many TM systems maintained a global version clock to reason about the happen-before relationship among events. This clock is read when accessing an object and updated when committing a transaction. This overhead cannot be easily analyzed because updating the global clock sometimes create a hot-spot during execution. Techniques [6, 31] have been proposed to relax the update condition.
- `Version Management`: Version management techniques focus on how to save the status of the accessed yet not committed objects. The choice of

version management usually affect the speed of commit/abort a transaction. When committing a transaction, the modified objects need to be committed under write-back strategy; when aborting a transaction, the modified objects need to fall back into the original state under write-through strategy. These operation usually takes O(W) or O(1) time in various TM systems depending on the choice of strategies, for example, committing takes O(1) time and aborting takes O(W) time under write-through strategy.

- `Conflict Resolution`: Various conflict resolution strategies have been proposed [75]. Some of the strategies made very simple decision such as aborting the offending transaction. Others make advanced decision based on the history of execution. This overhead varies largely among different strategies.
- Other overhead includes various low level overhead such as reallocating memory for read/write log and rewinding the stack when abort.

Note that researchers have also explored other aspects of TM such as nested transactions and vitalization for HTM. A counter to memorize the nesting level is a commonly used technique to deal with nesting problem [32, 25]. The outer transaction aborts when the inner transaction fails to commit. For simplicity, we do not include nested transactions in our study. As for virtualization, simple approaches such as aborting running transactions have been proposed, which works for most situations [107]. In [107], additional data structure called summary signature is added to continue isolating the memory addresses accessed in a transaction after the OS suspends a thread. Our study currently does not consider the impact of OS scheduling and context switches, and we leave the modeling of this design option to our future work.

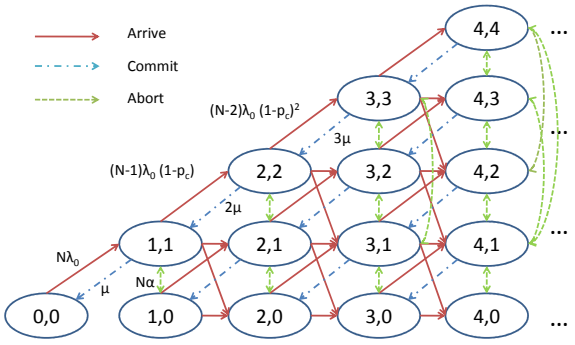



**Figure 2:** State Transition Diagram of the TM System Model

### 2.2.3 Analytical Model of TM-based Programs

In this section, we analyze the steady state probability distribution of the queuing based model, and derive the mean transaction completion time. The linear relation between the time period and the probability that a transaction arrives/finishes leads to a continuous time Markov Chain queuing model for the execution of transactions. In the queuing model, transactions issued by the threads are the clients entering the system that need to be processed by the target computing platform, which serves as the server. We first model TM programs without back-off strategy, i.e. a transaction will restart immediately after it is aborted. We will then extend the model to study back-off strategy.

#### *2.2.3.1 Model Without Back-off Strategy*

The model is illustrated in Figure 2. Each state of the system is described through a pair of parameters $(N_{tr}, N_{co})$ where $N_{tr}$ denotes the total number of transactions in the system and $N_{co}$ denotes the number of transactions that will commit. For example, at state $(3, 2)$, there are 3 active transactions in the system, 2 of which will commit (which also implies that the remaining transaction will abort). For notational purpose, we call the $N_{co}$ transactions (that will commit) the *will-commit* transactions, and the $N_{tr} - N_{co}$ transactions (that will abort) the *will-abort* transactions.

Before analyzing the transition rate between the states, we first re-list the parameters introduced in Section 2.2.2: (1) $\lambda_0$, the transaction arrival rate; (2) $\mu_0$, the basic transaction completion rate without any overhead (when the program runs with a single threads and without the TM system). The actual completion rate should be calculated according to the overhead and resource availability as in Eq. 10; (3) $k$, the average number of CDR points a transaction will encounter before it commits; (4) $d$, the size of the data set that is accessed by a transaction; (5) $D$, the size of the union of the data sets that are accessed by all the transactions; (6) $c$, the processing capability of the target computing platform; (7) $r$, the maximum processing capability each transaction demands for; (8) $N$, the total number of threads. (9) $o$, the initialization overhead factor of the TM system; (10) $a$, the CDR point overhead factor. (11) $b$, the thread overhead factor. Note that in Section 2.2.4 we explain how to estimate the value of overhead.

With the above notational preparation, we have the following calculations. The probability of a new transaction conflicting with an existing one is:

$$p_c = 1 - \frac{\binom{D-d}{d}}{\binom{D}{d}} \tag{2}$$

The average number of shared data elements accessed by $l$ transaction is:

$$n_l = D(1 - (\frac{D-d}{D})^l) \tag{3}$$

The probability of a new transaction conflicting with $l$ existing ones is:

$$p_{cl} = 1 - \frac{\binom{D-n_l}{d}}{\binom{D}{d}} \tag{4}$$

The expected size of overlapped data set between a new transaction and $l$ existing ones is:

$$n_c = \sum_{i=1}^{d} (\frac{\binom{n_l}{i}\binom{D-n_l}{d-i}}{\binom{D}{d}} i / p_{cl}) \tag{5}$$

Suppose currently the system is at state $(N_{tr}, N_{co})$, three events may occur and trigger state transitions as shown in Figure 2:

1. **Arrival of a new transaction**. For each thread that is not currently executing a transaction, a new transactions arrives at the rate of $\lambda_0$. The system-wide total arrival rate is therefore $(N - N_{tr})\lambda_0$. The arrival may cause three different state transitions:

   (a) If the new transaction does not conflict with any of the existing will-commit transactions, it will become another will-commit transaction. The probability of this transition happening within $\Delta t$ time is:

   $$p_{(N_{tr}+1,N_{co}+1)} = (N - N_{tr})(\lambda_0 \Delta t)(1 - p_c)^{N_{co}} \tag{6}$$

   The system will transit from state $(N_{tr}, N_{co})$ to state $(N_{tr} + 1, N_{co} + 1)$.

   (b) If the new transaction conflicts with $i$ $(2 \leq i \leq N_{co})$ of the existing will-commit transactions and causes them to abort (at forthcoming CDR points), the system will transit to state $(N_{tr} + 1, N_{co} + 1 - i)$. Based on the assumption in Section 2.2.2, when two transactions conflict, the resolution mechanism gives priority to the transaction that completes first. Consequently, the probability that the new transaction causes $i$ existing transactions to abort (or equivalently, the probability that this transition occurs) is

   $$\begin{aligned} p_{(N_{tr}+1,N_{co}+1-i)} &= ((N - N_{tr})\lambda_0 \Delta t)(\binom{N_{co}}{i}(1 - p_c)^{N_{co}-i} p_c^i) \\ &\quad (\int_0^\infty (1 - F(x))^i f(x) dx) \\ &= \frac{((N - N_{tr})\lambda_0 \Delta t)(\binom{N_{co}}{i}(1 - p_c)^{N_{co}-i} p_c^i)}{i + 1} \end{aligned} \tag{7}$$

   In the equation, $F(x)$ is the cumulative distribution function (CDF) of an exponential distribution. $f(x)$ is the probability density function (PDF) of an exponential distribution. The equation computes the probability

that the new transaction completes earlier than $i$ existing transactions and conflict with them.

(c) Otherwise, the new transaction will become a will-abort transaction. The system state will transit to $(N_{tr}+1,\ N_{co})$, and the probability that this transition occurs is:

$$p_{(N_{tr}+1,N_{co})} = (N - N_{tr})\lambda_0 \Delta t - p_{(N_{tr}+1,N_{co}+1)} - \sum_{i=2}^{N_{co}} p_{(N_{tr}+1,N_{co}+1-i)} \quad (8)$$

2. **Commit of a transaction.** A will-commit transaction commits. This type of event arrives at the rate of $\mu$. $\mu$ should be computed from the basic transaction completion rate $\mu_0$ according to processing capability and overhead. With the extra overhead introduced by the implementation of the TM system, $\mu$ should be computed from $\mu_0$ using

$$\mu = \frac{\mu_0}{1 + o + bN_{tr} + ak} \quad (9)$$

where $o$, $bN_{tr}$, and $ak$ are the initialization, thread number related, and CDR point related overhead respectively.

Depending on the values of $r$ and $c$, if the total demand of resources exceeds $c$, then transactions cannot proceed at the basic rate of $\mu_0$. We plug in Eq. 1 and thus convert $\mu$ to

$$\mu = \begin{cases} \dfrac{\mu_0 c}{N_{tr} r(1 + o + bN_{tr} + ak)} & \text{if } N_{tr} r(1 + o + bN_{tr} + ak) > c \\ \dfrac{\mu_0}{1 + o + bN_{tr} + ak} & \text{otherwise} \end{cases} \quad (10)$$

With the commit of the transaction, the system state transits to $(N_{tr} - 1,\ N_{co} - 1)$. The probability of this transition to occur within $\Delta t$ time is

$$p_{(N_{tr}-1,N_{co}-1)} = (N_{co}\mu\Delta t) \quad (11)$$

3. **Abort of a transaction.** A will-abort transaction checks for potential conflicts with the existing $N_{tr}$ transactions at its CDR points. We assume that the CDR points and accessed data are evenly distributed among the transactions. The average number of overlapped shared data items $n_c$ between the will-abort transaction and the others can be calculated as Eq. 5. The average number of CDR points passed before detecting one conflict data among the total number of $n_c$ is

$$E_k = \sum_{i=1}^{k} [1 - (\frac{k-i-1}{k})^{n_c}]i \tag{12}$$

The arrival rate of detecting a conflict will be

$$\alpha = \frac{k}{E_k}\mu. \tag{13}$$

If conflict is detected at the CDR point, the transaction will abort and restart. Similar to the discussion of the arrival of new transactions, the restart transaction may or may not conflict with existing transactions, and we have

(a) If the restarted transaction does not conflict with existing will-commit transactions, the system state will transit to ($N_{tr}$, $N_{co}$+1). The probability that this scenario occurs is

$$p_{(N_{tr},N_{co}+1)} = [(N_{tr} - N_{co})\alpha\Delta t](1-p_c)^{N_{co}} \tag{14}$$

(b) If the restarted transaction conflicts with $i$ existing will-commit transactions and wins the contention resolution, the system state will transit to ($N_{tr}$, $N_{co}+1-i$). The probability that this scenario occurs is

$$p_{(N_{tr},N_{co}+1-i)} = \frac{[(N_{tr} - N_{co})\alpha\Delta t](\binom{N_{co}}{i}(1-p_c)^{N_{co}-i}p_c^i)}{i+1} \tag{15}$$

**Table 1:** State transition rate when the system is at state $(N_{tr}, N_{co})$

| Destination State | Transition Rate |
|---|---|
| $(N_{tr}+1, N_{co}+1)$ | $(N-N_{tr})\lambda_0(1-p_c)^{N_{co}}$ |
| $(N_{tr}+1, N_{co}+1-i)$ | $\frac{(N-N_{tr})\lambda_0(\binom{N_{co}}{i}(1-p_c)^{N_{co}-i}p_c^i)}{i+1}$ |
| $(N_{tr}+1, N_{co})$ | $(N-N_{tr})\lambda_0(1-(1-p_c)^{N_{co}}-\sum_{i=1}^{N_{co}}\frac{\lambda_0\binom{N_{co}}{i}(1-p_c)^{N_{co}-i}p_c^i)}{i+1})$ |
| $(N_{tr}-1, N_{co}-1)$ | $N_{co}\mu$ |
| $(N_{tr}, N_{co}+1)$ | $[(N_{tr}-N_{co})\alpha](1-p_c)^{N_{co}}$ |
| $(N_{tr}, N_{co}+1-i)$ | $\frac{[(N_{tr}-N_{co})\alpha](\binom{N_{co}}{i}(1-p_c)^{N_{co}-i}p_c^i)}{i+1}$ |

(c) Otherwise, the system remains in state ($N_{tr}$, $N_{co}$), and the probability is

$$p_{(N_{tr},N_{co})} = 1-\lambda_0\Delta t - N_{co}\mu\Delta t - p_{(N_{tr},N_{co}+1)} - \sum_{i=1}^{N_{co}} p_{(N_{tr},N_{co}+1-i)} \quad (16)$$

The transition relation is summed up as Table 1: Based on the above calculation of state diagram transition probability, we can get the steady state transaction completion time. Let $\boldsymbol{\pi}$ denote the steady state probability vector $[P_{(0,0)}, P_{(1,0)}, P_{(1,1)}, P_{(2,0)}, \cdots]$. The intensity matrix $Q$ can be obtained as Eq. 17. In Eq. 17, $R_{(n_1,n_2)to(n_3,n_4)}$ is the probability rate that system state transits from $(n_1, n_2)$ to $(n_3, n_4)$. This probability rate can be obtained according to Table 1.

$$Q = \begin{pmatrix} 1-R_{(0,0)to(0,0)} & -R_{(0,0)to(1,0)} & -R_{(0,0)to(1,1)} & \cdots \\ -R_{(1,0)to(0,0)} & 1-R_{(1,0)to(1,0)} & -R_{(1,0)to(1,1)} & \cdots \\ -R_{(1,1)to(0,0)} & -R_{(1,1)to(1,0)} & 1-R_{(1,1)to(1,1)} & \cdots \\ \vdots & \vdots & \vdots & \ddots \end{pmatrix} \quad (17)$$

For the steady state probability, we have

$$\begin{cases} \boldsymbol{\pi Q} = 0 \\ \sum_{p\in\boldsymbol{\pi}} p = 1 \end{cases} \quad (18)$$

By solving Eq. 18, we can derive $\pi$, the steady-state probabilities for all states. Subsequently, the expected number of transactions in the system can be calculated as:

$E(L) = \sum_{p \in \boldsymbol{\pi}} N_{tr} p_{tr}$. The expected arrival rate of transactions is therefore $E(\lambda) = \sum_{p \in \boldsymbol{\pi}} (N - N_{tr}) p_{tr} \lambda_0$. By Little's Law, we can get the expected transaction execution time: $E(W) = \dfrac{E(L)}{E(\lambda)}$, which can be calculated numerically.

#### *2.2.3.2 Model With Back-off Strategy*

Although immediate restart is widely used by many TM designs including TinySTM and SwissTM, back-off based restart strategies have also been applied in a lot of research attentions (e.g., [11, 100]). Our model can be easily extended to describe these strategies.

By introducing a new intermediate state between transaction abort and restart, our model can describe back-off related activities. The new model is illustrated in Figure 3. To accommodate the newly introduced state, we denote the states with triplets in the form of $(N_{tr}, N_{co}, N_{rs})$. $N_{tr}$ represents the total number of transactions in the system, $N_{co}$ denotes the number of transactions that will commit. These two notations bear the same meaning as in Section 2.2.3.1. The newly introduced $N_{rs}$ denotes the number of transactions currently in back-off status (waiting to restart). As shown in Figure 3, when no transaction is in the back-off status (i.e. the $(N_{tr}, N_{co}, 0)$ states), the meaning of the states are the same as $(N_{tr}, N_{co})$ in Section 2.2.3.1. The system can enter a state with $N_{rs} \neq 0$ only when a transaction aborts.

The transition probabilities between the states can be calculated based on the transition between four events: Arrival, Commit, and Abort as previously discussed in Section 2.2.3.1, and a new event **Restart of an aborted transaction**. Assuming back-off period finishes at rate $\beta$, we have the following possible transitions:

1. **Arrival of a new transaction.** The transition probability when there is an arrival of a new transaction is similar to the model in Section 2.2.3.1 except that there are $N_{tr} + N_{rs}$ transactions in the system instead of $N_{tr}$.

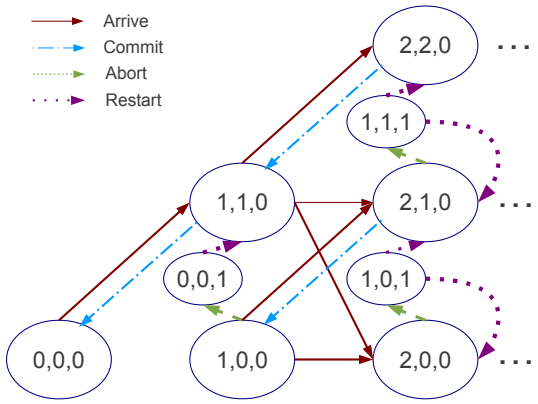



**Figure 3:** State Transition Diagram of the TM System Model with Back-off Strategy

2. **Commit of a transaction.** The probability of transition does not change when a commit event arrives.

3. **Abort of a transaction.** Aborting transactions will check for potential conflicts, abort and become a waiting-for-restart transaction instead of restart again immediately. The system therefore will transit from $(N_{tr}, N_{co}, N_{rs})$ to $(N_{tr} - 1, N_{co}, N_{rs} + 1)$. The transition rate is the same as discussed in Section 2.2.3.1.

4. **Restart of a transaction.** The transition from a back-off status to restart is at rate $\beta$.

The transition probability is summarized in Table 2. Following the same procedure in Section 2.2.3.1, we can derive the state transition matrix $\boldsymbol{Q}$, and subsequently obtain the expected transaction time.

### 2.2.4 Experimental Result

We first conducted experiments to validate our model. To demonstrate the effectiveness of the model, we then studied the impact of multiple design factors on the performance of TM.

**Table 2:** State transition rate when the system is at state $(N_{tr}, N_{co}, N_{rs})$

| Destination State | Transition Rate |
|---|---|
| $(N_{tr}+1, N_{co}+1, N_{rs})$ | $(N-N_{tr}-N_{rs})\lambda_0(1-p_c)^{N_{co}}$ |
| $(N_{tr}+1, N_{co}+1-i, N_{rs})$ | $\frac{(N-N_{tr}-N_{rs})\lambda_0(\binom{N_{co}}{i}(1-p_c)^{N_{co}-i}p_c^i)}{i+1}$ |
| $(N_{tr}+1, N_{co}, N_{rs})$ | $(N-N_{tr}-N_{rs})\lambda_0(1-(1-p_c)^{N_{co}}-\sum_{i=1}^{N_{co}}\frac{\lambda_0\binom{N_{co}}{i}(1-p_c)^{N_{co}-i}p_c^i)}{i+1})$ |
| $(N_{tr}-1, N_{co}-1, N_{rs})$ | $N_{co}\mu$ |
| $(N_{tr}-1, N_{co}+1, N_{rs}+1)$ | $(N_{tr}-N_{co})\alpha$ |
| $(N_{tr}+1, N_{co}+1, N_{rs}-1)$ | $N_{rs}\beta(1-p_c)^{N_{co}}$ |
| $(N_{tr}+1, N_{co}+1-i), N_{rs}-1$ | $\frac{N_{rs}\beta(\binom{N_{co}}{i}(1-p_c)^{N_{co}-i}p_c^i)}{i+1}$ |

We validated our model by comparing model prediction and actual execution results of STAMP benchmarks. TinySTM [32] was used as the TM system. STAMP is a widely accepted benchmark suite because it contains various real programs that reflect the typical workload for a TM system. According to its experiment results [15], the transactional execution behaviors are mainly dependent on the design options (e.g., lazy or eager) and do not depend on the implementation level (e.g., hardware or software). TinySTM is a recently published STM with very decent performance. Thus, we chose TinySTM as our sample TM system and STAMP as the sample workload.

The experiments were conducted on a system with four 6-Core Intel Xeon 2.4GHz processors. Each processor core had 32k L1 and 3MB L2 private cache, and each chip (6 cores) had a 12MB shared L3 cache. Linux kernel version 2.6.18 and GCC version 4.1.1 were used.

To obtain our model parameters, we profiled the STAMP benchmarks and the execution statistics are listed in Table 3.

- $\lambda_0/\mu_0$. The ratio represents the percentage of time that a thread spends inside and outside transactions in a real program. Let $T$ denote the percentage of time that a thread is in the transaction mode, we can calculate the ratio as

**Table 3:** Execution Characteristics of STAMP Benchmark

| | **genome** | **intruder** | **kmeans-high** | **kmeans-low** | **labyrinth** | **ssca2** | **vacation-high** | **vacation-low** | **yada** |
|---|---|---|---|---|---|---|---|---|---|
| Time in Tx(T) | 97% | 43% | 33% | 30% | 96% | 25% | 86 % | 86% | 97% |
| $\lambda_0/\mu_0$ | 32.3 | 0.75 | 0.5 | 0.4 | 24 | 0.33 | 6.1 | 6.1 | 32.3 |
| Contention | Low | High | High | High | Low | High | Low | Low | High |
| $d$ | 15 | 20 | 5 | 5 | 458 | 4 | 24 | 22 | 142 |
| $D$ | 10000 | 800 | 80 | 100 | 900000 | 100 | 10000 | 10000 | 130000 |
| Consumption | Low | High | Normal | Normal | High | High | High | High | High |
| r | 1 | 4 | 2 | 2 | 3 | 4 | 4 | 4 | 3 |
| Overhead(o) | 0.9 | 1.02 | 1.33 | 1.19 | 0.98 | 0.89 | 1.01 | 1.01 | 0.74 |
| Overhead(a) | 0 | 0.08 | 0.19 | 0.15 | 0.01 | 0.12 | 0.04 | 0.04 | 0.26 |

* **bayes** benchmark was excluded because of its non-deterministic finishing conditions as noted in [15], which made the comparison against the deterministic result generated by theoretical model less meaningful.

$$\frac{\lambda_0}{\mu_0} = \frac{T}{1-T} \tag{19}$$

We measured the time spent in transactions for the STAMP benchmark suite (listed in Table 3), which is in the range of $\lambda_0/\mu_0$ from 0.4 to 32.3.

- $d$ and $D$. $D$ is the size of the shared array and $d$ is the number of data elements accessed by each transaction. The value of $d$ is provided by [15]. The value of $D$ listed in Table 3 was estimated through a regression on the number of retries per transaction according to $d$.

- $k$. Since each access to a store data structure will issue a series of conflict checking process in TinySTM, we set the value of $k$ to to the number of protected data for each transaction.

- $N$, the total number of threads was varied between 2 and 16.

- $o$, $a$, and $b$. The three overhead factors describes the impact of TM initialization, number of CDR points, and number of threads. Because it is difficult to obtain these values theoretically, we estimated these parameters through regression analysis on experiment data. We executed STAMP by varying the number of threads and used the following simplified fitting model.

**Table 4:** Overhead Factors for the Micro benchmark, $o$, $a$ and $b$ are defined in Eq. 9

| | **o** | **a** | **b** |
|---|---|---|---|
| TinySTM | 2.05 | 0.15% | 80% |
| SwissTM | 1.91 | 0.30% | 59% |

$$y = o + bN \tag{20}$$

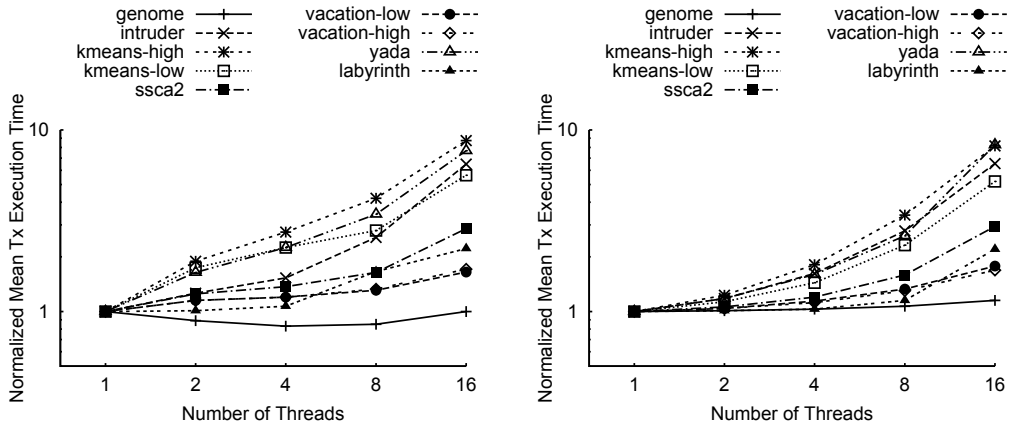
We omitted factor $a$ for this set of experiments because our later results (see Table 4) show that the impact of $a$ is insignificant compared to $o$ and $b$.

- $c$ and $r$. The ratio between $c$ and $r$ represents the processing capability of the target platform. It is affected by multiple factors such as the CPU frequency, cache size and latency etc., which makes it very difficult to measure in real programs. Because we can map all the other parameters for our model, we estimate the ratio according to a test run of a specifically designed micro benchmark (details in the next set of experiments).

We analyzed the STAMP benchmarks using our model (with the estimated parameters in Table 3). As shown in Figure 4, the analytical prediction is close to actual executions where the relative error is less than 35% with an average of 13%. This verifies our model's capability of describing the behaviors of various TM-based programs.

We further validated our model by a specially designed micro benchmark. We chose the micro benchmark method because it allows more flexible control of the design parameters than STAMP. We adjusted parameters such as the conflict rate and the number of CDR points etc. to thoroughly examine the model.

The transactions in the micro benchmark operated on a shared array where each transaction wrote to 10 randomly selected blocks in the array. The block size was fixed so that each block in the software TM system was mapped to just one lock.

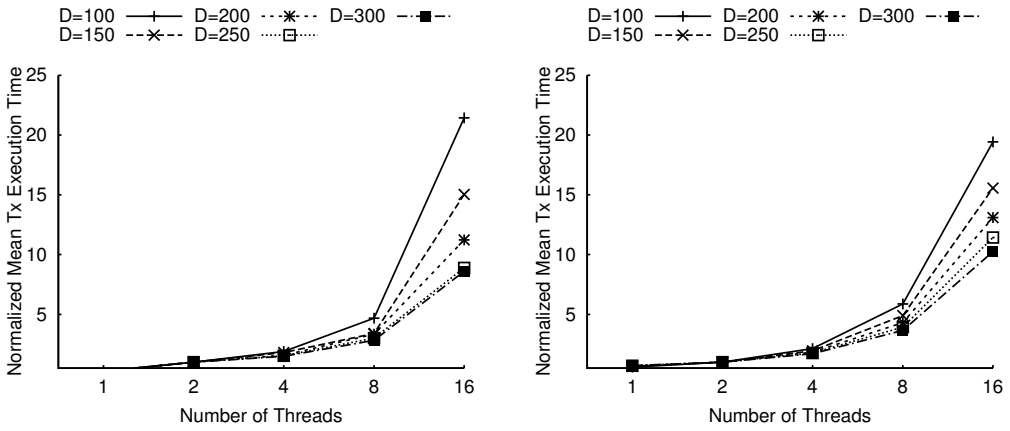



(a) Actual Execution (b) Analytical Prediction

**Figure 4:** Comparison of Experiment Execution and Model Prediction of STAMP Benchmark

Each thread executes a fixed number (10240) of transactions. The size of the array was adjusted to test the impact of transaction conflict. In addition to TinySTM, we also tested SwissTM [25], which was another state-of-the art TM systems with decent execution speeds.

Parameters in the micro benchmark of the model were set to closely reflect the scenarios of the STAMP benchmarks as listed in Table 3: (1) $\lambda_0/\mu_0$. For our micro benchmark, we set $\lambda_0/\mu_0 = 10$. Note that we can insert or remove extra instructions between transactions (that do not access the shared array) to adjust $\lambda_0/\mu_0$. (2) $d$ and $D$. We fixed $d = 10$ and varied $D$. The value $d = 10$ is typical for the STAMP benchmarks as shown in Table 3. For our micro benchmark, we varied $D$ in the range from 100 to 300. (3) $k$ was set to 10 as the number of CDR points is assumed to be equal to the number of protected data for each transaction. (4) $N$, the total number of threads was varied between 2 and 16. (5) $o$, $a$, and $b$. The parameters for TinySTM and SwissTM are listed in Table 4. The adjusted R-square of both experiments are larger than 0.9, which indicates that the fitting model has a high accuracy in describing the relationship. (6) $c$ and $r$. Those two parameters are also measured through a test run. To summarize, we set $\lambda_0/\mu_0 = 10$, $k = 10$, $d = 10$; the




(a) Actual Execution (b) Analytical Prediction

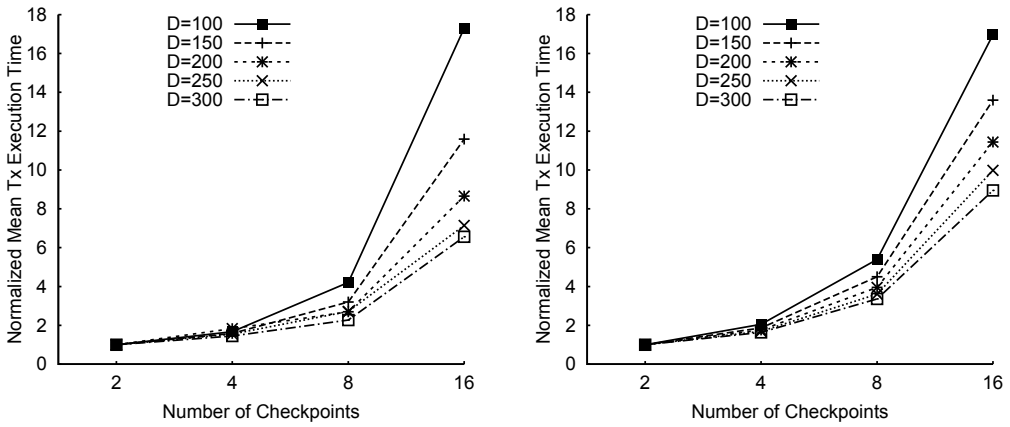
**Figure 5:** Comparison of Experiment Execution and Model Prediction on TinySTM System

overhead related parameters $o$, $a$, $b$ are set according to Table 4; $c/r$ is tested and set to be 200. We changed $D$ from 100 to 300.

Figure 5 to 6 illustrate the comparison between our model prediction and the actual execution of the micro benchmark. It can be seen that our model fits actual execution very well. The relative error between our prediction and execution result is under 30% with an average of 18%. Apart from validation of our model, this experiment also reveals a sharp rise in mean transaction completion time as the number of threads increases. Furthermore, conflict rate affects mean transaction completion time as well. With higher conflict rate, reducing conflict rate (e.g., from $D = 100$ to $D = 150$ when thread number is 16) will greatly reduce the mean transaction completion time. On the other hand, with a lower conflict rate, such as when $D = 300$, varying the conflict rate within a small range (e.g., $D$ from 300 to 250) will not significantly affect the mean transaction completion time.

To demonstrate the effectiveness of our model in the analysis of TM systems, we studied the impact of CDR points and overhead in the next two sets of experiments.

The debate between eager (early) and lazy (late) conflict detection has attracted a lot of research interests, and the number of CDR points directly affects the frequency

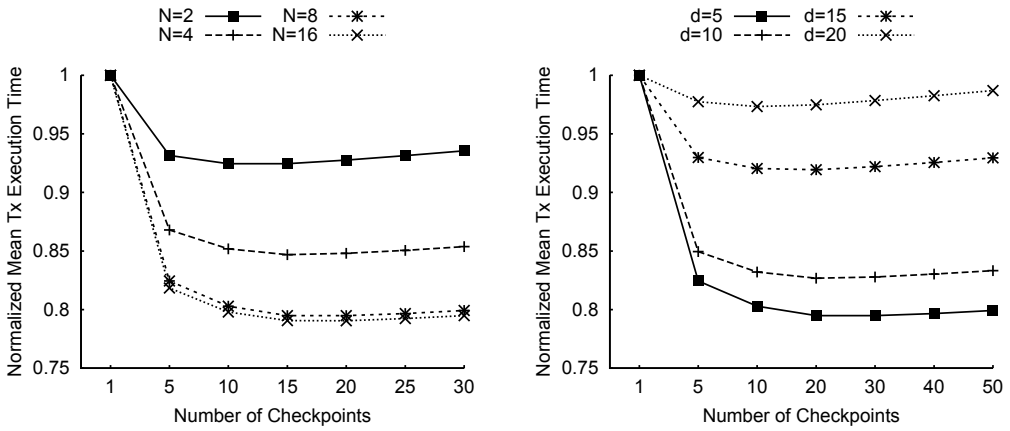



(a) Actual Execution (b) Analytical Prediction

**Figure 6:** Comparison of Experiment Execution and Model Prediction on SwissTM System

of performing early conflict detection. For this set of experiments, We set $D$ to 100, and other parameters to the same values in the validation experiments. We tested CDR points in the range between 1 and 50. Our model based analysis is presented in Figure 7(a). It shows that there exists an optimal number of CDR points. Depending on the number of threads, the optimal value of $k$ ranges from 5 to 20. The mean transaction completion time will increase when the number of CDR points deviates from the optimal value. We believe this is due to the double-sided effect of early abort: on one hand, eager conflict detection can help a transaction to abort earlier and thus make more forward progress; on the other hand, too much early abort will cause the restarted transactions to compete for resources as well. In Figure 7(b), we increase the contention level by increasing $d$. The result also shows that with lower contention level, increasing $k$ has a noticeable improvement on performance (more than 10%), while with higher contention levels, the impact of CDR point is much less significant than with low contention levels.

Table 4 shows that parameter $b$, which is related to the number of threads, dominates the overhead. We vary $b$ from 0.2 to 1.2 (TinySTM is 0.8 and SwissTM is 0.58), and number of threads from 2 to 16. The result is illustrated in Figure 8. It




(a) With Various Thread Number (b) With Various Conflict Rate

**Figure 7:** Impact of the Number of CDR points

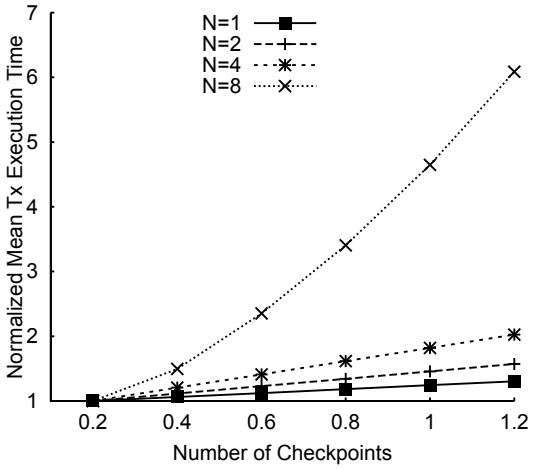



**Figure 8:** Impact of Thread-Related Overhead

shows that transactions can be completed significantly faster if the overhead can be reduced. For example, if the overhead factor $b$ for TinySTM can be reduced by 25% (from 0.8 to 0.6), the mean transaction completion time for our micro benchmark reduces by up to 35%. This result suggests that while it is crucial to explore the vast design space of TM systems (data logging, conflict detection, resolution, etc.), the quality of actual implementations is also an important factor that deserves further research. Figure 8 also reveals that the impact of $b$ is more significant when there are more threads $N$. This further exemplifies that reducing implementation overhead is

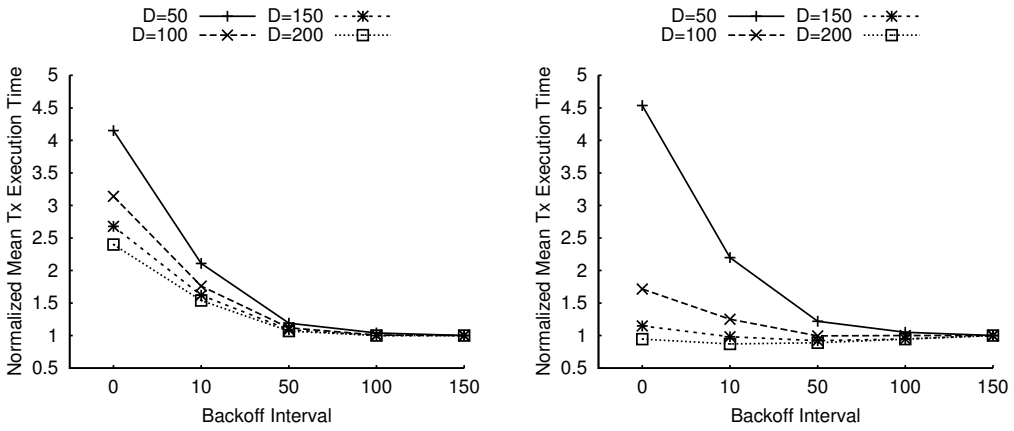
important for the scalability of TM systems.

With our extended model of back-off strategy we can study the impact of transaction back-off. Figure 9 shows the experimental results. The $x$ axis is the ratio between transaction arrival rate and transaction back-off rate $\lambda/\beta$. A larger ratio indicates a larger back-off interval. The number of threads was set to 16 in the experiments. The size of data set $D$ was set to the range from 50 to 200 to study the impact of conflict levels. The parameters $d$ and $k$ were fixed to 10. Other parameters were set the same as in the modelling of TinySTM system. Figure 9(a) shows that when the conflict level is high, the back-off strategy can reduce the mean transaction completion time significantly. However, the back-off strategy becomes less effective when the back-off interval is too long or there exists less conflicts. To evaluate the validity of our extended model, real executions were conducted and the results are shown in Figure 9(b). We used the same micro benchmark as in the previous validation experiments, i.e., each transaction in the micro benchmark operated on a shared array and wrote to 10 randomly selected blocks in the array with each thread executing a fixed number (10240) of transactions.

The back-off strategy we used in the real execution is random back-off with linear bound, which is widely used in various STM systems. Figure 9 shows that, although the actual quantitative results are different, our model based study revealed trends that is close to actual execution. The results show that our model can predict the impact of back-off intervals and conflict level.

## *2.3 Adaptive Contention Management for STM Systems*

In Section 2.2, we presented a queueing-based model to describe the impact of data contention and basic system behavior of TM systems. Yet, the analytical approach becomes less effective when used to decide an optimal implementation for contention management policy. In fact, such task is extremely complicated that even detailed




(a) Analytical Prediction (b) Actual Execution

**Figure 9:** Impact of the Conflict Rate and Back-off Time

simulation or experiment approaches become less satisfactory because of the difference in system configurations and a wide range of policy choices. In this chapter, we resort to a runtime adaptive approach to address the problem of choosing contention management policy.

Contention manager (CM) is a crucial component of STM systems. CM decides how to resolve a conflict after it is detected (e.g., abort one of the conflicting transaction) and how to avoid it from happening again (e.g., add a random back-off before restarting a new transaction). Because of its importance to the performance of STM systems, CM has received intensive research attentions and a large variety of schemes have been designed to explore the trade-offs between performance and run-time overhead [105, 96, 42, 54]. For example, a simple CM may always choose to self-abort a transaction to resolve all the conflicts. Such simple designs have low run-time overhead because the decision is pre-set. For another example, a complicated CM may favor an older transaction, which aims to preserve existing computing efforts but requires more bookkeeping and higher decision making cost. Unfortunately, as shown in Section 2.3.2, there does not exist a single CM that performs well for all the transactional workload, and the performance variation of the CMs can be significant.

More importantly, there is no general method to identify a suitable CM scheme

even when the workload is known. Given the large variety of proposed CM schemes, a natural solution would be profiling the workload with multiple CMs and then selecting the best one. However, existing STM systems do not support such automatic adaptation and require the programmers to manually perform the profiling and "hard-code" the best choice in the programs. This is against the design objective of TM — TM expects programmers to focus on determining where atomicity is necessary, rather than on the mechanisms that enforce it. The necessity for the manual profiling and selection would make TM less attractive.

We argue that adaptation is necessary and feasible for the contention management for STM systems. We demonstrate that the performance of CMs is sensitive not only to the type of workload but also to the underlying system platforms. We present an effective profiling method for the adaptation, and use it to develop an adaptive contention manager (ACM) on both TinySTM [32] and RSTM [75]. In our proposed method, we dynamically adjust two key parameters, i.e., the profiling interval and the profiling length of each CM, to reduce the profiling overhead for any type of workload and platforms. We also propose to use logic-time to measure the profiling length. The effectiveness of the proposed ACM schemes is validated through extensive experiments on two platforms (x86 and powerpc). The main contributions are as follows:

1. We propose a dynamic profiling framework that searches for and applies an optimal CM during the execution of STM workloads.

2. We propose two logic-time based methods to characterize the profiling length of each CM. Particularly, the abort-based method achieves better performance than traditional physical-time-based methods (up to 25%).

The rest of this section is organized as follows: Section 2.3.1 lists various of CM policies proposed by the research community. Section 2.3.2 justifies the necessity and feasibility of adaptation. We propose our profiling-based adaptive contention manager

in Section 2.3.3. Section 2.3.4 presents our implementation details, and Section 2.3.5 reports the experimental results that validate our new approach.

### 2.3.1 Contention Management Strategies

In an STM system, conflict resolution is handled by the CM. Three possible decisions may be made by a CM:

1. **Abort-other**: when a transaction detects that it conflicts with another transaction, it will kill the other transaction to ensure the validity of its own copy of the shared data.

2. **Abort-self**: when a transaction detects that it conflicts with another transaction, it will abort itself to ensure the data validity of the conflicting transaction.

3. **Back-off**: two types of back-off schemes exist: (1) when a transaction detects a conflict with another transaction, it stalls itself for a certain period of time, and then re-checks for data validity upon returning from the stall. (2) when a transaction aborts due to a conflict, it backs off for a period of time before it restarts. Note that other terminologies may be used to name these schemes. For example, scheme (1) is called "wait" in RSTM.

An ideal CM is expected to (1) minimize the wasted work, (2) avoid future conflicts, and (3) reduce the overhead of executing the CM itself. Because it is often difficult to achieve the three objectives simultaneously, a wide variety of CM schemes have been studied to explore the design trade-offs. We categorize CMs below based on their primary optimization objectives:

1. CMs that emphasize on minimizing the wasted work. These CMs evaluate the conflicting transactions and choose to abort the one that has performed less computation. Some CMs in this category may attempt to backoff before aborting a transaction. Example CMs include:

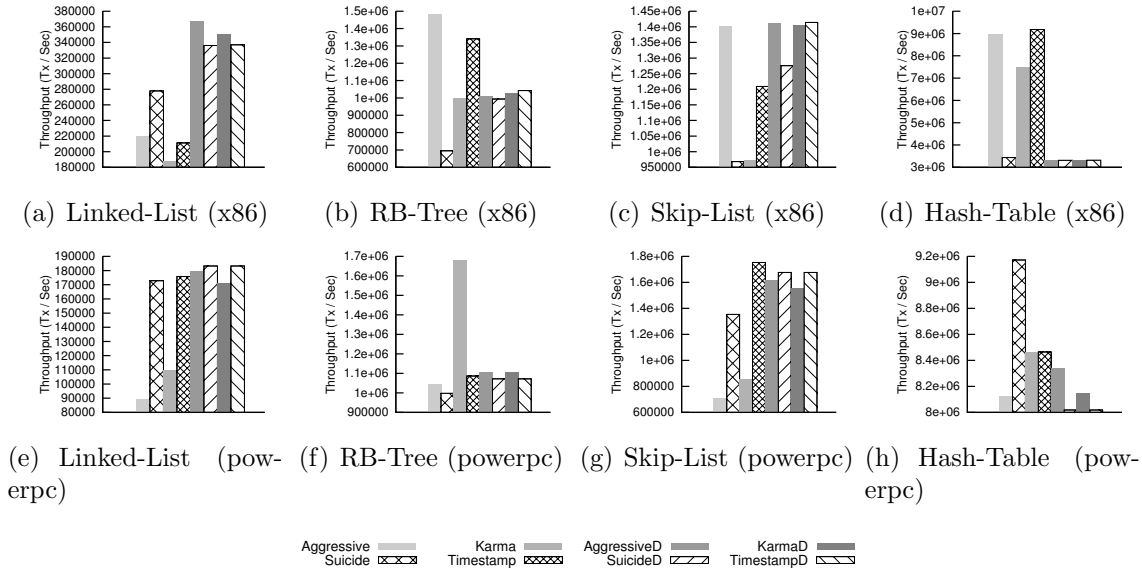
- `Timestamp`: always aborts the newer transaction. The start time can be read from the system clock (`Timestamp` in RSTM) or a globally maintained counter (`Greedy` in RSTM and `Timestamp` in TinySTM).
- `Karma`: always aborts the less-productive transaction. The productivity of a transaction can be evaluated by the size of its data set (reads and writes). Variations of `Karma` may assign more weight to writes (e.g., `Whpolka` in RSTM) or to transactions that already aborted others (e.g., `Eruption` in RSTM).

2. CMs that emphasize on reducing CM overhead. Such CMs often focus on implementation simplicity and does not perform bookkeeping. Example CMs include:

   - `Aggressive`: always aborts the other transaction.
   - `Suicide`: always aborts self (also called `Timid` in RSTM).
   - `Polite`: always exponentially backs off for a number of times, and eventually aborts the other transaction. (This CM is unavailable in TinySTM).

3. CMs that attempt to reduce future conflicts. These CMs are often derived from the above CMs and apply back-off to the transactions that were recently aborted. For example:

   - `AggressiveD`: always aborts the other transaction and asks it to back off for a certain period of time (`AggressiveD` in TinySTM asks a transaction to back off until the lock that caused the abort is released; A variation `AggressiveR` in RSTM backs off a fixed amount of time).
   - `SuicideD`: aborts self and backs off before a restart.
   - `KarmaD`: `Karma` with back-off.
   - `TimestampD`: `Timestamp` with back-off.




(a) Linked-List (x86) (b) RB-Tree (x86) (c) Skip-List (x86) (d) Hash-Table (x86)

(e) Linked-List (powerpc) (f) RB-Tree (powerpc) (g) Skip-List (powerpc) (h) Hash-Table (powerpc)

**Figure 10:** Comparison of different contention managers on different benchmarks and different platforms. (TinySTM, 16 threads)

### 2.3.2 The Necessity of Adaptive Contention Management

CM has attracted a lot of research attention because of its importance. A large set of CMs have been proposed in the literature and many STM systems are released with multiple choices of CMs. For example, the latest TinySTM 1.0.0 integrates five basic CMs and researchers can easily plug in other more complicated CMs; RSTM release 5 [75] includes a pool of over 20 CMs for programmers to choose from.

Based on different design heuristics, the CMs can be simple (e.g. `Suicide` that always causes a transaction to abort itself in case of conflict) or sophisticated (e.g., `Timestamp` that favors older transactions), where the more sophisticated ones are often designed to minimize the amount of wasted calculations. Given the variety of CMs, it is challenging, and still remains an open problem, to select the optimal CM for a given workload. This is primarily because the performance of CMs is sensitive to the workload as well as the underlying system platform.

We demonstrate the non-optimality of existing CM designs through experiments. We tested various CMs on both TinySTM [32] and RSTM [105, 96] distribution

packages on two hardware platforms. The first platform was equipped with four 2.93GHz quad-core Intel X7350 CPUs, and the other with one 3.0GHz quad-core IBM POWER7 CPUs where each core supported 4 hardware threads. We observed similar trends on both RSTM and TinySTM.

Figure 10 illustrates that the performance of CMs varies with the benchmarks as well as the system platforms.

- *benchmark dependence*. This is observed on both platforms. For example, on the x86 platform, CMs with backoff (`AggressiveD`, `SuicideD`, `KarmaD`, and `TimestampD`) outperform the CMs without back-off on Linked-List and Skip-List. `Aggressive` performs the best on RB-Tree, but under-performs on Linked-List. Similar trends can also be observed on the POWER7 platform.

- *platform dependence*. For the same benchmark program, a CM may exhibit different performance characteristics across the platforms. For example, `SuicideD` performs best for Hash-table on POWER7, but is one of the worse CMs for the same benchmark on x86.

In summary, there does not exist a static choice of CM that can guarantee optimal performance. The results show that (a) the choice of CM has a significant impact on the performance of STM system (e.g., more than 4× performance difference was observed on RB-Tree as in Figure 10(f)); (b) the optimality of CMs depends on the workload (benchmark) and platforms; and (c) choosing a fixed CM will lead to significant performance variations when the workload or platform changes. Methods to choose optimal CM include prediction through modeling or run-time profiling. Research works [55, 52] model transactional memory program behaviors and predict performance of some system configurations using abstract program specifications. Because the complication of the system and time-varying of the workload, those models are far less than enough to accurately predict the performance under different

CM strategies. Adaptive selection of an appropriate CM during run time is therefore essential to the performance of STM systems.

### 2.3.3 Profiling-based Adaptive Contention Management

To achieve adaptive contention management (ACM), we propose to periodically profile the CMs and dynamically select the one with the highest throughput. The whole process is illustrated in Figure 11. At every profiling point, each CM in the pool (the selection of the CM pool will be discussed in Section 2.3.3.3) will be switched in and run for a period of time. The throughput of a CM is calculated upon the completion of its profiling. After the throughput values of all CMs are collected, the CM with the highest throughput will be selected for subsequent execution until the next profiling point.

As shown in Figure 11, an STM program may encounter multiple profiling points during its execution. The profiling interval ($T$) controls the frequency of profiling, and the profiling length ($l_i$ for $\text{CM}_i$) affects the length of the profiling process. Profiling will inevitably cause overhead to the original program. Intuitively, the more profiling a program conducts (a smaller $T$ or larger $l_i$), the more overhead will be incurred since sub-optimal CMs will be applied more often. However, if the profiling process is not frequent enough ($T$ is too large), the ACM may not be responsive when the workload changes; furthermore, if the profiling process is not long enough ($l_i$ is too small), then the profiling results may be inaccurate and may cause the system to choose a sub-optimal CM.

Because the optimal values of $T$ and $l_i$ are workload and platforms dependent, a major challenge in designing our profiling-based ACM is the optimization of the profiling interval $T$ and profiling length $l_i$.

It is desirable to have $T \gg l_i$ since the objective of adaptation is to quickly select an optimal CM and then use it for the program execution. The consequence of

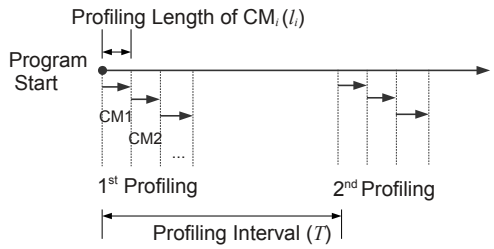



**Figure 11:** Periodic profiling process of CMs for the proposed adaptive ACM scheme

selecting a sub-optimal CM is therefore expensive. The profiling accuracy should be prioritized over the profiling overhead. Therefore, in our method, we will start from a small profiling interval $T$ that helps us quickly find for the minimum profiling length that satisfies the accuracy requirement, and then increase the profiling interval $T$ to reduce the overhead incurred by unnecessary profiling.

The profiling overhead also depends on the selection of the candidate CMs. The overhead consists of two parts: (1) the fixed overhead such as setting the system timer, switching between CMs, etc., and (2) during the profiling process, each CM will be tested for a certain period of time, sub-optimal CMs will lower the performance of the STM program. Part (1) of the cost is an implementation detail and is also related to STM designs. Experimental results suggest that this cost is marginal. Thus, we focus on part (2) of the overhead. For notational convenience, we assume that we have $k$ CMs in the pool. We use $Th_i$ to denote the profiled throughput of a CM$_i$, and $Th_{max}$ to denote the throughput of the optimal CM. We quantify the overhead by

$$O = \frac{\sum_{n=1}^{k}(Th_{max} - Th_i)l_i}{T} \tag{21}$$

which is the performance lost when profiling sub-optimal CMs.

Equation 21 indicates that the profiling overhead can be reduced by using a larger profiling interval $T$, minimizing the profiling length $l_i$ for each CM, and carefully choosing the candidates to have a small number of candidates that tend to perform well (smaller $k$ and $(Th_{max} - Th_i)$).

Next we discuss the three aspects of our design: Section 2.3.3.1 shows how we dynamically adjust profiling interval $T$ and profiling lengths $l_i$ to reduce unnecessary profiling; Section 2.3.3.2 depicts how we decide the profiling lengths to accommodate all types of workload; And Section 2.3.3.3 discusses how to choose the candidate CMs.

#### *2.3.3.1 Dynamic Adjustment of Profiling Interval and Profiling Length*

The optimal values of the profiling interval and profiling length depend on various factors, and a major one is the characteristics of the workload. For example, a high-throughput STM program would require a shorter period of time for an accurate profiling. Workload with time-varying characteristics would require more frequent adaptation. Thus, fixed profiling interval and length as in [34] is undesirable.

In our proposed method, we dynamically adjust the profiling interval and length according to the workload. The profiling interval should be adjusted to the degree of time-variance of the workload. If the workload varies fast, the profiling interval needs to be shorter to be responsive. If the behavior of the workload is stable, we should extend the profiling interval. Similarly, the profiling length also needs to be dynamically adjusted so that it is long enough to ensure the profiling accuracy, but not so long to cause unnecessary profiling overhead.

It is expensive to verify whether a profiling result is accurate or not. For example, it is possible to track the standard deviation across all the profiling results. But this will require extra storage and computation. Note that the objective of profiling is not to track the precise throughput for each CM, but to identify which CM is better than others for the current workload and platform. We can therefore tolerate some profiling errors as long as they do not affect the comparison of the CMs. To balance the accuracy and overhead, in our adaptation scheme (shown in Figure 12), we track the throughputs at two consecutive profiling points, and we consider the profiling results to be accurate if the difference is smaller than a threshold.

Figure 12 shows our algorithm for the dynamic adjustment of the profiling frequency and length. At the end of the $k^{th}$ profiling point, we compute the throughput for CM$_i$ ($Th_i^k$) and compare it with the previous results $Th_i^{k-1}$. We use $v_i$ to denote the throughput variance between two consecutive profiling for CM$_i$ (line 5). $v_i$ is compared against threshold VAR_THRES to decide if the current profiling result is accurate or not. If not, the profiling length for this CM will be doubled at next profiling point (line 6).

We also record the accumulated variance *var* to detect changes in the workload. In line 10, we compare *var* with VAR_THRES $\times$ NB_CMS (where NB_CMS denotes the number of CMs in the pool). If *var* $>$ VAR_THRES, it is indicative that the workload behavior has changed, so we will reset the profiling interval. In this case, $T$ will be reset to an initial value (INITIAL_INT = 250 $ms$ in our experiments). We are conservative in increasing the profiling interval and shrinking the profiling length, because the profiling accuracy should not be sacrificed for the overhead. Only when *var* $<$ VAR_THRES (which means none of $v_i$ is larger than VAR_THRES), we believe the profiling result has stabilized, so we double the value of $T$ and cut $l_i$ by half to reduce unnecessary profiling. In our design, $T$ will be capped by INT_BOUND (set to 4 seconds in our experiments) to maintain the responsiveness of the profiling procedure.

#### *2.3.3.2 Profiling Length*

The profiling length $l_i$ for each CM is another key design parameter. Two metrics can be used to time the profiling length: physical-time or logic-time. Frank in [34] chose to use the physical-time, which we call time-based profiling method (TPM). Instead, we can also use logic-time to measure the profiling length. In STM systems, commit and abort are two frequent and meaningful events, and are good candidates for tracking logic events. It is possible and convenient to profile each CM for a fixed

```
1: procedure ADJUST(Th_i^k)
2:     ▷ Th_i^k: the throughput of CM_i in the k^th profiling.
3:     ▷ l_i: profiling length for CM_i; T: profiling interval.
4:     var ← 0.0
5:     for all cm do
6:         v_cm ← (|Th_cm^k − Th_cm^{k−1}|) ÷ Th_cm^k
7:         if v_cm > VAR_THRES then
8:             l_cm ← l_cm × 2
9:         end if
10:        var ← var + v_cm
11:    end for
12:    if var > VAR_THRES × NB_CMS then
13:        T ← INITIAL_INT
14:    else
15:        if var < VAR_THRES and T < INT_BOUND then
16:            T ← T × 2
17:            for all cm do
18:                l_cm ← l_cm ÷ 2
19:            end for
20:        end if
21:    end if
22:    return l_i, T
23: end procedure
```

**Figure 12:** Adjustment of Profiling Interval and Profiling Length after the Profiling Ends

amount of commits/aborts instead of time ($l_i$ will be different for different CMs). We call these commit-based profiling method (CPM) and abort-based profiling method (APM) respectively. We next compare these three methods and show that APM is better than the other two.

We first quantify the performance overhead of the TPM, CPM and APM. As we showed in Equation. 21, the overhead is related to $T$, $l_i$ and $Th_i$. Let us assume TPM will profile $\text{CM}_i$ for $t_i$ seconds, thus we can replace $l_i$ to $t_i$ directly which represents the time spent by $\text{CM}_i$. The overhead of TPM can be calculated as

$$O_{TPM} = (\sum_{n=1}^{k}(Th_{max} - Th_i)t_i)/T \tag{22}$$

For CPM, we assume each CM is profiled for $C_i$ commits, therefore $l_i$ should be

expanded to $\frac{C_i}{Th_i}$, and the overhead of CPM is

$$O_{CPM} = \sum_{i=1}^{k} (\frac{C_i \cdot Th_{max}}{Th_i} - C_i)/T \tag{23}$$

Similarly, if we assume each CM is profiled for $A_i$ aborts, and the abort rate of $\text{CM}_i$ is $Ab_i$, we can calculate the overhead of APM as

$$O_{APM} = (\sum_{n=1}^{k} (Th_{max} - Th_i)\frac{A_i}{Ab_i})/T \tag{24}$$

It can be seen from the equations that $O_{TPM}$ is bounded if we set $t_i$ to a small value (compared with $T$). For $O_{CPM}$ and $O_{APM}$, because a CM may theoretically (and very rarely) take an arbitrary long period of time to commit or abort transactions, their values may be unbounded.

Although $O_{TPM}$ can be bounded, it is very difficult to set the profiling length for TPM. This is because workload may vary significantly. For example, we observed over 100× variances in transaction throughput for benchmarks in the STAMP suites [15]. Given any profiling length, say 1 second, it may be appropriate for workload A, but insufficient for workload B. Note that our adaptation scheme adjusts the profiling length automatically, but setting the initial profiling length is still a challenge for TPM. Besides, for a workload with time-varying characteristics, transaction length may vary significantly as the program executes, parameter $t_i$ has to be continuously adjusted, which will cause extra overhead (see experimental results in Section 2.3.5.3).

On the contrary, CPM and APM are both decoupled from physical-time, and do not have this drawback. Regardless of the transaction throughput of a workload, it will commit/abort transactions. We will be able to estimate the performance of the CM during the time period that certain number of commits/aborts occurred. For example, if we profile a CM for 1000 aborts, but see no commits, we can almost be sure that this CM is problematic. However, if we profile this CM for 1 second of time, and do not see any commits, we will not be able to tell whether this is a problem of

the workload (transactions are too long) or a problem of the CM. CPM and APM are therefore more flexible choices than TPM.

In practice, $O_{APM}$ rarely is unbounded. $O_{APM}$ has a $Th_{max} - Th_i$ term on the numerator. In practice, this term is small when the abort rate $Ab_i$ is small: a CM with low abort rate tends to result in high transaction commit rate. More importantly, if a CM on the contrary generates a low transaction throughput $Th_i$ and a low abort rate $Ab_i$ at the same time, then it is likely that this CM is causing the program not to commit and not to abort, which is a sign of deadlocks. However, deadlocks are guaranteed not to occur in any properly designed STM systems [47] which is actually a major advantage of STM. The overhead of APM is thus also bounded in practice.

Although properly designed STM systems can prevent deadlocks, under certain conditions, some CMs may still cause livelocks that results in a close to zero throughput $Th_i$. For example, in our experiments, `Aggressive` of RSTM with 16 threads on RB-Tree on x86 had a throughput of 44 transactions per second, while other CMs achieved more than $10^6$ transactions per second. For such cases, $O_{CPM}$ can be arbitrarily large (see experimental results in Section 2.3.5).

In terms of implementation cost, TPM is the highest. Because we only have one timer for both $t_i$ and $T$ in Linux, for each profiling process, TPM must adjust the interval of the timer at least twice for $t_i$ and $T$ respectively. Timer needs to be implemented through operating system support, which tends to be expensive. For CPM and APM, we can track the number of commits or aborts with a simple counter embedded in STM's commit or abort functions.

In summary, TPM has the advantage of bounded overhead, but it is inflexible in guaranteeing profiling accuracy and will cause higher implementation cost than CPM and APM. CPM and APM are more robust to variance in the workload with lower implementation overhead and better profiling accuracy, but CPM will be severely impacted if one of the candidate CM causes livelocks on the target workload and

platform. APM is therefore better than the other two for measuring the profiling length. Experimental comparison of the three methods are presented in Section 2.3.5.

#### *2.3.3.3 Selection of Candidate CMs*

Selecting a proper pool of candidate CMs is also important for our design. An important design parameter for the pool selection problem is the size of the pool. A larger pool increases the probability of finding a better CM, at the cost of longer profiling period as well as higher implementation cost (e.g. memory storage). Our experimental results showed that 4 to 8 are reasonable sizes.

Another important factor is in the selection of individual CMs. Our experimental results show that there are two types of CMs: (1) those that perform well on some benchmarks (e.g., `Aggressive` on HashTable), but poorly on others (e.g., `Aggressive` on RB-Tree); (2) those that perform reasonably well across all workload and platforms, but may not be the best. Type 1 CM is preferred for our ACM because we can dynamically identify suitable CMs for a given workload.

### 2.3.4 Implementation

To thoroughly test the performance of the proposed method, we built our ACM scheme on both TinySTM and RSTM. These two STM systems follow two very different design logics while both exhibiting good transactional performance.

TinySTM is a word-based STM system developed by Felber et al. [32]. It was implemented in C, and the design target was to keep the code as simple and efficient as possible. Thus, it provides less configurations than RSTM to the programmers. In its write-back-ETL mode, it supports run-time switching of CMs (but requires programmers to specify which CMs to switch). Five basic CMs including `Aggressive`, `Suicide`, `SuicideD` (they call it `Delay`), `Karma` and `Timestamp` are shipped with distribution package, but other CMs can be easily plugged in because of its modular design. Note that for the back-off decision returned by a CM, TinySTM only accept

```
1 stm_commit() {
2   ...
3   if (profiling) {
4     commits = ATOMIC_INC (&nb_commits[
      cur_CM]);
5#ifdef _CPM_
6     if (commits > max_commits[cur_CM]
      && thread_id == 0)
7       if (cur_CM == LAST_CM) {
8         profiling = False;
9         select_best_CM ();
10        adjust_prof_int_and_length ();
11        /* see Figure~\ref{fig:
      tmcm_profalgo} */
12        set_next_profiling_time (&
      timer_handler);
13      } else
14        switch_cm ();
15#endif
16  }
17  ...
18 }
```

```
20 stm_abort() {
21   ...
22#ifdef _APM_
23   if (profiling) {
24     aborts = ATOMIC_INC (&nb_aborts[
      cur_CM]);
25     if (aborts > max_aborts[cur_CM] &&
       thread_id == 0)
26       if (cur_CM == LAST_CM) {
27         profiling = False;
28         select_best_CM ();
29         adjust_prof_int_and_length ();
30         /* see Figure~\ref{fig:
      tmcm_profalgo} */
31         set_next_profiling_time (&
      timer_handler);
32       } else
33         switch_cm ();
34   }
35#endif
36   ...
37 }
```

```
39 stm_init() {
40   ...
41   cur_CM = 0;
42   reset_profiling_counters ();
43   profiling = True;
44#ifdef _TPM_
45   set_next_switch_time (&timer_handler
       );
46#endif
47   ...
48 }
```

```
49 timer_handler (){
50#ifdef _TPM_
51   if (profiling) {
52     if (cur_CM == LAST_CM) {
53       profiling = False;
54       select_best_CM ();
55       adjust_prof_int_and_length ();
56       /* see Figure~\ref{fig:
      tmcm_profalgo} */
57       set_next_profiling_time (&
      timer_handler);
58     } else
59       switch_cm ();
60   } else {
61     reset_profiling_counters ();
62     profiling = True;
63   }
64#elif defined(_APM_) || defined(_CPM_)
65   if (!profiling) {
66     reset_profiling_counters ();
67     profiling = True;
68   set_next_switch_time (&timer_handler
       );
69   }
70#endif
71 }
```

**Figure 13:** A snapshot of the added code of ACM for TinySTM

the scheme that backs off after a restart. The latest TinySTM version 1.0.0 is used in the experiments.

Differently, RSTM [75] is an object-based STM system implemented in C++. We used its release version 5. This version provides multiple choices for the configuration such as invisible-read or visible-read, lazy version management or eager version management, etc. Over 20 CMs are available in the package, though most of them are very similar. RSTM supports all three types of CM decisions including both backoff

schemes, and every CM was implemented in a separate C++ class so that RSTM is almost compatible with any CM.

Our ACM can be considered as an add-on to the original STM system. By monitoring the run-time behavior of the workload, our ACM can adaptively adjust the current CM to maximize the overall performance. Figure 13 shows a snapshot of our modifications for TinySTM. The majority of our modification is in three STM interface functions, `stm_init`, `stm_commit`, and `stm_abort`, which exist for all STM systems. We implement our ACM with all three profiling methods, TPM, CPM and APM. Our design is easy to implement with less than 200 lines of code in total.

As shown in Figure 13, we use a flag `profiling` to denote if the program is currently being profiled, `cur_CM` to denote the current CM being used, and a counter `nb_commits` for each CM to record the number of commits. For APM, we also need one additional counter `nb_aborts` for each CM to record the number of aborts. Initially, when `stm_init` is called, we reset all the counters , set the current CM to the first one in the pool (`cur_CM=0`), and then set the profiling flag to True. If TPM is used, we also need to install a timer during `stm_init`. We use `setitimer(ITIMER_REAL, ...)` to set the timer. Upon the expiration of the timer, a SIGALRM will be delivered and our installed `timer_handler` function will be called. TPM will switch the CM in `timer_handler` function. For CPM and APM, CM switching is triggered in functions `stm_commit` and `stm_abort` respectively. When profiling completes for all the CMs, the best CM will be selected and the next profiling interval and length will be adjusted (line 8-11 and line 26-29).

### 2.3.5 Experimental Results

In this section, we present the experimental results and analysis. Two architectural platforms were used in the experiments:

- **x86_64**: The system was equipped with four 2.93GHz quad-core Intel X7350 CPUs and 128GB memory. Linux kernel version was 2.6.32, and gcc version 4.4.4 was used.

- **powerpc**: The system was equipped with one 3.0GHz quad-core IBM POWER7 CPU where each core supports four simultaneous hardware threads. 4GB memory was installed in the system. Linux kernel version was 2.6.32, and gcc version 4.4.4 was used.

We implemented our ACM scheme for both TinySTM and RSTM. TinySTM supports both x86 and powerpc platforms and works in 64-bit mode. We tested it on both platforms in the 64-bit mode. RSTM release 5 is 32-bit only and does not support Linux on powerpc systems. We tested it on the x86 platform using 32-bit mode.

We selected four similar benchmarks from the TinySTM and RSTM. Linked-List, RB-Tree, Skip-List, and Hash-Table were chosen for TinySTM. Linked-List, RB-Tree, DList (Doubly Linked-List), and Hash-Table were chosen for RSTM. These benchmarks covered a broad range of typical transactional workload. They differed in transaction size, transaction issue rate as well as the conflict rate. For example, the transaction sizes were random in Linked-List, but remained almost constant in Hash-Table. We used the default configuration for both TinySTM and RSTM, and each benchmark program was executed 10 seconds for each run. All the throughput values were averaged over five runs (we observed less than 10% variance).

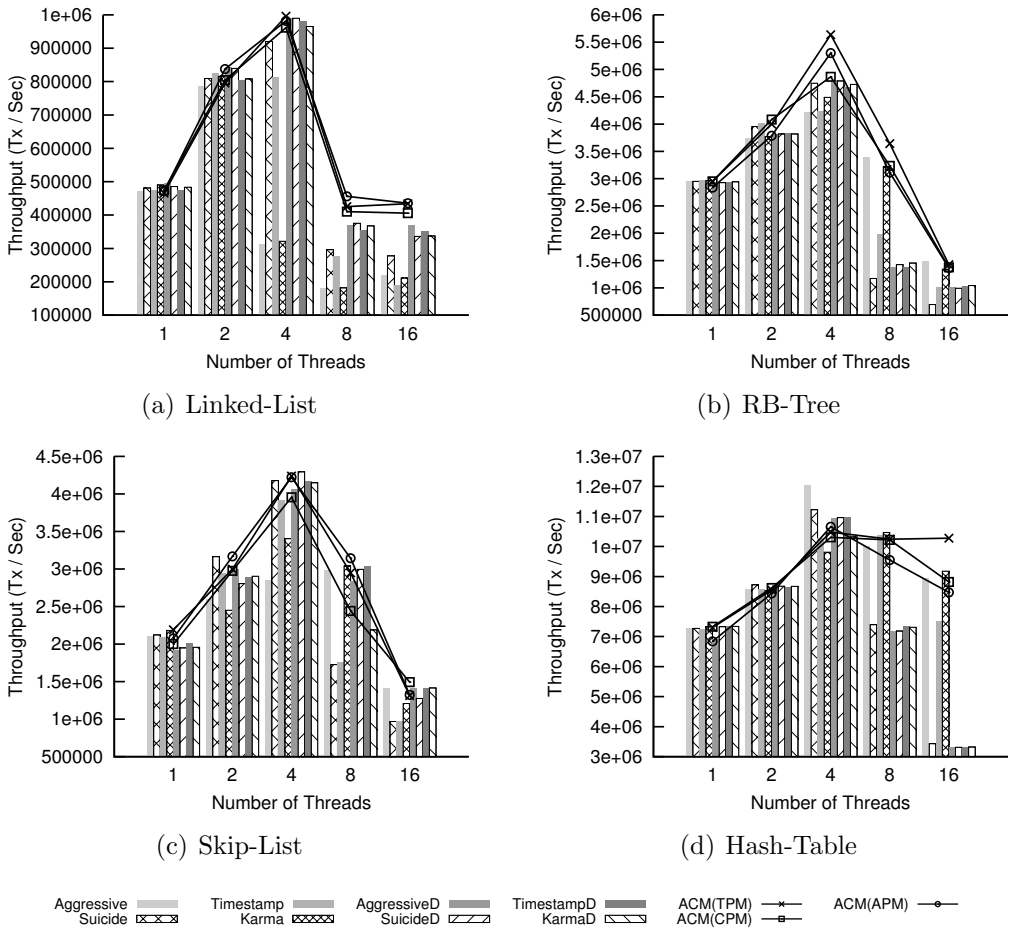
Eight candidate CMs were chosen for both TinySTM and RSTM. This was a relatively large pool as we assume we have no *a priori* knowledge of the target workload and the STM system itself. In practice, if the STM designer knows which CMs are likely to be better in his/her system, CM candidate pool size can be reduced, which will reduce the overheads of our ACM scheme, and improve its performance as well.

**Table 5:** Summary of the tested CMs

| CM | Description |
|---|---|
| `Aggressive` | always aborts the other transaction. |
| `Suicide` | always aborts self (called `Timid` in RSTM) |
| `Polite` | always backs off before aborting the other transaction (not available in TinySTM) |
| `Karma` | always aborts the newer transaction (In RSTM, it will back off before aborting self). |
| `Timestamp` | always aborts the less-productive transaction (In RSTM, it is called `Greedy`, and it will back off before aborting self). |
| `AggressiveD` | `Aggressive` with back-off before a restart (called `AggressiveR` in RSTM). |
| `SuicideD` | `Suicide` with back-off before a restart (called `TimidR` in RSTM). |
| `PoliteR` | `Polite` with back-off before a restart (not available in TinySTM) |
| `KarmaD` | `Karma` with back-off before a restart |
| `TimestampD` | `Timestamp` with back-off before a restart |

Table 5 lists the pool of candidate CMs.

For all the experiments, we used the same initial values to set up our ACM scheme. INITIAL_INT was set to 250 $ms$, INT_BOUND to 4 seconds, and VAR_THRES to 0.2. For the initial profiling length, we used 1 $ms$ for TPM, 128 commits for CPM and 128 aborts for APM. The experimental results are presented in Figures 14, 15 and 16. It can be seen that on all benchmarks and platforms, the performance of




(a) Linked-List (b) RB-Tree

(c) Skip-List (d) Hash-Table

**Figure 14:** Performance comparison of ACM and static CMs on x86 platform for TinySTM.

CMs varied significantly. On x86 platform, the performance variance of CMs reached 40.5% for TinySTM (Hash-Table with 16 threads) and 86% for RSTM (RB-Tree with 16 threads). Similarly, on powerpc platform, this variance could be as high as 32% (Skip-List with 16 threads).

The proposed ACM schemes were able to adaptively choose an optimal CM during run time and consistently achieved performance that was close to the best static CM for all benchmarks and platforms. On some benchmarks, the performance of our ACM was even higher than that of the best CM in the candidate pool. For example, on x86 platform with TinySTM, our ACM(APM) outperformed the best static CM `SuicideD` by 18% for 8 and 16 threads on Linked-List(Figure 14(a)); on powerpc platform with

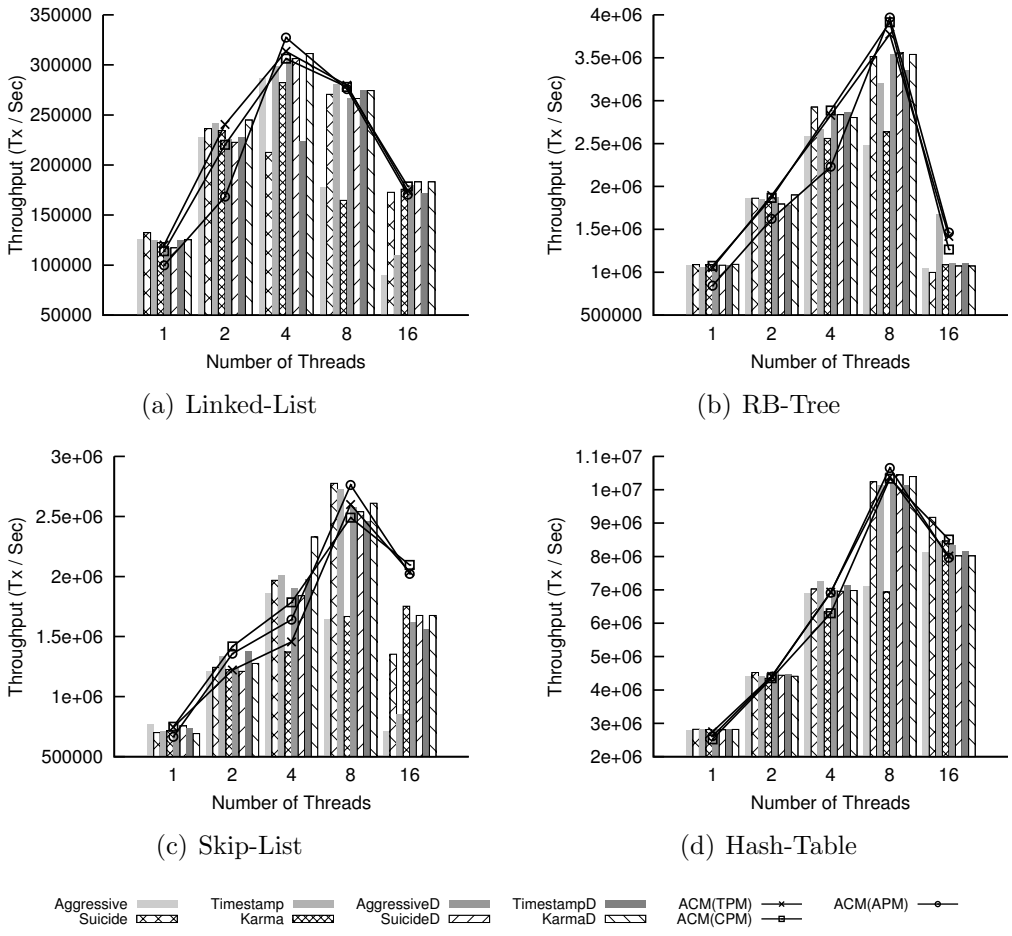



**Figure 15:** Performance comparison of ACM and static CMs on powerpc platform for TinySTM.

TinySTM of 16 threads, ACM(APM) generated a 13% higher throughput than the best static CM `Timestamp` (Figure 15(c)). This is because these benchmarks had time-varying behavior during the execution. Any CMs in the pool, because they are static, would not be optimal throughout the execution of the benchmarks. On the contrary, our ACM scheme was capable of adapting the optimal choice of CMs during run time, and thus outperformed all the static CMs in the candidate pool.

Next we analyze and compare the three schemes of choosing profiling length, TPM, CPM and APM.

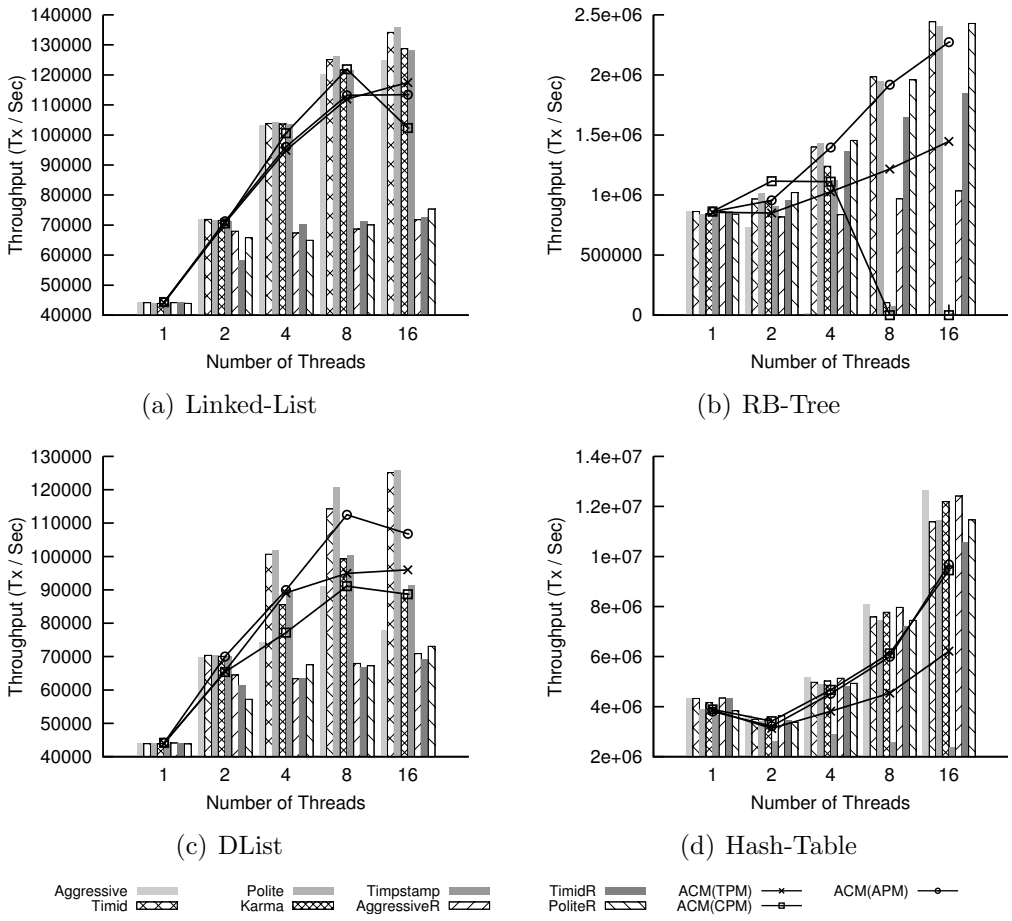



**Figure 16:** Performance comparison of ACM and static CMs on x86 platform for RSTM.

#### *2.3.5.1 Implementation Overhead*

As we discussed in Section 2.3.3.2, TPM has the implementation overhead that is mainly caused by frequently setting the timer. On the Hash-Table benchmark of RSTM (Figure 16(d)), all the candidate CMs had similar performance so that the adaptation frequently switched CMs. Moreover, because these CMs performed very stably, the profiling length $l_i$ and profiling interval $T$ were adjusted frequently (line 13-16 in Figure 12. With TPM, each adjustment would require one extra timer re-installation. On high-throughput benchmarks such as Hash-Table, this overhead was magnified so that TPM performed worse than both CPM and APM (up to 55%).

On the other hand, APM has one additional overhead than CPM which is the

extra atomic operation in `stm_abort` (line 23 in Figure 13). This overhead was not obvious for most benchmarks, but on Hash-Table of TinySTM (Figure 14(d)), this overhead (extra $10^6$ atomic operations per second) caused APM to perform slightly worse than CPM by 4%. We could possibly used thread-local counters to avoid some atomic operations and thus reduce this overhead, but this is the implementation detail and not the focus of this study.

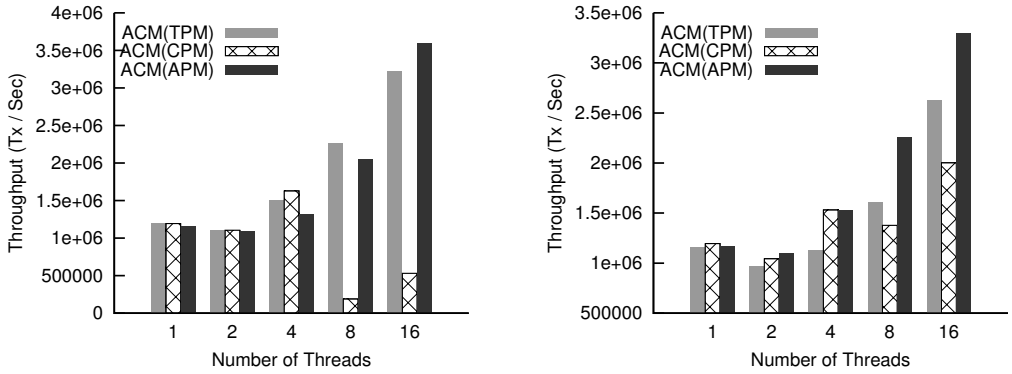
#### *2.3.5.2 Livelock CMs*

The RB-Tree on RSTM was an interesting case. Multiple CMs in the candidate pool caused livelocks. We observed numerous aborts but almost zero commit. The experimental results are demonstrated in Figure 16(b)).

For example, with 16 threads, `Aggressive` had only 44 commits per second and `Karma` had only 4853 commits per second, both of which were significantly lower than other benchmarks (more than $10^6$ commits per second). When profiling these CMs, CPM stalled and waited for commits that were almost not occurring. CPM thus had a very low performance in this case (only 69 commits per second while TPM and APM were higher than $10^6$). It is worth noting that TPM performed worse than APM by up to 36.6% when 16 threads were used. This was because TPM wasted certain amount of time profiling these livelock CMs. APM performed best for this scenario because the livelock CMs generated excessive aborts so that APM quickly collected enough aborts and switched to other well-performing CMs.

#### *2.3.5.3 Time-varying Workloads*

To demonstrate that the performance of TPM is workload sensitive, we synthesized a new benchmark for RSTM. This new benchmark integrated four types of workload Linked-List, RB-Tree, DList and Hash-Table from the benchmark suite of RSTM. The synthesized workload alternated through the above four types of workload, running each of them for a pre-set number of seconds. Figure 17 shows the results of two




(a) Workload Switch Time: 10 seconds (b) Workload Switch Time: 2 seconds

**Figure 17:** Performance comparison of TPM, CPM and APM on the synthetic benchmark (RSTM, x86).

experiments. In both experiments, the new benchmark was run for 40 seconds in total. We set the switch time to 10 seconds in the first experiment and 2 in the second. TPM and APM exhibited similar performance, and CPM performed poorly on 8 and 16 threads. This is because some CMs stalled when RB-Tree is switched in for CPM. In the second experiment, when we decreased the switch time to 2 seconds, which simulated a more volatile workload that changes its behavior frequently. As shown in Figure 17(b), APM outperformed TPM, and achieved a performance increase by 25% on 16 threads.

## *2.4 Summary*

In this chapter, we focused on the data contention of transactional memory systems. We used an analytical approach to quantify the impact of data contention on performance with respect to various system configurations [122]. We also proposed an adaptive contention management mechanism to automatically select the best policy [53].

The analytical model we presented was based on continuous-time Markov chain. It considered the impact of resource capacity, transaction issue rate, implementation

overhead, and more importantly, the spatial and temporal aspects of transaction conflict patterns. The model was validated through extensive experiments. We further explored the impact of two factors on the execution efficiency. Our results demonstrated that it is possible to describe the impact of various key parameters on data contention, basic behavior and system performance in transactional memory systems. We observed that the performance is greatly degraded with higher probability of conflicts which in turn is affected by the implementation overhead.

The model made several simplifications such as omitting the differences in various conflict resolution which are important aspects in TM system design. These issues were addressed by investigating a profiling-based adaptive contention management method for software transactional memory. We examined the performance of existing CM policies with a wide variety of benchmarks and platforms, from which we concluded that adaptation of CMs would be crucial to the performance of TM systems. We then presented our profiling based adaptive CM, and proposed to use logic-time (abort and commit events), instead of the physical-time, to determine the profiling length. We analyzed the profiling overhead for the proposed methods. We showed that the physical-time based method is sensitive to workload and incapable of handling volatile workload. The experimental results validated our proposed method and showed that APM outperforms both TPM and CPM.

# CHAPTER III

# DATA CONTENTION ON GEO-REPLICATED TRANSACTIONAL DATA STORE

With the rapid advancement in Cloud Computing and networking techniques, many online service providers have deployed their systems via geographically-distributed centers that replicate each other. For example, Megastore [8] and Spanner [19] were designed with these supports in mind which were reported to be the back-end storage systems for Google's email service. Such systems provide two benefits: tolerance to the data center outage and low latency for local read requests.

The Geo-replicated systems provide transactional data access abstraction while also hides the fact of distributed replica across multiple data centers from clients. That is, data access in such systems support both transaction-based and distributed-system-based data access abstraction. Specifically, this abstraction maintains a "replica consistency" for the client. Similar to transactional memory systems, to maintain such consistency while effectively exploiting parallelism, data contention management is crucial to system implementation. On the other hand, because of the distributed nature of such system, the contention management strategies have distinguished feature against transactional memory systems discussed in Chapter 2. This chapter aims to discuss the impact of data contention and various design options on the performance of such system.

We begin by elaborating on the "replica consistency" requirement for such systems. Replica consistency regards the question on how the systems allow the replica states to differ from each other. The replica state transitions can be expressed by a sequential log of transactions. Replicas with the same sequential log history have the

same states given the same initial data state. The replica states (i.e., the history log) can differ in two ways: (1) a replica is in a stale state (i.e., the log history is a prefix of others) which could result from a crash or message delays when new transactions are not yet committed to the replica; (2) a replica is in a conflicted state of another which is caused by lack of synchronization when updating replicated data.

At the two ends of the design spectrum, a system can allow both or neither types of differences. In practice, both design extremes have drawbacks: conflicted replica state must resort to manual resolution which is often infeasible; and disallowing stale replicas make the system vulnerable to faults. In this study, we focus on the middle ground of the spectrum where a subset (e.g. a minority) of the replicas are allowed to have stale states but conflicted states are not allowed. Such a replica consistency requirement is common among geographically-replicated systems (e.g., [19, 8, 113, 66]).

The replica consistency requirement necessitates a replication protocol that ensures a shared sequential transaction order (the history log) among replicas. This adds two important aspects to the system design besides the usual focus on concurrency control: (1) interaction between the replication protocol and execution (i.e., concurrency control); and (2) selection of the replication protocol. We study two system types (Execute-Before-Replicate and Replicate-Before-Execute) and three replication protocol schemes (Single Leader Paxos, Fast Paxos and Epoch-based Paxos), which are representative design options for the above mentioned two aspects.

The design options explore various trade-offs. The Execute-Before-Replicate (EBR) and Replicate-Before-Execute (RBE) systems vary in whether execution happens before or after the replication protocol, which result in different drawbacks that EBR incurs higher probability of conflict and RBE has less degree of concurrency. The Single Leader Paxos (SP), Fast Paxos (FP) and Epoch-based Paxos (EP) protocol schemes also outperform each other under different circumstances: SP may have an

extra delay, FP has to deal with collisions and EP is sensitive to network latency variation and time drift.

Although the two aspects (replication protocol and its interaction with execution) are fundamental system design decisions, the impact of the options on performance is affected by various other factors (e.g. network latency, workload arrival rate, and dataset size, etc.). As such, systematic understanding of the performance characteristics of these design options becomes necessary. Analytical models are critical for system deployment. Hence, we aim to develop a quantitative analysis to guide the selection of the design choices for geographically-replicated datastores. Specifically, we make the following contributions in this chapter:

- Categorize fundamental design options and identify their trade-offs including: (1) EBR and RBE systems and (2) SP, FP and EP replication protocol schemes.
- Develop a complete framework to model systems with combinations of the design options.
- Validate models through extensive detailed simulations.
- Study the impact of workload/system parameters on the performance of the system variations.

Our models establish relationships between workload/system configurations and transaction response time for each design option and hence enable quantitative comparison on the trade-offs. It can be shown that network latency has super linear impact on EBR while transaction arrival rate has super linear impact on RBE. EBR can achieve higher maximum throughput under low network latency. Our models also reveal that FP outperforms SP when load occupancy is under around 25%; the performance of EP is degraded by large network latency variance and time drift among replicas.

The rest of the chapter is organized as follows: We first list the related works in Section 3.1. Section 3.2 overviews design options. Section 3.3 presents our methods to model these options. Section 3.4 validates our model with simulation and presents insights on comparison among the options. Section 3.5 summarizes this chapter.

## *3.1 Related Work*

Spanner [19] used long lived leaders with Paxos to replicate the execution results which is similar to EBR. Megastore [8] assigned dedicated leader for each log position which was a variation of RBE. Calvin [113] pre-assigned timestamps and then executed the transactions in timestamp order which was a variation of RBE. Several optimizations of such systems existed in Paxos-CP [87], Spanner and MDCC [66], etc. Other research efforts, such as Dynamo [22] and Walter [104], etc., targeted different consistency requirements.

Two surveys [84, 111] summarized research works on modeling the performance of distributed or centralized database. A common method used was mean value analysis [110, 111, 38, 39]. We followed this line of work. Our models focus on the important aspects of replication consistency, and thus are capable of describing the drawbacks of EBR and RBE such as long lock holding time and less degree of concurrency and the trade-offs of three Paxos protocol variants.

Other methods were used to quantify various aspects of concurrency control and distributed protocol. In [77] two phase commit protocol was modeled using queuing theory. Methods were described in [62] to compute the impact of network delay on conservative timestamp algorithm in DDBS. In [5], the authors discussed multiple aspects of three protocols that guarantee serializability and transaction atomicity. In [14] the authors employed a simulation study for performance of multi version and distributed two-phase locking.

Classic and Fast Paxos were described in [69] and [70]. There were few efforts in

modeling the performance of Paxos and the systems using Paxos. In [63] the authors discovered cases when Fast Paxos performed worse than classic Paxos.

## *3.2 Systems Design Options Overview*

The consistency requirement brings out two important aspects to the system design, i.e., the replication protocol and its interaction with execution. This section overviews the representative design options regarding these two aspects.

### 3.2.1 The EBR and RBE Systems

EBR and RBE are two representative types that differ on when to invoke the replication protocol. The two types explore trade-offs between high probability of conflict (EBR) and less degree of concurrency (RBE).

#### *3.2.1.1 The EBR System*

In EBR, a master replica executes transactions while other replicas update the results. Client requests are directed to the master replica; the master replica executes transactions in parallel employing the traditional dynamic locking scheme: locks are requested on demand each time when a transaction accesses data.

The SP scheme is invoked after locks are acquired and before transaction commit. The protocol updates the transaction order and execution result to other replicas. The replication protocol must be invoked before, instead of after, transaction commit for fault tolerance. Otherwise, it could happen that the master fails at the point between the commit of a transaction $T$ and the invocation of the protocol for $T$. In this case, the system continues with a new master not knowing the existence of $T$. The failed master replica, however, recovers with $T$ committed which leads to a conflicted state. The necessity to commit after the protocol exposes transactions to longer lock holding time and hence higher probability of conflict.

#### *3.2.1.2 The RBE System*

In RBE, upon new transactions arrive, the order is determined and agreed upon among replicas through a replication protocol. Each replica follows this same order and executes transactions individually. At each replica, when a transaction is "ready" to execute, it proceeds to the locking stage. A transaction is "ready" only when all its predecessors have finished their locking stage. Transactions acquire all locks at the locking stage. If the required locks cannot be granted, the transaction is appended in the FIFO queues of the locks and its execution is suspended. Transactions holding all the locks continue to data accesses, commit and lock release.

The constraint that execution must follow the pre-determined order causes the problem of less degree of concurrency. The problem exhibits at two levels. At the replication protocol level, a transaction cannot proceed to execution if its preceding transactions have not finished their replication protocol. For example, due to network latency variance, it is possible that the messages of transaction 1 arrives earlier than messages of transactions 0. In such case, the replica cannot execute transaction 1 until after it learns about transaction 0. This causes extra delay for transaction 1. We term this impact *reorder overhead*. At the execution level, The pre-determined order may also have an adverse impact on concurrency because of the conflict among transactions. For example, there are three transaction $T_1$ (Write A, Write B), $T_2$ (Write B, Write C) and $T_3$ (Write C); Under dynamic locking in EBR, because $T_1$ and $T_3$ do not have overlapped data, they could be executed in parallel with $T_2$ blocked after $T_1$; However, for RBE, when the order is determined to be $T_1$, $T_2$ and $T_3$, only one transaction can be executed at a time.

### 3.2.2 Replication Protocol Schemes

The replication protocol ensures the consistency requirement that replicas view the same transaction order regardless of faults and arbitrary network delay. This is done

using the Paxos protocol. Paxos [69, 70] is a distributed protocol to reach consensus among a set of agents. In our cases, the consensus is made on each slot in the shared order (e.g., executing $T_a$ as the $10^{th}$ transaction). The protocol is conducted among agents termed proposers, acceptors and learners. Proposers issue proposals, acceptors help make decisions and learners learn the decisions. The procedure of one slot finishes when a quorum of learners learn the decision. Each replica has one acceptor and one learner. There could be one leader proposer or multiple proposers depending on the scheme.

We abstract away complex algorithmic details in the Paxos protocol but only present behaviors that are related to our performance analysis. For example, though quite some efforts are made in the Paxos protocol to handle faults, we choose not to include the impact of fault recovery because replica faults are rare during normal execution.

#### *3.2.2.1 The SP Scheme*

SP employs one leading proposer for every slot in the transaction order. When The proposer receives a new transaction, it sends an accept request to acceptors for the transaction to be placed in the next slot in the order; acceptors accept the transaction for that slot and forward "accepted" responses to the learners; learners learn the responses when received messages from a quorum of acceptors and replicas can proceed to the execution.

SP costs two times the cross-replica latency (a message chain from proposer to acceptors then to learners). However, if the leader is on a replica different from the client location, an extra cross-replica message delay is needed, which makes the delay three times the cross-replica latency.

#### *3.2.2.2 The FP Scheme*

FP [70] attempts to optimize SP by eliminating the extra cross-replica message delay resulted from the location difference between client and the leader. FP allows multiple proposers, each on one replica, to send proposals when they receive new transactions. The agents proceed similarly as in SP except (1) a larger quorum of acceptors is needed for learners to learn the decision for a slot [70]; (2) multiple proposals of one slot can cause a collision which needs to be recovered by starting a new round of messages and having a coordinator decide for that slot.

FP typically costs approximately two times the cross-replica latency as SP does. When collision occurs, however, one of the proposals will succeed and be learned for that slot; Other proposals will fail and the failed proposers propose for the next slot. This results in two kinds of extra delay in the response time: one is that a proposal have to restart multiple times before it succeeds; another is that the successful proposal will also incur a delay for collision recovery.

#### *3.2.2.3 The EP Scheme*

The drawbacks in previous schemes result from (1) the distance between client and leader in SP; (2) collisions by multiple proposers in FP. EP attempts to alleviate these overhead by letting each replica host a proposer for different slots, e.g., the proposer on replica 0 proposes for the $0^{th}, 5^{th}, 10^{th}, \ldots$ slots, assuming 5 replicas. Thus every replica can issue proposal and replicas do not compete for the same slot. However, for RBE, the assigned proposers need to send proposals in a bounded time (even if no transaction is received) because otherwise the transaction execution of later slots will be stalled. To alleviate this problem, an epoch batch approach is adopted. Replica local time is divided into epochs (small fixed length of time pieces). Proposers on each replica propose batches of transactions for the assigned slot. A batch contains all transactions arrived in the last epoch. An empty batch is proposed if no client

**Table 6:** Trade-offs Among Design Options.

| System | Related Work | Performance Characteristics |
|---|---|---|
| EBR, SP | Spanner [19] | 3x cross-replica latency; long lock holding time. |
| RBE, SP | Megastore [8] | 3x cross-replica latency; reorder overhead; lower degree of concurrency. |
| RBE, FP | MDCC [66] | 2x cross-replica latency; proposer collision; reorder overhead; lower degree of concurrency. |
| RBE, EP | Calvin [113] | 2x cross-replica latency; waiting time for epoch end; reorder overhead; lower degree of concurrency. |

request is received in the last epoch. Using this method, an underloaded proposer will not block the execution of other transactions indefinitely.

EP costs approximately two times the cross-replica latency. However, it incurs extra delay resulted from the waiting time for each epoch end and a reorder overhead from both network latency variance and time drift among replicas.

Table 6 summarizes the performance trade-offs among the design options. We also listed related systems in the table. Note that these systems incorporate a wide range of techniques and many details that we cannot capture in the scope of this study, e.g., these systems need techniques to handle consistency among partitions which we do not consider. However, the characteristics of our abstraction still apply on these implementations.

## *3.3 System Models*

The model is desired to help compare the design options in Table 6. Such analysis should study the execution, replication protocol and their interaction. It should reveal the performance traits and trade-offs including the high probability of conflict in EBR, loss of concurrency in RBE, and various features among the three replication scheme. To simplify the modeling analysis, we first decouple the transaction execution and the

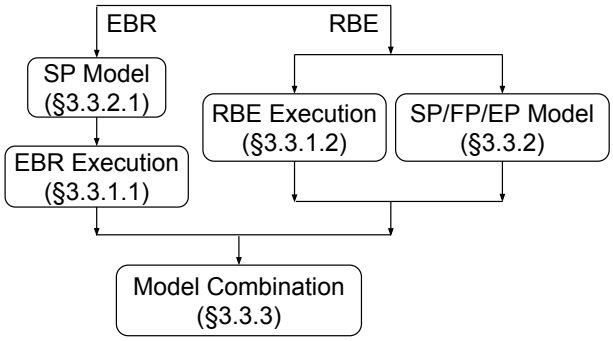



**Figure 18:** Models of EBR and RBE Systems

replication protocol and combine them later. The execution models abstract away the replication protocol and only analyze the trade-off between EBR and RBE execution. The protocol models studies the trade-offs among replication schemes including the loss of concurrency issue at protocol level (the reorder overhead) for RBE. Figure 18 summarizes the analytical models of the two system types and the three schemes.

### 3.3.1 Models of EBR and RBE Execution

For both system types we model a closed system with a fixed number of transactions, denoted by $m$. The justification for considering a closed system is given in [110]. Transactions in the system are assumed to be of the same size, denoted by $k$, which is the number of locks a transaction requests. Transactions request locks uniformly from a pool of $d$ locks. Locks are acquired in exclusive mode. Models with more general assumptions can be extended from such a basic model: for example, systems with different transaction sizes can be extended using methods in [111]; the effect of non-uniform access and shared locks is equivalent to the case of exclusive uniform access with a larger lock pool [110]. To simplify the computation, we also assume that lock conflict is rare (i.e., $km \ll d$). Each of the $k$ steps in a transaction takes some processing time with mean value $s$. After acquiring all the locks the transaction commits which takes a mean time of $c$. The above assumptions are adopted by many related works (see survey [111]). Table 7 lists the common variables used.

**Table 7:** Variables Used in the Execution Models

| | |
|---|---|
| $d$ | Total number of locks |
| $m$ | Number of concurrent transactions in the system |
| $k$ | Number of locks acquired by each transaction |
| $s$ | Mean time takes for each lock step |
| $c$ | Mean time takes for commit step |

#### *3.3.1.1 The EBR System Execution Model*

Following the assumption stated above, a transaction in EBR goes through a fixed number of steps. In each step, it acquires a lock. The transaction waits if the requested lock is not available. After acquiring all the locks, the transaction takes the commit step (while holding the locks) which invokes the replication protocol. The analysis extends the model in [112] by considering the impact of a long commit time.

The mean response time $res_{EBR,exec}$ can be calculated as

$$res_{EBR,exec} = ks + c + p_s k W_s + W_d \approx ks + c + p_s k W_s, \tag{25}$$

where $ks + c$ is the response time without any lock conflict; $p_s$ is the probability the transaction is blocked for each step; $W_s$ is the mean waiting time for each step if a transaction is blocked; $p_s k W_s$ is the average blocking time for a transaction; $W_d$ is the overhead of restart because of deadlocks which is ignored when lock conflict is rare.

The probability of lock conflict can be approximated by

$$p_s \approx \frac{(m-1)\overline{L}}{d}. \tag{26}$$

where $\overline{L}$ is the mean number of locks held by a transaction. The probability that an active transaction is in its $j^{th}$ stage is assumed to be proportional to the processing time of that stage (e.g., when commit time is long, the system tends to find a transaction in its commit stage). Therefore, the average number of locks an active transaction holds is $L_a = \sum_{j=1}^{k} j \frac{1}{k+\frac{c}{s}} + \frac{k}{k+\frac{c}{s}}$. Under the assumption that lock conflicts

are rare, we can use the mean number of locks of active transactions to approximate number of locks of the system, that is, $\overline{L} \approx L_a$.

$W_s$ is the waiting time when a transaction is blocked. Blocked transactions form a *waits-for* graph where nodes represent transactions and edges represent the waits-for relationship. Because only exclusive locks are considered, the graph is a forest of trees. Active transactions are at the roots of the trees and designated to be at level zero. Transactions blocked by active transactions are at level one and so on. To compute $W_s$, we first compute $W_1$, the waiting time of level one transactions that are blocked by an active transaction. Assume that the probability, $p_{b,j}$, that an active transaction is at its $j^{th}$ step when a level one transaction is blocked by it, is proportional to the number of locks that the active transaction holds and the mean time it remains in that state. Then the probability is computed as $p_{b,j} = \frac{js+(j-1)u}{norm}$, where $u = p_s W_s$ is the average total waiting time of a transaction, and $norm$ is a normalization factor; The probability that the active transaction is in its commit stage is $p_{b,c} = \frac{kc}{norm}$. The normalization factor is $norm = \sum_{j=1}^{k}[js + (j-1)u] + c$. The variable $u$ is unknown and can be ignored [112] under the rare lock conflict assumption since $u \ll s$. The waiting time $W_1$ is then the time for the active transaction to finish, which can be computed as $W_1 = \sum_{j=1}^{k}[s' + (k-j)s + c]p_{b,j} + c'p_{b,c}$, where $s'$ is the mean residual time of each lock step and $c'$ the commit step. From renewal theory, the mean residual time per lock step $s' = \frac{\sigma_s^2+s^2}{2s}$ and that of the commit step $c' = \frac{\sigma_c^2+c^2}{2c}$ [64]. For fixed distribution, $s' = \frac{s}{2}, c' = \frac{c}{2}$; for exponential distribution, $s' = s, c' = c$.

To compute $W_s$ from $W_1$, we introduce the probability that a transaction is blocked, denoted $\beta$.

$$\beta = \frac{m - m_a}{m} \approx \frac{kp_s W_s}{res_{EBR,exec}}, \tag{27}$$

where $m_a$ denotes the mean number of active transactions in the system. The second equality follows the Little's Law, i.e., $\beta$ can be also expressed as a ratio of the

mean transaction delay in the blocked state and the mean transaction response time. From [111, 112], the probability that a transaction is at level $i$ is approximated by $P_b(i) = \beta^{i-1}, i > 1$, and $P_b(1) = 1 - \beta - \beta^2 - \ldots$. The mean waiting time at level $i > 1$ is approximated by $W_i = (i - 0.5)W_1$. Therefore, the waiting time $W_s$ is a weighted sum of delays of all levels.

$$W_s \approx W_1[1 - \sum_{i \geq 1} \beta^i + \sum_{i>1} (i - 0.5)\beta^{i-1}] \tag{28}$$

The probability that a level one transaction is blocked is $\alpha = \frac{kp_sW_1}{res_{EBR,exec}} \approx \frac{kp_sW_1}{(k+\gamma)s+kp_sW_1}$. Because $\beta$ is unknown, $\alpha$ is good approximation of $\beta$ (i.e., $\beta \approx \alpha$) since most blocked transactions are at level one when conflict rate is low.

#### *3.3.1.2 The RBE System Execution Model*

For RBE, before access data, transactions start lock acquisition in the order of their arrival. A transaction is blocked when a requested lock is held by an earlier transaction. Blocked transactions are appended in the FIFO queues associated with the requested locks. Locks released by committed transactions are granted to the next transaction in its queue. Transactions successfully acquired all the locks become active and start execution.

The mean response time of a transaction is the time the transaction originally takes plus the waiting time in the lock FIFO queue. That is,

$$res_{RBE,exec} = ks + c + p_tW_t = ks + p_tW_t. \tag{29}$$

where $ks + c$ is the response time without conflict; the commit time $c = 0$ since the replication protocol is invoked before execution; $p_t$ is the probability that a transaction is blocked during lock acquisition; $W_t$ is the average waiting time on the queue.

The probability that a new transaction will be blocked by the previous $m - 1$ transactions can be approximated as

$$p_t \approx 1 - (\frac{d - (m-1)k}{d})^k, \tag{30}$$

under the rare conflict assumption. The equation simply uses the fact that for each lock, the probability that the transaction does not conflict with the other $m-1$ transactions can be approximated by $\frac{d-(m-1)k}{d}$. The probability that a transaction has lock conflicts with another transaction can be approximated using the same reasoning: $p_w \approx 1-(\frac{d-k}{d})^k$. The probability that the $i^{th}$ arrived transaction in the system is active equals to the probability that the $i^{th}$ transaction does not conflict with the previous ones, which is $(1-p_w)^{i-1}$. Therefore, the average number of active transactions observed by the $m^{th}$ transaction is $a = 1+(1-p_w)+(1-p_w)^2+\cdots+(1-p_w)^{m-2} = \frac{1-(1-p_w)^{m-1}}{p_w}$. The mean number of transactions an active transaction blocks is then $h \approx \frac{m-1}{a}$, which is also the mean number of transactions the $m^{th}$ transaction has to wait given it encounters conflicts. Therefore, the waiting time of a blocked new transaction is

$$W_t = r + (h-1)ks, \tag{31}$$

where $r \approx 0.5ks$ is the mean residual time of the active transaction and $(h-1)ks$ is the mean time to wait for the blocked transactions of higher levels to finish.

The probability that a transaction is blocked is

$$\beta = \frac{m-m_a}{m} = \frac{p_t W_t}{res_{RBE,exec}}. \tag{32}$$

### 3.3.2 Models of Replication Protocol Schemes

In this section, we study the response time of the replication protocol. We assume the cross-replica network delays are random variables that are independent and identically following the same distribution; the local network latency is small enough to be ignored. The arrivals of client requests on all replicas follow Poisson process with the same arrival rate $\lambda$. We analyze the performance of the protocols under normal cases. We do not consider replica failure and recovery since faults are rare and the performance of many recovery schemes mainly depends on the implementation detail.

Throughout the derivation, we use $L$ to denote the random variable of cross-replica latency, $F_L(t) = Pr(L \leq t)$ be the probability distribution of $L$, and $f_L(t)$ be the density function. We use $n$ to denote the number of replicas of the system.

#### *3.3.2.1 The SP Scheme*

The SP scheme involves two kinds of delay: (1) a client sends a transaction to the leader replica; (2)*quorum delay* including: the leader proposer proposes the transaction for the next slot and sends messages to acceptors on all the replicas; the acceptors send messages to the learners on each replica; the learner learns the transaction when it receives messages from a majority quorum of acceptors.

The delay (1), denoted $C$, equals to the node-to-node latency $L$ if client and the leader replica is in different region or zero if they are in the same one. Assuming the load is balanced across regions, the distribution of the delay $C$ is

$$F_C(t) = Pr(C \leq t) = \frac{1}{n} + \frac{n-1}{n} F_L(t), \tag{33}$$

and $E[C] = \frac{n-1}{n} E(L)$.

To compute quorum delay in (2), denoted $Q$, we introduce a round trip delay random variable $R = L_i + L_j$ to denote the delay of the message chain from the proposer to an acceptor then to a learner, which is the summation of two iid node-to-node delays. The probability density function of $R$ is $f_R(t) = \int_0^\infty f_L(t) f_L(t-u) du$. The learner learns a transaction when it receives messages from $q$ messages acceptors. In SP, $q = \lceil \frac{n+1}{2} \rceil$. Given that one of the message chains is local and can be ignored (the proposer, acceptor and learner are all in the same replica), the quorum delay equals to the value of the $(q-1)^{th}$ smallest of $n-1$ iid round trip delay random variables.

$$F_Q(t) = \sum_{j=q-1}^{n-1} \binom{n-1}{j} F_R^j(t) (1 - F_R(t))^{n-j-1}. \tag{34}$$

For EBR, quorum delay equals the transaction commit time and thus the probability distribution $F_Q(t)$ (specifically, the first two moments of the distribution) is

used in the execution model (§ 3.3.1.1) for the mean commit and residual time.

For RBE, we should further compute the reorder overhead. The delay including the reorder overhead, denoted by $D$, is the time between that the proposer propose a transaction $T_i$ and that the learner learns the transaction as well as all the previous ones before it. To compute the distribution of $D$ from $F_Q(t)$, we follow the same method used in [62]: the probability $F_D(t) = Pr(D \leq t)$ can be computed by first obtaining the conditioned probability, $Pr(D \leq t|s)$, given that the time the transaction $T_i$ is proposed is $s$ and then letting $s \to \infty$. The conditioned probability can be computed as $Pr(D \leq t|s) = Pr(X)Pr(Y|s)$, where $X$ is the event that $T_i$ is learned in time less than $t$ ; and $Y$ is the event that all other previous transactions are learned before $s + t$. The probability $Pr(X) = Pr(Q \leq t) = F_Q(t)$ by definition. To derive $Pr(Y|s)$, consider the $i$ transactions that are proposed before transaction $T_i$. Each of those transactions has to be learned before $s + t$ and thus a transaction proposed at time $u$ can only have a quorum delay less than $s + t - u$, i.e., $Q \leq s + t - u$. Given that the transaction arrivals follow a Poisson distribution, the time these $i$ transactions are proposed is independent and uniformly distributed in $[0, s]$ [64]. By unconditioning on $u$ and summing over $i$, we can get $Pr(Y|s) = \sum_{i=0}^{\infty} \frac{(\lambda s)^i}{i!} e^{-\lambda s} [\int_0^s \frac{F_Q(s+t-u)}{s} du]^i = e^{-\lambda \int_t^{t+s} (1-F_Q(u))du}$. Therefore, the distribution of delay $D$ follows

$$F_D(t) = Pr(D \leq t) = F_Q(t) e^{-\lambda \int_t^{\infty} (1-F_Q(u))du}. \tag{35}$$

The mean response time of the replication protocol for RBE can be calculated as:

$$res_{sp} = E(C) + E(D). \tag{36}$$

#### *3.3.2.2 The FP Scheme*

The model for FP differs from SP in that multiple proposals of the same slot causes extra delay in addition to the quorum delay and the reorder overhead. The extra

delay has two parts: (1) a proposal for a slot fails and new slots are proposed until success; (2) for a successful proposal, a collision adds the delay of an extra message. Therefore, the response time of FP can be expressed as $res_{fp} = t_{fail} + t_{succ} + t_{reorder}$. Here $t_{fail}$ is the mean time from that a proposer proposes for a slot until that the last failed proposal finishes; $t_{succ}$ is the mean time for the successful and the last proposal of a slot. A successful proposal takes the time of a quorum delay computed by Eq. 34 (with a larger quorum, see [70]) when there is no collision. It takes an extra quorum delay for a coordinator to resolve the collision if there is one. Therefore, $t_{succ} = E(Q)(1 + Pr(collison))$. From simulation, we find that the impact of collision is insignificant compared to the impact of failed attempts. Hence, we can approximate $t_{succ}$ by $t_{succ} \approx E(Q)$. Furthermore, the reorder overhead is also negligible compared to the delay of failed attempts.

To compute the proposal delay, we further make the simplification that the network latency is constant. Under this assumption, if there is only one proposal, it is learned after a quorum delay (previously calculated in Eq. 34) which assumes to be a fixed interval. If there are multiple proposals competing for a slot, there will be failed proposals which will restart and reach the acceptors at the same time when competing for the next slot; one of the restarted proposals will be learned, leave the competition and move onto execution. Such a mechanism can be modeled as an M/D/1 queue where proposals arrive following a Poisson process and one proposal can leave the system after a fixed amount of time. The constant network latency approximation greatly simplifies competition process, but the model is more accurate when the load is low.

By simplifying the system into a M/D/1 queue, we can apply the Pollaczek-Khinchine formula [64], i.e., the number of proposals in the system in steady state is $N = \lambda E(Q) + \frac{(\lambda E(Q))^2}{2(1-\lambda E(Q))}$. By Little's Law, the average delay is:

$$res_{fp} = \frac{N}{\lambda}. \tag{37}$$

#### *3.3.2.3 The EP Scheme*

EP involves three kinds of delays: (1) the client sends a request to the local proposer which is batched for the next epoch; (2) the quorum delay for the replication of an epoch batch ; (3) the reorder overhead.

We first compute the average waiting time $E(W)$ in (1). Following the property of Poisson arrivals, given that there are $i$ transactions sent by the clients to a local replica on a period $[0, e]$, the arrivals of $i$ transactions are independent and uniformly distributed over the period [64]. For each transaction $r$, given $i$, the waiting time is then $E(W_r|i) = \frac{e}{2}$. Because the arrivals of these $i$ transactions are independent, the waiting time is then $E(W|i) = \frac{e}{2}$. By summing over all $i$, we get $E(W) = \sum_{i=1}^{\infty} Pr(i)\frac{e}{2} = \frac{e}{2}$. Next we compute the delay in (2) and (3) including both quorum delay $Q$ (the same as Eq. 34) and reorder overhead. We denote this delay $D$ which is the time between a transaction batch is proposed and the batch can be executed. Each transaction batch is identified by a tuple $(i, j)$ from replica $i$ and the $j^{th}$ epoch. Consider the delay $D_{I,J}$ of a transaction batch starting at time $Je$. The event $D_{I,J} \leq t$ is equivalent to that the learner learns all the transaction batches $(i, j)$ where, $i = 1, 2, \ldots n$ and $j \leq J$ before $Je + t$. The probability that the learner learns all the transaction batches for $j = J$ is $Pr(B_J \leq t) = F_Q^n(t)$; and $Pr(B_{J-1} \leq t) = F_Q^n(Je+t-(J-1)e) = F_Q^n(t+e)$; and so on. Therefore, the distribution of the delay for epoch $J$, $D_J$ can be computed as $Pr(D_J \leq t) = \prod_{j=0}^{J} F_Q^n(t + je)$. By letting $J \to \infty$, we can get the distribution of the delay of step (2) and (3), $F_D(t) = Pr(D \leq t) = \prod_{j=0}^{\infty} F_Q^n(t + je)$. Furthermore, we take the time drift among the replicas into account. To simplify computation, we assume the time differences between the epoch start on any two replicas are random variables, denoted $S$, that are independent and identically following the distribution $F_S(t)$. Let $Q' = Q + S$ denote the delay between the time of batch proposal and the time of its arrival at a learner taking the time drift between the replica of the proposer

and that of the learner into account. The probability of the delay $D$ becomes

$$F_D(t) = Pr(D \leq t) = \prod_{j=0}^{\infty} F_Q(t + je) F_{Q'}^{(n-1)}(t + je). \tag{38}$$

The average response time of Epoch-based Paxos is

$$res_{ep} = E(W) + E(D). \tag{39}$$

### 3.3.3 Combined System Models

For EBR, when transactions commit during execution, the replication protocol is invoked, therefore the total response time is the response time of the execution model with the protocol response time as an input variable.

$$res_{EBR} = E(C) + res_{EBR,exec}(Q). \tag{40}$$

where $C$ is the delay for a request to be sent from client to the leader in SP and is computed using Eq. 33; and $Q$ is the quorum delay random variable following the distribution computed in Eq. 34, the distribution is treated as an input for the execution model but only the first two moments are needed; $res_{EBR,exec}$ is computed using Eq. 25.

For RBE, the replication protocol is decoupled from transaction execution, therefore the response time is the summation of the execution and protocol response times:

$$res_{RBE} = res_p + res_{RBE,exec}. \tag{41}$$

where $res_p$ is the latency of the replication protocol latency which is computed using Eq. 36, Eq. 37 or Eq. 39 depending on the scheme; $res_{RBE,exec}$ is computed using Eq. 29.

If the system is a closed system with the concurrent number of transactions $m$, the execution time $res_{EBR,exec}$ and $res_{RBE,exec}$ can be readily computed. If the system is an open system with arrival rate $\lambda$, an iterative method is required to compute the

response time. Using Little's law, the initial value of the number of transactions in the system $m$ can be approximated as $m_0 = \lambda(ks+c)$. For each iteration the response time can be computed using Eq. 40 and Eq. 41 and the number of transactions for the next iteration is

$$m_i = \lfloor \lambda res_{i-1} \rfloor. \tag{42}$$

## *3.4 Experimental Results*

We first validated our model against simulation and then studied the impact of workload and system parameters and compare the design options.

### 3.4.1 Model Validation

#### *3.4.1.1 Simulation Settings*

We resorted to simulation to validate our analytical models, which gave us the freedom of testing with various parameters in a controlled manner and profiling the internal details without instrumentation overhead. The simulation was built on top of SimPy [103] which is a discrete event-based simulation framework. Our simulation tried to capture implementation details with respect to the locking mechanism and the Paxos protocol.

The execution of transactions was simulated by launching threads executing transaction operations. Each data access waited a randomly generated amount of time according to a pre-specified distribution. Locking was simulated in details such as thread suspend and wakeup. Potential deadlocks were resolved by using the Tarjan's algorithm [109] and restarting offending thread instead of using timeout technique due to the difficulty to determine the timeout interval. The network delay was simulated by randomly generating a wait time following a specified distribution when a message is sent. The complete algorithms of the Paxos protocols were implemented.

Event statistics, such as blocking time, were probed and profiled in the system to verify our model in a fine granularity. For each simulation experiment, we ran at

least 5000 transactions and ensured the system is in a steady state by monitoring the transaction finishing rate.

#### *3.4.1.2 Validation Experiments*

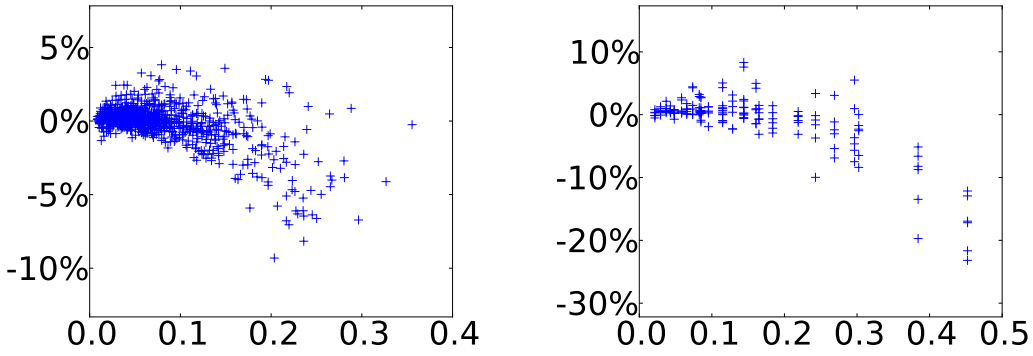
We validated our model against simulation with a wide range of workload and system parameters. The metrics of error (the y axes in the figures) was computed as $(res_{model} - res_{sim})/res_{model}$.

Both the execution model of EBR(Eq. 25) and RBE (Eq. 29) was validated under various workload parameters: the number of items $d = 4096$, 8192 and 16384; the number of concurrent transactions $m = 12$, 16 and 20; the number of items per transaction $k = 8$, 12 and 16; the interval $s = 10$, 20 and 30; for EBR systems, the commit time $c = 0, 20, 40, 60$ and 80; for RBE systems, $c = 0$; the execution and commit step followed both fixed and exponential distribution.

Figure 19(a) shows the comparison between simulation and model of EBR. The x axis is the probability that a transaction is blocked. The figure shows that the model follows the simulation result up until a high conflict rate (around 40% of the transactions are blocked). The error rate is at most 10%. Detailed profiling results reveals that the inaccuracy mainly results from an underestimation of the lock waiting time $W_s$.

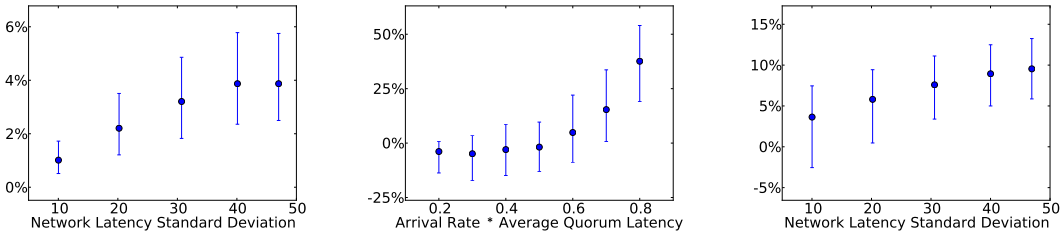
Figure 19(b) shows the comparison between simulation and model of RBE. The x axis is the probability that a transaction is blocked. The error rate of our model is at most 25% under high conflict rate (around 50% of the transactions are blocked) and is less than 10% when the probability of transaction blocked is less than 30%. Detailed results shows the error is due mostly to the underestimation of the waiting time $W_t$ for high lock contention cases.

To validate protocol schemes, we set the number of replicas $n = 5$, 7 and 9; and the cross-replica network latency distribution as a log-normal distribution with




(a) EBR Execution (b) RBE Execution

**Figure 19:** Validation of Execution Models. The y-axis shows the error rate; The x-axis shows the blocking probability ($\beta$).




(a) SP (b) FP (c) EP

**Figure 20:** Validation of Replication Protocols. Bars show the average, min and max error rate.

$\sigma = 0.1, 0.2, 0.3, 0.4, 0.5$ and $\mu$ was set such that the mean is fixed as 100.

Figure 20(a) shows the comparison between simulation and our model for SP (Eq. 36). The x axis is the standard deviation of the network latency. From the figure, it is shown the error rate is less than 6%. The error rate increases with the standard deviation of the network latency. A possible source of the error results from the numerical approximation of the solution.

To validate the model for FP (Eq. 37), we varied the arrival latency as well. Since the system is unstable when the arrival rate is larger than service rate, we selected the approximated load occupancy $\lambda T$ ($\lambda$ is the arrival rate and $T$ is two times the mean cross-replica latency which is approximately the quorum delay) ranging from 0.2 to 0.8. Figure 20(b) shows the comparison between model and simulation. The x axis is

the load occupancy. Despite our simplifications, the result is relatively accurate when the arrival interval is small. The error rate is less than 25% when $\lambda T \leq 0.6$. The error rate becomes larger when arrival rate is comparable to the service rate, which results from the simplification of the process of proposer competing slots and retrying and a constant latency.

To validate the model for EP (Eq. 39), we also varied the epoch length $e = 30, 50, 70$ and time drift upper bound $h = 0, 10, 20$. Figure 20(c) shows the result of validation of EP. The x axis is the standard deviation of the network latency. The error rate is less than 15% for our model.

### 3.4.2 Study of System Design Options

Figure 21 studies the performance drawbacks of EBR and RBE under open system with arrival rate $\lambda$ and assuming network latency is constant. When the network latency is low, EBR has a lower response time; the difference between the two systems increases with arrival rate $\lambda$. When increasing the network latency, the response time of EBR execution has a super linear growth. In fact, if we further approximate the model with $m = \lambda(ks + c)$, $k \gg 1$, $s \gg u$, and $W_1 \approx W_s$, our EBR model(Eq. 40) can be simplified into $res_{EBR} \approx \frac{\lambda k^2}{d} cc' + (\frac{\lambda k^2}{2d} + 1)c + \frac{\lambda k^4 s^2}{6d} + ks$ where $c$ is average commit time and $c'$ is the residual time of the commit step (i.e., the quorum delay). For fixed distribution $c' = 0.5c$, and for exponential distribution $c' = c$. When both $c$ and $c'$ is large, EBR incurs long response time. For the RBE model, if we further approximate our model using $m = \lambda ks$, $k \gg 1$, and $d \gg k^2$, the model(Eq. 41) can be simplified into $res_{RBE} \approx \frac{2\lambda k^4 s^2}{2d - \lambda k^3 s} + ks$. When $\lambda k^3 s \ll 2d$, the response time grows linearly with $\lambda$, but once $\lambda$ is large enough, it has a much larger impact on response time.

The models can also find peak system throughput by maximizing the active number of transactions $(1 - \beta)m$ ($\beta$ is the probability a transaction is blocked, see Eq. 27

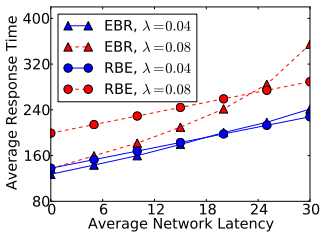



**Figure 21:** Impact of Arrival Rate $\lambda$ and Average Network Latency on Execution Time for Two System Types.

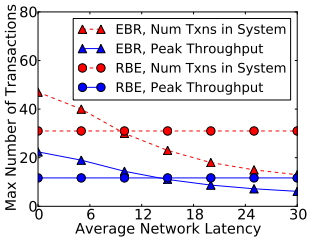



**Figure 22:** Impact of Average Network Latency on Maximum Throughput for Two System Types.

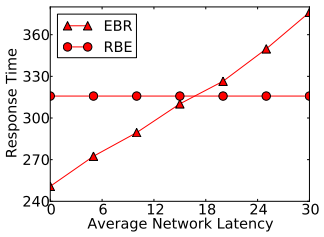



**Figure 23:** Impact of Average Network Latency on Execution time Under Max Throughput for Two System Types.

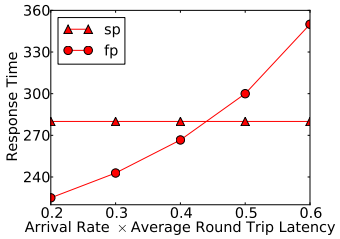



**Figure 24:** Comparison Between SP and FP.

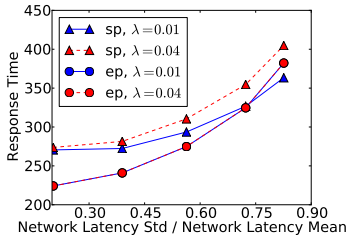



**Figure 25:** Comparison Between SP and EP.

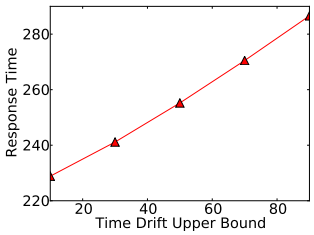



**Figure 26:** Impact of Time Drift on EP.

and Eq. 32) when increasing the number of transactions $m$. Figure 22 demonstrates the impact of network latency on peak system throughput and the number of concurrent transactions. When network latency is low, EBR can obtain a higher throughput than RBE (e.g., around two times when network latency is negligible). However, the peak throughput decreases drastically when network latency increases for EBR. The increase of network latency has no observable impact on the peak throughput of RBE as expected. Figure 23 also shows the response time when the system is at peak throughput. EBR always has a longer response time when network latency is large.

Next we compare the three schemes for the replication protocol. The number of replicas is set to 5. The network latency distribution is set to a log-normal distribution with average latency 100. Figure 24 demonstrates the impact of load occupancy (arrival rate $\times$ average service time) on FP when compared with SP. The figure

shows that when the occupancy is low (less than 45%), it is more beneficial to use the FP protocol to save the extra cross-replica delay for client to contact the leader. Since our model is an overestimation of the response time for FP, the threshold 45% is an underestimation. If we further simplify by letting $E(L) = T$ and $E(Q) = 2T$ and ignoring the reorder overhead of SP, the threshold is further underestimated and becomes the solution to $\frac{N}{\lambda} = 2\lambda T + \frac{(2\lambda T)^2}{2(1-2\lambda T)} \leq (2 + \frac{n-1}{n})T$, where $\frac{N}{\lambda}$ is the response time of FP from Eq. 37 and $(2+\frac{n-1}{n})T$ is an approximation of response time of SP from Eq. 36 without considering the reorder overhead. The result is $\lambda T = \frac{n-1}{2(2n-1)} \approx 25\%$. Therefore, a system with occupancy less than 25% should use FP as the replication protocol.

Figure 25 compares SP and EP with varying arrival rate $\lambda$ and network latency variance. As is shown in the figure, both protocols have an increased response time when network latency variance increases. This is resulted from the reorder overhead in RBE. Furthermore, the network variance has a larger impact on EP than SP. Therefore, when the network latency has a large variance, EP is less attractive. On the other hand, it is also shown in the figure that arrival rate has an impact on SP as well. This is again because of the impact of reorder overhead. When the arrival rate is large, there is a higher probability that a transaction with an earlier slot delays later ones. The arrival rate does not affect the EP since slots are proposed periodically which is independent of the arrival rate. Moreover, we can study the impact of time drift among replicas on the response time of EP. Assuming the distribution of time drift across replicas for each epoch is uniformly distributed from 0 to an upper bound, Figure 26 shows that the response time increases linearly with the time drift upper bound. This suggests for EP to work properly, the time drift across replicas should be kept within a reasonable limit.

## *3.5 Summary*

This chapter studied performance characteristics of geographically-replicated transaction datastores [125]. We focused on replica consistency where the system allows stale states on a subset of the replicas but disallows conflicting states. For such consistency requirements, we analyzed two types of systems (EBR and RBE) and three replication protocol schemes (SP, FP and EP) that are fundamental design options.

We developed analytical models to capture the performance characteristics of each design option. Models were validated through simulation with adequate details. We believe the assumptions made are reasonable and our models have sufficient accuracy under these assumptions. For our future work, we plan to generalize some of the assumptions. Specifically, we will consider cases when the transactions can have multiple classes; transactions acquire both shared and exclusive locks; the system can have both replicas and partitions; and data can support multiple version snapshot reads.

# CHAPTER IV

# NETWORK RESOURCE CONTENTION ON MAPREDUCE SYSTEMS

In Chapter 2 and Chapter 3, we show that data contention is a major limiting factor for the transactional data access abstraction. Many distributed systems, on the other hand, adopted an opposite philosophy that aims to share as little data as possible between processes/tasks. This is critical for distributed systems due to the high cost of communications. Various designs (e.g., [33, 82, 97]) have been proposed along this line of work. Among them, the MapReduce system is quickly gaining popularity because of its cost-effective design as well as growing community support.

The MapReduce programming model builds applications with two operators: map and reduce. Map tasks process the input data and subsequently emit key/value pairs. Each reduce task collects a list of values with the same key from all map tasks and generates new key/value pairs. In typical implementations of the program model, the input data is hosted on a distributed file system. The file system breaks large files into a series of blocks, which are replicated and randomly distributed across the cluster. Users define the functionality of map and reduce tasks as well as where the input files are. The execution system automatically orchestrates the execution of jobs, which includes scheduling, data transfer among the tasks, load balancing, as well as fault tolerance and user interactions.

Two methods are adopted by MapReduce systems to reduce data contention, and thus reducing potential serialization of the tasks, which is costly in distributed systems: (1) concurrent writing of a shared file is not allowed; (2) all applications need to be expressed by the programming model (i.e, there cannot be other operators

other than map and reduce). While these two methods do prevent certain applications from adopting the MapReduce model (mostly those applications that require intensive inter-process communications), it turns out that this model fits many distributed applications very well, especially those Big Data applications that can be formulated via the MapReduce programming model. With such a design, tasks do not need to compete for concurrent access to the same piece of data, either in memory or on disk, which, as discussed in previous chapters, is a major obstacle for transaction-based abstractions to achieve optimal parallelism. System scalability can therefore be significantly improved for applications that fit this program model.

While contention for shared data access becomes less of a concern, network resource contention emerges as a critical to system performance:

- Applications are often data-intensive. Typical applications of the MapReduce system have relatively high IO/Computation ratio, and often need to process data beyond memory capacity, which requires the usage of disks, and more importantly, requires the usage of multiple nodes. Data access occupies a large portion of execution time in MapReduce jobs which occurs during three phases: reading map inputs, shuffling the intermediate results emitted by mappers, and writing reducer outputs. Remote data accesses are involved in these phases: input data may be read from a remote data host when computation node is different from the storage node; shuffle stage has to pass data from all map tasks to all reduce tasks which inevitably involve communications among nodes across the cluster; and output data is written to multiple locations for fault tolerance.

- Scarcity of network resource becomes severe. The typical network architecture targeted by MapReduce has a tree topology where a cluster consists of multiple racks that connect to a top level switch while each rack is filled with leaf

compute nodes that connect to rack-level switch(es). For cost considerations, commodity hardware is often used, which has less over-all bisection bandwidth than those more expensive technologies. Under such circumstances, the top-rack switch bandwidth becomes a very scarce resource, especially as clusters scale to hundreds or thousands of servers.

In fact, MapReduce implementations are striving to reduce the impact of such contentions. Locality [21] becomes a critical concern and attracted extensive research interests (e.g. [2, 126, 43, 60, 28]). In our work, we attempt to attack the network resource contention problem from the following two aspects:

- To improve map input locality, the current approach assume a simple data access pattern for map tasks (i.e., each map task takes only one file partition as input). The approach, though covers a wide range of applications, becomes less effective when the assumption does not stand (i.e., each map task takes two file partitions as input such as in those pattern matching applications). Extending the current mechanisms to satisfy the map locality will allow a much wider range of applications to benefit from MapReduce.

- Reduce input locality (data transfer during the shuffle stage to feed data to the reduce tasks) is ignored by current implementation. However, this phase of data transfer is important for many applications whose mappers produce relatively large volume of data. Because the inherited all-to-all data access pattern of the shuffle stage, improving reduce locality would require clustering the map tasks close together, and this requires consideration in the trade-off between reducer and mapper localities.

In this chapter, we describe our approaches to improve data locality and reduce network resource contention in MapReduce systems. The rest of the chapter is organized as follows: We begin with a brief introduction of MapReduce system and

a review of relevant prior studies in Section 4.1. In Section 4.2, we introduce our solution to better support dual-input map tasks. Section 4.3 presents our grouping-blocks strategy and discusses the mechanisms to mitigating the undesirable effects of such strategy. Finally, in Section 4.4, we conclude this chapter with discussions and directions for future research.

## *4.1 Background and Prior Work*

In this section, we give an overview of the MapReduce/Hadoop systems. We then list the related works including locality in MapReduce and the MapReduce ecosystem in general.

### 4.1.1 MapReduce Overview

The MapReduce system adopts a programming model which is originally from functional languages. The programming model consists of only two user-defined functions: map and reduce. The map function takes in key/value pairs as input and generates new(intermediate) ones. The reduce function takes one(intermediate) key and a set of values associated with that key and generates new key/value pairs.

The jobs in the MapReduce system have strictly defined control flow, as is shown in Figure 27. Each job has one map phase followed by zero or one reduce phase. Input data is segmented into several splits; each map task iteratively apply map function on key/value pairs of the splits; the generated intermediate key/value pairs are shuffled to reduce tasks; reduce task, upon receiving a range of keys and the associated value lists, apply reduce function iteratively; and finally write output to file system. Users only need to write code for map and reduce function and the runtime system takes care of the rest of the control flow.

There are many MapReduce implementations targeting various hardware platforms such as graphic processing unit or multi-core architecture. In our context, we

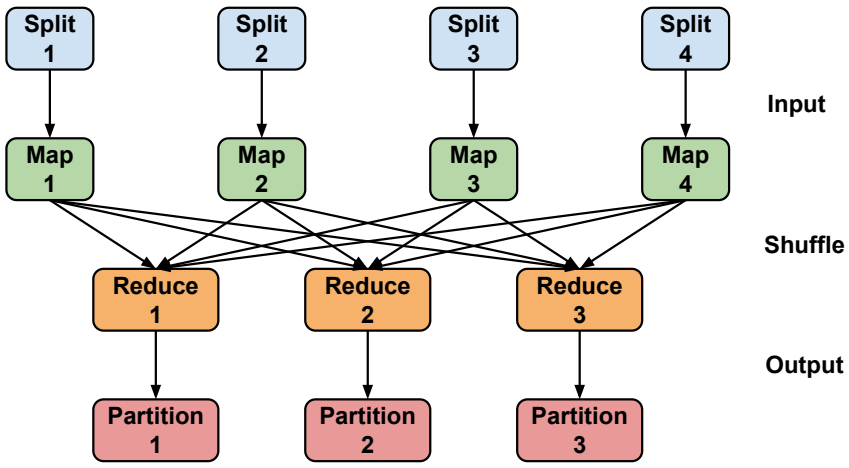



**Figure 27:** Phases of A MapReduce Program, including: (1) read input from splits; (2) apply map function; (3) shuffle intermediate data; (4) apply reduce function; (5) write output.

refer the MapReduce system specifically to the implementation designed in [21] targeting large scale distributed systems. Hadoop [117] is a widely used open source implementation of that design.

In Hadoop, the MapReduce system is composed of a storage system, Hadoop File System(HDFS), and an execution system which are responsible for storing data and executing tasks respectively. The two systems are assumed to be co-located on the same set of physical machines.

Documents in HDFS are partitioned into blocks, replicated and distributed to the storage nodes (called DataNodes) in the system. A centralized server node called the NameNode maintains the file system data structure and tracks the location of each replica block. Aside from the traditional APIs such as file open and read/write stream, the user interface of NameNode also provide a function call to obtain the locations of file blocks, which is used by the MapReduce execution system for map locality.

In the MapReduce execution system, a centralized job submission and scheduling server called the JobTracker is responsible for receiving client job submission and

scheduling tasks onto TaskTrackers. TaskTrackers are responsible for executing the tasks. TaskTrackers are co-located on the same physical compute nodes as DataNodes. In general, jobs are created by specifying the input documents, the output directory, the method to create splits from input documents, the map function, the reduce function and the number of reduce tasks. The JobTracker initializes one map task for each splits and several reduce tasks according to job specification. Map/reduce tasks are launched on TaskTrackers; the task runners call HDFS APIs and the key/value iterator of shuffling stage for data access; and map/reduce functions are applied iteratively on input/intermediate data.

TaskTrackers contact JobTracker periodically using remote procedure call (RPC). Both load balancing and fault tolerance are achieved through such heartbeat-like communication. Load balancing is achieved by including resource usage status in the heartbeat message from TaskTracker; new tasks are assigned when a TaskTracker has free resources. The heartbeat message also includes progress report of each running tasks such that when a task fails, it can be automatically launched elsewhere; furthermore, if the JobTracker does not receive heartbeat message from a TaskTracker for a long period, the TaskTracker is marked as unhealthy.

### 4.1.2 Locality in MapReduce Systems

The locality problem for MapReduce system is distinctive comparing with traditional computing systems in that:

- In the MapReduce system, the computation and the storage systems are co-located on the same physical machines. Traditional systems usually have separate physical entities dedicated to either storage or computation. For example, in desktop computers, computation is done inside the CPU pipeline; data requests are sent to memory and disk controllers which manage the memory or

disk where data is stored. As another example, in grid systems, there are compute nodes and storage nodes linked by high speed network; parallel file systems are developed to support data accessing from compute nodes to storage nodes. In such architecture, the system usually consists of a hierarchical storage system. The most used data is stored on nodes closest to the compute node in the hierarchy such that data can be brought into compute nodes fast. The distinctive architecture of MapReduce system is due to the consideration that such design is more cost-effective for data-center-scale setup.

- The MapReduce programming model has a well-defined control flow, i.e., every MapReduce jobs have at most three phases: map, shuffle and reduce. Map/Reduce task are specified before the job execution. There is no programming primitives that "fork" some new tasks during execution which is not only possible but very likely in multi-core and grid models. Such limitation on programming model is deliberately designed to ease the burden for the runtime system. In the context of locality, this gives the advantage that the scheduler can easily obtain and track locality information as well as ensure locality-aware task assignment.

- The MapReduce jobs have relatively regular data access pattern: (1) input data of map stage is usually in large chunks, often one or several HDFS blocks; the data access is mostly sequential instead of random access; (2) shuffling stage is an all to all broadcast from the map tasks of the same job executed; thus reduce tasks should be placed close to its corresponding map tasks.

Given the above observation that the MapReduce systems provide new context to data locality, next we discuss the problem in detail.

Figure 28 illustrates the cases in MapReduce systems where data access occurs including:

- Input Data Access. For each job, the user provide methods to create splits on

input documents and the MapReduce system creates one map task for each splits. A common unit of the document splits is HDFS block, i.e., each map task works on one HDFS block. As such, remote access incurs if the host of a map task is different from the host of input block. Figure 28(a) shows a data placement and task scheduling situation where node-local ($M1, M4$), rack-local ($M2$) and off-rack ($M3$) data access occurs. To obtain optimal locality, current MapReduce/Hadoop system adopts a simple mechanism for task scheduling. The HDFS is inquired for split location; a data structure is built for preferred map tasks (shown in Figure 28(a)); when a TaskTracker has free resources, the data structure is queried for a local map task to schedule. Such simple strategy only works when it is suitable to use one HDFS block as a split unit.

- Shuffle Data Access. The data access for shuffling exhibits an all-to-all communication pattern. Figure 28(b) shows a task scheduling case where node-local ($M3$), rack-local ($M1, M2$) and off-rack ($M4$) data access occurs. Note that it is difficult to avoid off-rack communication without sacrificing map locality. For example, we can schedule $M4$ on $N4$ to avoid off-rack access for reduce task; however, the input data access for $M4$ from $I4$ on $N7$ becomes off-rack. Because of such challenge, the current MapReduce/Hadoop implementation does not consider reduce task locality during task scheduling.

- Output Data Access. To ensure fault tolerance of output data, HDFS streams output data to multiple nodes on multiple racks. A commonly used replication factor is three. Figure 28(c) demonstrate the case where the output block is replicated on three nodes. The default mechanism of HDFS put one replica on the writing node and the other two on nodes within another rack. This strategy is the result of considering the trade-off between fault tolerance and off-rack data access, i.e., putting replicas on different racks can enhance resilience

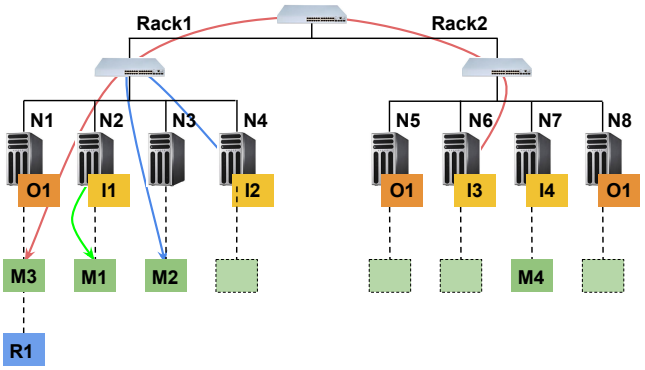
to rack failure while increasing the off-rack communication as well. To avoid off-rack data access, the third replica is put on the same rack with the second.

In summary, the locality problem in MapReduce system is distinctive from traditional computing system because of the special architecture targeted and the programming model. The data access usually occurs in map input, shuffle and output stage. As for the current system, there are still opportunities for improvement for locality in both map input and shuffle stage.

### 4.1.3 Related Works

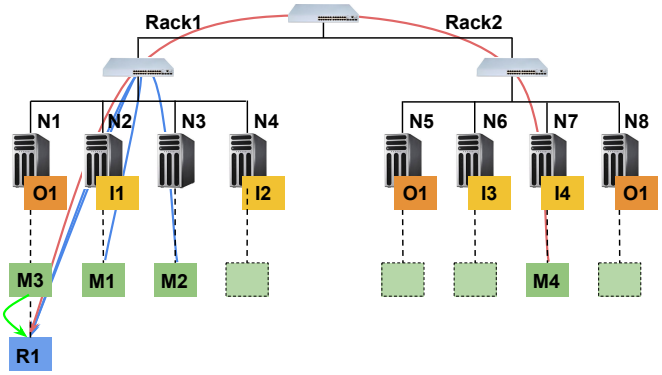
As locality is key to the performance of MapReduce systems, it has attracted a lot of research attention recently.

Some research efforts reduced this problem into some equivalent scheduling or graph problem. In[43], the authors discussed the non-optimality of the default Hadoop scheduling algorithm and found a well studied optimizer to improve it. The non-optimality was resulted from the simple decision that only assigned the tasks on available nodes one by one. It fails to consider cases where some nodes had sets of data which were local to many other tasks. In their work, the authors proposed to schedule available nodes and tasks together and optimize for map task locality. The optimization is reduced to a linear sum assignment problem. Quincy [60] transformed the scheduling problem into a minimum cost flow problem which took fair constraint into account as well. The graph vertices represented tasks, nodes and racks. Each edge was a possible task assignment on a node or a rack. Edge cost and capacities represented the locality cost and the fairness constraints. Bar [61] proposed a dynamic scheduling algorithm that calculated the best assignment of tasks iteratively during execution. The algorithm had two phases where a balance phase calculated the initial optimal assignment for all tasks and a reduce phase that tuned the initial decision dynamically according to the network and server load status.

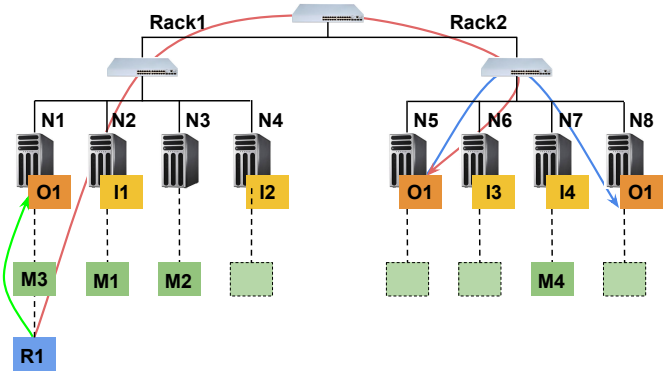



| Preferred Map Tasks | |
|---|---|
| N1 | M3 |
| N2 | M1 |
| N3 | |
| N4 | M2 |
| N5 | |
| N6 | |
| N7 | M4 |
| N8 | |
| Rack1 | M1, M2, M3 |
| Rack2 | M4 |

(a) Input Data Access




(b) Shuffle Data Access




(c) Output Data Access

**Figure 28:** Data Access in MapReduce Systems.

Delay Scheduling [126] identified several pathological cases of the existing Hadoop scheduling algorithms under a fairness constraint. It then proposed a "wait" technique to improve the locality: when no task from a job can be assigned to a local node, the scheduler waits for several turns such that more nodes will be available to schedule the job.

Several research efforts modified the architecture of the MapReduce system to better support the locality of a type of application. Haloop [13] extended Hadoop to improve the execution of iterative applications and Twister [27] proposed a light-weight MapReduce runtime for these applications. The two methods shared some similarities - they separated user data into a static set and a dynamic set so that the static user data was cached and reused across iterations.

Data placement strategies such as co-locating data or increasing replication factor have also been studied. In CoHadoop [28], an extension of HDFS was proposed to exploit co-locality among data blocks to reduce network data transfer. The co-locality information was manually discovered and configured. In Scarlett [2], the replication factors were adjusted for hot HDFS files to alleviate competition for such files. Demands for such adjustments of hot files were detected and analyzed periodically. Several strategies were discussed to prioritize the demands under disk capacity limits. In [67], the data placement, replica selection and co-location problems were considered all together. The problem was transformed into a hyper-graph representation. Data sets were represented by vertices and tasks were represented by hyper-edges. Heuristics were proposed to decide which data to replicate and which nodes to place the data.

Purlieus [86] discussed the data placement in terms of locality for both map and reduce stages. The authors proposed to couple the scheduling decision on both data placement and task placement to reduce the amount of data transfer in a cloud environment. Applications were categorised into map-input heavy, map-and-reduce-input

heavy and reduce-input heavy jobs. Placement techniques were proposed for each of the job type. LARTS [46] and CoGRS [45] discussed optimal reduce task placement. The placement of reduce tasks was obtained by choosing the node closest to all the map tasks in a rack that hosts the maximum number of map tasks. Furthermore, in CoGRS [45] data size skewness in map tasks was also considered to alleviate the reduce straggler problem. Orchestra [18] proposed techniques to schedule the shuffling stage in a finer granularity.

Memory locality was considered in PACMan [3]. A major discovery in their work was that hit-ratios do not necessarily improve job completion times and therefore special coordinated cache replacement policies were developed. Piccolo [90] and Spark [127] were in-memory runtime systems that share many similarity with the MapReduce system. They were optimized for iterative machine learning workloads.

MapReduce is currently under intensive study. We further list here several important lines of the ongoing researches aside from locality.

Many applications are discovered suitable for the MapReduce model and some corresponding libraries or systems are accordingly built. In [83], the authors illustrated that many machine learning applications can be applied to use the MapReduce model. Pegasus [72] showed the applicability of MapReduce model to graph algorithms. Algorithmic modifications were also discussed to improve performance. Mahout [73] was developed as a scalable data mining library developed on top of MapReduce system. Latin [85] and Map-Reduce-Merge [119] were developed on top of Hadoop to better support rational operations. Hive [114], developed by Facebook, was a warehouse system for Hadoop to support SQL-like languages. HBase [35] was a data storage system that implements the google BigTable architecture. Giraph [37] was a graph processing platform on top of Hadoop.

Other research efforts have been made to build similar or enhanced systems for different design focus. Sector [41] was a system originally developed for efficient

data access over wide area networks. It claimed Sector performs 2-20 times faster than Hadoop. Several works [127, 27, 90] developed in-memory systems by exploiting the fact that some applications reuse dataset, especially for iterative workloads. Mesos [59] and YARN [120] were redesigns of the MapReduce system that separated the responsibility of scheduling and progress monitoring into two components such that (1) the system becomes more scalable and (2) systems other than MapReduce, such as MPI, can also run under this framework.

Resource allocation and fairness is also a topic under extensive research. Hadoop Fair Schedulers [30] and Hadoop Capacity Scheduler [17] were schedulers developed in Facebook and Yahoo to solve the small job starvation problem in the default FIFO scheduler in Hadoop. Flex [118] developed a framework to optimize any of a variety of standard scheduling theory metrics. In [36], the problem of fairness among different resource types, such as CPU bound or memory bound workloads, were considered. The authors proposed a fairness model called dominant resource fairness, which satisfied several highly desirable properties.

Because progress monitoring and speculative execution are also major focuses of MapReduce system, researchers have also explored some techniques in this area. Zaharia et al. [128] improved the scheduling algorithm for a heterogeneous environment. A scheduling algorithm called LATE was proposed to accurately estimate the progress of tasks and was therefore highly robust under heterogeneous environment. In [4], the authors presented a system that monitors tasks and identifies the outliers of a job. The proposed system then restarted outliers in a network-aware manner and protected outputs of tasks to improve performance. ARIA [116] proposed a framework to let a scheduler meet service level objectives. The framework first built job profiles according to executed task history; then a model was used to estimate the amount of resource required to meet the deadline; finally a soft deadline based scheduler was used to determines the job ordering and the amount of resources required for each

job.

## *4.2 Improving Map-locality for Dual-input Applications*

In the current MapReduce/Hadoop systems, the locality-awareness of Hadoop is based on a relatively strong assumption that *a task is expected to work on a single data split*. In practice, a split typically consists of one data block, or a part of it. After all, this is what allows Hadoop to label a compute node as *local* or *remote* for scheduling purposes. This is in accordance with the MapReduce programming model, which defines one map task over each logical data split and thus requires users to describe the mapper function as a unary operator, applicable to only one single logical data split.

The unary-input requirement works well for many applications such as document processing. However, many other applications require more flexible operators. For example, a task in a pattern matching application would naturally take two inputs: one record of the template data, and another record of the stored data. For such applications, the unary input oriented Hadoop system has multiple limitations: (1) Developers need to work around the unary input requirement, which makes it less natural to program the applications. (2) When a workaround method is used, the built-in locality awareness of Hadoop becomes less effective or non-effective. (3) As dual-input tasks often share their data blocks, there are many unique locality optimization opportunities in these applications that cannot be exploited by existing Hadoop.

Motivated by the above observation, we study Dual-Hadoop, an extension of the Hadoop system to improve the execution of dual-input applications. We make the following contributions in this section:

1. We designed an easy-to-use interface for users to describe the association between a task and its inputs.

2. We developed a task scheduling algorithm that is able to exploit data locality for dual-input applications.

3. We designed and implemented a caching mechanism to accelerate data reads. The caching mechanism is an integral part of our extension that materializes the improved data locality exposed by our scheduling algorithm.

Extensive experiments were conducted to verify the effectiveness of Dual-Hadoop. The performance of the scheduling algorithm was tested against a wide range of dual-input task patterns, which shows that our algorithm reduces remote data reads by up to 48% when compared with existing Hadoop scheduling algorithm. Experiments on two actual applications (a pattern matching application and PageRank) were conducted on a 64-node Amazon EC2 cluster. The results show that our method improve the execution speed of these applications by up to 3.3x over the native Hadoop system.

The rest of the section is organized as follows: Our motivation is presented in detail in Section 4.2.1. In section 4.2.2 we give an overview of our extension. Section 4.2.3 presents the detailed design of our extension. Section 4.2.4 illustrates the experimental results.

### 4.2.1 Motivation for Dual-Hadoop

An extension for the map locality of the dual-input applications is necessary because (1) Hadoop currently does not naturally support such applications; (2) such data access patterns exhibit special locality.

#### *4.2.1.1 Lack of Support for Dual-Input Applications in Hadoop*

While Hadoop has good support for unary input applications, it does not handle dual-input applications very well. Take tiled matrix vector multiplication algorithms for example, a task would naturally consist of one matrix block and one vector block. Since existing Hadoop only supports unary input tasks, users need to use one of the

following workaround methods to program such applications:

- use an extra level of indirection in the input format to give the system an illusion of single input split that is actually one matrix block plus one vector block. This workaround method involves user defining a new InputFormat class which would significantly increase programming difficulty.

- use the Hadoop *distributed cache* utility to duplicate one input set at `all` the nodes so they can locally access this set of data, and Hadoop only needs to handle the other (one) input set. This method will not work when input data set exceeds the storage capacity of the distributed cache. Furthermore, duplication of an entire data set is often wasteful since each node will most likely access only part of the set.

- use an extra round of MapReduce job to concatenate the two input blocks for each task, and save the new *merged* data blocks onto HDFS for the actual MapReduce job. This workaround method needs to move a significant amount of data and cause expensive overheads. Additionally, if a block is to be shared by multiple tasks, the block will be duplicated multiple times in this data-preprocessing stage, which further increases the overheads.

- use two file system calls directly in the user-supplied mapper functions to read the two splits. This workaround method is easy to program. But the data reads are invisible to the Hadoop system. Without knowing which data blocks may be accessed by a task, the Hadoop scheduler cannot perform locality-aware scheduling, which will result in excessive data transfer overheads.

#### *4.2.1.2 Data Locality in Dual-Input Applications*

Tasks in dual-input applications often share their input data blocks. For example, in tiled matrix multiplication algorithms, the same block from matrix $A$ will multiply

with multiple blocks from matrix $B$. For another example, in a genome comparison application, an individual genome may need to be compared against multiple other genomes in a database. Apparently, the tasks (multiplying two matrix blocks, or comparing a pair of genomes) share their inputs.

This introduces a unique type of data locality: if tasks can be grouped together such that their overall data footprint can be minimized, then the group of tasks can benefit from reduced data transfer if they are co-assigned to the same compute node. The amount of data transfer can be further reduced if the assignment can utilize data blocks that are local to the compute node. Note that existing Hadoop is unable to exploit such data locality as it is designed with unary-input applications in mind.

Clearly it is necessary to improve Hadoop to better support dual-input applications. In the following discussion, we will present our proposed Hadoop extension that can significantly improve the data locality for such applications.

### 4.2.2 Dual-Hadoop Design Challenges and Overview

Our extension aims to address the following design challenges:

- *Programmability.* We need to extend the programming interface so that users can specify the inputs to the tasks. We want the interface to be easy to program so that users can focus on the main functionality of their applications, rather than dealing with the interface itself.

- *Transparency.* The next goal is to make it easier for users to achieve high performance. For this reason, we do not want the users to be even aware of the data locality issues. Dual-Hadoop is simply an improved version of Hadoop that can execute dual-input applications faster. For this reason, we reject all design alternatives that require users to track/handle the locations of the data blocks.

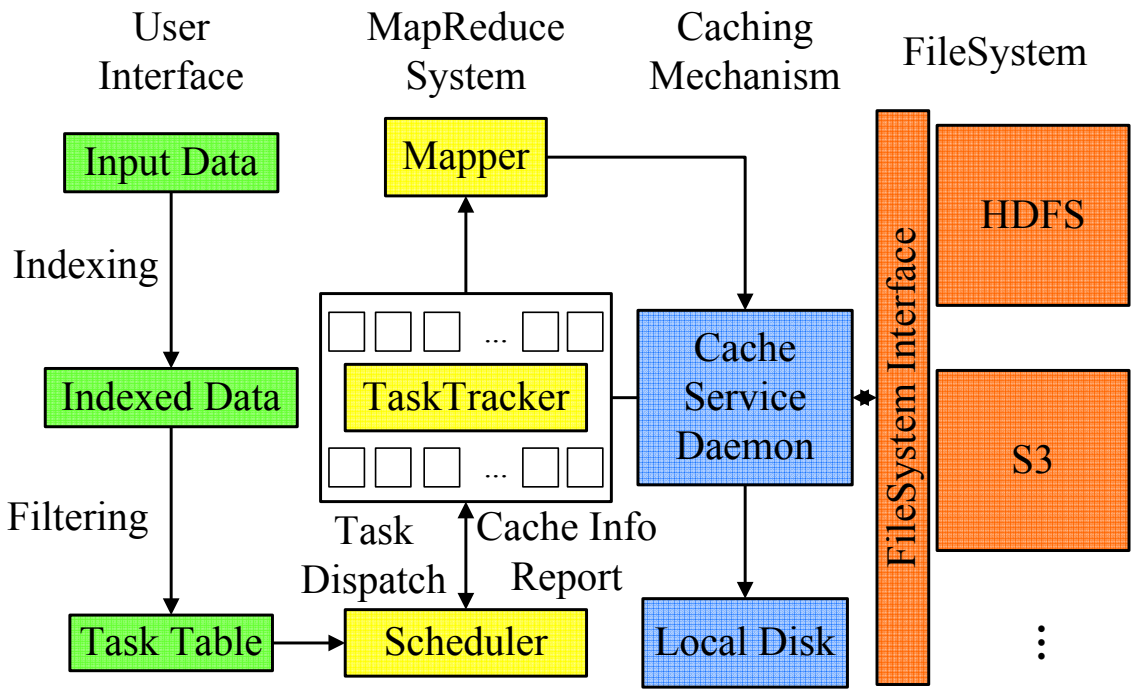



**Figure 29:** Dual-Hadoop Extension System Overview

- *Non-intrusiveness.* Dual-Hadoop takes certain amount of memory resources at the compute nodes to speed up dual-input applications. We want Dual-Hadoop to be fully bypassable when the system is executing unary-input applications, with close-to-zero overheads. Furthermore, if a user application tries to use up the available memory, we want Dual-Hadoop to gracefully disappear in the background and yield the resources to the user application.

Figure 29 illustrates an overview of Dual-Hadoop, which contains the following components: (1) the input interface, (2) the caching subsystem, and (3) the dual-input locality-aware scheduler.

- *Input interface.* This component assigns IDs to splits. Dual-Hadoop inherits the default Hadoop output format and adds a hook so that an ID can be designated to each split. In Dual-Hadoop, tasks are generated by calling a user-defined filter function that specifies which two splits would form a valid task. Tasks are internally represented as a 2-D matrix using either dense or sparse format

depending on the application. The IDs assigned by the user will be used to identify the splits in the user filtering functions as well as by the scheduler.

- *Scheduler*. The scheduler first obtains the task representation by applying the filter function from the user interface, then gathers information about the locations of the input data blocks (which DataNode has which blocks), and subsequently calculates a locality-optimized execution schedule. During the course of the execution, the scheduler monitors the content of the caches on the fly, and fine-tunes the schedule according to dynamic locality information supplied by the caching subsystem.

- *Caching subsystem*. This component runs on each compute node and is designed to cache the input splits accessed by existing tasks (with the expectation that they will also be needed by subsequent tasks). The caching subsystem sits between MapReduce system and HDFS system, therefore it is user transparent. It supplies a split to the requesting map task if the split is in the cache, and will seamlessly resolve to the native HDFS data read mechanism when the requested split is missing in the cache.

In the following, we will discuss the details of Dual-Hadoop design, and analyze why it is capable of exploring data locality for dual-input applications.

### 4.2.3 Dual-Hadoop Design Details

In the following, we discuss our design in detail including the user interface, the scheduling algorithm and the cache mechanism.

#### *4.2.3.1 User Interface*

The Dual-Hadoop user interface is designed to assign IDs to splits by letting the user name splits with strings.

```
1 // The Dual-Hadoop filter interface
2 public interface BiHFilter {
3  public boolean accept(String split0Id, String split1Id);
4 }
5
6 // A usage example: matrix-vector multiply
7 public class MatVecMulFilter implements BiHFilter {
8  public boolean accept(String split0Id, String split1Id) {
9   if (!isAMatrix(split0Id)) return false;
10  if (!isAVector(split1Id)) return false;
11  colId = getMatrixColId(split0Id);
12  rowId = getVectorRowId(split1Id);
13  if (colId == rowId) return true;
14  return false;
15 }
16 }
```

**Listing 4.1:** Dual-Hadoop User Interface for Defining Tasks

An application can have one of the following input data formats: (1) Each user input file has a granularity small enough to define a split. In such case, file names can be naturally used as the split ID. Users can set a flag in Dual-Hadoop to specify such configuration. (2) Each input file is structured and contains multiple file splits. In this case, users just need to provide an ID file according to a predefined format, listing split IDs and the file segments that each split maps to. (3) The input files are unstructured. In this case, users need to do a preprocessing step to structure their input files. Similar methods was also used in other works [72, 28, 24] to pre-process unstructured data. The resulting structured files can then be handled as in case (2). We extend the default Hadoop OutputFormat class to provide simple utilities so that the name files can be easily generated along with the preprocessing step.

Users specify the map tasks by customizing a filter class, which returns true if a pair of split IDs form a task, and false otherwise. Listing 4.1 illustrate the simple interface and a usage example of matrix vector multiplication. Users can manipulate the ID strings in a customized fashion (such as $getMatrixColId()$ in Listing 4.1) to identify the file split and form the tasks.

#### *4.2.3.2 The Locality-aware scheduler*

The scheduler weaves all the components together in Dual-Hadoop. Once a MapReduce job is submitted, the scheduler first creates an internal presentation of the tasks. The scheduler will then monitor the locality of the data splits (disk replicas and copies in the cache subsystem), exploit data sharing pattern among the tasks, and assign tasks to optimize data localities for the tasks. Scheduling in Dual-Hadoop is performed in three phases:

*Phase 1: Task Generation*

Dual-input applications have two sets of input splits, $A$ and $B$, and a task will take a split from $A$ and another one from $B$. Note that $A$ and $B$ may overlap, either partially or completely.

In this phase, we run the user-supplied filter function (discussed in the user interface) and generate an internal presentation of the tasks in the form of an incidence matrix $I$. The matrix uses one row (and column) to represent a split in $A$ (and $B$). If there exists a task whose input splits are $a \in A$ and $b \in B$, then we have $I(a, b) = 1$ indicating the presence of this task. Note that the matrix may be dense or sparse depending on the characteristics of the job. Subsequently, the storage of matrix $I$ will take the dense or sparse forms accordingly.

Figure 30 illustrates an example of the incidence matrix for matrix-vector multiplication. The matrix is of size 2 by 4 blocks and the vector is 4 blocks. Each value 1 in the matrix indicates a task that multiplies a matrix block with a vector block.

| | 1, $V_0$ | 2, $V_1$ | 3, $V_2$ | 4, $V_3$ |
|---|---|---|---|---|
| 1, $M_{00}$ | 1 | | | |
| 2, $M_{10}$ | 1 | | | |
| 3, $M_{01}$ | | 1 | | |
| 4, $M_{11}$ | | 1 | | |
| 5, $M_{02}$ | | | 1 | |
| 6, $M_{12}$ | | | 1 | |
| 7, $M_{03}$ | | | | 1 |
| 8, $M_{13}$ | | | | 1 |

**Figure 30:** Incidence Matrix Example

*Phase 2: Static Task Grouping*

Taking the incidence matrix as input, we partition the rows and columns into groups such that tasks within the same group share their input file splits. The algorithm is listed in Figure 31.

In the algorithm, groups are formed such that a row should be included in a group if 80% of the columns from this row has the same value(0 or 1) as the columns from the representative row in the group. The representative row is chosen as the row in the group with the most 1's. Similarly, the algorithm is also used to form column grouping. Both row and column grouping results will be used in the next phase.

The static grouping phase provides an insight into the relation between tasks: tasks from the same group are likely to share (some) input splits, and if we assign them to a common compute node, we will see reduced data transfers.

*Phase 3: Dynamic Task Dispatching*

The dynamic task dispatching phase is executed during run time, it decides which node should execute which tasks, and the goal is to reduce data transfers while maintaining load balancing across the compute nodes. To achieve the goal, this phase uses the static grouping result as a guide to reduce data transfers, and further considers the following input information: (1) what replicas does each node have? (2) what splits is a node currently caching?

Before describing our dynamic dispatching algorithm, let us define a new term: a

```
1: procedure STATIC_GROUPING(taskMatrix)
2:     for all row ∈ taskMatrix do
3:         foundGroup ← False
4:         for all group ∈ groupList do
5:             repr ← group.repr
6:             smaller ← getSmaller(row, repr)
7:             if size(row ∩ repr) > threshold * size(smaller) then
8:                 group.add(row)
9:                 foundGroup ← True
10:            end if
11:        end for
12:        if foundGroup == False then
13:            groupList.addNewGroup(row)
14:        end if
15:    end for
16:    return groupList
17: end procedure
```

**Figure 31:** Algorithm for the Static Grouping Phase

*task pack*, which is the set of tasks that will be executed together by a compute node. Task pack is the unit of actual task dispatching. Our scheduler takes two steps to form a task pack: first we choose a group from Phase 2 that most splits local, then we pick a pack from the group so that it can fit into the cache of the local compute node.

We use the following criteria when forming task packs: (1) the number of tasks in a pack should not exceed the total number of tasks dividing the number of nodes; (2) the difference between the number of row splits and the number of column splits is small; (3) at most half of the cache will be used for row splits or column splits. The first criterion is a heuristic to ensure load balance; the second criterion is to ensure data reuse and thus reduce data transfers; and the third criterion is a heuristic to the following optimization problem: maximize $n_1 * n_2$ subject to the constraint that $s_1 n_1 + s_2 n_2 \leq C$, where $n_1, n_2$ denote the number of row and column splits, $n_1 * n_2$ denote the number of tasks to be executed and $s_1, s_2$ denote the size of row and column splits and $C$ denote the cache capacity.

```
1: procedure DYNAMIC_DISPATCHING(groupList, cache, replica)
2:     ▷ Find best group
3:     local ← cache + replica
4:     max ← 0
5:     for all group ∈ groupList do
6:         count ← #blocks in both local and group
7:         if count > max then
8:             best ← group
9:             max ← count
10:        end if
11:    end for
12:    ▷ Packing
13:    calculate maxRowSize, maxColSize, maxSize
14:    numRows ← 0
15:    numCols ← 0
16:    size ← 0
17:    for all rowsplit ∈ best.rows() do
18:        if numRows ≥ maxRowSize then
19:            break
20:        end if
21:        if size ≥ maxSize then
22:            break
23:        end if
24:        if not replica.contains(rowsplit) then
25:            size ← size + rowsplit.size()
26:        end if
27:        rowSplits.add(rowsplit)
28:        numRows ← numRows + 1
29:    end for
30:    ▷ do the same for columns
31:    ...
32:    for all rowsplit ∈ rowSplits do
33:        for all colsplit ∈ colSplits do
34:            if hasTask(rowSplit, colSplit) then
35:                pack.add(getTask(rowsplit, colsplit))
36:            end if
37:        end for
38:    end for
39:    return pack
40: end procedure
```

**Figure 32:** Algorithm for the Dynamic Dispatching Phase

Figure 32 outlines this dynamic dispatching phase. Task packs will be created such that (1) tasks in a pack share their input splits; and (2) the input file splits are likely to be already has a local replica or in cache. Tasks in a pack are then scheduled onto the corresponding compute nodes (represented by its TaskTracker) one by one.

To take load balance into account, when no more packs can be formed for an idle TaskTracker, i.e., when all tasks have been already assigned to some pack, our scheduler falls back to the default Hadoop scheduling algorithm so that this idle TaskTracker can steal tasks from some pack that is assigned to some other TaskTracker. While this gives the TaskTracker some work to do, it may cause less than optimal data transfers (as this is an out-of-pack task). Nonetheless, this is no worse than what the native Hadopp would do.

#### *4.2.3.3 Caching Subsystem*

The caching subsystem has two components: a file handler object and a service daemon. The handler object is constructed when opening files. The service daemon sits on top of the file system abstraction of each compute node (between MapReduce and HDFS systems). We design it in this way because we want the service daemon to remember the caching history among jobs. Furthermore, the service daemon can function for any Hadoop supported file system.

When users want to open a cache-enabled file, they use an openCachedReadOnly function. The function returns our specialized file handler and users read data as usual using this handler. The openCachedReadOnly function accepts an optional versionID parameter besides the usual path parameter. We expect users to change this versionID if the data is modified. If the cached version is not equal to the user provided version, the block will be re-fetched. The handler checks if the current reading position is within the cached block or local replica boundary. If yes, the handler continues to read, otherwise, the handler sends a cache block request to

the service daemon using a remote procedure call(RPC). Upon receiving the RPC response, the handler updates its status and proceeds to read.

The service daemon serves handlers' requests, manages the cached blocks and reports caching status to the TaskTracker for scheduling. When it receives a caching request, it checks if the required data is in the cache. If not, the daemon uses the usual file system API(such as HDFS API) to read the data and saves blocks into local file system. The maximum block size is fixed(default 64M) for our design. If the underlying file system has a block size larger than our cache block size configuration, that block is segmented into smaller blocks. The cached blocks are evicted using a least recently used policy if the capacity is reached. Each time when a TaskTracker needs to send a heartbeat, it also reports the status of the cache with the heartbeat.

#### *4.2.3.4 Developer Transparency*

With Dual-Hadoop, users will only need to perform one extra piece of work than with existing Hadoop: specifying which two splits form a task. Other than this, they just write Hadoop programs as they currently do: focusing on the mapper and reducer functions. The three phases of our scheduler as well as the caching subsystem are transparent to the users. The users do not need to know how tasks are grouped together to reduce their data footprint, or how task packs are formed to take advantage of local data replicas and cached splits, or how splits enters and leaves the caches. Dual-Hadoop executes in the background to accelerate the execution of dual-input applications.

#### *4.2.3.5 Non-intrusiveness*

The non-intrusiveness of Dual-Hadoop has two aspects:

1. If the application is unary-input as with existing Hadoop applications, our scheduler will detect the non-existence of the user-supplied input filter, and

immediately hand everything over to the existing Hadoop scheduling subsystem. The overhead in this case will be minimal. The caching service will still run on the compute nodes. But they will not be activated and thus will only consume a minimum amount of resources.

2. Dual-Hadoop also consumes certain amount of memory when the caching subsystem attempts to cache file blocks at the nodes. To this end, Dual-Hadoop takes advantage of the existing Linux file caches, which by itself is elastic, meaning that it yields memory to user programs as they demand more memory. In such cases, Dual-Hadoop will seamlessly yield memory to user programs (by caching less). Note that the application will not benefit from the faster speed provided by caches, but they will still benefit from our task grouping process (with reduced data footprint).

### 4.2.4 Experimental Results

Extensive experiments were conducted to evaluate Dual-Hadoop. We performed (1) simulation-based studies to verify the effectiveness of our scheduling algorithm, and (2) execution of real applications to demonstrate the performance improvement over the existing Hadoop system.

#### *4.2.4.1 Effectiveness of the Scheduling Algorithm*

For this set of experiments, we studied a wide range of task sharing patterns to evaluate how well our scheduling algorithm can reduce remote data reads. We simulated a 64-node Hadoop system running HDFS with the following parameters:

- HDFS data blocks were 64MB (the default Hadoop setting), and were randomly distributed across the cluster with a uniform distribution.
- Compute nodes became available (and subsequently request new tasks) in a random order. This emulated random task execution time.

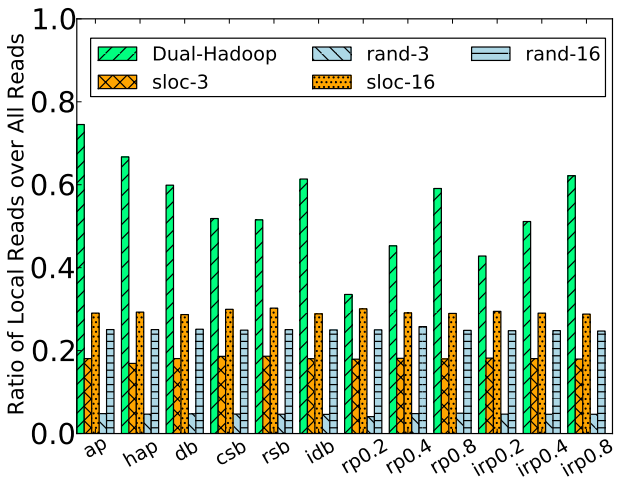
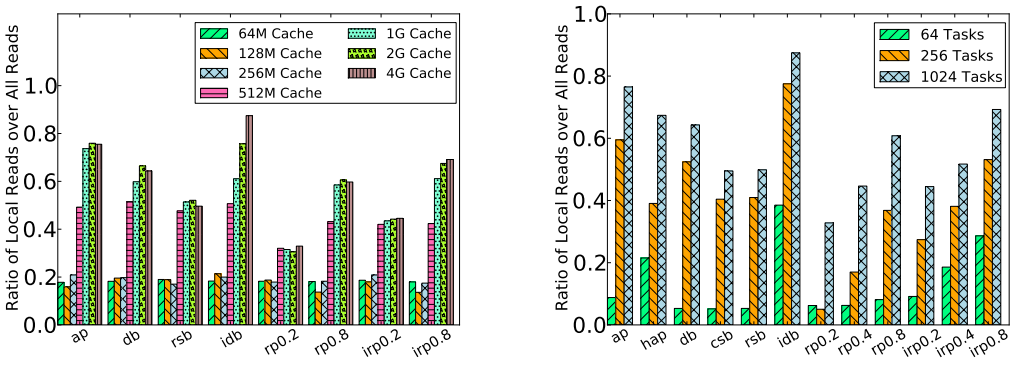



**Figure 33:** Dual-Hadoop Scheduling Performance Comparison with Default Hadoop. `sloc-*` and `rand-*` are workaround methods of Hadoop with replication factor set to 3 and 16

The following task input patterns were evaluated: (1) `All Pair Pattern(ap)`, which has a full incident matrix;(2) `Half All Pair Pattern(hap)`, which has a triangular incident matrix; (3) `Diagonal Block Pattern(db)`, which has a diagonal block incident matrix; (4) `Circular Shuffled Diagonal Block Pattern(csb)`, which circularly shuffles the rows and columns of diagonal block matrix; (5) `Random Shuffled Diagonal Block Pattern(rsb)`, which randomly shuffles the rows and columns of diagonal block matrix; (6) `Iterative Diagonal Block Pattern(idb)`, which is a series of jobs with Diagonal Block Pattern; (7) `Random Pair Pattern(rp0.x)`, which whether an entry is 1 or 0 is randomly generated according to the density level x; (8) `Iterative Random Pair Pattern(irp0.x)`, which is a series of jobs with Random Pair Pattern. Among the patterns, All Pair Pattern, Half All Pair Pattern and Iterative Diagonal Block Pattern are extracted from real applications, the others are synthetic patterns.

Figure 33 compares the scheduling algorithm between Dual-Hadoop and the default Hadoop. As discussed previously, the default Hadoop needs to use a workaround




(a) Cache Capacity

(b) Total Number of Tasks

**Figure 34:** Performance Impact of Cache Capacity And Number of Tasks

method for dual-input applications. We tested the following two workaround methods: (1)`sloc`, which updates Hadoop's InputFormat such that Hadoop can schedule according to the locality of one data split (still cannot do anything about the second split), (2) `rand`, in this workaround method, the two splits are directly accessed using file system API calls inside the map functions, and Hadoop can only randomly assign the tasks because it does not know what data a task may request.

In this set of experiment, there were 1024 tasks, each having two 128MB input splits. The cache capacity was set to 1024M bytes. As is shown in Figure 33, the Dual-Hadoop scheduling algorithm consistently outperforms the Hadoop scheduling methods (both sloc and rand), by up to 75%. In an attempt to help Hadoop increase the ratio of local reads, we increased the Hadoop replication factors from 3 to 16 (sloc-16 and rand-16 in Figure 33). But Hadoop only benefited from a moderate increase of local reads, which are still up to 60% lower than Dual-Hadoop scheduling. This is because our scheduling and caching mechanism tend to group data blocks that are co-accessed or re-used by the tasks. Blindly increase the replication factor does not help much, for example, the probability that two data blocks are co-located on one node is only 25% even for a replication factor high as 16 in a 64-node system.

We then studied the impact of cache capacity and the results are shown in Figure 34(a). The experiments were configured with 1024 tasks, each having two 128MB input splits. The per node cache capacity varied between 64MB and 4096MB. The results show that the ratio of local reads stayed relatively low at 20% when the cache is smaller than 256MB, which is reasonable because 256MB is the minimum size to accommodate the input data for at least one task. Once cache capacity passes this threshold, the hit rate increases with larger caches, and can ensure that ∼50-70% data reads are locally satisfied when we have 1024MB or larger caches.

The number of tasks also has an impact on our Dual-Hadoop scheduling algorithm. This is illustrated in Figure 34(b). We varied the number of tasks from 64 to 1024, and the results show that a larger number of tasks leads to more local data reads. This is because more tasks per node provides more opportunity to exploit data sharing among the tasks.

In summary, this set of experiments show that, compared with the existing Hadoop task scheduling strategy, our Dual-Hadoop scheduling algorithm can significantly reduce the amount of remote data reads for dual-input applications. And this is achieved with a relatively low requirement on the cache capacity. Furthermore, our scheduling algorithm works better when the application scales up: the more tasks there are, the better our algorithm can exploit the data sharing characteristics among the tasks.

#### *4.2.4.2 Experiments with Actual Applications*

For this set of experiments, we ran real applications using a cluster of Amazon EC2 medium instances. Each instance was configured with 3.75GB memory, 2 EC2 compute units, 410 GB instance storage and runs 64-bit Amazon Linux AMI. We applied our extension on Hadoop stable release 1.0.4. HDFS replication factor was set to 3 (default setting) if not mentioned otherwise.

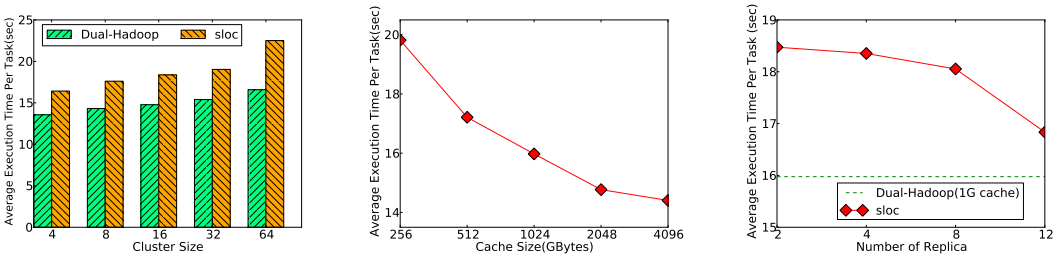
Two applications were tested: (1) a pattern matching application that features

the all pairs sharing pattern, and (2) PageRank where tasks share inputs with an iterative block diagonal matrix pattern.

The pattern-matching application is abstracted from the earthquake analysis problem studied in [78] where each template earthquake waveform is compared against each recorded waveforms (to automatically identify aftershocks). The programming implementation had two sets of floating-point arrays, $A$ - the templates, and $B$ - the recorded waveforms. Each array in $A$ will perform a correlation analysis against each array in $B$. For the baseline Hadoop implementation, we implemented a workaround solution that combines two arrays from set A and B as a split. This has the similar effect as the `sloc` scheduling strategy we simulated.

Figure 35(a) compares the performance between Dual-Hadoop and the baseline. In this set of experiments, all arrays were 256M bytes. Set A always contained 4 arrays. The number of arrays in set B was set to 4 times the number of nodes. Cache size was set to 2GB per node. Figure 35(a) shows that Dual-Hadoop out-performs the baseline in all cluster sizes by up to 26.3%. The speedup increases with cluster size (from 17.4% at 4 nodes to 26.3% at 64 nodes), which shows Dual-Hadoop scales better than the Hadoop baseline.

Figure 35(b) shows the impact of cache size. Obviously the execution time decreases when the cache size increases. Furthermore, changing cache size from 256MB to 1GB already accounts for more than 70% of the speedup, which verifies the simulation results in Section 4.2.4.1 that our algorithm has a relatively low requirement on cache capacity to exploit the data sharing patterns. Figure 35(c) compares our extension with a naive strategy that simply increases the replication factor of HDFS. It can be seen that simply increasing the number of replication helps very little unless the replication factor reaches 12 replicas per block, which will unfortunately need an excessive amount of storage for the replicas and still cannot outperform Dual-Hadoop that only uses 1GB cache per node.




(a) Cluster Size (b) Cache Capacity For Dual-Hadoop (c) Replication Factor For Hadoop Naive Approach

**Figure 35:** Performance Impact of Various Configurations For Pattern Matching. `sloc` is a workaround method of Hadoop.

Next we study the effectiveness of our approaches on PageRank application. PageRank is a well-known algorithm to rank web pages according to their significance. It is heavily used by search engines and is often implemented using MapReduce. The core of the algorithm is an iterative matrix vector multiplication, where the matrix represents the link connectivity among the web pages, and the vector represents the ranking of the pages.

The baseline Hadoop implementation is an optimized variation of a common implementation [72, 13, 71] that uses two MapReduce jobs in each iteration. As Dual-Hadoop extension naturally supports dual-input applications, we were able to design a matrix-vector multiplication based algorithm. The same Amazon EC2 Cloud instances were used for this set of experiments. We used a semi-synthetic dataset called Livejournal [7] as the input data. The size of the original data is expanded from 2GB to 32GB in accordance with the increase in cluster size.

Figure 36 shows that our method achieves speed up of up to 3.3x over the baseline. Two factors contribute to the speedup: (1) our method eliminated the extra MapReduce job in the baseline, which was used to copy data just to work around Hadoop's unary input limitation; (2) our locality-aware scheduling algorithm and caching subsystem further reduces the execution time. A breakdown of the execution time shows that factor (1) contribute to around 3x speed up and our extension can

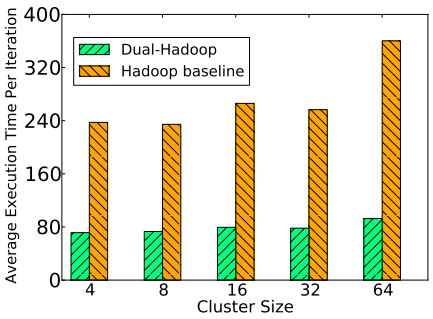


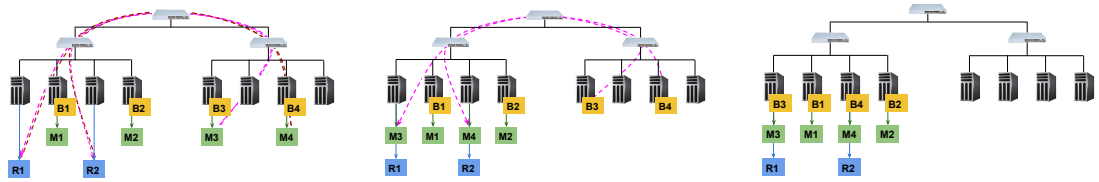

**Figure 36:** Performance Comparison for PageRank

further improves around 10% performance on top of that. Figure 36 also shows that as the cluster size increases, native Hadoop experienced larger increases in the average execution time per iteration (34.1% from 4 nodes to 64 nodes), this validates the better scalability of Dual-Hadoop.

## *4.3 Improving Map/Reduce Co-locality with Grouping-Blocks Strategy*

In the previous section, we extend the MapReduce runtime to better support map locality for dual-input applications. We further discuss the problem of reduce locality in this section. Little effort has been made for the reduce-task locality in the current MapReduce/Hadoop implementation. On the other hand, many applications have a large amount of shuffle data comparable to map input, for example, more than 90% applications shuffle more than half the size of map input data in three production clusters reported in [94], which makes the locality problem in reduce phase more significant.

Reduce tasks acquire data from every map task of the same job. It is inherently difficult to avoid off-switch network communication due to the all-to-all nature of the shuffle operation. To achieve minimum off-switch data access, both map and reduce tasks must be scheduled within a few racks. Such scheduling decision, however, may often hurt map locality because input blocks are highly likely scattered across the

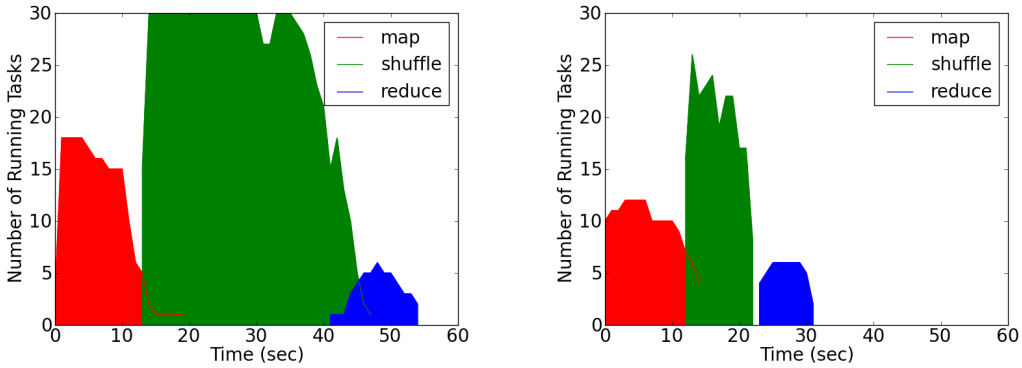



(a) Scattered Blocks with Map Locality (b) Scattered Blocks with Reduce Locality (c) Grouped Blocks with Both Map and Reduce Locality

**Figure 37:** Demonstration of the Benefit of the Grouping-blocks Strategy. When blocks are scattered, it is difficult to satisfy both map and reduce locality. Minimum off-switch data access can be achieved by grouping blocks in a few racks.

whole cluster. To satisfy both map and reduce locality, we propose to group the input blocks in a few racks. Figure 37 illustrates and compares the three data placement and scheduling options; by grouping data blocks and scheduling tasks accordingly, minimal off-switch data access can be achieved. Figure 38 illustrates the impact of the grouping-blocks strategy on job execution time for a multi-job workload with two Sort jobs and two Data Generation jobs. We ran four jobs at the same time each with a 4 GB dataset in a 4-rack 24-node cluster. It showed that when we grouped the input data of Sort jobs, the performance of both applications were improved. By limiting the amount of off-switch data access, the Sort jobs spent less time on shuffling which result in a less execution time; in addition, more bandwidth were used by the file system for replicating data blocks, hence the improvement on Data Generation jobs.

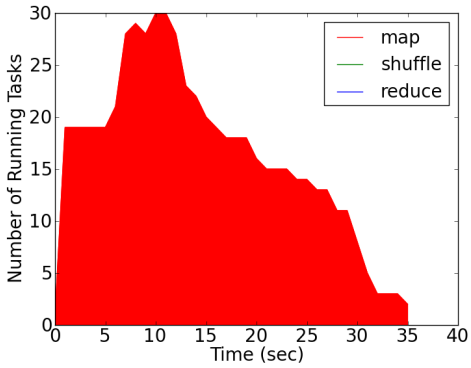
Given the above observation, a natural question is how the MapReduce runtime accommodate such grouping-blocks strategy. The presence of grouped blocks brings out new concerns for data placement and task scheduling. For data placement, we need to answer the questions such as: Which files to group? How many racks to host the grouping blocks? For task scheduling, the questions include: How to schedule tasks according to the locations of the grouped blocks? And when such assignment is effective? In general, the solutions revolve around the trade-off between the amount of off-switch data access and parallelism. Specifically, the down side of the grouping-blocks strategy is loss of parallelism which manifests into two effects termed “sticky



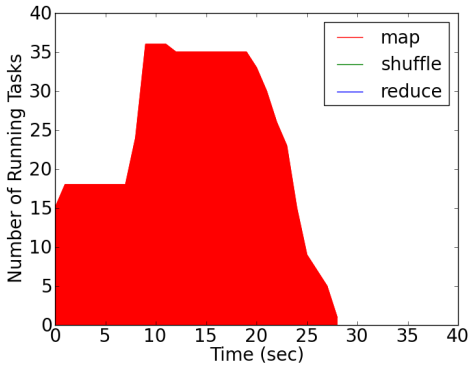

(a) Sort with Blocks Scattered




(b) Sort with Blocks Grouped




(c) TextGen with Sort Blocks Scattered




(d) TextGen with Sort Blocks Grouped

**Figure 38:** Demonstration of the Impact of the Grouping-blocks Strategy. When the data blocks of the Sort application were grouped, the job execution time of both Sort and TextGen was improved.

effect" and "conflict effect" in this paper. Consider a case where we group all the blocks of a job inside one single rack and schedule all the map and reduce tasks on that rack. In such case, we perfectly avoid all the off-switch data access. However, such decisions may have an adverse impact on parallelism: (1) the "sticky"" effect occurs because we enforce all tasks to be executed on the assigned rack such that the job cannot run more tasks even when nodes in other racks have available computation slots; (2) the "conflict"" effect occurs if other jobs also group the blocks in the same rack such that all these jobs compete for the limited number of slots on this rack. In a word, accommodating the grouping-blocks strategy in MapReduce requires new mechanisms to mitigate the "sticky"" and "conflict"" effects in both data placement

and task scheduling.

In this section, we studied the impacts of taking advantage of grouped blocks in a MapReduce system and designed new mechanisms to control data placement and task scheduling to mitigate the disadvantage. Specifically we make the following contributions:

- We proposed a grouping-blocks strategy that can greatly reduce the off-switch data access and hence the job execution time.
- We identified the trade-off of such grouping-blocks strategy between the improvement on off-switch data access and loss of parallelism.
- We proposed data placement and task scheduling mechanisms to mitigate the adverse impact of grouping-blocks strategy on parallelism.
- We conducted extensive experiments to study the impact of the grouping-blocks strategy on off-switch data access and job execution time and to validate the effectiveness of our proposed mechanisms.

The rest of the section is organized as follows: We present an overview of the issues when accommodating the grouping-blocks strategy in the system in Section 4.3.1. Section 4.3.2 and 4.3.3 presents the detailed design of our data placement and task scheduling mechanisms. Section 4.3.4 illustrates the experimental results.

### 4.3.1 Problem Overview for Accommodating Grouping-Blocks Strategy

In this section, we present the desired characteristics of the data placement and task scheduling mechanisms to accommodate the grouping-blocks strategy in MapReduce/Hadoop.

Each file block in the MapReduce file system has multiple replicas. Our strategy gathers one set of the replicas within a few racks and leaves the other replicas untouched. For simplicity, we call those special files with a subset of replicas grouped

"G-files"; the special racks that host these grouped blocks "G-racks"; and the special jobs that execute on the G-files "G-jobs". For now, we assume concurrently only one G-job executes on a G-file, otherwise, we can extend our mechanism to adopt the approaches in Scarlett [2], increase the replication factor and treat the file as multiple G-files.

#### *4.3.1.1 Trade-off of the Grouping-blocks Strategy*

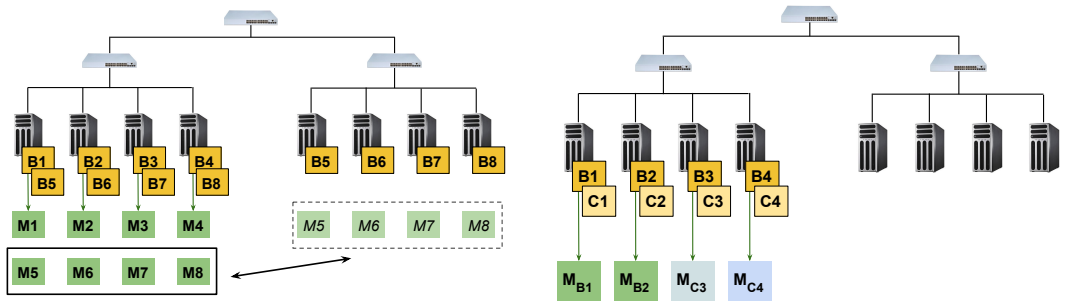
As is stated above, the problem of both data placement and task scheduling with respect to the grouping-blocks strategy concerns the trade-off between the amount of off-switch data access and parallelism. While grouping blocks can avoid much of the off-switch access, it can have an adverse impact on parallelism, which exhibits in two ways. Figure 39 illustrates the two situations where the parallelism of a G-job is degraded.

- Sticky effect. The parallelism of a G-job is limited by the capacity of its G-racks. A larger set of G-racks promises more parallelism. A smaller set of G-racks, however, produces less off-switch data access. Therefore, both data placement and task scheduling methods should minimize for the amount of off-switch data access while keeping the capacity of the G-racks acceptable.

- Conflict effect. G-jobs competing for the same set of G-racks degrades the parallelism of both. The grouping-blocks strategy is more effective with more G-jobs. On the other hand, this increases the chance of competition. Therefore, both data placement and task scheduling methods should minimize the probability of conflict.

#### *4.3.1.2 Data Placement for the Grouping-blocks Strategy*

We adopted an off-line data placement mechanism, i.e., the mechanism is launched by a separate file system process without interfering with the job execution. Using




(a) "Sticky" Effect (b) "Conflict" Effect

**Figure 39:** Demonstration of loss of parallelism. Because we force the tasks to execute on the G-racks to reduce off-switch data access, the job cannot execute more tasks even if the cluster is under load. Moreover, other jobs may compete for the G-rack which further degrades the parallelism.

collected job execution statistics, our mechanism places a replica for each block of the G-files in a designated location. The mechanism is also assumed to operate within a storage budget to control the interference with regular cluster operation.

The concerns of the data placement manifest into two problems:

- Selecting the candidate jobs for grouping. Some jobs are more suitable to apply the grouping-blocks strategy, for example, jobs with only map tasks do not need the strategy for reduce locality. The data placement mechanism needs to select the set of G-jobs that maximize the improvement on off-switch access under the budget limit. Furthermore, the selected candidates is required to be less susceptible to the "sticky" effect.

- Deciding the locations for the grouped blocks. After candidate selection, the mechanism computes the locations of the G-racks while avoiding the "conflict" effect.

#### *4.3.1.3 Task Scheduling for the Grouping-blocks Strategy*

Given the existence of G-racks, the task scheduler is required to perform the following functionalities:

- Discovering the locations of G-racks. It is possible to identify the G-racks by passing the additional information through the interface between file system and the scheduler. However, a preferable design is to keep the current interface intact which requires the scheduler computes the locations of G-racks on its own.

- Applying the grouping-blocks strategy. The scheduler should assign tasks of G-jobs according to the locations of G-racks. On the other hand, even though the data placement mechanism has taken into account the "sticky" and "conflict" effect during G-rack placement, the scheduler should still judge the impact of such effects and turn off the grouping-blocks strategy in case the decision made by the data placement mechanism is less effective.

- Mitigating the "sticky" effect. When a G-job requires more computation capacity than that the G-racks can afford, the scheduler needs to detect such case and turn off the strategy.

- Mitigating the "conflict" effect. When two G-jobs compete for a G-rack, the scheduler should turn off the strategy for one of the jobs.

### 4.3.2 Detailed Techniques for Data Placement

The data placement mechanism selects candidate G-jobs and decides the locations of G-racks for those jobs. The candidate selection attempts to maximize the improvement on off-switch data access under a budget limit while avoiding the "sticky" effect. The location of the G-racks is decided such that the "conflict" effect is mitigated.

#### *4.3.2.1 Collecting Statistics and Hints*

We list the statistics and hints used by our mechanism in Table 8. For each file $f_i$ and its corresponding job, we collect the file size $s_i$, map input/output ratio $\mu_i$, maximum number of map slots $m_i$, maximum number of reduce slots $r_i$, file occurrence

**Table 8:** Useful Statistics and Hints.

| | | |
|---|---|---|
| $s_i$ | file size | required for candidate selection and location decision. |
| $\mu_i$ | map input/output ratio | required for candidate selection. |
| $m_i$ | max number of map slots | required for candidate selection. |
| $r_i$ | max number of reduce slots | required by candidate selection. |
| $q_i$ | occurrence frequency | optional for location decision, default to 1. |
| $n_i$ | non-concurrent group name | optional for location decision, default to file name. |

frequency $q_i$ and the non-concurrent set name $n_i$. The file size $s_i$ is used for both candidate selection and location decision which is maintained in the file system and readily obtainable; $\mu_i$ is used to maximize the improvement on off-switch data access which can be obtained from counters in job execution history; $m_i$ and $r_i$ are used for mitigating the "sticky" effect which can be obtained in scheduler history statistics; both $q_i$ and $n_i$ are hints for avoiding the "conflict" effect; $q_i$ can be computed through execution history; $n_i$ is a hint provided by the user; a non-concurrent set is a set in which the files cannot occur simultaneously, which is explained in detail in Section 4.3.2.4.

#### *4.3.2.2 Deciding the Number of Racks*

The "sticky" effect occurs when the capacity of the G-racks cannot keep up with the demand of the G-job. Hence a sufficient condition to avoid such effect would be setting the capacity of the G-racks larger than the maximum demand, namely,

$$R_i = max(\lceil \frac{m_i}{C_m} \rceil, \lceil \frac{r_i}{C_r} \rceil), \tag{43}$$

where $R_i$ is the number of G-racks for file $f_i$; $C_m$ and $C_r$ are the capacity of map and reduce slots for a rack.

While such sufficient condition seems to yield a large $R_i$ which undermines the improvement on off-switch data access, we list cases when $R_i$ can be small:

- Small jobs, that is, the total number of map and reduce tasks are small. For example, with a configuration of 3G file size, 128M block size, 4 map slots per node and 6 nodes per rack, the blocks of the file can be fit into one rack. Small jobs can be prevalent in some cluster workload (e.g., M45 in [94]).

- Limited job pool capacity, that is, some job pools are of lower priority such that the administrators set small values for the maximum number of map and reduce slots to limit the total demand of jobs in that pool.

- Busy cluster, that is, when there are a large amount of jobs running in the cluster, each job can only have a small share of map and reduce slots in a fair share setting.

#### *4.3.2.3 Selecting the Candidates*

Given that the "sticky" effect is mitigated by setting a proper number of G-racks, $R_i$, we further select G-jobs to maximize the improvement on off-switch data access under the budget limit. The problem can be formalized as

$$\begin{aligned} \text{maximize} \quad & \sum I(f_i) \\ \text{subject to} \quad & \sum s_i \leq B \end{aligned}$$

Where $B$ is the budget limit and $I(f_i)$ is the improvement on off-switch data access. The improvement has three parts: map input, shuffle and reduce output, that is, $I(f_i) = I_m(f_i) + I_s(f_i) + I_r(f_i)$. The current scheduling can achieve nearly perfect map locality [126] and so is the case when using the grouping-blocks strategy, therefore, $I_m(f_i) \approx 0$. Since we did not modify the file system replica location algorithm,

```
procedure SELECT_CANDIDATE(B, file list)
    Used budget, used ← 0
    F ← List of files sorted in descending order by key function skey_muRs
    Candidate set, C ← empty set
    for all file f_i ∈ F do
        if used + s_i > B then
            break
        end if
        add f_i to C
        used ← used + s_i
    end for
    return C
end procedure
function skey_muRs(f_i, s_i)
    ▷ For each file, generate the weighted improvement for sorting
    return I(f_i)/s_i
end function
```

**Figure 40:** Candidate Selection for Data Placement

$I_r(f_i) \approx 0$. Therefore, $I(f_i) \approx I_s(f_i)$. To compute $I_s(f_i)$, for simplicity, we ignore the skewness among job tasks. The improvement is the difference between with or without using grouping-blocks strategy, i.e., $I_s(f_i) = D_o(f_i) - D_w(f_i)$. The total amount of shuffle data is $H_i = \mu_i s_i$. Without our strategy, most of the shuffle data goes through top-rack switch, i.e., $D_o(f_i) \approx H_i$; with the strategy, each reduce task receives $\frac{1}{R_i}$ of the data from the same rack, i.e., $D_w(f_i) = (1 - \frac{1}{R_i})H_i$. Therefore,

$$I(f_i) = \frac{\mu_i}{R_i} s_i. \tag{44}$$

The optimization is an integer linear programming problem which is NP-hard. We resorted to a heuristic shown in Figure 40. The files are sorted by a weighted improvement $I(f_i)/s_i$ such that smaller jobs with larger improvement are preferable candidates and selected first. The procedure stops until we reach the budget limit.

#### *4.3.2.4 Deciding the Locations of the G-racks*

The major focus when choosing the locations of G-racks is to avoid the "conflict" effect, that is, we want to minimize the maximum probability that two G-jobs conflict

for any G-racks. We adopted a greedy approach shown in Figure 41. For each file $f_i$ to be placed, we find a set of $R_i$ racks that have the least probabilities that another file $f_j$ will conflict with $f_i$. Therefore, for $f_i$, we sort all racks according to the value $q(r) = Pr(A \mid B)$ in descending order, where $A$ is the event that none of the G-files (other than $f_i$) on rack $r$ has its G-job being executed; $B$ is the event that the G-job of $f_i$ being executed on $r$. Let $A_j$ be the event that the G-job of $f_j$ *not* being executed on $r$. If events in $A_j$ are mutually independent, then $Pr(A \mid B) = \prod Pr(A_j \mid B)$.

The conditional probability depends on the joint probability of $A_j$ and $B$, which is difficult to obtain. To simplify the computation, we focused on only two common types of dependency relationships: (1) two G-job executions are totally independent; (2) two G-job executions cannot occur simultaneously. Relationship (1) can be common among jobs of different users, i.e., the execution of jobs of one user does not depend the others; relationship (2) can be common among jobs for the same user, i.e., a user executing jobs one after another. Therefore,

$$Pr(A_j \mid B) = \begin{cases} 0, & A_j \text{ and } B \text{ only one can happen} \\ Pr(A_j), & \text{otherwise, } A_j \text{ and } B \text{ independent} \end{cases}$$

We let users provide the hints of relationship (2) by defining a non-concurrent set name $n_i$ for each file $f_i$. Files with the same $n_i$ will not run simultaneously.

Accurately computing the probability that the G-job of file $f_j$ being executed (i.e., $1 - Pr(A_j)$) is difficult. We estimate such by assuming that the probability increases linearly with the occurrence frequency $f_i$, the file size $s_i$ and the reciprocal of number of G-racks $\frac{1}{R_i}$, that is, when a file occurs more frequently and has larger size, the corresponding job occupies a rack for a longer time; if the file blocks spread on multiple G-racks, the job occupies all racks for a shorter time. Therefore, we can obtain

$$p_i = \frac{1}{norm} \frac{q_i s_i}{R_i}, \tag{45}$$

where, $norm = \sum \frac{q_i s_i}{R_i}$ is a normalization factor. And $q_j = Pr(A_j) = 1 - p_j$.

Interestingly, if we assume all jobs are independent and have occurrence frequency equal to 1, the sort key becomes

$$q(r) = \prod (1 - \frac{\frac{s_j}{R_j}}{norm}) \approx 1 - \frac{\sum \frac{s_j}{R_j}}{norm},$$

that is, our algorithm reduces to selecting the racks with least amount of G-files for the next G-file, which is a simple yet intuitive algorithm to balance the workload which is also adopted by the MapReduce file system for replica placement.

### 4.3.3 Detailed Techniques for Task Scheduling

The task scheduling mechanism discovers the locations of G-racks from the input location information from file system. It then decides whether to apply the grouping-blocks strategy when executing the job. Such decision is based on whether the job is susceptible to both the "sticky" and "conflict" effect.

#### *4.3.3.1 Discovering the G-rack Locations*

To keep the interface between scheduler and file system intact, we adopted a design that the scheduler figures out the locations of G-racks by itself. The location information passed from the file system under the original interface contains the replica locations of all blocks. Based on such information, we attempted to find a set of racks of minimum size that contains all the input blocks. Such problem can be reduced to a minimum vertex cover problem where racks are represented by (hyper-)edges and blocks vertices, which is an NP-complete problem. Therefore we resorted to a greedy approach. We first count the number of block replicas for each rack and sort the racks in descending order according to the count. Then the racks are selected until all the blocks are covered. Figure 42 lists the pseudo code for discovering the G-racks.

```
procedure DECIDE_LOCATION(G-file list)
    Table of files on each rack, T ← empty table
    for all f_i ∈ G-file list do
        R_i ← Number of G-racks for f_i
        n_i ← Name of the non-concurrent group for f_i
        Set of G-racks, S ← pick(n_i, R_i, T)
        Place blocks of f_i on S
        update(S, f_i, T)
    end for
end procedure
function pick(n_i, R_i, T)
    S ← empty set of racks
    A ← List of racks sorted in descending order by key function skey_prob
    add first R_i racks to S
    return S
end function
function skey_prob(r, n_i, T)
    ▷ for each rack r, given n_i and T, generates a probability for sorting
    Table of files on r, F ← T[r]
    Prob. of no conflict, q ← 1
    ▷ F is indexed by non-concurrent set names
    for all n ∈ F.keys() do
        if n ≠ n_i then
            Prob. of occurrence, p = 0
            for all f_i ∈ F[n] do
                p_i ← Prob. of f_i using G-rack r in Eq. 45
                p ← p + p_i
            end for
            q ← (1 − p)q
        end if
    end for
    return q
end function
function update(S, f_i, T)
    for all rack r ∈ S do
        n_i ← Name of the non-concurrent set for f_i
        add f_i to T[r][n_i]
    end for
end function
```

**Figure 41:** Location Decision for Data Placement

#### *4.3.3.2 Applying the Grouping-blocks Strategy*

During task scheduling, we require the mechanism to enforce the task assignment of G-jobs onto their designated G-racks. Furthermore, flexibility to disable the strategy under various circumstances is also required. Figure 43 shows our modification to the

```
procedure DISCOVER_GRACKS(blockLocations)
    G-rack set, G ← empty set
    Count number of block replicas for each rack
    L ← List of racks sorted in descending order by replica count
    Included block set, S ← empty set
    while S.size() < total number of blocks do
        r ← pop first from L
        add r to G
        add all blocks on r to S
    end while
    return G
end procedure
```

**Figure 42:** Discovering the G-rack Locations

default scheduling algorithm. Each job has a *Decider* to judge whether the grouping-blocks strategy should be turned on or off. When a node has free slot and the decider choose to apply the strategy, we check if the node is within the G-rack; if yes we assign a task from this job, otherwise we skip the assignment for the node.

We maintain a timer to track the time skipped for the task assignment of the job. Because when the strategy is turned on, the job can only assign tasks to the G-racks. Such design is vulnerable to faulty conditions, i.e., if the G-racks are not responding, the jobs are delayed indefinitely. Adding a timer can avoid such situation.

#### *4.3.3.3 Deciding to Use the Grouping-blocks Strategy*

The decision of whether to apply the grouping-blocks strategy depends on whether the job is susceptible to the "sticky" or "conflict" effects. We check several criteria illustrated in Figure 44. We first judge whether the job is suitable for the strategy by checking the map input output ratio and the size of G-racks against some threshold (in our experiments $Thr_{\mu} = 0.5$ and $Thr_{R} = 3$). Such check is necessary since the scheduler computes the G-rack locations on its own not knowing the decisions in the candidate selection. Next we check if the job can be affected by "sticky" effect by checking both the fair share of the job and the history average of the pool it belongs

```
procedure ASSIGN_TASKS(node)
    maxWait ← the maximum time we wait for fault tolerance
    ▷ Maintain two variables for each job j, initialized as
    j.wait ← 0, j.skipped ← false
    whenA heartbeat is received from node n
    for all job j do
        if j.skipped = true then
            j.wait ← j.wait+ time since last heartbeat
            j.skipped ← false
        end if
    end for
    if n has a free slot then
        Sort jobs as is the way in the default scheduling
        for all j ∈ jobs do
            apply ← j.decider.judge()
            G ← Set of G-racks for j
            if j.wait > maxWait or not apply then
                Use default scheduling algorithm
            else if n ∈ G then
                Assign a task in j to n
            else
                j.skipped ← true
            end if
        end for
    end if
end procedure
```

**Figure 43:** Applying the Grouping-blocks Strategy to Task Scheduling

to. Last we check if the G-racks is already occupied by some other G-jobs to avoid the "conflict" effect. If all criteria are met, we update the occupied G-racks and decide to apply the strategy.

### 4.3.4 Experimental Results

In this section, we present the experimental results and analysis. Extensive experiments were conducted to evaluate the impact of grouped blocks and the effectiveness of our mechanisms.

```
procedure JUDGE(job, scheduler)
    ▷ Check if suitable for grouping
    if job.μ ≤ Thr_μ then
        return false
    end if
    G ← Set of G-racks of job
    if G.size() ≥ Thr_R then
        return false
    end if
    ▷ Check "sticky" effect
    if job.fairShare > G.capacity then
        return false
    end if
    pool ← scheduling pool of the job
    ave ← Average fair share of the pool in history
    if ave > G.capacity then
        return false
    end if
    ▷ Check "conflict" effect
    O ← Occupied G-racks maintained by scheduler
    if G ∩ O ≠ ∅ then
        return false
    end if
    ▷ Update and return
    O ← O ∪ G
    return true
end procedure
```

**Figure 44:** Deciding to Use the Grouping-blocks Strategy

#### *4.3.4.1 Methodology and Settings*

We resorted to both real execution and simulation in our experiments. Real execution experiments were conducted to study (1) the impact on job execution time of grouping-blocks strategy; (2) the effectiveness of the candidate selection in data placement; (3) the effectiveness of the method to avoid "sticky" effect in task scheduling; (4) the effectiveness of the methods to avoid "conflict" effect in both data placement and task scheduling. Simulations are conducted to study the impact of grouped blocks in large scale including the impact on map task locality and the number of occupied G-racks.

The real execution experiments were run on a private cluster of 24 servers. Each server was equipped with two 2.33GHz quad-core Intel Q8200 CPUs, 8GB memory, a disk with 50GB capacity and 3.0Gbit/s SATA interface, and was running CentOS with kernel version 2.6.32. We used and implemented our mechanisms on Hadoop version 1.2.2. The network topology was configured with 4 racks each had 6 servers and the 4 racks were connected by a top-rack switch. Each switch had a 2000M bps data transmission rate. The Hadoop cluster was configured with 3 map slots and 1 reduce slots for each server because of the number of CPUs and the fact that most applications we ran had more map tasks than reduce tasks. HDFS replication factor was set to 3 and block size 128MB.

We used two types of applications: (1) real application including WordCount, Sort and TextGen, and (2) synthetic applications. The real applications were selected from example applications in the Hadoop repository and were widely used in other research works [108, 45, 86]. The applications were chosen to represent jobs with different map-input-output ratio: WordCount has large amount of map input with very few output; Sort has equal large amount of map input and output; TextGen has no map input but a large amount of output. Synthetic applications were implemented to set arbitrary map-input-output ratio. The input size of the applications ranged from 2 GB to 6 GB in all experiments.

We used a tuple of parameters $(u, j, w_u, w_j)$ to construct the arrival of applications for experiment workloads where $u$ is the number of users, $j$ is the number of jobs each user attempts to run in sequence, $w_u$ is the waiting time between the users starting their first job (user 2 starts $w_u$ seconds later than user 1) and $w_j$ is the waiting time between subsequent jobs for each user. Such construction is general enough to represent a range of workload. For example, when $u, w_u$ is small, the workload simulates a closed system with multiple users iterating the cycle of submitting a job and waiting for the result; when $w_j$ is large, it simulates an open system with multiple

batches of a $w_j$ interval and $u$ jobs each batch.

We implemented our simulator using a Python simulation framework, Simpy [103]. The simulator implemented the MapReduce system in detail including the exact procedure of map-locality-aware scheduling, the Fair scheduler and the control flow of map/shuffle/reduce tasks execution. Fault tolerance and speculative execution was omitted as they are less relevant; Job launching and scheduling overhead was also omitted. For simplicity, we ignored the interference of resource sharing, e.g., disk I/O and network bandwidth; and set tasks as of fixed execution time. We simulated a cluster of 256 servers and 64 racks; 512 jobs were submitted; jobs arrived following Poisson process; all jobs had a fixed amount of map and reduce tasks (36 maps and 12 reduces) with a fixed amount of execution time (10 sec map, 1 sec shuffle and 10 sec reduce).

#### *4.3.4.2 Impact of Grouping-blocks Strategy*

We first studied the impact of the grouping-blocks strategy with respect to various applications, amount of G-files, G-file sizes and G-rack sizes.

We constructed a workload with the parameter tuple $(5-8, 3, 5, 5)$. That is, we had 5 to 8 users; each user ran 3 jobs; users started with five-second intervals and intervals between jobs were also 5 seconds. We separated the users by 2 groups: 4 of them ran applications including WordCount, Sort and TextGen; the rest always ran Sort. We increased the budget of the candidate selection algorithm in accordance with the number of users such that the amount of G-files ranged from 20% to 50% of the total data (the algorithm automatically selected the Sort files first). Each data file is of the size 2 GB. For each workload, we ran two experiments with and without our grouping-blocks mechanisms.

Figure 45 shows the experiment result with/without grouping-blocks strategy. For most of our experiment settings, the workload with the grouping-blocks strategy

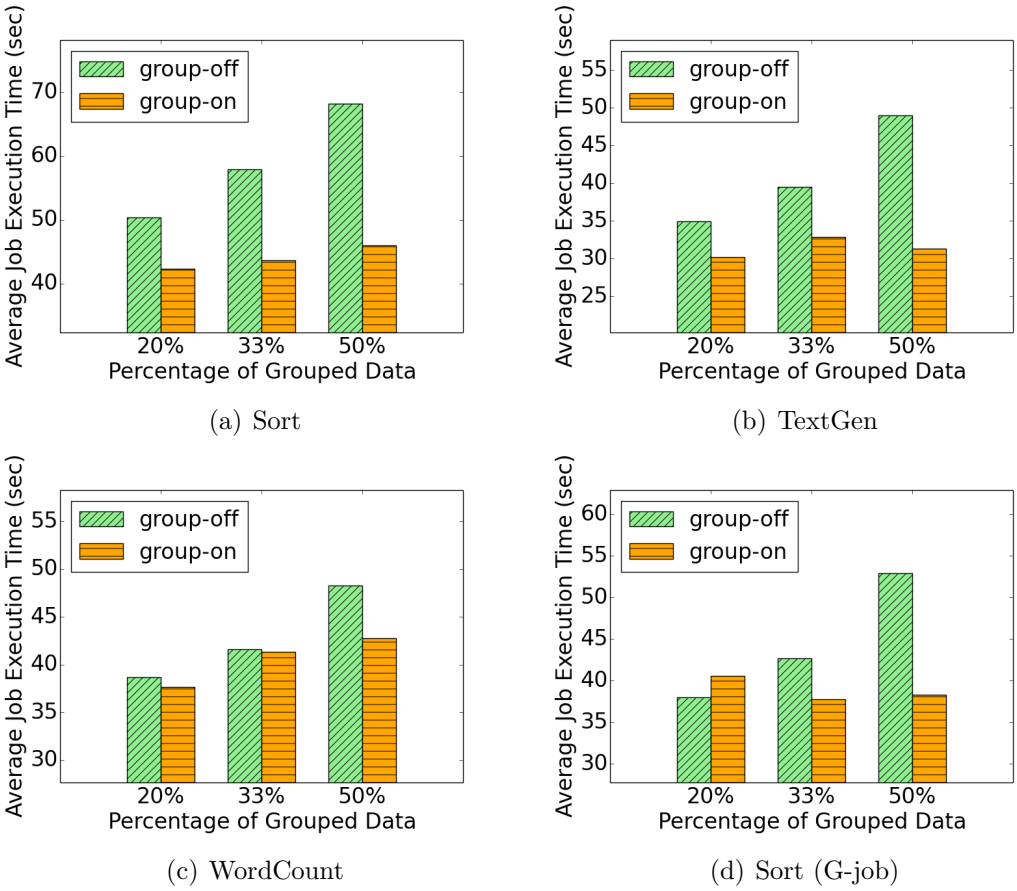



**Figure 45:** Impact of Grouping-blocks Strategy with Different Amount of G-files on a Workload with Three Applications: Sort, TextGen and WordCount. Sort has improvement up to 48% on job execution time; TextGen 56%.

outperforms that when the strategy is turned off. When the amount of G-files is larger, such improvement is more obvious. The job execution time is reduced by up to 48% for Sort and 56% for TextGen. There is less improvement in the WordCount application since the job has small amount of shuffle and output data and hence less requirement for the off-switch bandwidth. Table 9 shows detailed statistics: the speed up for Sort application is resulted from a 4.5x speed up in the shuffle stage and accordingly a 48% speed up in reduce time; the speed up for TextGen is resulted from a 2x speed up in the map stage.

We further studied the impact of the size of G-files and G-racks. We set the number of users to 8 and again divide the users by two groups: 4 of the users ran

**Table 9:** Detailed Time(sec) for the Impact of Grouping-blocks Strategy.

| App | Amount of G-files | Task | on | off |
|---|---|---|---|---|
| Sort | 20% | shuffle | 0.41 | 1.45 |
| | | reduce | 16.73 | 29.90 |
| | 50% | shuffle | 0.36 | 1.59 |
| | | reduce | 23.03 | 34.19 |
| TextGen | 20% | map | 15.40 | 19.97 |
| | 50% | map | 15.87 | 32.14 |

jobs of 6 GB data; the rest of the users ran jobs of size ranging from 2 GB to 6 GB. We limited the number of G-racks $R_i$ of the jobs from the second group to 1 or 2 by creating a pool for each job and limiting the map and reduce task capacity ($m_i$ and $r_i$) for each pool. The candidate selection algorithm automatically selected the files from the second group due to a smaller $R_i$. Figure 46 shows the result of our experiment. The speed up increases with the G-file size. This is because larger G-files have a increased requirement on the off-switch bandwidth and therefore have a larger impact when the amount of off-switch data is reduced. The speed up also decreases with the G-rack size. This can be predicted from Equation 44 that the improvement decreases drastically with large $R_i$. One exception is for the 2 GB case which is because the file is small enough to fit in one rack and increasing the pool capacity has no effect on $R_i$ in such case.

#### *4.3.4.3 Effectiveness of Candidate Selection*

We verified the effectiveness of our candidate selection mechanism in this experiment. We constructed a workload with the parameter tuple $(32, 1, 5, 0)$, that is, the workload had a batch with 32 jobs arriving in five-second intervals. Each job ran a synthetic application with a randomly selected configuration of $\mu_i$ ranging from 0.1 to 1.0, $s_i$ ranging from 2 GB to 6 GB and $R_i$ ranging from 1 to 2. Figure 47 compares the experiment results of no grouping-blocks strategy (labeled "off"), random selection (labeled "rand") and our candidate selection algorithm (labeled "muRs") with the

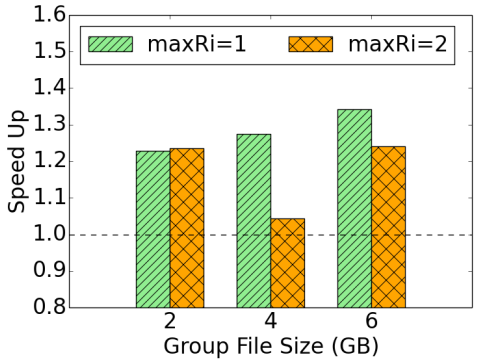



**Figure 46:** Impact of Grouping-blocks Strategy with Different G-file and G-racks size on a Workload of Sort. Speedup increases with G-file size and decreases with G-racks size.

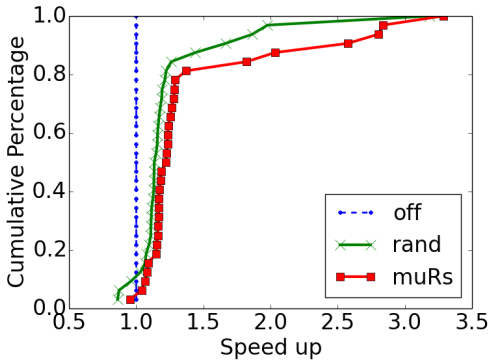
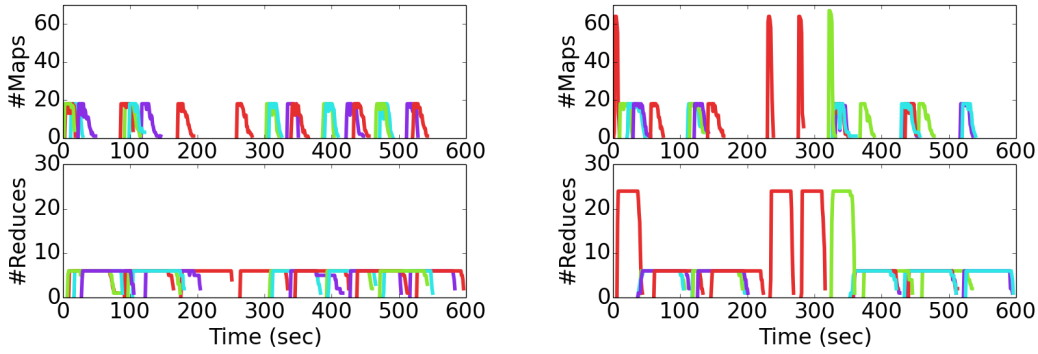



**Figure 47:** Effectiveness of Candidate Selection. We compare three cases: no grouping-blocks strategy ("off"), random selection ("rand") and our mechanism ("muRs") in Figure 40. Our mechanism has an average of 19% speed up over random selection.

same workload. It is shown that our mechanism achieves better performance for each job over random selection. This is because our mechanism can choose the files and jobs that benefit the most from the strategy while random selection sometimes chooses jobs that are less affected. The random selection obtains an average of 25% speed up over no grouping-blocks strategy; our mechanism obtains an average of 44% speed up which is around 19% speed up over random selection.




(a) Without Checking (b) With Checking

**Figure 48:** Avoiding the "Sticky" Effect in Task Scheduling. Figure shows the number of running maps and reduces for each job. With the checking mechanism, when the capacity of the pool was changed (time around 200-300), the grouping-blocks strategy was turned off and the execution was not limited to G-racks.

#### *4.3.4.4 Avoiding the "Sticky" Effect in Task Scheduling*

The "sticky" effect is prevented by setting a suitable $R_i$ in data placement. However, once the data placement is done, it is the responsibility of the task scheduling to mitigate the "sticky" effect when the workload changed. Figure 48 shows the capability of our task scheduling mechanism under such case. We constructed a workload with four pools in the cluster. During time 0 to 200, all four pools had jobs running and limited the capacity to 25% of the cluster; during time 200 to 300, only one pool had job running and the capacity was changed to 100% of the cluster. After time 300, the other three pools ran jobs again. It is shown that with our checking mechanism, the scheduler can dynamically turn off the strategy to prevent the "sticky" effect during time 200 to 300.

#### *4.3.4.5 Avoiding the "Conflict" Effect*

Next we validated our mechanisms in data placement and task scheduling to avoid the "conflict" effect. We constructed a workload with the parameter tuple $(4, 5, 360, 10)$. The workload had 5 batches with 360-second intervals; each batch had 4 users each submitting a work flow. Each users had two work flows to choose with a probability


**Figure 49:** Avoiding the "Conflict" Effect in Data Placement and Task Scheduling. We compare between two data placement approaches: random distribution ("rand dist") and probability distribution ("prob dist") in Figure 42; and two task scheduling approaches: with/without checking conflict.

of 0.8 and 0.2 respectively; each work flow was consist of 4 jobs running consecutively. All the files are G-files. Such construction of the work flow created G-files with varied frequency of 0.8 and 0.2; moreover, G-files within one work flow belongs to one non-concurrent set. We feed such hints to our location decision mechanism.

Figure 49 shows the result of our experiment. When the checking in task scheduling is disabled (lines without the "check" label), the CDF of job execution time has a distinctive step around 40 sec and 80 sec. This indicates the conflicts occur and some jobs have to wait until the conflicting jobs to finish. Our data placement mechanism avoids around 20% of conflicts: the step begins from 60% in the "prob dist" line as opposed to 40% in the "rand dist" line. When the checking in task scheduling is enabled, there is no distinctive step, i.e., conflicts are avoided. The average job execution time of the four configurations (i.e., probability dist with checking, probability dist, random dist with checking and random dist) is 61.71, 63.28, 67.94 and 79.43 respectively, that is, our mechanisms obtain a 28% improvement over random distribution with no checking.

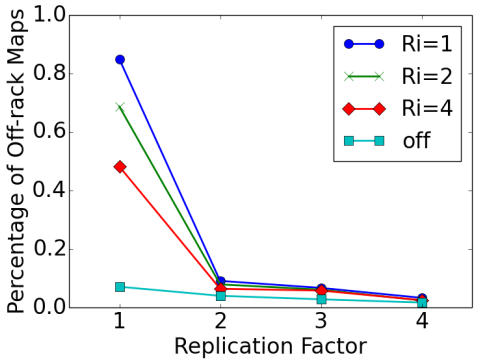



**Figure 50:** Impact of Grouped Blocks on Map Locality. The figure shows the impact of the number of replication and size of G-racks ($R_i$) on percentage of off-switch map tasks.

#### *4.3.4.6 Impact of Grouped Blocks on Map Locality*

We used our simulator to study how grouped blocks affect map locality for non-G-jobs and hence provided insights on whether grouping blocks of one replica can hurt the map locality. We varied the replication factor from 1 to 4 and the number of G-racks $R_i$ from 1 to 4 as well as no grouped blocks. Figure 50 shows the result of the simulation: the default data placement always have a better map locality but the difference diminishes quickly with larger replication factor. With three replica, all settings can achieve a percentage of off-switch map task less than 10% (with 6.7% for $R_i = 1$ and 2.8% for no grouped blocks). Therefore, with the grouping-blocks strategy, the replication factor should be at least 2; and with 4 replica, under a uniform access pattern simulated in our environment, the impact of grouped blocks is negligible.

#### *4.3.4.7 Impact on Number of Occupied G-racks*

The grouping-blocks strategy is more effective when more racks are being used as G-racks. For example, the best case is all racks run G-jobs all the time, which saves the most of off-switch data access. Therefore, in this experiment, we studied the average percentage of G-racks occupied during execution with respect to the amount

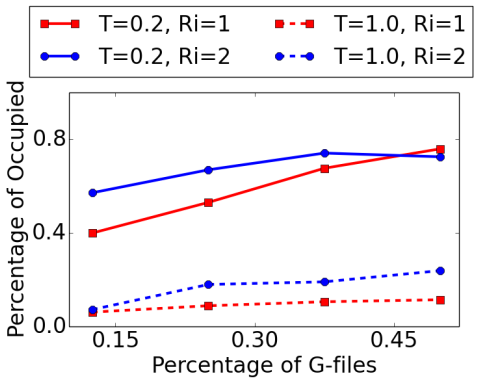



**Figure 51:** Impact on Percentage of Occupied G-racks. The figure shows the impact of the amount of G-files, the size of G-racks ($R_i$) and the job arrival interval ($T$).

of G-files, the size of G-racks and job arrival rate. Figure 51 shows our simulation result. From the figure, we can observe that increasing the amount of G-files does not necessarily adding more opportunity for the strategy. When arrival rate is low ($T = 1.0$), the number of occupied G-racks is limited by the number of concurrent jobs; when the arrival rate is high, conflicts limit the number of G-racks occupied. Such observation suggests when cluster is under load, the grouping-blocks strategy is less useful and the budget should be set to a low value; when the cluster load is high, there is also a threshold when increasing the budget does not help anymore. We can also observe that increasing the number of G-files of a smaller $R_i$ is more effective when arrival rate is high. Hence the candidate selection should give more weight to these files.

## *4.4 Summary*

In this chapter, we focused on the network resource contention problem of the MapReduce system. Our improvement over the current state-of-art design consists of two aspect: (1) an extension to better support map locality for dual-input applications [123] and (2) a grouping-blocks strategy for map and reduce co-locality [124].

We first studied the inefficiency of Hadoop when executing dual-input applications. We presented Dual-Hadoop, our extension to Hadoop to better support such

applications. Dual-Hadoop integrated an easy-to-use user interface, a dual-input aware scheduler, and a user-transparent caching mechanism. Our extension is able to exploit the unique data locality characteristics of dual-input applications. This was verified through extensive experiments, which shows 48% reduction in remote data reads and up to 3.3x improvement in application execution time than the default Hadoop. Dual-Hadoop is expected to be extensible to support multiple input applications. Works following our contribution can further investigate on this opportunity and develop scheduling algorithms that improve Hadoop on even more general task sharing patterns.

Next, we studied a grouping-blocks strategy for Hadoop/MapReduce system. Such strategy can greatly improve both map and reduce locality; on the other hand, it suffers from loss of parallelism problem, i.e., the "sticky" effect and the "conflict" effect. Therefore, we proposed several mechanisms from both data placement and task scheduling aspect to mitigate the loss of parallelism problem. Extensive experiments were conducted to validate the effectiveness of grouping-blocks strategy and our mechanisms. The grouping-blocks strategy is shown to improve job execution time by up to 56%; furthermore, our proposed mechanisms is successful on mitigating the loss of parallelism issue in various situations. Future work can investigate more on the relationship between job execution time and number of G-racks to provide more accurate conclusion on choice of number of G-racks. We also plan to validate our mechanisms on a real production workload in the future.

# CHAPTER V

# CONCLUSION

Parallel and distributed computing systems become more and more important as the volume of data that needs processing grows extremely large. Data access is an essential part in any program and hence optimizing data access parallelism is important to the performance of such systems. To ease the use of parallel and distributed systems, data access operations are often abstracted in the form of programming models. In this dissertation, we study two important and widely used abstractions: Transaction-based Abstraction and Distributed-system-based Abstraction. To improve the parallelism for the transaction-based abstraction, the implementation needs to focus on the data contention which occurs when multiple transactions try to access conflicted data simultaneously and therefore must be serialized. For the distributed-system-based abstraction, the major concern is the network resource contention which limits the performance when a large amount of data needs to be transferred through a top-level switch.

In Chapters 2 and 3, we illustrated the impact of data contention on performance with respect to various system and workload parameters. In Chapter 2, we presented an analytical model based on continuous-time Markov chain and modeled each transaction as client requesting services from the computing system. In Chapter 3, we adopted the mean value analysis approach to study the impact of data contention with various system schemes. Both analytical methods can be used to estimate the system performance which were validated through extensive experiments. On the other hand, the two methods (queueing model and mean value analysis) have different strength in express different systems. Queueing methods are more intuitive to

describe open systems with the inherited assumption of Poisson arrival; it can easily describe the variance of system capability with respect of number of transactions in the system. Therefore, it is more suitable to adopt such methods to describe the competition on shared computer resources in transactional memory system. Mean value analysis is more intuitive in describing closed systems and can be very powerful to describe the system in a modularized fashion. Therefore, it is easier to describe various system designs and compose the modules to model the whole system.

From our models in both chapters, we can observe that the data contention problem is sensible to workload parameters. Many parameters, such as data set size and transaction size, have a super linear relationship with the system performance. In Chapter 3, it is shown that a batched deterministic system scheme can be more beneficial under high contention situations especially when the time to commit a transaction is large (e.g., large network delay in Geo-replicated systems).

Many other detailed factors such as implementation overhead and contention resolution decisions can affect the data contention problem and the performance for the transaction-based abstraction. In Chapter 2, we adopted an adaptive method for contention management to select the best policy for the workloads and platforms. We showed that adaptive contention management is necessary and feasible. Our scheme is a profiling-based method that would choose a suitable CM for a given workload and system platform during run-time.

Because of the sensitive nature of transaction-based abstraction to data contention which makes it difficult to be deployed in distributed environment, many distributed systems (e.g., MapReduce) are designed to avoid the potential serialization of transaction-based systems, by splitting data access among independent tasks and reducing communication among the tasks. While serialization due to data contention is reduced in these systems, network resource contention becomes a limiting factor for such systems. In Chapter 4, we studied network resource contention in

MapReduce systems and identified two opportunities to improve the state of art: (1) the support of dual-input map tasks, and (2) the issues with map/reduce co-locality. For dual-input applications, we extended the MapReduce implementation with a caching system and developed cache-aware scheduling strategies. To improve locality for both map and reduce tasks, we proposed a strategy that groups the map input data in a few racks and enhanced the current implementation to mitigate the side effect of parallelism degradation. A common feature in both of the enhancement is that we collected hints and statistics from user and history execution and fully utilized such information. We developed special data placement and task scheduling mechanisms according to such information to minimize unnecessary data communication. While existing methods can handle general situations well, our study shows that, for a focused (and representative) group of applications (e.g. dual-input applications or many small map-reduce-input-heavy jobs), it is possible to greatly reduce the network resource contention and thereby reducing job execution time.

# REFERENCES


[1] Ananian, C. S., Asanovic, K., Kuszmaul, B. C., Leiserson, C. E., and Lie, S., "Unbounded transactional memory," *IEEE Micro*, vol. 26, no. 1, pp. 59–69, 2006.

[2] Ananthanarayanan, G., Agarwal, S., Kandula, S., Greenberg, A., Stoica, I., Harlan, D., and Harris, E., "Scarlett: coping with skewed content popularity in mapreduce clusters," in *Proceedings of the sixth conference on Computer systems*, EuroSys '11, (New York, NY, USA), pp. 287–300, ACM, 2011.

[3] Ananthanarayanan, G., Ghodsi, A., Wang, A., Borthakur, D., Kandula, S., Shenker, S., and Stoica, I., "Pacman: coordinated memory caching for parallel jobs," in *Proceedings of the 9th USENIX conference on Networked Systems Design and Implementation*, NSDI'12, (Berkeley, CA, USA), pp. 20–20, USENIX Association, 2012.

[4] Ananthanarayanan, G., Kandula, S., Greenberg, A., Stoica, I., Lu, Y., Saha, B., and Harris, E., "Reining in the outliers in map-reduce clusters using mantri," in *Proceedings of the 9th USENIX conference on Operating systems design and implementation*, OSDI'10, (Berkeley, CA, USA), pp. 1–16, USENIX Association, 2010.

[5] Anderson, T. A. and others, "Replication, consistency, and practicality: Are these mutually exclusive?," in *SIGMOD Conference* (Haas, L. M. and Tiwary, A., eds.), pp. 484–495, ACM Press, 1998.

[6] Avni, H. and Shavit, N., "Maintaining consistent transactional states without a global clock," in *SIROCCO '08: Proceedings of the 15th international colloquium on Structural Information and Communication Complexity*, (Berlin, Heidelberg), pp. 131–140, Springer-Verlag, 2008.

[7] Backstrom, L., Huttenlocher, D., Kleinberg, J., and Lan, X., "Group formation in large social networks: membership, growth, and evolution," in *Proceedings of the 12th ACM SIGKDD international conference on Knowledge discovery and data mining*, KDD '06, (New York, NY, USA), pp. 44–54, ACM, 2006.

[8] Baker, J., Bond, C., Corbett, J. C., Furman, J., Khorlin, A., Larson, J., Leon, J.-M., Li, Y., Lloyd, A., and Yushprakh, V., "Megastore: Providing scalable, highly available storage for interactive services," in *Proceedings of the Conference on Innovative Data system Research (CIDR)*, pp. 223–234, 2011.

[9] Baugh, L., Neelakantam, N., and Zilles, C., "Using hardware memory protection to build a high-performance, strongly atomic hybrid transactional memory," in *Proceedings of the 35th Annual International Symposium on Computer Architecture*, ACM Press, June 2008.

[10] Baugh, L., Neelakantam, N., and Zilles, C., "Using hardware memory protection to build a high-performance, strongly-atomic hybrid transactional memory," in *Proceedings of the 35th Annual International Symposium on Computer Architecture*, ISCA '08, (Washington, DC, USA), pp. 115–126, IEEE Computer Society, 2008.

[11] Blake, G., Dreslinski, R. G., and Mudge, T. N., "Proactive transaction scheduling for contention management.," in *MICRO* (Albonesi, D. H., Martonosi, M., August, D. I., and Martnez, J. F., eds.), pp. 156–167, ACM, 2009.

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

[13] Bu, Y., Howe, B., Balazinska, M., and Ernst, M. D., "Haloop: efficient iterative data processing on large clusters," *Proc. VLDB Endow.*, vol. 3, pp. 285–296, Sept. 2010.

[14] Burger, A. and others, "Performance of multiversion and distributed two-phase locking concurrency control mechanisms in distributed databases.," *Inf. Sci.*, vol. 96, no. 1-2, pp. 129–152, 1997.

[15] Cao Minh, C., Chung, J., Kozyrakis, C., and Olukotun, K., "STAMP: Stanford transactional applications for multi-processing," in *IISWC '08: Proceedings of The IEEE International Symposium on Workload Characterization*, September 2008.

[16] Cao Minh, C., Trautmann, M., Chung, J., McDonald, A., Bronson, N., Casper, J., Kozyrakis, C., and Olukotun, K., "An effective hybrid transactional memory system with strong isolation guarantees," in *Proceedings of the 34th Annual International Symposium on Computer Architecture*, ACM Press, Jun 2007.

[17] "Hadoop capacity scheduler." `http://hadoop.apache.org/docs/stable/capacity_scheduler.html`.

[18] Chowdhury, M., Zaharia, M., Ma, J., Jordan, M. I., and Stoica, I., "Managing data transfers in computer clusters with orchestra," in *Proceedings of the ACM SIGCOMM 2011 conference*, SIGCOMM '11, (New York, NY, USA), pp. 98–109, ACM, 2011.

[19] Corbett, J. C., Dean, J., Epstein, M., Fikes, A., Frost, C., Furman, J. J., Ghemawat, S., Gubarev, A., Heiser, C., Hochschild, P., Hsieh, W., Kanthak, S., Kogan, E., Li, H., Lloyd, A., Melnik, S., Mwaura, D., Nagle, D., Quinlan, S., Rao, R., Rolig, L., Saito, Y., Szymaniak, M., Taylor, C., Wang, R., and Woodford, D., "Spanner: Google's globally-distributed database," in *Proceedings of the 10th USENIX Conference on Operating Systems Design and Implementation*, OSDI'12, (Berkeley, CA, USA), pp. 251–264, USENIX Association, 2012.

[20] Damron, P., Fedorova, A., Lev, Y., Luchangco, V., Moir, M., and Nussbaum, D., "Hybrid transactional memory," in *ASPLOS-XII: Proceedings of the 12th international conference on Architectural support for programming languages and operating systems*, (New York, NY, USA), pp. 336–346, ACM, 2006.

[21] Dean, J. and Ghemawat, S., "Mapreduce: simplified data processing on large clusters," in *Proceedings of the 6th conference on Symposium on Opearting Systems Design & Implementation - Volume 6*, OSDI'04, (Berkeley, CA, USA), pp. 10–10, USENIX Association, 2004.

[22] DeCandia, G. and others, "Dynamo: Amazon's highly available key-value store," in *Proceedings of Twenty-first ACM SIGOPS Symposium on Operating Systems Principles*, SOSP '07', (New York, NY, USA), pp. 205–220, ACM, 2007.

[23] Dice, D., Shalev, O., and Shavit, N., "Transactional locking ii," in *In Proc. of the 20th Intl. Symp. on Distributed Computing*, 2006.

[24] Dittrich, J., Quiané-Ruiz, J.-A., Jindal, A., Kargin, Y., Setty, V., and Schad, J., "Hadoop++: making a yellow elephant run like a cheetah (without it even noticing)," *Proc. VLDB Endow.*, vol. 3, pp. 515–529, Sept. 2010.

[25] Dragojević, A., Guerraoui, R., and Kapalka, M., "Stretching transactional memory," in *Proceedings of the 2009 ACM SIGPLAN conference on Programming language design and implementation* (Hind, M. and Diwan, A., eds.), PLDI '09, (New York, NY, USA), pp. 155–165, ACM, 2009.

[26] Dragojević, A., Guerraoui, R., and Kapalka, M., "Stretching transactional memory," in *PLDI '09: Proceedings of the 2009 ACM SIGPLAN conference on Programming language design and implementation*, (New York, NY, USA), pp. 155–165, ACM, 2009.

[27] Ekanayake, J., Li, H., Zhang, B., Gunarathne, T., Bae, S.-H., Qiu, J., and Fox, G., "Twister: a runtime for iterative mapreduce," in *Proceedings of the 19th ACM International Symposium on High Performance Distributed Computing*, HPDC '10, (New York, NY, USA), pp. 810–818, ACM, 2010.

[28] Eltabakh, M. Y., Tian, Y., Özcan, F., Gemulla, R., Krettek, A., and McPherson, J., "Cohadoop: flexible data placement and its exploitation in hadoop," *Proc. VLDB Endow.*, vol. 4, pp. 575–585, June 2011.

[29] Ennals, R., "Efficient software transactional memory," Tech. Rep. IRC-TR-05-051, Intel Research Cambridge Tech Report, Jan 2005.

[30] "Hadoop fair scheduler." `http://hadoop.apache.org/docs/r1.1.2/fair_scheduler.html`.

[31] Felber, P., Fetzer, C., Marlier, P., and Riegel, T., "Time-based software transactional memory," *IEEE Transactions on Parallel and Distributed Systems*, vol. 21, pp. 1793–1807, 2010.

[32] Felber, P., Fetzer, C., and Riegel, T., "Dynamic performance tuning of word-based software transactional memory," in *PPoPP '08: Proceedings of the 13th ACM SIGPLAN Symposium on Principles and practice of parallel programming*, (New York, NY, USA), pp. 237–246, ACM, 2008.

[33] Forum, T. M., "Mpi: A message passing interface," 1993.

[34] Frank, J. and Chun, R., "Adaptive software transactional memory: A dynamic approach to contention management," in *PDPTA* (Arabnia, H. R. and Mun, Y., eds.), pp. 40–46, CSREA Press, 2008.

[35] George, L., *HBase: The Definitive Guide*. O'Reilly Media, 1 ed., 2011.

[36] Ghodsi, A., Zaharia, M., Hindman, B., Konwinski, A., Shenker, S., and Stoica, I., "Dominant resource fairness: fair allocation of multiple resource types," in *Proceedings of the 8th USENIX conference on Networked systems design and implementation*, NSDI'11, (Berkeley, CA, USA), pp. 24–24, USENIX Association, 2011.

[37] "Apache giraph." `http://incubator.apache.org/giraph/`.

[38] Gray, J. and others, "A straw man analysis of the probability of waiting and deadlock in a database system.," in *Berkeley Workshop*, p. 125, 1981.

[39] Gray, J. and others, "The dangers of replication and a solution," in *Proceedings of the 1996 ACM SIGMOD International Conference on Management of Data*, SIGMOD '96', (New York, NY, USA), pp. 173–182, ACM, 1996.

[40] Gross, D. and Harris, C. M., *Fundamentals of Queueing Theory (Wiley Series in Probability and Statistics)*. Wiley-Interscience, February 1998.

[41] Grossman, R. and Gu, Y., "Data mining using high performance data clouds: experimental studies using sector and sphere," in *Proceedings of the 14th ACM SIGKDD international conference on Knowledge discovery and data mining*, KDD '08, (New York, NY, USA), pp. 920–927, ACM, 2008.

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

[43] Guo, Z., Fox, G., and Zhou, M., "Investigation of data locality and fairness in mapreduce," in *Proceedings of third international workshop on MapReduce and its Applications Date*, MapReduce '12, (New York, NY, USA), pp. 25–32, ACM, 2012.

[44] Hammond, L., Wong, V., Chen, M., Carlstrom, B. D., Davis, J. D., Hertzberg, B., Prabhu, M. K., Wijaya, H., Kozyrakis, C., and Olukotun, K., "Transactional memory coherence and consistency," in *Proceedings of the 31st Annual International Symposium on Computer Architecture*, p. 102, IEEE Computer Society, Jun 2004.

[45] Hammoud, M., Rehman, M. S., and Sakr, M. F., "Center-of-gravity reduce task scheduling to lower mapreduce network traffic," in *Proceedings of the 2012 IEEE Fifth International Conference on Cloud Computing*, CLOUD '12, (Washington, DC, USA), pp. 49–58, IEEE Computer Society, 2012.

[46] Hammoud, M. and Sakr, M. F., "Locality-aware reduce task scheduling for mapreduce," in *Proceedings of the 2011 IEEE Third International Conference on Cloud Computing Technology and Science*, CLOUDCOM '11, (Washington, DC, USA), pp. 570–576, IEEE Computer Society, 2011.

[47] Harris, T., Cristal, A., Unsal, O. S., Ayguade, E., Gagliardi, F., Smith, B., and Valero, M., "Transactional memory: An overview," *IEEE Micro*, vol. 27, no. 3, pp. 8–29, 2007.

[48] Harris, T. and Fraser, K., "Language support for lightweight transactions," in *Object-Oriented Programming, Systems, Languages, and Applications*, pp. 388–402, ACM Press, Oct 2003.

[49] Harris, T., Marlow, S., Peyton-Jones, S., and Herlihy, M., "Composable memory transactions," in *PPoPP '05: Proceedings of the tenth ACM SIGPLAN symposium on Principles and practice of parallel programming*, (New York, NY, USA), pp. 48–60, ACM, 2005.

[50] "Hdfs architecture guide." `http://hadoop.apache.org/docs/r1.2.1/hdfs_design.html`.

[51] He, Z. and Hong, B., "On the performance of commit-time-locking based software transactional memory," *The 11th IEEE International Conference on High Performance Computing and Communications (HPCC-09)*, 2009.

[52] He, Z. and Hong, B., "Modeling the run-time behavior of transactional memory," in *MASCOTS*, pp. 307–315, 2010.

[53] He, Z., Yu, X., and Hong, B., "Profiling-based adaptive contention management for software transactional memory," in *Parallel Distributed Processing Symposium (IPDPS), 2012 IEEE 26th International*, pp. 1204–1215, May 2012.

[54] Heber, T., Hendler, D., and Suissa, A., "On the impact of serializing contention management on stm performance," in *OPODIS '09: Proceedings of the 13th International Conference on Principles of Distributed Systems*, (Berlin, Heidelberg), pp. 225–239, Springer-Verlag, 2009.

[55] Heindl, A. and Pokam, G., "An analytic framework for performance modeling of software transactional memory," *Comput. Netw.*, vol. 53, no. 8, pp. 1202–1214, 2009.

[56] Herlihy, M., Luchangco, V., Moir, M., and Scherer, III, W. N., "Software transactional memory for dynamic-sized data structures," in *PODC '03: Proceedings of the twenty-second annual symposium on Principles of distributed computing*, (New York, NY, USA), pp. 92–101, ACM, 2003.

[57] Herlihy, M., Moir, M., and Luchangco, V., "A flexible framework for implementing software transactional memory," in *Proceedings of the 21th ACM SIGPLAN Conference on Object-Oriented Programing, Systems, Languages, and Applications*, pp. 253–262, oct 2006.

[58] Herlihy, M. and Moss, J. E. B., "Transactional memory: architectural support for lock-free data structures," in *ISCA '93: Proceedings of the 20th annual international symposium on Computer architecture*, pp. 289–300, ACM Press, 1993.

[59] Hindman, B., Konwinski, A., Zaharia, M., Ghodsi, A., Joseph, A. D., Katz, R., Shenker, S., and Stoica, I., "Mesos: a platform for fine-grained resource sharing in the data center," in *Proceedings of the 8th USENIX conference on Networked systems design and implementation*, NSDI'11, (Berkeley, CA, USA), pp. 22–22, USENIX Association, 2011.

[60] Isard, M., Prabhakaran, V., Currey, J., Wieder, U., Talwar, K., and Goldberg, A., "Quincy: fair scheduling for distributed computing clusters," in *Proceedings of the ACM SIGOPS 22nd symposium on Operating systems principles*, SOSP '09, (New York, NY, USA), pp. 261–276, ACM, 2009.

[61] Jin, J., Luo, J., Song, A., Dong, F., and Xiong, R., "Bar: An efficient data locality driven task scheduling algorithm for cloud computing," in *Proceedings of the 2011 11th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing*, CCGRID '11, (Washington, DC, USA), pp. 295–304, IEEE Computer Society, 2011.

[62] Jing, F. R., Takahashi, Y., and Hasegawa, T., "Analysis of impact of network delay on multiversion conservative timestamp algorithms in ddbs.," *Perform. Eval.*, vol. 26, no. 1, pp. 21–50, 1996.

[63] Junqueira, F. and others, "Classic paxos vs. fast paxos: Caveat emptor," in *Proceedings of the 3rd Workshop on on Hot Topics in System Dependability*, HotDep'07, (Berkeley, CA, USA), USENIX Association, 2007.

[64] Kleinrock, L., *Queueing Systems*, vol. I: Theory. Wiley Interscience, 1975. (Published in Russian, 1979. Published in Japanese, 1979. Published in Hungarian, 1979. Published in Italian 1992.).

[65] Knight, T., "An architecture for mostly functional languages," in *LFP '86: Proceedings of the 1986 ACM conference on LISP and functional programming*, pp. 105–112, ACM Press, 1986.

[66] Kraska, T. and others, "Mdcc: Multi-data center consistency," in *Proceedings of the 8th ACM European Conference on Computer Systems*, EuroSys '13', (New York, NY, USA), pp. 113–126, ACM, 2013.

[67] Kumar, K. A., Deshpande, A., and Khuller, S., "Data placement and replica selection for improving co-location in distributed environments," *CoRR*, vol. abs/1302.4168, 2013.

[68] Kumar, S., Chu, M., J. Hughes, C., Kundu, P., and Nguyen, A., "Hybrid transactional memory," in *Proceedings of Symposium on Principles and Practice of Parallel Programming*, Mar 2006.

[69] Lamport, L., "Paxos Made Simple," *SIGACT News*, vol. 32, pp. 51–58, Dec. 2001.

[70] Lamport, L., "Fast paxos," 2005.

[71] Lin, J. and Dyer, C., *Data-Intensive Text Processing with MapReduce*. Synthesis Lectures on Human Language Technologies, Morgan & Claypool Publishers, 2010.

[72] Lin, J. and Schatz, M., "Design patterns for efficient graph algorithms in mapreduce," in *Proceedings of the Eighth Workshop on Mining and Learning with Graphs*, MLG '10, (New York, NY, USA), pp. 78–85, ACM, 2010.

[73] "Apache mahout." `http://mahout.apache.org/`.

[74] Marathe, V. J., Scherer III, W. N., and Scott, M. L., "Design tradeoffs in modern software transactional memory systems," in *Proceedings of the 7th Workshop on Languages, Compilers, and Run-time Systems for Scalable Computers*, (Houston, TX), Oct 2004.

[75] Marathe, V. J., Spear, M. F., Heriot, C., Acharya, A., Eisenstat, D., Scherer III, W. N., and Scott, M. L., "Lowering the overhead of software transactional memory," Tech. Rep. TR 893, Computer Science Department, University of Rochester, Mar 2006.

[76] "Memcachd." `http://memcached.org/`.

[77] Menasce, D. A. and Nakanishi, T., "Performance evaluation of a two-phase commit based protocol for ddbs.," in *PODS* (Ullman, J. D. and Aho, A. V., eds.), pp. 247–255, ACM, 1982.

[78] Meng, X., Yu, X., Peng, Z., and Hong, B., "Detecting earthquakes around salton sea following the 2010 mw7.2 el mayor-cucapah earthquake using gpu parallel computing," *Procedia CS*, vol. 9, pp. 937–946, 2012.

[79] M.Herlihy, "Apologizing versus asking permission: optimistic concurrency control for abstract data types," *ACM Transactions on Database Systems*, vol. 15, no. 1, pp. 96–124, 1990.

[80] Moir, M., "Practical implementations of non-blocking synchronization primitives," in *Symposium on Principles of Distributed Computing*, pp. 219–228, 1997.

[81] Moore, K. E., Bobba, J., Moravan, M. J., Hill, M. D., and Wood, D. A., "Logtm: Log-based transactional memory," in *Proceedings of the 12th International Symposium on High-Performance Computer Architecture*, pp. 254–265, ACM Press, Feb 2006.

[82] Moretti, C., Bui, H., Hollingsworth, K., Rich, B., Flynn, P., and Thain, D., "All-pairs: An abstraction for data-intensive computing on campus grids," *IEEE Trans. Parallel Distrib. Syst.*, vol. 21, pp. 33–46, Jan. 2010.

[83] Ng, A. Y., Bradski, G., Chu, C.-T., Olukotun, K., Kim, S. K., Lin, Y.-A., and Yu, Y., "Map-reduce for machine learning on multicore," in *NIPS*, 12/2006 2006. ¡p¿Selected for Oral Presentation¡/p¿.

[84] Nicola, M. and Jarke, M., "Performance modeling of distributed and replicated databases," *IEEE Trans. on Knowl. and Data Eng.*, vol. 12, pp. 645–672, July 2000.

[85] Olston, C., Reed, B., Srivastava, U., Kumar, R., and Tomkins, A., "Pig latin: a not-so-foreign language for data processing," in *Proceedings of the 2008 ACM SIGMOD international conference on Management of data*, SIGMOD '08, (New York, NY, USA), pp. 1099–1110, ACM, 2008.

[86] Palanisamy, B., Singh, A., Liu, L., and Jain, B., "Purlieus: locality-aware resource allocation for mapreduce in a cloud," in *Proceedings of 2011 International Conference for High Performance Computing, Networking, Storage and Analysis*, SC '11, (New York, NY, USA), pp. 58:1–58:11, ACM, 2011.

[87] Patterson, S. and others, "Serializability, not serial: Concurrency control and availability in multi-datacenter datastores," *CoRR*, vol. abs/1208.0270, 2012.

[88] Porter, D. E., Hofmann, O. S., and Witchel, E., "Is the optimism in optimistic concurrency warranted?," in *Proceedings of the 11th USENIX workshop on Hot topics in operating systems* (Hunt, G. C., ed.), HOTOS'07, (Berkeley, CA, USA), pp. 1:1–1:6, USENIX Association, 2007.

[89] Porter, D. E. and Witchel, E., "Modeling transactional memory workload performance," in *PPoPP '10: Proceedings of the 15th ACM SIGPLAN symposium on Principles and practice of parallel programming*, (New York, NY, USA), pp. 349–350, ACM, 2010.

[90] Power, R. and Li, J., "Piccolo: building fast, distributed programs with partitioned tables," in *Proceedings of the 9th USENIX conference on Operating systems design and implementation*, OSDI'10, (Berkeley, CA, USA), pp. 1–14, USENIX Association, 2010.

[91] Rajwar, R. and Goodman, J. R., "Speculative lock elision: enabling highly concurrent multithreaded execution," in *Proceedings of the 34th annual ACM/IEEE international symposium on Microarchitecture*, MICRO 34, (Washington, DC, USA), pp. 294–305, IEEE Computer Society, 2001.

[92] Rajwar, R., Herlihy, M., and Lai, K., "Virtualizing transactional memory," in *Proceedings of the 32nd Annual International Symposium on Computer Architecture*, pp. 494–505, IEEE Computer Society, Jun 2005.

[93] Ramadan, H. E., Rossbach, C. J., Porter, D. E., Hofmann, O. S., Bhandari, A., and Witchel, E., "Metatm/txlinux: transactional memory for an operating system," in *ISCA '07: Proceedings of the 34th annual international symposium on Computer architecture*, (New York, NY, USA), pp. 92–103, ACM, 2007.

[94] Ren, K., Gibson, G., Kwon, Y., Balazinska, M., and Howe, B., "Abstract: Hadoop's adolescence; a comparative workloads analysis from three research clusters," in *Proceedings of the 2012 SC Companion: High Performance Computing, Networking Storage and Analysis*, SCC '12, (Washington, DC, USA), pp. 1452–, IEEE Computer Society, 2012.

[95] Saha, B., Adl-Tabatabai, A., and Jacobson, Q., "Architectural support for software transactional memory," in *39th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO-39 2006), 9-13 December 2006, Orlando, Florida, USA*, pp. 185–196, 2006.

[96] Scherer, III, W. N. and Scott, M. L., "Advanced contention management for dynamic software transactional memory," in *PODC '05: Proceedings of the twenty-fourth annual ACM symposium on Principles of distributed computing*, (New York, NY, USA), pp. 240–248, ACM, 2005.

[97] "Sector/sphere." `http://sector.sourceforge.net/index.html`.

[98] Shavit, N. and Touitou, D., "Software transactional memory," in *Proceedings of the 14th ACM Symposium on Principles of Distributed Computing*, pp. 204–213, ACM Press, Aug 1995.

[99] Shavit, N. and Touitou, D., "Software transactional memory," *Journal of Distributed Computing*, vol. 10, no. 2, pp. 99–116, 1997.

[100] Shriraman, A. and Dwarkadas, S., "Refereeing conflicts in hardware transactional memory," in *Proceedings of the 23rd international conference on Supercomputing* (Gschwind, M., Nicolau, A., Salapura, V., and Moreira, J. E., eds.), ICS '09, (New York, NY, USA), pp. 136–146, ACM, 2009.

[101] Shriraman, A., Spear, M. F., Hossain, H., Marathe, V., Dwarkadas, S., and Scott, M. L., "An integrated hardware-software approach to flexible transactional memory," in *Proceedings of the 34rd Annual International Symposium on Computer Architecture*, ACM Press, Jun 2007.

[102] Shriraman, A., Spear, M. F., Hossain, H., Marathe, V. J., Dwarkadas, S., and Scott, M. L., "An integrated hardware-software approach to flexible transactional memory," in *Proceedings of the 34th Annual International Symposium on Computer Architecture*, ISCA '07, (New York, NY, USA), pp. 104–115, ACM, 2007.

[103] "Simpy." `http://simpy.readthedocs.org/en/latest/`.

[104] Sovran, Y. and others, "Transactional storage for geo-replicated systems," in *Proceedings of the Twenty-Third ACM Symposium on Operating Systems Principles*, SOSP '11', (New York, NY, USA), pp. 385–400, ACM, 2011.

[105] Spear, M. F., Dalessandro, L., Marathe, V. J., and Scott, M. L., "A comprehensive strategy for contention management in software transactional memory," *SIGPLAN Not.*, vol. 44, no. 4, pp. 141–150, 2009.

[106] Spear, M. F., Shriraman, A., Hossain, H., Dwarkadas, S., and Scott, M. L., "Alert-on-update: a communication aid for shared memory multiprocessors," in *Proceedings of the 12th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, PPOPP 2007, San Jose, California, USA, March 14-17, 2007*, pp. 132–133, 2007.

[107] Swift, M., Volos, H., Goyal, N., Yen, L., Hill, M., and Wood, D., "OS support for virtualizing hardware transactional memory," in *TRANSACT '08: 3rd Workshop on Transactional Computing*, (Salt Lake City, Utah, USA), feb 2008.

[108] Tan, J., Meng, X., and Zhang, L., "Coupling task progress for mapreduce resource-aware scheduling," in *Proceedings of the IEEE INFOCOM 2013, Turin, Italy, April 14-19, 2013*, pp. 1618–1626, 2013.

[109] "Tarjan's strongly connected components algorithm." http://en.wikipedia.org/wiki/Tarjan's_strongly_connected_components_algorithm.

[110] Tay, Y. C., Goodman, N., and Suri, R., "Locking performance in centralized databases.," *ACM Trans. Database Syst.*, vol. 10, no. 4, pp. 415–462, 1985.

[111] Thomasian, A., "Concurrency control: Methods, performance, and analysis," *ACM Comput. Surv.*, vol. 30, pp. 70–119, Mar. 1998.

[112] Thomasian, A. and Ryu, I. K., "Performance analysis of two-phase locking.," *IEEE Trans. Software Eng.*, vol. 17, no. 5, pp. 386–402, 1991.

[113] Thomson, A., Diamond, T., Weng, S.-C., Ren, K., Shao, P., and Abadi, D. J., "Calvin: Fast distributed transactions for partitioned database systems," in *Proceedings of the 2012 ACM SIGMOD International Conference on Management of Data*, SIGMOD '12', (New York, NY, USA), pp. 1–12, ACM, 2012.

[114] Thusoo, A., Shao, Z., Anthony, S., Borthakur, D., Jain, N., Sen Sarma, J., Murthy, R., and Liu, H., "Data warehousing and analytics infrastructure at facebook," in *Proceedings of the 2010 ACM SIGMOD International Conference on Management of data*, SIGMOD '10, (New York, NY, USA), pp. 1013–1020, ACM, 2010.

[115] Tomić, S., Perfumo, C., Kulkarni, C., Armejach, A., Cristal, A., Unsal, O., Harris, T., and Valero, M., "Eazyhtm: eager-lazy hardware transactional memory," in *Micro-42: Proceedings of the 42nd Annual IEEE/ACM International Symposium on Microarchitecture*, (New York, NY, USA), pp. 145–155, ACM, 2009.

[116] Verma, A., Cherkasova, L., and Campbell, R. H., "Aria: automatic resource inference and allocation for mapreduce environments," in *Proceedings of the 8th ACM international conference on Autonomic computing*, ICAC '11, (New York, NY, USA), pp. 235–244, ACM, 2011.

[117] White, T., *Hadoop: The Definitive Guide*. O'Reilly Media, Inc., 1st ed., 2009.

[118] Wolf, J., Rajan, D., Hildrum, K., Khandekar, R., Kumar, V., Parekh, S., Wu, K.-L., and balmin, A., "Flex: a slot allocation scheduling optimizer for mapreduce workloads," in *Proceedings of the ACM/IFIP/USENIX 11th International Conference on Middleware*, Middleware '10, (Berlin, Heidelberg), pp. 1–20, Springer-Verlag, 2010.

[119] Yang, H.-c., Dasdan, A., Hsiao, R.-L., and Parker, D. S., "Map-reduce-merge: simplified relational data processing on large clusters," in *Proceedings of the 2007 ACM SIGMOD international conference on Management of data*, SIGMOD '07, (New York, NY, USA), pp. 1029–1040, ACM, 2007.

[120] "Hadoop yarn." http://hadoop.apache.org/docs/current/hadoop-yarn/hadoop-yarn-site/YARN.html.

[121] Yen, L., Bobba, J., Marty, M. M., Moore, K. E., Volos, H., Hill, M. D., Swift, M. M., and Wood, D. A., "Logtm-se: Decoupling hardware transactional memory from caches," in *Proceedings of the 13th International Symposium on High-Performance Computer Architecture(HPCA)*, ACM Press, Feb 2007.

[122] Yu, X., He, Z., and Hong, B., "A queueing model-based approach for the analysis of transactional memory systems," *Concurrency and Computation: Practice and Experience*, 2013.

[123] Yu, X. and Hong, B., "Bi-hadoop: Extending hadoop to improve support for binary-input applications," in *The 13th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing*, CCGrid '03, 2013.

[124] Yu, X. and Hong, B., "Grouping blocks for mapreduce co-locality," in *Parallel Distributed Processing Symposium (IPDPS), 2015 IEEE 29th International*, May 2015.

[125] Yu, X., Meng, S., Tan, J., Meng, X., Zhang, L., and Hong, B., "Analyzing designs of geographically replicated transactional datastores," submitted to InfoComm 2015 under review.

[126] Zaharia, M., Borthakur, D., Sen Sarma, J., Elmeleegy, K., Shenker, S., and Stoica, I., "Delay scheduling: a simple technique for achieving locality and fairness in cluster scheduling," in *Proceedings of the 5th European conference on Computer systems*, EuroSys '10, (New York, NY, USA), pp. 265–278, ACM, 2010.

[127] Zaharia, M., Chowdhury, M., Franklin, M. J., Shenker, S., and Stoica, I., "Spark: cluster computing with working sets," in *Proceedings of the 2nd USENIX conference on Hot topics in cloud computing*, HotCloud'10, (Berkeley, CA, USA), pp. 10–10, USENIX Association, 2010.

[128] Zaharia, M., Konwinski, A., Joseph, A. D., Katz, R., and Stoica, I., "Improving mapreduce performance in heterogeneous environments," in *Proceedings of the 8th USENIX conference on Operating systems design and implementation*, OSDI'08, (Berkeley, CA, USA), pp. 29–42, USENIX Association, 2008.

[129] Zilles, C. and Baugh, L., "Extending hardware transactional memory to support nonbusy waiting and nontransactional actions," in *Proceedings of the First ACM SIGPLAN Workshop on Languages, Compilers, and Hardware Support for Transactional Computing*, ACM Press, Jun 2006.

On Contention Management for Data Accesses in Parallel and Distributed Systems

Xiao Yu

160 Pages

Directed by Professor Sudhakar Yalamanchili

Data access is an essential part of any program, and is especially critical to the performance of parallel computing systems. The objective of this work is to investigate factors that affect data access parallelism in parallel computing systems, and design/evaluate methods to improve such parallelism - and thereby improving the performance of corresponding parallel systems. We focus on data access contention and network resource contention in representative parallel and distributed systems, including transactional memory system, Geo-replicated transactional systems and MapReduce systems. These systems represent two widely-adopted abstractions for parallel data accesses: transaction-based and distributed-system-based. In this thesis, we present methods to analyze and mitigate the two contention issues.

We first study the data contention problem in transactional memory systems. In particular, we present a queueing-based model to evaluate the impact of data contention with respect to various system configurations and workload parameters. We further propose a profiling-based adaptive contention management approach to choose an optimal policy across different benchmarks and system platforms. We further develop several analytical models to study the design of transactional systems when they are Geo-replicated.

For the network resource contention issue, we focus on data accesses in distributed systems and study opportunities to improve upon the current state-of-art MapReduce systems. We extend the system to better support map task locality for dual-map-input applications. We also study a strategy that groups input blocks within a few racks to balance the locality of map and reduce tasks. Experiments show that both

mechanisms significantly reduce off-rack data communication and thus alleviate the resource contention on top-rack switch and reduce job execution time.

In this thesis, we show that both the data contention and the network resource contention issues are key to the performance of transactional and distributed data access abstraction and our mechanisms to estimate and mitigate such problems are effective. We expect our approaches to provide useful insight on future development and research for similar data access abstractions and distributed systems.