Scheduling for High Throughput and Small Latency in Parallel and Distributed Systems

Zhe Wang
Washington University in St. Louis

Follow this and additional works at: https://openscholarship.wustl.edu/eng_etds

Part of the Computer Sciences Commons

Recommended Citation
https://openscholarship.wustl.edu/eng_etds/806

This Dissertation is brought to you for free and open access by the McKelvey School of Engineering at Washington University Open Scholarship. It has been accepted for inclusion in McKelvey School of Engineering Theses & Dissertations by an authorized administrator of Washington University Open Scholarship. For more information, please contact digital@wumail.wustl.edu.
Scheduling for High Throughput and Small Latency in Parallel and Distributed Systems
by
Zhe Wang

A dissertation presented to the McKelvey School of Engineering of Washington University in partial fulfillment of the requirements for the degree of Doctor of Philosophy

December 2022
St. Louis, Missouri
# Table of Contents

List of Figures .............................................................................................................................. vi
List of Tables ................................................................................................................................. ix
Acknowledgments .......................................................................................................................... x
Abstract ........................................................................................................................................ xiii

## Chapter 1: Introduction .................................................................................................................. 1
  1.1 Scheduling for high throughput and small latency ................................................................. 1
  1.2 Contributions ......................................................................................................................... 2
  1.3 Organization ........................................................................................................................... 5

## Chapter 2: An adaptive-scheduling system for multiprogrammed parallel computing .......... 6
  2.1 Introduction ............................................................................................................................. 6
  2.2 Background ............................................................................................................................ 10
  2.3 System design ........................................................................................................................ 12
    2.3.1 A client-server architecture ........................................................................................... 12
    2.3.2 Policy-customization interface ....................................................................................... 13
    2.3.3 Decentralized scheduling framework ............................................................................. 15
  2.4 A responsive, low-cost mechanism of processor reallocation ............................................. 18
    2.4.1 Responsive processor reallocation ................................................................................ 18
    2.4.2 Efficient work resumption ............................................................................................. 20
  2.5 Support for shared cloud environments ............................................................................... 22
    2.5.1 Subscription of runtime information ............................................................................. 23
    2.5.2 Resource occupancy control and admission control ................................................... 24
  2.6 Evaluation ............................................................................................................................... 25
    2.6.1 System overhead ............................................................................................................. 26
    2.6.2 Cache partitioning and memory bandwidth allocation .................................................. 29
  2.7 Case studies ............................................................................................................................ 30
2.7.1 Online scheduling to minimize average flow time ............................ 30
2.7.2 Elastic parallel real-time scheduling ............................................. 33
2.7.3 Adaptive scheduling using parallelism feedback .............................. 35
2.7.4 Co-scheduling throughput and tail-sensitive jobs ............................ 37
2.7.5 Subscription of runtime information ............................................ 40

2.8 Related works................................................................................... 41

2.8.1 Dynamic core reallocation between parallel jobs. ............................ 41
2.8.2 Scheduling multiprogrammed parallel workloads. ............................ 42

Chapter 3: Static schedules with sub-linear overhead for fault-tolerant transmission ............................................................................. 44

3.1 Introduction..................................................................................... 44
3.2 Background...................................................................................... 47

3.2.1 Identity mapping and collision-free mappings.................................. 47
3.2.2 Examples of collision-free mappings and their constructions............. 49

3.3 Generating fault-tolerant mapping with divisible collision-free mappings .... 52

3.3.1 Divisible collision-free mapping.................................................. 54
3.3.2 Divisible collision-free mapping $M(n, f, 1)$ guarantee fault-tolerance .... 55
3.3.3 Modulo mapping is divisible....................................................... 57

3.4 Generating fault-tolerant mapping with reducible collision-free mappings ...... 58

3.4.1 Reducible collision-free mapping ................................................. 59
3.4.2 Achieving fault-tolerance by repetitions........................................ 60
3.4.3 Applying the collision-free mapping for $\alpha = 1/2$ of small size .......... 69

3.5 Evaluation ....................................................................................... 72

3.5.1 Divisible collision-free mapping for $\alpha = 1$................................. 73
3.5.2 Reducible collision-free mapping for $\alpha = 1/2$ ............................. 75

3.6 Application implications .................................................................. 78

3.6.1 Analysis under average fault model ............................................. 78
3.6.2 Considering messages with different periods ................................. 80
3.6.3 Retransmission mappings for dual-criticality scenarios ..................... 81

3.7 Related works................................................................................... 83
Chapter 4: Provably-good strategies for data placement in distributed Key-Value stores

4.1 Introduction

4.2 System model and performance metrics

4.3 Deterministic policies
   4.3.1 Simple policies work well against random clients
   4.3.2 No deterministic policy works against an adversarial client

4.4 Randomized policies with no transfers
   4.4.1 Upper Bounds for Balls into Bins
   4.4.2 Lower bound on algorithms with no data transfer
   4.4.3 Constraining the request sequence

4.5 Randomized policies with transfers
   4.5.1 An overview of the system $Y$
   4.5.2 Bounding the execution time of a batch process
   4.5.3 Bounding the number of rejected requests

4.6 Related work

Chapter 5: Simulation of data-placement policies for distributed key-value stores

5.1 Introduction

5.2 Evaluating assumptions and estimate parameters
   5.2.1 Evaluating request processing
   5.2.2 Evaluating data-transfer model
   5.2.3 Estimating data-transfer latency and bandwidth

5.3 Simulating workloads
   5.3.1 Adversary workload
   5.3.2 Zipfian workload
   5.3.3 Cloud workload

5.4 Simulation design
   5.4.1 Simulation framework of distributed key-value store
   5.4.2 Implementations of data-placement policies

5.5 Evaluation
5.5.1 Comparing throughput between the Rand strategy and the Batch strategy ................................................................. 124
5.5.2 Comparing throughput between the Batch strategy and the Batch* strategy under different design choices ....................................... 126
5.5.3 Comparing latency between the Rand strategy and the Batch* strategy 130

Chapter 6: Conclusion ........................................................................... 132
References .............................................................................................. 134
# List of Figures

| Figure 2.1: | AMCilk’s Client-Server Architecture. | 13 |
| Figure 2.2: | Implementation of DREP scheduler via AMCilk policy-customization interface. | 15 |
| Figure 2.3: | AMCilk scheduling framework. | 16 |
| Figure 2.4: | Runtime snapshot. Container 1 is allocated with 4 cores with 4 active workers, while Container 2 is allocated with 2 cores. | 17 |
| Figure 2.5: | A job’s worker array, storing all its workers sorted in a way that makes it easy for active workers to mug and steal. | 21 |
| Figure 2.6: | Time cost of key functionalities | 27 |
| Figure 2.7: | Average flow time with Bing Search Workload (left) and Finance Workload (right). Flow time is measured as the time span from the moment when a request arrives at the system to the moment when the system completes the request. AMCilk implements preemptions at frame boundaries and this experiment compares the result to preemptions compared at steals as proposed by prior work. | 32 |
| Figure 2.8: | Average flow time with different request frequency with Bing Search Workload. Flow time is measured as the time span from the moment when a request arrives at the system to the moment when the system completes the request. | 33 |
| Figure 2.9: | Elastic scheduling with task 1 and task 2. The period of task 1 reduces to 1/3 of its original value at an arbitrary time while the period of task 2 remains the same. We show the deadline miss rate (vertical axis) of the task 1 with different original task 1’s period (horizontal axis). | 34 |
| Figure 2.10: | Adaptive scheduling. We run two jobs at the same time with three schedulers (horizontal axis): (1) No partition where each job occupies all cores, and so every core has two jobs running; (2) Eql-partition where each job exclusively uses half of number of cores in system; (3) A-STEAL where each job gets cores on demand according to its runtime parallelism (our scheduler). | 37 |
Figure 2.11: We co-run streaming and latency-sensitive jobs with four settings (horizontal axis): (1) No CP/MBA where neither cache partitioning nor memory bandwidth allocation is set between the two jobs; (2) CP where only cache partitioning is set; (3) MBA where memory bandwidth allocation is set; (4) CP+MBA where both cache partitioning and memory bandwidth allocation are set. We run the four settings with two workloads (shown in legend): (1) Medium workload where the frequency of job requests is medium; (2) High workload where the request frequency is high. 

Figure 2.12: The number of steal cycles on processor 2 in the subscription of runtime information. We run a program which repeatedly switch between high-parallelism phase (red) and low-parallelism phase (white). At 0.1 second intervals, we measure the steal cycles (vertical axis) of processor 2 which is the total time (nanoseconds) the processor 2 spends on work stealing in the previous interval. This information is published to the subscriber and the black line represents the subscriber’s view of this information.

Figure 3.1: Numerical results of DIV. Figure 1a presents the ratio of DIV length to DUAL length for $n \in [1000, 10000]$ and $f \in [20, 100]$; Figure 1b keep $f$ constant and shows the sub-linear trend of DIV length as $n$ increases, comparing to the linear trend of DUAL; Figure 1c presents the trend of $r_1, r_2, r_3, r_4$ as $n$ increases, where $r_i$ denotes the minimal $n/f$ such that the length of DIV is shorter than DUAL length over $i$. DUAL mapping is the baseline.

Figure 3.2: The trend of $r_1, r_2, r_3, r_4$ of RDC as $n$ increase, where $r_i$ denotes the minimal $n/f$ such that the length of RDC is shorter than DUAL length over $i$.

Figure 5.1: 50%, 75%, 90%, and maximum latencies of data transfer under different settings.

Figure 5.2: Normalized frequency of latency of standalone data moves.

Figure 5.3: Request frequency of chunks in the Zipfian workload.

Figure 5.4: Request frequency of chunks in the Cloud workload with different numbers of requested chunks.

Figure 5.5: Mean, maximum, 90%, and 50% reject ratios of the Batch strategy and the Rand strategy under the Adversary workload, the Zipfian workload, and the Cloud workload with different speedups and different values of $s$. RandX denotes the Rand strategy with a processing speedup of $X$. BatchX denotes the Batch strategy with a speedup of $X$. 

vii
Figure 5.6: Mean, maximum, 90%, and 50% reject ratios of the Batch strategy and the Batch* strategy under the Adversary workload, the Zipfian workload, and the Cloud workload for different options of request consuming. The SecondaryOnly option denotes the strategy that only consumes the request in the secondary queue. The SecondaryFirst option consumes the primary queue only if the secondary queue is empty. The Isolated option spends a half busy time consuming the primary queue and a half busy time consuming the secondary queue. The PrimaryFirst option consumes the secondary queue only if the primary queue is empty. $s = 100$. 

Figure 5.7: Mean, maximum, 90%, and 50% reject ratios of the Batch strategy and the Batch* strategy under the Adversary workload and the Zipfian workload for different design options of destination selection for data transfers. The 2Q option denotes the strategy which selects the destination server with the minimal length of the secondary queue when moving data. The Q option denotes the strategy which selects the destination server with the minimal total length of both queues when moving data. $s = 100$. 

Figure 5.8: Mean, maximum, 90%, and 50% latencies of the Batch* strategy and the Rand strategy under the Adversary workload, the Zipfian workload, and the Cloud workload with different speedups and different values of $s$. RandX denotes the Rand strategy with a processing speedup of $X$. Batch*X denotes the Batch* strategy with a processing speedup of $X$. $s = 100$ and $s = 500$. 

viii
# List of Tables

<table>
<thead>
<tr>
<th>Table 2.1:</th>
<th>Running time of data-intensive jobs</th>
<th>30</th>
</tr>
</thead>
<tbody>
<tr>
<td>Table 3.1:</td>
<td>Comparison of number of slots between RDC and DUAL (baseline) for small $n$’s and $f$’s</td>
<td>76</td>
</tr>
<tr>
<td>Table 5.1:</td>
<td>The minimal time span (milliseconds) of processing a single request when running the default YCSB workloads. The request includes read, scan, insert, and update</td>
<td>112</td>
</tr>
<tr>
<td>Table 5.2:</td>
<td>Throughput of the default YCSB workloads. Each value is the number of requests processed per second in a particular workload</td>
<td>113</td>
</tr>
</tbody>
</table>
Acknowledgments

Throughout my Ph.D. period, I have received a great deal of support and assistance. First and foremost, I want to thank my advisor, Dr. Kunal Agrawal. She is a talented person with much experience in theoretical research. Her insightful feedback always pushes me to sharpen my idea and make my work to a higher standard. Moreover, I learned a lot about thinking and expressing ideas cleanly, which is incredibly beneficial when facing complicated problems in my future career.

I want to thank my research collaborators, especially Dr. Jing Li, Dr. Jeremy T. Fine-man, and my lab mate, Jinhao Zhao. Jing gave me a great help to start my Ph.D. journey. Further, she introduced me to an opportunity at Apple, where I had a fantastic internship. Jinhao is a very interesting and reliable teammate who is very good at theoretical research. I enjoyed working with him and learned a lot from him, particularly how he thinks. Jeremy is a reliable collaborator. I still remember his great support of my RTSS paper, which is my first theoretical paper.

I want to thank other committee members: Dr. Sanjoy Baruah, Dr. I-Ting Angelina Lee, and Dr. Christopher D. Gill. Especially, I am very grateful to Dr. Sanjoy Baruah, the committee chair, for his organization of the dissertation proposal and defense.

Thank the Department of Computer Science and Engineering for making this place our second home. I greatly appreciate the department staffs, who are always willing to help and support me during my study.

I want to thank my lovely girlfriend, Minyang Dai, for her accompany and support during the most challenging time of my Ph.D. period.

I want to end this list of acknowledgments by showing my most profound gratefulness to my Dad and Mom. With their support, I start the Ph.D. journey.
The work described in this dissertation is supported, in part, by the National Science Foundation (USA) under Grant Numbers CNS–1948457, CCF–1733873, CCF–1725647, CCF–1918989, and CCF–2106759.

Zhe Wang

Washington University in St. Louis

December 2022
Dedicated to my parents,
Qingyu Wang and Chunrong Wu,
who have supported me to start this journey.
Parallel and distributed systems are pervasive, such as web services, clouds, and cyber-physical systems. We often desire high throughput and small latency in the parallel and distributed system. However, since the system is distributed and the input is online, scheduling for high throughput while keeping the latency small is often challenging. In this dissertation, we developed scheduling algorithms, policies, and mechanisms to approach high throughput with small latency in various parallel and distributed applications. First, we developed AMCilk runtime system for running multi-programmed parallel jobs on many-processor machines. When running parallel jobs, the allocation of processors to the parallel jobs often changes over time. Since AMCilk conducts responsive reallocation of processors between parallel jobs, various applications achieve significant throughput boosts and latency reductions compared to existing solutions. Second, we built static schedules for the real-time transmission of messages in shared communication media. Medium errors can cause transmission failures, and they are unpredictable. Since our schedules tolerate medium errors with low overhead, real-time applications can transmit messages at high throughput. Finally, we analyzed various data-placement strategies for distributed key-value stores. In distributed key-value stores, a large amount of data is stored across multiple servers. When a request to access data arrives, it is routed to the appropriate server, queued, and eventually processed.
If the queue is full, then requests can be rejected. We analyze various data-placement strategies and compare their capacities of avoiding rejections and achieving high throughput with small queue sizes.
Chapter 1

Introduction

1.1 Scheduling for high throughput and small latency

Parallel and distributed systems are pervasive, such as interactive cloud services [1,79,83,136] and cyber-physical systems [86,92,104]. In the parallel and distributed system, throughput and latency are often measured to capture the system performance. Throughput indicates how many jobs the system completes within a specified period. System administrators often pursue high throughput for effective use of resources. On the other hand, users often care about latency. Latency indicates how long users have to wait to get the results of their jobs. A short or bounded latency is often critical in many applications, such as interactive and real-time applications, where latency impacts user experience and safety. In this dissertation, we design algorithms, policies, and mechanisms to achieve solutions, providing high throughput while keeping latency small for various parallel and distributed systems.

However, throughput and latency are often competing criteria in the parallel and distributed system. As system load increases, throughput increases, but latency also grows. Under sufficiently heavy loads, the latency can be unacceptably large. Various mechanisms can limit latency, but these mechanisms may diminish the throughput. For example, interactive cloud services often reject requests to bound the latency and prevent the system from overloading. However, inefficient designs and implementations can lead to many rejections when the system is underutilized, resulting in low throughput. We must carefully design
and implement the parallel and distributed system to achieve high throughput with small latency.

When developing parallel and distributed systems, achieving high throughput with small latency is often challenging, mainly due to the following two reasons:

- **Distributed nature.** System components are distributed. A component often has limited runtime information of another component. Thus, it is hard for the system scheduler to make the globally optimal decision.

- **Online nature.** System input and environment are dynamic and unpredictable. Therefore, the system scheduler has difficulty to make the optimal decision without knowing the information in advance.

Moreover, due to their distributed and online natures, scheduling policies face difficulties to implement efficiently, which degrades the application throughput and latency further.

### 1.2 Contributions

In this dissertation, we explored scheduling algorithms, policies, and mechanisms to provide high throughput with small latency for various parallel and distributed applications, including multiprogrammed parallel computing on many-processor machines, fault-tolerant transmission on shared media by static schedules, and distributed key-value stores. While these applications are quite different, we face the common challenge of their distributed and online natures. This dissertation presents the following theoretical and experimental results that provide high throughput with small or bounded latency:

I. **An adaptive-scheduling system for multiprogrammed parallel jobs**

Modern parallel platforms, such as clouds or servers, are often shared among multiple different parallel jobs. To run multiprogrammed parallel jobs, various scheduling
policies are developed, such as online scheduling to minimize average latency [1], adaptive scheduling using parallelism feedback [4], and elastic parallel real-time scheduling [104]. These policies often change processor allocation for parallel jobs over time. However, existing parallel runtime systems are optimized for only running a single parallel job [27, 55]. Directly using the existing runtime systems to run multiple parallel jobs often incurs high overhead and inefficiency, resulting in degradation of the application throughput and latency.

We developed AMCilk, a runtime system designed to run multiprogrammed parallel jobs on many-processor machines. In the AMCilk system, the processor reallocation between parallel jobs is fast without incurring high overhead. Thus, significant throughput improvements and latency reductions have been achieved in various applications. The results of this work have been published in the following peer-reviewed conference and the peer-reviewed journal:


II. Static schedules with sub-linear overhead for fault-tolerant transmission

Shared communication media, such as a shared bus and wireless network, are widely used in real-time applications. In real-time applications, it is often beneficial to use static schedules (where the time slot to send each message is determined in advance) to exchange messages due to a deterministic upper bound on message delivery time.
When transmitting messages in shared media, medium errors may happen, resulting in transmission failures. If a medium error happens to a message, the message must be repeatedly sent until the message is correctly transmitted. A static schedule must reserve sufficient slots for each message to tolerate medium errors in transmission. It is crucial that the messages be delivered in a timely manner, and hence we are interested in minimizing the length of the schedule that achieves the desired level of fault tolerance. Since medium errors are unpredictable, achieving short static schedules is challenging.

We provided an efficient algorithm for producing a schedule for $n$ messages with total length $n + O(f^2 \log^2 n)$ that can tolerate $f$ medium errors. We also prove that fault-tolerant schedules with length $n + O(f \log f \log n)$ exist. Since $n$ steps are required to transmit $n$ messages, the overhead of fault tolerance is characterized by the additive terms of $O(f^2 \log^2 n)$ and $O(f \log f \log n)$, respectively. These terms are sublinear in $n$ and represent asymptotic improvements to the previously best-known schedule [3], which has overhead $fn/2$. As a result, our schedules achieve higher throughput of real-time transmissions. Results of this study have been published in the following peer-reviewed conference:


III. Provably-good strategies for online data placement in distributed key-value stores

Distributed key-value stores have been widely developed for academia and commercial use [46, 59, 79, 83, 123, 136], where the data is distributed among servers. When a request to access data comes in, it is routed to the appropriate server, queued, and
eventually processed. If a queue of a server is full, then requests may be rejected. With inappropriate data placement, some servers can get a large fraction of the requests and become overloaded, resulting in many rejections and low throughput. Existing related works are either empirical [79, 136] or making stochastic assumptions on the arrival pattern of requests [42, 107].

We consider the problem from a theoretical and online perspective. We assume that input requests are generated by an adversary and compare various strategies under the worst-case setting. We proved that no deterministic strategy works well. We showed that a simple randomized strategy could consume a constant fraction of requests at most. Finally, our result indicates that resource augmentation, constraint request frequency, and data transfer between servers are essential to achieve almost optimal throughput with small queues in the presence of the adversary.

The results of this research will be presented at the following peer-reviewed conference:

- Zhe Wang, Jinhao Zhao, Kunal Agrawal, Jing Li, He Liu, and Meng Xu. "Provably Good Randomized Strategies for Data Placement in Distributed Key-Value Stores." In 2023 Principles and Practice of Parallel Programming (PPoPP)

1.3 Organization

The rest of this dissertation is organized as follows. Chapter 2 considers a scenario of multi-programmed parallel computing and presents the AMCilk runtime system. Chapter 3 considers a real-time transmission problem on shared media and develops several static schedules to transmit messages at high throughput. Chapter 4 and Chapter 5 consider an online data-placement problem in distributed key-value stores and explore various strategies theoretically and experimentally to achieve high throughput with short queue capacity. Finally, Chapter 6 summarizes all contributions presented in this dissertation.
Chapter 2

An adaptive-scheduling system for multiprogrammed parallel computing

2.1 Introduction

Motivation  In recent years, the number of cores on multiprocessor and multicore systems has been increasing at a rapid rate. With this trend, there is an increasing interest in running many parallel jobs on a single machine at the same time, especially in the context of shared environments such as clouds and shared clusters. However, most parallel runtime systems, such as Cilk variants [8, 20, 56, 71], OpenMP [103], and TBB [70], are designed to run a single parallel job. To run multiprogrammed workloads, one must frequently instantiate one runtime system for each job. Since these runtime systems are unaware of being in a multiprogrammed environment and often assume that they have a certain number of cores, say $p$ (often the entire machine), dedicated to running their single job, they create $p$ pthreads, pin them to each of these cores and use them to execute for the duration of the job. This leads to suboptimal performance for jobs in these environments.

For multiprogrammed environments, the system scheduler must decide how to allocate system resources among the different jobs in the system. This allocation depends on the performance goal of the system and different applications with multiprogrammed workloads may have different performance goals. For instance, an interactive web service running on a cloud may care about minimizing some function of the latency of the jobs. On the other
hand, a real-time application running on an embedded device may require that jobs meet their deadlines. There has been significant theoretical research on designing schedulers for various performance goals, e.g., minimizing some function of the job latencies [1,2,4,5,51,63, 75,88,109,115] and guaranteeing no deadline misses [73,84,86,90,100,104]. However, most of these schedulers have either not been implemented or implemented using a custom-built system for that application scenario.

**Problem**  In this work, our goal is to design a high-performance, flexible and extensible framework for enabling multiprogrammed workloads in a shared environment. Since the different multiprogrammed parallel workloads have various job arrival patterns, job memory access characteristics, requirements and performance objectives, we want to design the parallel runtime system that enables the following functionalities: (1) **Online arrival:** Jobs can arrive online, and the scheduler does not need to know what jobs will arrive in the future; (2) **Dynamic reallocation:** The scheduler can dynamically increase or decrease the number of cores allocated to a job while the job is executing; (3) **Efficient execution:** The job must efficiently use the cores that are assigned to it at any moment using an efficient parallel scheduling algorithm such as work-stealing [8]; (4) **Cache management:** The job scheduler can support cache partitioning and memory bandwidth allocation, as a complement to core allocations, to mitigate the cache and memory bandwidth contention and support quality of service; and (5) **External Resource Control:** When AMCilk is sharing a machine with other processes (say in a shared cloud environment) an external scheduler should be able to control the resource occupancy by the AMCilk runtime.

In most parallel runtime systems, dynamically changing the number of cores allocated to the job is difficult and expensive for multiple reasons. Since multiprogrammed systems often run each job in its process, deallocating a core from one job and allocating it to another often involves an operating system (OS) call. Since the OS may not be aware of what is happening within the job, the thread running on a deallocated core may be holding a lock or
be in some unsafe state when it is de-scheduled, compromising the efficiency of the parallel program. Moreover, the kernel operations involved when reallocating cores are likely to be expensive. Finally, the job scheduler may have high inter-process communication overhead for collecting runtime information required to make scheduling decisions.

Contribution In this chapter, we take a different approach. We design AMCilk (adaptive multiprogrammed Cilk), a parallel runtime framework extending the Cilk runtime systems to efficiently support multiprogrammed scenarios in a shared environment. Specifically, AMCilk has the following features:

- AMCilk allows a system administrator to implement their preferred scheduling policy to allocate cores among different jobs to optimize the application-specific performance criterion by exposing an easy interface. The AMCilk framework then transparently (to the system administrator) implements this policy by automatically reallocating cores as dictated by the policy.

- AMCilk’s client-server architecture allows jobs to be submitted online, start new jobs dynamically and return results of completed jobs to clients.

- AMCilk concurrently runs multiple parallel jobs in a single runtime system, so that the AMCilk scheduler can access the full runtime information of jobs and enforce core reallocation with low overhead.

- AMCilk develops a safe, low-cost, and responsive preemption mechanism, which allows reallocating cores between jobs in microseconds while the jobs are running. Thus, it has little performance penalty on the jobs. Note that the “preemption” in this chapter denotes the action of stopping the execution of a parallel job on a processor, and the AMCilk runtime system enables this preemption mechanism.
• AMCilk exposes interfaces that use the hardware-level cache partitioning and memory bandwidth allocation to restrict the interference between jobs and to control the quality of service when multiprogrammed jobs compete for the last-level cache and memory bandwidth.

• AMCilk can be run in an environment where other processes share the hardware resources with it. To enable this, AMCilk exposes its resource allocation interface to external schedulers which can then control the resource occupancy of AMCilk. AMCilk then seamlessly adapts to this new resource allocation.

• In order to enable an intelligent allocation by the external scheduler, AMCilk exposes its runtime information to these schedulers using a subscription model.

Therefore, when building applications using AMCilk, system administrators can customize the core allocation, cache partitioning and memory bandwidth allocation policy via AMCilk interfaces, without needing to understand the implementation details.

Our evaluation indicates that the overheads of starting a new job, completing a job, reallocating cores, etc., within AMCilk are small, and the core reallocation adds a minimal performance penalty on job executions. Moreover, we implemented application scenarios using AMCilk to understand whether AMCilk provides performance improvement to their application-specific criteria. In particular, we developed four applications by implementing their scheduling algorithms via the AMCilk policy-customization interface. The first one [1] has the goal of minimizing the average latency of online parallel jobs, such as those in interactive services. We find that the implementation based on AMCilk provides a performance advantage of between 60 to 70% over the previous implementation (which was used in the experiment of [1]), which uses the same scheduling policy — the performance improvement is purely due to AMCilk’s ability to reallocate cores faster than the previous implementation. The second one is an elastic real-time application [104] with periodic tasks that must
meet deadlines, where some tasks can vary their demand causing other tasks to adjust their deadlines accordingly. Again, we see that AMCilk provides better responsiveness to the demand change, providing better performance to the application. The third application is an application that dynamically adapts the number of cores according to the parallelism of the applications and requires that we monitor the jobs to adjust the core allocation. We see that the AMCilk implementation successfully adapts to the changing parallelism providing better performance than the best static allocation. The forth application demonstrates the importance of cache and memory bandwidth partitioning in multiprogrammed environments. In addition, the final experiment shows the efficiency of the subscription functionality for external schedulers.

2.2 Background

AMCilk is implemented for the Cilk language using a home-grown Cheetah runtime system, which is similar to Intel’s Cilk Plus runtime system [71]. Cilk [20] is a parallel programming language that extends C, while Cilk Plus was designed later for C++. Here we describe the key features of Cheetah that are critical for understanding the design of AMCilk.

Cilk Plus language and Cheetah runtime system. Cilk Plus extends C++ with additional keywords, principally including spawn and sync. A function that is spawned may execute in parallel with the continuation of its parent function. The sync keyword indicates that all function instances spawned by the current function must return before the next instruction. Therefore, the programmer expresses the logical parallelism of the program, while the Cheetah runtime system is responsible for scheduling this program on the given number of cores. The compiler and linker compile the program by inserting calls to the runtime system at function spawn, return, and sync. The program’s main function is compiled as the cilk_main function, while the newly added main function performs runtime initialization by creating p threads, one for each core, and pins them on their cores. It also
sets up data structures for scheduling this program on these threads. One key data structure
is a **worker** for each thread, which keeps track of information about that thread from the
perspective of the program — for most of this chapter, we will use the term worker and
thread interchangeably. After initialization, the runtime calls the `cilk.main` function to
begin executing the program.

**Work-Stealing.** Work-stealing [8] is a theoretically good and practically efficient schedul-
ing algorithm used by many programming languages and libraries, such as Cilk variants [8,
20,56], OpenMP [103], and Intel’s TBB [70]. Same as common Cilk variants, in the Cheetah
runtime system, each worker maintains its own deque (a double-ended queue) of stack frames
and pushes/pops stack frames from the bottom of the deque. If a worker’s deque is empty,
it becomes a **thief**, picks a random victim among the other workers, and steals the frame
from the top of the victim’s deque and starts executing it.

**THE Protocol.** A worker pushes and pops frames from the bottom of its own deque, while
a thief might steal work from the top of another worker’s deque. Therefore, if there is only
one frame on a deque, any thief who tries to steal it must synchronize with the owner to
ensure consistency. The Cheetah runtime system employs the **THE** protocol [56] to perform
the synchronization efficiently. The THE protocol uses three shared atomic variables: T, H,
and E. T and H mark the head and tail of the deque, and E is an exception pointer and
marks a place where T cannot cross over.

Generally, E and H both point at the head of a deque, while T points at the tail. When
a worker pushes a frame on the deque, it simply increments T. When a thief tries to steal
from the top of the deque, it grabs the lock of the victim’s deque and increments E. If E ≤
T, the thief steals the top frame and increments H; otherwise, it gives up and restores E. It
then releases the deque lock. When a worker tries to pop a frame, it decrements T and then
compares it with E. If E ≤ T, then the worker can pop without getting any locks. If E > T,
the worker calls an exception handler within the runtime system. Generally, this means that
some thief is trying to steal while the victim is trying to pop. In this case, the victim also tries to get the deque lock, and either the thief or the victim wins based on who gets the lock.

This E pointer can also be used to trigger exceptions of other kinds — essentially, by setting E to be larger than T, we can force the thread to enter the exception handling routine within the runtime system and then modify the exception handling routine to perform other operations. We will use this functionality in AMCilk to inform the worker to perform core reallocations — described in Section 2.3.

2.3 System design

2.3.1 A client-server architecture

Figure 2.1 illustrates the conceptual client-server architecture of AMCilk. A client (i.e., user) creates a job request struct, which stores the program id (indicating which program to run) and its input parameters. It submits the job request to the server via a pipe. AMCilk has a dedicated request receiver thread (pinned to a dedicated core) that listens for requests and on receiving a request, pushes it into a FIFO job request buffer. The AMCilk scheduler takes job requests from the head of the buffer, parses the request, and prepares to run the executable of the corresponding program. When a job finishes, the server sends the result to the client. The result is the return value or the location where the return value stored. Both request receiver and AMCilk scheduler are nonblocking — they do not wait for a job request to complete before starting on the next one.

AMCilk runs multiple Cilk jobs in a single runtime system. Recall that (Section 2.2) the original Cilk runtime system is designed to run a single job, where the main function of the job’s executable initializes the runtime and calls cilk_main as an entry into the user code. In contrast, the AMCilk runtime system is pre-initialized as a server and sets up the
basic data structures needed to execute jobs. The parallel programs are pre-compiled and pre-linked with the runtime system and have their cilk\_main functions. To run multiple jobs, the server runs each job within a data structure called a container which contains all the metadata required to run Cilk jobs. Since jobs arrive and leave online, the number of active containers changes over time. However, creating a container from scratch is relatively expensive, so AMCilk creates a pool of containers at initialization and reuses the containers. When a new job arrives, the server selects an inactive container and calls the appropriate cilk\_main function to start executing the job. When all containers are busy\(^1\), any new arriving job is buffered. When a container becomes available, it picks a job from the buffer in a FIFO order.

### 2.3.2 Policy-customization interface

AMCilk provides an interface that allows the system administrator to customize the policy for allocating cores, cache, and memory bandwidth between concurrent jobs. We provide some useful allocation policies “out of the box” — these are the policies we used in our case studies described in Section 2.7, namely (1) DREP; (2) ELASTIC\_RT; and (3) PARALLELISM\_FB. System administrators can design their own policies and implement them using a simple interface provided within AMCilk.

\(^1\)This case rarely happens, since we use a large pool — we set the number of container to be equal to the number of cores used for executing jobs.
The reallocation decision interface is event-driven. AMCilk provides four events: (1) \texttt{START\_JOB}; (2) \texttt{EXIT\_JOB}; (3) \texttt{TIMER}; (4) \texttt{REQUESTED}. When any event happens, the \texttt{job\_scheduler(e)} function is called — this is the function that the system administrator implements in order to design their own core-allocation policy. The \texttt{job\_scheduler(e)} has an argument \texttt{e} indicating which event triggered the current function call (to the \texttt{job\_scheduler(e)}). The system administrator can use this argument to distinguish different events and define appropriate response to different events (or ignore some events).

Within this function implementation, the system administrator can use pre-defined functions to both get information about the current state of the runtime system and to change the allocation of cores, memory bandwidth and cache. In general, to perform core-reallocation, one must (1) analyze the runtime information; (2) make a core-reallocation decision; (3) assign cores to jobs. AMCilk collects the runtime information in the backend, and the interface exposes the information to the system administrator, like the number of running jobs, the number of available cores and the current scheduling state showing which core belongs to which job. The interface also exposes in-depth runtime details, like the number of cycles when each core was working vs. stealing in the previous interval. Within \texttt{job\_scheduler(e)}, the system administrator can call various functions to access this information and use this information to make scheduling decisions. The scheduling decisions can be communicated to the AMCilk scheduler by using setter functions — for example, AMCilk defines \texttt{core\_id} to denote a core and \texttt{container\_id} to denote a container, and the system administrator can use \texttt{give\_core\_to\_container(core\_id, container\_id)} to allocate a core to a container. AMCilk will then automatically enforce this reallocation using a safe, responsive, and low overhead preemption and core reallocation method described in Section 2.4.

Using the policy-customization interface can greatly simplify the system implementation. For example, Figure 2.2 shows an implementation of DREP system [1] in AMCilk with the policy-customization interface. DREP is a system designed for online scheduling of
multiprogrammed parallel jobs to minimize average flow time. Basically, when a new job arrives, for each core, the DREP scheduler gives this core to the new job with probability of $p = 1/n_t$ where $n_t$ is the number of unfinished jobs. When a job finishes, for each core, the scheduler randomly picks an unfinished job to give that core. As shown in Figure 2.2, the system is implemented in simply 18 lines (without comments), and no system details are needed in the implementation.

AMCilk provides a similar interface to customize cache partitioning and memory bandwidth allocation policies. Again, the system administrator can access runtime information via the interface, like cache misses, and the administrator can use the interface to allocate cache blocks and set maximum memory bandwidth usage of each container. Note that AMCilk is extensible, and system experts could develop their own runtime information collectors and events under our scheduling framework.

### 2.3.3 Decentralized scheduling framework

The AMCilk scheduling framework enables concurrent running of multiple parallel jobs and reallocates computing resources, including core, last-level cache, and memory bandwidth, between jobs according to the customized scheduling policy. Figure 2.3 zooms into the architecture of the scheduler itself. The Runtime Monitoring Module keeps track of the runtime
The AMCilk scheduling framework provides interfaces that allow the system administrator to easily customize the scheduling policy for its application scenario in the Resource Allocation Module (step 5). Furthermore, AMCilk exposes an interface that allows external systems to control the resources used by AMCilk via sending the demand to the request receiver thread (step 6), which invokes the AMCilk scheduler to enforce the allocation demand (step 7).

To perform the cache partitioning and memory bandwidth allocation decided by the scheduling policy, Resource Enforcement Module calls the interfaces provided by third-party infrastructures. For example, Intel RDT [69] that we use in this work provides interfaces for allocating last-level cache and memory bandwidth to core groups. So the AMCilk scheduler groups the cores assigned to each running job and calls Intel RDT to perform the allocation to the core groups.
To support concurrent execution and dynamic core allocation of multiple parallel jobs, AMCilk decouples the concept of the core (physical processing unit) and the worker (software abstraction of a core). For a machine with $p$ cores (excluding the core dedicated to the request receiver thread), AMCilk creates $p$ workers (threads) for each container dedicated to a job, and each of these workers is pinned to a different core. Hence, each core has multiple workers, one for each container. The Resource Enforcement Module ensures that each running job occupies a disjoint set of cores according to the core allocation decision, by activating at most one worker on each core. An example snapshot is shown in Figure 2.4.

![Runtime snapshot](image)

**Figure 2.4:** Runtime snapshot. Container 1 is allocated with 4 cores with 4 active workers, while Container 2 is allocated with 2 cores.

For each job, the cores allocated to this job must complete its work using a modified work-stealing scheduler that we augmented to support three novel functionalities needed by the AMCilk scheduling framework: decentralized scheduling, core reallocation, and work resumption. We explain the decentralized scheduling here and the other two mechanisms in the next subsections.

Although a core is dedicated for the AMCilk scheduler (leaving $p - 2$ cores allocated by the scheduling policy for executing jobs$^2$), instead of a dedicated centralized thread for the scheduler, each container handles its own allocation by setting its worker 1 be a dedicated

---

$^2$One core is dedicated for the AMCilk scheduler, and one core is dedicated for receiving job requests, leaving $p - 2$ cores to execute jobs.
scheduler worker when *starting a job* and *removing a job*. Specifically, to start a new job from the request buffer, a container from the container pool is activated by waking up its worker 1. This worker prepares all the necessary data structures for this job and decides which cores should be allocated to this job, based on the customized scheduling policy provided by the system administrator. This will trigger reallocation so that these cores are allocated to this new job. At this point, the *cilk main* function of this new job is called and the job execution begins. When a container completes a job, one random worker returns from the *cilk main* and enters the runtime. This worker will activate the worker 1 of its container before putting itself to sleep. Then this worker 1 will clean up the data structures for this job, trigger the core reallocation per the scheduling policy, and inactivate itself (and this container) once done. If other scheduling events occur, for instance, due to external triggers or timing triggers, a dedicated thread pinned on core 1 for the AMCilk scheduler will wake up to make the new scheduling decision and trigger the core reallocation.

### 2.4 A responsive, low-cost mechanism of processor reallocation

#### 2.4.1 Responsive processor reallocation

During job execution, the scheduling policy may decide to change the core allocation of jobs, i.e., some job(s) must give some of their cores to other jobs, and some job(s) may reclaim the cores it gives out in the previous scheduling, triggering AMCilk’s core reallocation mechanism. Reallocating a core $x$ that is currently used by job $a$ to job $b$ involves two procedures: putting the running worker of job $a$ on core $x$ to sleep and waking up the worker of job $b$ on core $x$. The second procedure can be achieved by simply sending a signal to wake up the corresponding thread. If the woken-up worker has some work on its deque, then it resumes working on its deque. Otherwise, it immediately starts stealing.
The first procedure, namely *worker preemption* where a worker stops working and goes to sleep, is the key to core allocation. This operation must be *safe* (i.e., we don’t want to preempt a worker while it is holding a lock, for example), *responsive* (i.e., given a reallocation decision, the worker should go to sleep as soon as possible), and *low overhead* (i.e., its overhead should have minimal impact on performance).

There are a few options for implementing worker preemption. One possibility is to use the priority mechanism of the operating system (OS). Say the scheduling policy decides to allocate a core $x$ to job $b$ while it is currently allocated to job $a$. The containers for both jobs have a thread pinned to core $x$, so the scheduler could increase the priority of the job $b$’s thread on core $x$ and decrease the priority of the job $a$’s thread. One disadvantage of this method is that this context switch has high overheads. More importantly, it is difficult to ensure correctness and performance since the thread of job $a$ might be holding a lock when it is put to sleep by OS, causing it to block other threads from doing work.

To ensure that the thread is put to sleep when it is safe to do so, another approach, taken by Agrawal et. al [1], is to allow worker preemption only when the worker attempts to steal. In particular, on receiving the decision that a worker $w$ must be put to sleep, the corresponding work-stealing scheduler waits until worker $w$ has no work on its deque and is about to steal. At this point, it puts the thread to sleep. This is, in some sense, the safest and easiest place to implement a preemption within the runtime system since, as described in Section 2.2, the worker is not working on anything and does not have any work on its deque. However, this mechanism would not be very responsive since the worker may not steal for a long time. Therefore, the time between the occurrence of the decision that some core $x$ should be moved from job $a$ to job $b$ and the time when job $a$ actually puts its worker on core $x$ to sleep can be huge.

In contrast, we employ the middle road and use the exception mechanism of the Cilk runtime system (described in Section 2.2) to implement preemption. When the AMCilk
scheduler decides to take away core \( x \) from a job, it sets the exception pointer \( (E) \) of the worker \( w \) on core \( x \) to a large number. When worker \( w \) finishes its current frame, it finds that \( E > T \) and jumps to the exception handling routine. This routine then sets up the state indicating that worker \( w \) is now inactive and puts the associated thread to sleep. It is important to note that the preempted worker may still have work on its deque but it may never be woken up again, so efficient work resumption, explained in Section 2.4, is needed to complete the work left on this deque by other workers of the same job.

Our design choice for worker preemption is reasonably responsive since it implements preemption at frame (function) boundaries — the worker to be preempted is preempted as soon as it finishes the function it is currently executing. For most fine-grained parallel code, the individual functions are reasonably small. In addition, since the preemption is handled by the runtime system, it can ensure that the thread is not holding locks when it is preempted.

### 2.4.2 Efficient work resumption

As discussed above, since AMCilk implements preemption at frame boundaries, a worker \( w \) of job \( a \) can go to sleep while there is still work (frames) on its deque. This work must be resumed by some workers of job \( a \) so that job \( a \) can successfully complete. To facilitate work resumption, each worker has a status field. Before an active worker \( w \) goes to sleep, it first checks if its deque has any remaining work. If there is remaining work, it marks its status as inactive with work; otherwise, it marks its status as inactive without work.

All workers of the job are stored in an array of size \( p \), where \( p \) is the number of (active and inactive) workers. This array is sorted to store all the inactive with work workers at the beginning and the active workers in the middle, followed by all the inactive without work workers. We also maintain two auxiliary pointers pointing to the last location storing
an `inactive_with_work` worker and the first location storing an `inactive_without_work` worker, as shown in Figure 2.5.

![Worker Array Diagram](image)

**Figure 2.5:** A job’s worker array, storing all its workers sorted in a way that makes it easy for active workers to mug and steal.

In addition to the above data structures, we implement the key operation, called **mugging**, for efficient work resumption. Recall that, in original work-stealing, when a worker runs out of work, it randomly picks a victim and steals work from the top of the victim’s deque. In AMCilk, when an active worker of job $a$ runs out of work (i.e., its deque is empty), it first checks the worker array to see if there are any `inactive_with_work` workers. If so, it picks one as the victim and mugs the victim’s worker by swapping the victim’s nonempty deque with its own empty deque. It then moves the victim to the last portion of the worker array (the `inactive_without_work` portion, since this worker now has an empty deque) and updates both auxiliary pointers. Once there is no `inactive_with_work` worker, regular work-stealing among the active workers is resumed efficiently by storing the active workers contiguously. With the help of the two auxiliary pointers, AMCilk avoids the unsuccessful steal attempts from sleeping workers with empty deques.

Our design for the work resumption mechanism has the advantage that it maintains the theoretical and practical performance guarantees provided by work-stealing [8]. Intuitively, these guarantees depend on the fact that if there are $d$ total deques for a job, then $d$ random steal attempts will reduce the critical-path length of the job with high probability. However, if we have more deques, we need more steal attempts to make progress. In AMCilk, if there are sleeping workers with nonempty deques, we prioritize making their deques empty and never
steal from sleeping workers with empty deques. Therefore, if the job has $x$ active workers, this design only needs $x$ steal attempts to reduce the critical-path length — in systems with many jobs, the number of cores may be much larger than $x$ and this design is efficient. The theoretical guarantees provided by some multiprogrammed application scenarios [4] depend on this mechanism.

2.5 Support for shared cloud environments

Thus far, we have presented AMCilk as though it fully occupies the entire physical machine. However, in some shared environments such as clouds, the AMCilk process may share the resources, such as cores and memory bandwidth, with other processes. In these scenarios, an external scheduler, like a cluster scheduler, should be able to dynamically control the resource occupied by the AMCilk process.

This section introduces the key features of AMCilk that allow the AMCilk system to run in such shared cloud environments. In particular, AMCilk runtime system supports: (1) Subscription of runtime information — an external scheduler can monitor the runtime information of each job running in the AMCilk process; (2) Resource occupancy control — an external scheduler can set the upper bound of resources used by the entire AMCilk process (AMCilk will still control how to allocate resources to each of its own jobs based on the mechanisms described in the previous section); (3) Admission control — the external scheduler can set the maximum buffer length for arriving jobs for AMCilk, thereby providing admission control for AMCilk jobs. As a result, the external scheduler can gather runtime information for the AMCilk process, use this information to decide how much resources to provide the AMCilk process, and also put constraints on the buffer length and the resource occupancy of the entire AMCilk process. Note that this external scheduler is outside of AMCilk and AMCilk has no control over it.
2.5.1 Subscription of runtime information

There are three components in the runtime information provided by AMCilk: (1) hardware usage, including processors, memory, last-level cache, off-chip memory bandwidth. This is the basic information showing how busy the AMCilk system is. However, this hardware usage information is often misleading. Recall that each parallel job running in the AMCilk is executed by a work-stealing scheduler. If a job is allocated more processors than its parallelism, some worker of the job can be busy for stealing attempts. In this case, although the processor usage is high, this job does not need as many as the processors it is allocated. Thus, it may be useful for the external scheduler to know (2) the internal runtime information for each job, such as working cycles and stealing cycles to know the actual resource utilization for each job. Furthermore, the external scheduler need to distinguish whether the AMCilk system is overloaded when the resource utilization is high. Therefore, AMCilk exposes (3) the length of request buffer to the cluster scheduler.

AMCilk also provides interfaces which allow system designers to define any metric based on the runtime information, and AMCilk system automatically exposes the metric to the external scheduler via the subscription. For example, if the external scheduler needs to know the current average flow time of jobs, the system designer sets the AMCilk to gather the flow time of each jobs and sets a time window to calculate the average within the window. Currently, AMCilk supports the subscription of average flow time, maximum flow time, minimum flow time and percentile latency. System designers can either choose one of these metrics or design their own metric — AMCilk will then calculate this metric and provide the information through to the subscriber.

The subscription is implemented following publish–subscribe pattern, which decouples the AMCilk and the external scheduler and reduces data size in communications. Since the external scheduler may be located at a different machine, the runtime information should be
visible by a different machine. The AMCilk system runs as a publisher which periodically updates the runtime information to keep updated. Noting that the external scheduler may not need all runtime information that AMCilk exposes. Thus, the runtime information is organized in key-value pairs, where keys are the name of resources, like ”stealing_cycle”. The runtime information is updated in a transactional manner — either all keys are updated or no key is updated. The external scheduler runs as a subscriber. By providing the keys of interest, the external scheduler can get the specific values for those keys from the latest runtime information on demand.

### 2.5.2 Resource occupancy control and admission control

AMCilk runtime exposes interfaces to the external scheduler to control the occupancy of three resources: (1) the number of processors; (2) last-level cache; (3) off-chip memory bandwidth. When the external scheduler decides to adapt the resources occupied by the AMCilk process, the AMCilk process leverages the mechanisms we described in Section 2.3 to adapt to these new resources. In particular, once the AMCilk runtime receives a request to adapt resource occupation from the external scheduler, the internal AMCilk scheduler runs and changes the allocation of its jobs based on its current policy. Once the enforcement is done, AMCilk notifies the external scheduler. Since the resource reallocation of jobs in the AMCilk is fast, the external scheduler can adapt the resource occupancy of AMCilk with low latency. Note that given a change in resources, the actual allocation of resources to the jobs running under AMCilk is based on the scheduling policy provided to AMCilk via the AMCilk interface. Therefore, when running in a shared environment, system designers must provide scheduling policies which can intelligently respond to resource changes to get maximum benefit from this feature.

Recall that the AMCilk runtime system runs as a server which receives job requests from clients. Once a job request is received, this request is appended to a request buffer. If
the resources owned by the AMCilk runtime is insufficient to handle the request frequency, the number of buffered job may increase unboundedly causing long delays in processing of jobs. In order to avoid this, we allow the external scheduler to do dynamic admission control where it can change the AMCilk’s maximum buffer length by communicating with the AMCilk server. Then the AMCilk still admits the jobs in a first-come-first-serve order, but it rejects any requests which cause the buffer size to grow larger than the maximum buffer length.

2.6 Evaluation

We evaluate AMCilk performance using two types of benchmarks. In this section, we try to understand the efficiency of AMCilk implementation by quantifying the system overhead and examining the advantage of cache and memory bandwidth allocation functionalities. In the next section, we will try to understand the impact of AMCilk on multiprogrammed applications to see if AMCilk can provide a performance boost for their application-specific metrics.

We conducted the evaluation on a machine with two 2.40GHz Intel Xeon Gold 6148 Processors and 754GB memory. Each processor has 20 physical cores with 27.5 MB L3 Cache, and the system has 40 physical cores in total. The two processors support Intel RDT which provides capabilities for cache and memory allocation and monitoring. The Linux version is 4.15.0. AMCilk uses Intel(R) RDT Software Package to control the hardware-level cache partitioning and memory bandwidth allocation. In the experiments, we disabled hyperthreading. Two cores are reserved for the request receiver and the AMCilk scheduler, respectively; the remaining 38 cores are used to execute jobs.

---

3The package is open source and maintained at https://github.com/intel/intel-cmt-cat.
2.6.1 System overhead

We first conduct experiments to quantify the time costs of the four core functionalities that AMCilk promises (as discussed in Section 2.3): (1) starting a job; (2) removing a job; (3) core reallocation; (4) work resumption.

Experimental Design. We measured the overhead by instrumenting each individual operation and running a latency-sensitive application [88]. We develop a simple client to generate the workload. The workload includes a series of online requests. The work of each request is random and follows Bing Search Server Request Work Distribution [88]. Each request is computationally intensive and each request is parallelized by using parallel-for loops. Note that since we evaluate the time span of scheduling actions on parallel jobs, the result will not change much between different computations since the individual functions are reasonably small for most fine-grained parallel code. We use Poisson distribution to decide inter-arrival time between requests. By configuring the average number of requests that arrive per second, we can set the utilization of the system. We wanted to examine whether the load affects the overhead and varied the total load of the application, i.e., machine utilization from 60% to 90%, by changing the average number of requests arrived per second. We observed that the machine utilization has a very small impact on the overhead, so we only report the results for a machine utilization of 75%. The experiments were run long enough, and we measured the time to run each operation for 100,000 times and report the mean and standard deviation. To improve the readability of boxplots, we randomly sample 1,000 of the 100,000 measurements to draw Figure 2.6. We found 15 outliers in the measurements in total, and we removed the outliers when calculating the maximum latency (and 50th/95th/99th/99.5th percentile).

Evaluation Results.
As explained in Section 2.3.3, starting a job includes taking a job from the request buffer, setting up the container for this job, and allocating resources to this job. In our evaluation, this functionality takes $295\mu s$ on average with a standard deviation of $489\mu s$. Half of the measurements take no more than $291\mu s$, and almost all measurements are no more than $440\mu s$. In particular, the 95th percentile is $369\mu s$, and the 99th percentile is $414\mu s$, and the 99.5th percentile is $440\mu s$, and the maximum measurement is $8834\mu s$. Note that allocating resources to a new job often involves reallocating cores, so this time cost is dominated by core reallocation ($272\mu s$). Recall that in our design, containers are created at AMCilk system initialization and are reused upon job arrivals. We evaluate this design choice with an experiment where we create containers from scratch every time a new job arrives. As expected, always creating containers is significantly more expensive with a mean overhead of $4379\mu s$, due to the cost of creating pthreads for workers and allocating and (more importantly) initializing the data structures for the closures, frames, and fibers that the runtime system uses.

Removing a job involves deallocating cores (and other resources) of the completed job and releasing the container back to the container pool. This functionality costs $10.0\mu s$ on average with a standard deviation of $21.5\mu s$. Half of the measurements take no more than $8.71\mu s$, and almost all measurements are no more than $197\mu s$. In particular, the 95th percentile is $12.8\mu s$, and the 99th percentile is $61.2\mu s$, and the 99.5th percentile is $197\mu s$, 27
and the maximum measurement is 457µs. Removing a job takes a significantly shorter time than starting a job because it only deallocates cores. The reallocation of these cores is either performed in starting a new job or performing the core reallocation for the active jobs based on the scheduling policy.

**Core reallocation** includes deciding the resource allocation for jobs according to the customized scheduling policy and enforcing the decision. Of the 272µs average overhead (std. 480µs). Half of the measurements take no more than 267µs, and almost all measurements are no more than 401µs. In particular, the 95th percentile is 345µs, and the 99th percentile is 384µs, and the 99.5th percentile is 401µs, and the maximum measurement is 8812µs. On average only 17.5µs is spent on making the decision, so enforcing the decision introduces the major overhead. Recall that enforcing the decision involves putting a worker to sleep for one job and activating a worker for another job. Activating a worker costs 57.5µs, while putting a worker to sleep costs 85.2µs. The latter operation takes more time because it includes waiting until the worker reaches the frame boundary. Obviously, this overhead would be significantly higher if the worker has to reach a steal boundary instead.

**Work resumption** starts when a worker with a non-empty deque goes to sleep and ends when another worker successfully jumps to the user code after finding and mugging this nonempty deque of a sleeping worker. This functionality costs 7.20µs on average with a standard deviation of 7.50µs. Half of the measurements take no more than 5.54µs, and almost all measurements are no more than 53.2µs. In particular, the 95th percentile is 16.4µs, and the 99th percentile is 32.8µs, and the 99.5th percentile is 53.2µs, and the maximum measurement is 220µs. For resuming the work of inactive workers, we could let a thief steal from the victim’s deque one frame at a time, instead of mugging the entire deque. To verify our choice of mugging, we measure the overhead of both operations. We observe that a mugging operation costs 0.363µs (std. 0.204µs), which is actually less than the cost of 1.44µs (std. 3.00µs) of a successful steal. This result is as expected since a successful steal
involves taking multiple locks, manipulating data structures, and promoting the child frame to make it ready for a potential future steal. Mugging is much simpler; we just grab a lock and change some pointers around. Therefore, mugging not only reduces the number of active deques, but also has a smaller overhead.

The experiments show that all operations have small average costs, but their variations are not negligible. The variations come from contention, instead of noise. In particular, the measured time includes the operation of locking data structures before modifying them. Therefore, the cost is higher when we have to wait on the lock. Additionally, some optimizations — there are fast paths and slow paths depending on the particular situation — also lead to variation.

2.6.2 Cache partitioning and memory bandwidth allocation

Since the overheads of cache and memory bandwidth allocation of AMCilk are the same as Intel RDT, we do not measure these costs. Instead, we demonstrate their capability of reducing interference in a scenario where data-intensive parallel jobs co-run with streaming applications.

Experimental Design. We use a parallel_sort program, which takes an array as the input and returns the sorted array, as the data-intensive job. We randomly generate an array with 50,000,000 64-bit elements. We use AMCilk to run 4 such jobs concurrently, where each job is allocated with 4 cores. We also design a parallel streaming job that repeatedly loads data from memory, modify the data, and store the data into the memory. When co-running with the 4 data-intensive jobs, this streaming job is allocated with the remaining cores in the platform. We measure the running time of the 4 data-intensive jobs in 4 cases: (1) only running the 4 jobs; (2) co-running the 4 jobs with the streaming job; (3) partitioning the cache between the 4 jobs and the streaming job; (4) restricting the memory bandwidth usage of the streaming job. For each case, we record the running time for 1,000 times.
Evaluation Results. As shown in Table 2.1, when co-running with the streaming job, the data-intensive job's running time increases by 13.4%. With cache partitioning (CP), the job running time reduces by 2.8%. With memory bandwidth allocation (MBA), where we restrict 10% for the streaming job, the job running time decreases back to the time of running alone. This simple experiment shows that cache and memory bandwidth allocation can effectively reduce interference between jobs and providing this functionality is crucial to enable the design of efficient multiprogrammed systems using AMCilk.

Table 2.1: Running time of data-intensive jobs

<table>
<thead>
<tr>
<th></th>
<th>(1) Alone</th>
<th>(2) Co-run</th>
<th>(3) Co-run+CP</th>
<th>(4) Co-run+MBA</th>
</tr>
</thead>
<tbody>
<tr>
<td>Mean (second)</td>
<td>1.86</td>
<td>2.11</td>
<td>2.05</td>
<td>1.87</td>
</tr>
<tr>
<td>Std. (second)</td>
<td>0.0264</td>
<td>0.0600</td>
<td>0.0360</td>
<td>0.0286</td>
</tr>
</tbody>
</table>

2.7 Case studies

Multiprogrammed applications are ubiquitous. In this section, we present five concrete examples of multiprogrammed application scenarios with differing needs. We implemented all five scenarios using AMCilk on the hardware used in Section 2.6 and ask the following question: does the AMCilk implementation provide improved performance to these applications for the criteria that these applications care about — in other words, do the responsive and low-overhead core reallocation and cache partitioning and memory bandwidth allocation provide a measurable impact on the application-specific performance of these applications?

2.7.1 Online scheduling to minimize average flow time

In the context of interactive services, users send requests to the service, and the service must process the requests while optimizing some service-wide performance criteria. We consider the online scenario where the jobs (computation done to satisfy requests) are
parallel and the service does not know the characteristics of the jobs (such as their running times or arrival times). One of the most commonly used quality-of-service metrics is the **average flow time** of all jobs, where the flow time of a job is the elapsed time between the job’s arrival time and its completion time.

Several scheduling algorithms have been designed and theoretically analyzed for minimizing average flow for parallel jobs [1,2,51,115]. The only one that has been implemented is the **Distributed Random Equi-Partition (DREP)** algorithm [1], which was shown to have good performance theoretically and practically. Given the DREP scheduler, when a job arrives at time $t$, each processor decides to give itself to the new job with probability $1/n_t$ where $n_t$ is the number of unfinished jobs at time $t$. When a job completes, each processor assigned to that job randomly picks an unfinished job and gives itself to the picked job. Agrawal et al.’s implementation [1] shows their scheduler design has strong practical performance, but the performance can significantly increase further if the DREP scheduler is implemented based on AMCilk system, due to a faster core reallocation mechanism. In particular, in Agrawal et al.’s implementation [1], preemption only occurs at steal boundaries (as described in Section 2.3). When a new job arrives, the DREP scheduler allocates certain cores to it which were allocated to other jobs. The cores only stop working on their current jobs and start working on the new jobs when their deque becomes empty and they try to steal. In contrast, AMCilk implements preemptions at frame boundaries, leading to more responsive reallocations.

We compared the frame-boundary preemption of AMCilk and the steal-boundary implementation\(^4\) using the workload distribution from real applications: *bing search workload* and *finance server workload* [88]. For each workload, we vary the average number of jobs arrived per second to generate three different system loads: *low*, *medium*, and *high loads*, where

\(^4\)The implementation in [1] was based on the Cilk Plus runtime system. For a fair comparison with AMCilk, we implemented their steal-boundary preemption in the Cilk-based Cheetah runtime system.
the average system utilizations are approximately 60%, 75%, and 90%. For each setting, we randomly generate 100,000 jobs and record the average flow time. Figure 2.7 shows the results for the Bing search workload and the finance workload. The result indicates that the frame-boundary implementation reduces the average flow times by 60-70% compared to the steal-boundary implementation for the Bing search workload, and the reduction of average flow time is even larger for the finance workload. Figure 2.8 compares both systems by increasing the job arrival rate of the Bing workload. We can see that AMCilk supports the job arrival rate of up to 230 jobs per second without being overloaded (where overloading is indicated by having the average flow time increase unboundedly as time passes) while the frame-boundary implementation supports at most 160 jobs per second — an improvement of 43.8%, indicating that fast preemption can indeed lead to measurable impact on service-level performance for this application.
2.7.2 Elastic parallel real-time scheduling

In cyber-physical systems, such as autonomous vehicles and robotics, sensors periodically collect environment data, and the computing component must process the data to calculate the control demands by the end of the period. Abstractly, such a system contains a set of real-time tasks — each task $\tau_i$ is defined by a tuple $\{C_i, T_i\}$, where $C_i$ is the maximum execution requirement of each job of the task and the task can release jobs with a period (minimum inter-arrival time) of $T_i$. In the simplest scenario, each job has a deadline of $T_i$ — it must complete in $T_i$ time after it is released.\(^5\)

We are interested in parallel real-time tasks where the jobs of real-time tasks may contain internal parallelism — in particular, we focus on elastic real-time tasks \cite{104}. In this model, tasks can change or tolerate a change in their utilizations $U_i = C_i/T_i$ (by changing either $C_i$ or $T_i$ or both) due to the change in the physical system — for instance, if the system enters a less stable state and requires a more expensive or faster control algorithm. The tasks that can increase its utilization are demanding tasks. To satisfy the utilization increase of a demanding task, additional cores must be given to this task (to meet its deadline) by reducing the cores given to the non-demanding tasks. Orr et al. \cite{104} established an elastic scheduling algorithm to calculate the core allocation for all tasks when a demanding task

\(^5\)In the general setting, the deadline may be different from the period.
Figure 2.9: Elastic scheduling with task 1 and task 2. The period of task 1 reduces to 1/3 of its original value at an arbitrary time while the period of task 2 remains the same. We show the deadline miss rate (vertical axis) of the task 1 with different original task 1’s period (horizontal axis).

changes its demand — the details are complex and not relevant to this discussion — the key is that the platform running these applications must be able to reallocate cores among jobs due to external stimuli.

Orr et al. [104] conducted experiments on elastic scheduling using OpenMP; however, they did not have access to a platform with responsive and low-cost core reallocation mechanism while jobs were running. In their system, after the elastic scheduler computes a new allocation, a demanding task gets additional cores only after the currently running jobs of non-demanding tasks have completed. Hence, the delay between demanding more cores and actually getting these cores depends on the other tasks’ period. In contrast, AMCilk allows reallocation at any time during the job’s execution, so the demanding tasks get additional cores much more quickly.

We demonstrate the benefit of fast reallocation on the performance of elastic task systems by running a simple experiment with 2 tasks. Both tasks calculate the 42nd Fibonacci number. Note that the performance in this application scenario depends on the latency of core reallocation, which is evaluated in Section 2.6. We vary Task 1’s period from 10 to 600 milliseconds, while fixing Task 2’s period as 50 milliseconds. For each setting, we run task 1 for 1000 iterations. We randomly select 10 iterations to let task 1’s period be
reduced to 1/3 of its original value and let this change lasts for a random length from 1 to 10 iterations. Figure 2.9 shows task 1’s deadline miss rate — the number of jobs missing their deadlines divided by the total number of jobs. In real-time systems, the goal is to not miss any deadlines. Since AMCilk allows for fast core reallocation regardless of tasks’ period, task 1 never misses any deadlines. In contrast, the deadline miss rate of Orr et al.’s system depends heavily on the periods of the two tasks. As task 1’s period gets smaller (compared to task 2’s period), task 1 misses more deadlines.

The ability of AMCilk to reallocate cores with predictable delays that are independent of job periods is a huge advantage for real-time systems. The goal of real-time system is to provide an a priori guarantee on the timing properties of the system. AMCilk makes it easier to provide such guarantees, since the predictable delays can be incorporated into the a priori timing analysis, while this is harder to do so when the delay depends on the job characteristic.

2.7.3 Adaptive scheduling using parallelism feedback

Fine-grained multithreaded jobs, such as those written using Cilk, can change parallelism as they execute. Thus, statically allocating a fixed number of cores when a job arrives is often inefficient, as the number of cores that can be used by the job depends on whether it is in its low- or high-parallelism phases. Thus, Agrawal et al. [4] proposed an adaptive scheduling strategy that dynamically adapts the number of cores allocated to a job based on an estimate of the job’s dynamic parallelism. Given a job, this scheduler periodically collects the number of steal-cycles and work-cycles and mug-cycles on each processor allotted to the job in runtime and uses this information to decide whether the job needs more processors or whether a program occupies too many processors. The scheduler dynamically adapts the number of processors of the program accordingly. While the details are not relevant,
this scheduler monitors all jobs’ runtime characteristics and periodically changes the core allocation based on these characteristics.

We implemented this adaptive scheduling algorithm using AMCilk. This implementation demonstrates an interesting feature of AMCilk that the previous examples don’t. For DREP, the core allocation changes only when new jobs arrive or when jobs complete. In elastic scheduling, core allocation changes due to external signals. In adaptive scheduling, AMCilk monitors the internal characteristics of the jobs and changes the allocations based on these characteristics.

We evaluate the AMCilk implementation of adaptive scheduling using a simple experiment with 2 jobs that change their parallelism frequently: each job repeatedly switches between high- and low-parallelism phases for 10 times, where the phase of one job is opposite to the other job. In the high-parallelism phase, the job has one large parallel for-loop with 12,800,000 iterations, while in the low-parallelism phase, the job has 4000 small parallel for-loops, each with 100 iterations. AMCilk should capture the switch between the high- and low-parallelism phases of the two jobs quickly and adapt the core allocation of the two jobs responsively. As a result, the running time for both jobs should be smaller than the static partition case for better core utilization.

According to [4], the period of core adaptations should be long enough to amortize the time for core reallocations. Since AMCilk core reallocation is $272\mu$s, we gradually decrease the period from 10ms to 0.5ms and explore the value of the period such that the running time of the two jobs is minimal. Finally, we set the period as 1ms in our experiment.

There is no existing implementation of adaptive scheduling, so we compare against static allocations. We measure the running times of the jobs and normalized them using the running time of 1.65 seconds when each job run individually on all (38) cores. As shown in Figure 2.10, if we do not partition the cores and let the two jobs share the 38 cores, their running times become 2.4 times of their solo running times. If we statically and equally partition the cores,
Figure 2.10: Adaptive scheduling. We run two jobs at the same time with three schedulers (horizontal axis): (1) No partition where each job occupies all cores, and so every core has two jobs running; (2) Eql-partition where each job exclusively uses half of number of cores in system; (3) A-STEAL where each job gets cores on demand according to its runtime parallelism (our scheduler).

i.e., giving each job 19 cores, they complete in 2.32 and 2.34 seconds. Using the AMCilk implementation of adaptive scheduling (with a reasonable setting of parameters), the two jobs complete in 1.86 and 1.87 seconds — 19.8% and 20.1% reductions over equal-partition. This is because our implementation is able to monitor the parallelism of jobs and give fewer cores (about 8 cores) to the job in the low-parallelism phase and more cores (about 30 cores) to the job in the high-parallelism phase. More specifically, when a job changes from low-parallelism to high-parallelism, it experiences 8 times of getting more cores decided by the adaptive scheduling policy, which takes 47.9 milliseconds in total. The functionalities provided by AMCilk makes it possible to implement the adaptive scheduling efficiently for multiple parallel jobs with dynamic parallelism.

2.7.4 Co-scheduling throughput and tail-sensitive jobs

The previous experiments have explored the impact of the fast core-reallocation ability of AMCilk. The final experiment explores the impact of its cache and memory bandwidth partitioning functionality. On many shared platforms, throughput-oriented applications and latency-sensitive applications may be scheduled together — for instance, an interactive application and a streaming application may share the system. While the applications may occupy disjoint cores, they share memory resources such as the last-level cache and memory
bandwidth. Therefore, the latency-sensitive application may have unexpected performance slow down due to interferences.

As explained in Section 2.3, modern hardware often enables cache partitioning and memory bandwidth allocation to control the interference between jobs and improve the quality of service. AMCilk exposes these functionalities to the AMCilk scheduler through an easy-to-use interface allowing the system administrator to manage cores, last-level cache, and memory bandwidth at the same time.

To understand the impact of these functionalities on performance, we run one latency-sensitive application along with a streaming application. The streaming application runs in parallel and repeatedly loads data from memory, modifies it, and stores it back. The latency-sensitive application is an interactive service where clients send requests to the service and the service tries to minimize average flow time (using the DREP scheduler described above in Section 2.7.1). Since we wish to understand the impact of cache and bandwidth, each job in this latency-sensitive application is a sorting job (since sorting is moderately memory intensive) and the size of jobs vary — 95% of the jobs are short (sorting 500,000 numbers) and the other 5% are long (sorting 50,000,000 numbers). We run the latency-sensitive application on cores 2–23 and the streaming application on cores 24–39. In the experiment, the strategy of cache partitioning and bandwidth allocation is simple, since we only want to emphasize the importance of the two functionalities. We allocate a small number of cache columns and a little memory bandwidth to the streaming application while giving a large amount of cache and bandwidth to the latency-sensitive application. The allocation of the cache and memory bandwidth does not change throughout the execution. We compare the average flow time of jobs of the latency-sensitive application between the settings with the cache and memory bandwidth partitioning and the setting without the partitioning.

Figure 2.11 shows the impact of the streaming application on the average flow time of the interactive application. As a baseline, we ran the interactive application alone (without the
Figure 2.11: We co-run streaming and latency-sensitive jobs with four settings (horizontal axis): (1) No CP/MBA where neither cache partitioning nor memory bandwidth allocation is set between the two jobs; (2) CP where only cache partitioning is set; (3) MBA where memory bandwidth allocation is set; (4) CP+MBA where both cache partitioning and memory bandwidth allocation are set. We run the four settings with two workloads (shown in legend): (1) Medium workload where the frequency of job requests is medium; (2) High workload where the request frequency is high.

streaming application) and use its average flow time to normalize the results of different co-running scenarios. When co-running without any cache or memory bandwidth partitioning, the average flow time increases to 5.29 times for medium load and 18.3 times for high load. Only applying cache partitioning already improves the performance significantly, especially for medium load where the impact of the streaming application virtually disappears. In the setting, we only give 3 cache columns to the streaming application while we give 8 columns to the interactive application. Cache partitioning has minimal impact on its performance since the streaming application itself is insensitive to cache size. For high load, we see further improvement as we apply memory bandwidth allocation. In the setting, we give 10% bandwidth to the streaming application and 90% bandwidth to the interactive application. Finally, we get virtually all of the performance back when we use both cache partitioning and memory bandwidth allocation. Noted that reducing memory bandwidth allocation does have an impact on the streaming application – causing about 150% slowdown (reducing the processing speed from 1855.64 to 724.14 Mflop/sec).

This experiment shows that it is crucial to use cache partitioning and memory bandwidth allocation if we wish to get good performance in multiprogrammed environments. AMCilk
allows system administrators to easily access these functionalities using an easy-to-use interface.

### 2.7.5 Subscription of runtime information

We now show that the subscription of runtime information is efficiently implemented in AMCilk. Since the overhead incurred by the collection of runtime information is negligible, and the information publishing does not block the critical path of the job execution, we focus on presenting the timing precision to see whether the information in the subscription represent the runtime internals of AMCilk accurately.

We developed a program which has two phases — a high parallelism phase and a low parallelism phase — and the program alternates between the two. The high-parallelism phase keeps all processors busy and almost all the processors are idle during the low parallelism phase. Recall that when a worker runs out of work, it tries to steal work from others. When parallelism is low, workers have a hard time finding work and therefore, they repeatedly steal. Therefore, we expect many steal attempts during the low parallelism phase and very few steal attempts during the high-parallelism phase.
We used this benchmark to measure the accuracy of the information published by the AMCilk runtime system. We set the information to be published every 0.1 seconds. Figure 2.12 shows the number of steal cycles on processor 2 (as an example), collected by an external process (on the same machine) via the subscription of AMCilk runtime information. The red zone is the time when the program is in the high-parallelism phase. The white zone is the timing when the program is in the low-parallelism phase. The black line denotes the number of steal cycles viewed by the external process via the subscription. We see that when the program enters the low-parallelism phase (white), the number of steal cycles dramatically increases, and the subscriber is able to see this change quite rapidly. Similarly, when the program enters the high-parallelism phase, the number of steal cycles drops to 0 and the subscriber is able to see this change rapidly after it happens. This experiment provides evidence that the subscriber can rapidly get an accurate view of the runtime information allowing it to make appropriate scheduling decisions.

2.8 Related works

2.8.1 Dynamic core reallocation between parallel jobs.

The primary feature of AMCilk is the fast and low-overhead core reallocation mechanism between parallel jobs. There has been intensive prior work over a decade ago. Some prior works [54, 105, 110] consider dynamic core reallocation between parallel jobs in threading primitives. In these works, the parallel job is implemented with lightweight threads. AMCilk is different from these works since the parallel jobs running in AMCilk are written in fork-join primitives with language support. As a result, the problem and the design of core reallocation in AMCilk are different from those prior works. Moreover, AMCilk is ready to use since existing legacy written in Cilk language can run in AMCilk without any modification.
Similar to AMCilk, there are prior works [36, 37, 64, 118] that consider dynamic core reallocation between the applications written in fork-join primitives with language support. However, the parallel job in these prior works executes in a work-sharing model, where each worker iteratively takes a chunk of work from a centralized queue and processes it. AMCilk is different from those prior works, where the parallel job executes in the work-stealing model. In the work-stealing model, the work is assigned to workers in a decentralized manner. Thus, the problem and the solution of core reallocation in AMCilk are different from those prior works.

There are prior works [1, 25, 30, 89] that consider the dynamical core reallocation between the applications in the work-stealing model. However, in these works, putting a worker to sleep is either achieved at the steal boundary or lacking in description. In Cilk-AP [119], putting a worker to sleep is achieved at the frame boundary. However, Cilk-AP handles the leftover work of a sleeping worker by work stealing. On the other hand, AMCilk handles the leftover work by mugging, which maintains the number of deques to steal to be equal to the number of cores of the job. Thus, AMCilk keeps the theoretical bound of the work-stealing scheduler [8]. Moreover, none of those systems supports customizable scheduler nor cache and memory bandwidth management. In addition, those systems do not support resource occupancy control or admission control, or subscription of runtime information. On the other hand, AMCilk supports all these functionalities, which makes AMCilk be an efficient runtime system for multiprogrammed parallel applications in shared environments.

2.8.2 Scheduling multiprogrammed parallel workloads.

There is extensive theoretical work on scheduling multiprogrammed parallel workloads in various situations and for different metrics. For example, Edmonds et al. [50] designed a dynamic equipartitioning strategy, which provides a variety of theoretical advantages. For online systems, researchers have considered minimizing average flow time [1, 2, 51, 115],
maximum flow time \([5, 109]\), makespan \([4]\) and tail-latency \([63, 75, 88]\). Various real-time scheduling policies for parallel jobs also require support for multiple jobs running in a single machine \([73, 84, 86, 90, 100, 104]\). AMCilk is specifically designed to support the above types of scheduling algorithms in an efficient manner.

Several platforms were implemented for various real-world applications, from interactive cloud services \([1, 63, 72, 88]\) to parallel real-time systems \([84, 91, 104]\). Among them, some platforms can only run the jobs of the specifically modified application program \([63, 72, 84]\); some create one runtime system for each program and can only support their particular scheduling algorithms \([91, 104]\); the others use one runtime for multiple jobs, but do not support responsive core reallocation nor the different scheduling algorithms \([1, 88]\). AMCilk is an efficient platform that meets the requirements of real-world applications and various scheduling algorithms.

In addition, there is intensive prior work on co-scheduling the mix of parallel applications for various performance goals \([18, 31, 41, 58, 82, 94, 95, 108]\), which relies on dynamic core reallocation between the applications and exposure of application runtime. We believe that the fast core reallocation and the support of the shared environment of AMCilk make those co-scheduling designs efficient in implementations.
Chapter 3

Static schedules with sub-linear overhead for fault-tolerant transmission

3.1 Introduction

Motivation  Shared communication medium such as a shared bus, wireless network, CAN, are commonly used in myriad systems. On safety-critical systems and/or on embedded devices, it is often beneficial to generate a static schedule — where time for sending each message is predetermined — since these enable both simpler implementation and a deterministic upper bound on message delivery time. For instance, static schedules are used on time-triggered systems, like TTA [80], where timeline is divided into a series of identical and exclusive time slots, and messages are sent in pre-allocated slots.

A transmission algorithm on (especially wireless) channels for messages must tolerate noise or transmission errors in the medium — such noise leads to message failure or corrupts the message. When a message encounters noise, it must be sent repeatedly until it is correctly transmitted. Therefore, a static schedule for message transmission must reserve sufficient slots for each message to tolerate medium errors in transmission.

Problem  In this chapter, we consider the scenario where we have $n$ messages to be transmitted (possibly by different senders) on a shared medium where time is divided into slots.
We assume that the channel has up to \( f \) medium errors — that is, up to \( f \) time slots experience noise. In addition, if more than one message is transmitted in the same time slot, a collision occurs and neither message will be sent. If a time slot experiences a medium error or a collision, the message(s) transmitted during the slot fail. We assume that the sender knows when a message fails and must retransmit.\(^6\) In static schedules, each sender has a list of slots. It transmits its message on each successive slot assigned to it until the message succeeds and then stops transmitting the message. We assume that senders are not aware of and cannot react to other runtime conditions (such as which other messages have successfully transmitted or how many medium errors or collisions have occurred so far).

The goal is to generate a fault-tolerant static schedule which minimizes makespan while guaranteeing that all \( n \) messages are transmitted successfully for all possible settings of up to \( f \) slots experiencing medium errors. This strong correctness requirement means that we do not make any assumptions about the distribution of medium errors and the schedule must guarantee that all messages successfully transmit even if an adversary that knows the static schedule generates (upto \( f \)) medium errors. We are interested in finding fault-tolerant schedules with length as short as possible. Even with \( f = 0 \), all schedules must have length at least \( n \) (and in fact with \( f = 0 \) achieving length \( n \) is trivial). We thus focus on how much we need to add to the schedule length in order to achieve fault tolerance as \( f \) increases. Specifically, if the schedule has length \( n + T \), we say that the fault-tolerance overhead, or simply the overhead, of this schedule is \( T \). The goal is to minimize this overhead.

It is clear that each message must be assigned to at least \( f + 1 \) slots under this adversarial model — if any message \( m \) is assigned to fewer slots, the \( f \) medium errors can occur on all the slots where \( m \) is assigned and \( m \) will fail to transmit. The simplest schedule is to send each message for \( f + 1 \) times which uses \( n + n \times f \) slots and the overhead is \( n \times f \). This

\(^6\)This is a fairly standard assumption and is often implemented using some sort of acknowledgment mechanism, like the use of a short "acknowledgement" frame in AirTight [24].
is the best we can do if we assign at most one message to each slot in the static schedule. One can take advantage of the fact that the sender knows when a message has transmitted successfully and will not transmit such a message again to generate shorter schedules. The state-of-the-art schedule [3] uses \( n + n \times f/2 \) slots and the overhead is \( n \times f/2 \), which is better than the naive one. These overheads increase linearly along with \( n \) — as \( n \) increases, the overhead can be quite large even for a small \( f \).

**Contribution** In this chapter, we present techniques for designing static schedules with sublinear overheads for the model described above. In particular, we provide two mappings, one with overhead \( O(f^2 \log^2 n) \) and the other with overhead \( O(f \log f \log n) \). While the problem (fixed number of messages and medium errors) and the objective (minimize overhead) considered may not directly apply to real systems, this simple model allows us to investigate whether sublinear overheads are even possible using static schedules and we answer this question in the affirmative. We believe that these insights can be applied to more complex and realistic problems as well.

The main results of this chapter are:

- (Section 3.2.2.) We give an algorithm for generating fault-tolerant schedules with overhead \( O(f^2 \log^2 n) \) and hence total length \( n + O(f^2 \log^2 n) \). The overhead depends only logarithmically on \( n \) and is asymptotically better than the state of the art algorithm [3] whenever \( f = o(n/\log^2 n) \), and can be significantly better when \( f \) is much smaller than \( n \). This mapping can be generated efficiently.

- (Section 3.4.) We prove the existence of fault-tolerant schedules with overhead \( O(f \log f \log n) \) and hence the static schedule length is \( n + O(f \log f \log n) \). The algorithm is randomized, and generates a fault-tolerant mapping with a non-zero probability. Although the algorithm for generating the schedule is efficient, verifying that the schedule is fault tolerant takes time exponential in \( f \). Therefore, this may not be a viable approach for
large $f$, but can be practical for small $f$ since the schedule need be generated just once offline.

- (Section 3.5.) While our methods are asymptotically better than prior methods, they do have constant factors. We analytically compute schedule lengths of our methods and compare against the method proposed by Agrawal et al. [3] for concrete values of $n$ and $f$ to verify that under certain conditions, sublinear mappings are shorter.

3.2 Background

This section defines identity mapping and collision-free mappings as building blocks of fault-tolerant schedules and summarizes existing works of collision-free mappings. Then, the section provides concrete examples of collision-free mappings and their constructions.

3.2.1 Identity mapping and collision-free mappings

We build static schedules by concatenating multiple subschedules that we call mappings. A mapping is defined over a contiguous interval of slots, and it specifies which messages transmit in each slot. There is no technical difference between the terms, but we use the word mapping to refer to components of the schedule and schedule to refer to the whole. We assume throughout that messages have distinct IDs from $1, 2, \ldots, n$ and use these to define mappings. The simplest mapping is the identity mapping.

**Definition 1** (Identity mapping — $I(n)$). The identity mapping for $n$ messages spans $n$ slots. Message $i$ is mapped to slot $i$.

There is no apparent advantage in assigning multiple messages to a slot the first time a message is scheduled. All fault-tolerant schedules thus begin with the identity mapping of length $n$. We refer to what follows the identity mapping as the retransmission mapping.
or retransmission schedule. The retransmission mapping is the interesting part of fault-tolerant schedules and the overhead due to fault-tolerance is the length of the retransmission mapping. For example, the dual mapping [3] has retransmission mapping length of \( nf/2 \).

We will rely on the notion of collision-free mapping to generate sublinear length retransmission mappings, defined as following:

**Definition 2** (Collision-free mapping — \( M(n, m, \alpha) \)). A collision-free mapping \( M(n, m, \alpha) \) allots \( n \) messages to a series of slots. For any subset of \( m \) messages, the mapping guarantees to send at least \( \alpha m \) messages without collision.

When applying collision-free mappings to retransmissions, the notion of collision-free mappings is a property that (lower) bounds the number of collision-free slots with respect to subsets of remaining messages. Collision-free slots are essential to retransmissions. In collision-free slots, remaining messages are guaranteed to send without collision. If no medium error happens to a collision-free slot, the remaining message assigned to this slot will be successfully transmitted.

There are existing mappings which are conceptually equivalent to collision-free mappings [12, 34, 45, 68, 78]. Collision-free mappings have been called \( \alpha \)-good mappings and selectors in literature. There is extensive work on selectors in particular in the context of distributed computing. Some existing collision-free mappings are constructive. Bender et al. [12] and Kautz et al. [78] developed collision-free mappings for \( \alpha = 1 \) with length \( O(m^2 \log^2 n) \). Indyk [68] constructed a collision-free mapping for \( \alpha = 3/4 \) with length \( O(m \text{ polylog } n) \). Chlebus et al. [34] gave an explicit collision-free mapping with length \( O(\frac{m^2}{m-\alpha m+1} \text{ polylog } n) \) for any \( \alpha \in (0, 1] \). Furthermore, many works explored the existence of small collision-free mappings. Dyachkov et al. [49] indicated that the size of collision-free mapping for \( \alpha = 1 \) can be \( O(m^2 \log n) \). Bender et al. [12] presented that the mapping length can be \( O(m \log n) \) for \( \alpha = 1/2 \). Bonis et al. [45] showed that the size of collision-free mapping can be \( O(\frac{m^2}{m-\alpha m+1} \log \frac{n}{m}) \), for any \( \alpha \in (0, 1] \).
3.2.2 Examples of collision-free mappings and their constructions

In this subsection, we introduce examples of collision-free mappings. These results were generated for $\alpha$-good mappings [12], but we are going to present them as collision free mappings. Here $X = \{1, 2, \ldots, n\}$ is the set of $n$ message IDs and $m \leq n$ is the size of the subsets. As given by the definition of collision-free mapping, in a collision-free mapping $M(n, m, \alpha)$: for all subsets $S \subset X$ of size $m$, at least an $\alpha$ fraction of the messages in $S$ are mapped to a slot that does not contain any other messages from $S$. This means that if we have $m$ remaining messages to transmit, at least $\alpha m$ messages are, effectively, collision free, no matter which $m$ messages remain.

To clarify this concept, consider some examples. The first example of a collision-free mapping $M(4, 2, 1)$.

\[ \langle \{X_1, X_2\}, \{X_2, X_3\}, \{X_3, X_4\}, \{X_1, X_4\} \rangle, \]

where $X_i$ represents the message $i$ and "\{\}" represents a slot — this schedule contains 4 slots, and each slot contains 2 messages. Since $m = 2$, the mapping must ensure that for each pair of messages $x, y$, there exists a slot that contains $x$ and not $y$ and vice versa. To see why this is useful, consider that $X_3$ and $X_4$ have transmitted before this schedule, but $X_1$ and $X_2$ remain. If there are no errors during the transmission of this schedule, then $X_2$ will successfully transmit during slot 2 and $X_1$ will successfully transmit during slot 4. This is true regardless of which two messages remain at the beginning of the mapping. Note that this mapping is not particularly short — we could also use the schedule where each message transmits in its own slot, which is trivially collision-free for $\alpha = 1$; however, for larger values of $n$ and $m$, non-trivial 1-good schedules where messages share slots can be shorter, as we will soon see.
Now let us consider an example where $\alpha = 1/2$. We have $n = 4$ messages $X_1, X_2, X_3, X_4$ and $m = 2$. It turns out that the schedule

$$\langle \{X_1, X_2\} \{X_2, X_3\} \rangle$$

is collision-free for $\alpha = 1/2$. Note that for a collision-free mapping for $\alpha = 1/2$, we have fewer constraints. Now for each pair of messages $x, y$, either $x$ has to have a slot which does not contain $y$ or $y$ has to have a slot that does not contain $x$. This allows us to generate this counter-intuitive mapping where $X_4$ is never transmitted. For instance, consider $X_1$ and $X_4$ — since the first slot contains $X_1$, but not $X_4$, it satisfies the condition. We can check similarly for all pairs.

The advantage of collision-free mappings is realized when the size of the mapping is relatively small. Bender et al. [12] give the following methods for creating a collision-free mapping for $\alpha = 1$ and a collision-free mapping for $\alpha = 1/2$ with sizes $O(m^2 \log^2 n)$ and $O(m \log n)$, respectively. Importantly, for both methods if $m \ll n$, then the number of slots is far below $n$. Both of their constructions involve concatenating several simple mappings that they call collections. A collection is a mapping in which each message is assigned to exactly one slot.

We first describe the modulo mapping which is a small collision-free mapping.

**Definition 3** (Modulo mapping). Let $n$ be the number of messages and let $1 \leq m \leq n$ be a parameter of the mapping. Let $C = m \lceil \log n / \log(m \log n) \rceil$, and let $p_1, p_2, \ldots, p_C$ denote the $C$ smallest prime numbers greater than $m \log n$. The modulo mapping consists of the concatenation of $C$ submappings called collections. Collection $i$ spans $p_i$ slots numbered consecutively from 0, and all messages $x \in X$ with $x \equiv j \pmod{p_i}$ are assigned to slot $j$ in collection $i$. 

50
Bender et al. [12] prove the following bound on the length of this mapping — it follows from the density of primes, allowing all primes considered to have value \( O(m \log n) \).

Lemma 1 (From [12]). The modulo mapping with parameter \( n \) and \( m \) is a collision-free mapping \( M(n, m, 1) \), and it spans a total of \( O(m^2 \log^2 n) \) slots.

Without doing the full proof, here is the intuition for why the modulo mapping is collision-free for \( \alpha = 1 \). Consider two integers \( j \) and \( k \) where \( j \neq k \). In the modulo mapping, \( j \) and \( k \) collide in collections \( i \) iff \( j \equiv k \pmod{p_i} \), where \( p_i \) is the prime used for collection \( i \). Consider the first \( C_0 = \log n / \log(m \log n) \) collections. For \( j \) and \( k \) to collide in all these collections, we must have \( |j - k| \geq \prod_{i=1}^{C_0} p_i \) for all primes \( p_i \) used in these collections. Since all the primes \( p_i \) are large (\( \geq m \log n \)), this necessarily implies \( |j - k| \geq \prod_{i=1}^{C_0} p_i \geq (m \log n)^{C_0} = n \). Therefore, if our messages are numbered from 1 to \( n \), the difference between the two numbers is at most \( n - 1 \), and no two messages can collide in all \( C_0 \) collections. This observation can be generalized: In particular, since we have a total of \( C = mC_0 = m \log n / \log(m \log n) \) collections, we can show that there exists a collection such that \( j \) will not collide with any other subset of \( m - 1 \) messages implying that this is a collision-free mapping for \( \alpha = 1 \).

Now consider the size of the mapping. The size of the mapping is the \( \sum_{i=1}^{C} p_i \) where \( p_i \) is the \( i \)th prime number larger than \( m \log n \). Since primes are relatively dense, any interval of size \( N \log N \) contains approximately \( N \) primes for sufficiently large \( N \). Therefore, if we consider an interval \( (m \log n, km \log n) \) for some constant \( k \), we will find \( C \) primes. Thus, \( p_i = O(m \log n) \) for all \( p_i \), implying that the mapping size is \( O(m^2 \log^2 n) \).

Definition 4 (Ball-bin method). Let \( n \) be the number of message IDs and let \( 1 \leq m \leq n \) be a parameter of the mapping. The mapping is also parameterized by constant \( d, c > 0 \). The ball-bin mapping is the concatenation of \( \lceil d \log n \rceil \) collections, each spanning \( \lceil cm \rceil \) slots. In each collection, each message is assigned to one slot independently and uniformly at random.
By construction, the number of slots used by the ball-bin method is bounded. But because ball-bin method is randomized, the resulting mapping is not always collision-free. Bender et al. [12] show that the ball-bin method is likely to produce a collision-free mapping for $\alpha = 1/2$, as stated by the following lemma. Note that the method simply returns a mapping, without any indication of whether the mappings is collision-free for $\alpha = 1/2$ or not. Thus, we need to verify the quality of the mapping before incorporating it into the fault-tolerant schedule, repeating the random generation in the unlucky event that the mapping is not collision-free for $\alpha = 1/2$.

Lemma 2 (From [12]). There exist constant settings of parameters $c > 0$, $d > 0$ and constant probability $p > 0$ such that: for any setting of parameter $m$, with probability at least $p$ the ball-bin method produces a mapping that is a collision-free mapping $M(n, m, 1/2)$. Moreover, the mapping produced always uses $O(m \log n)$ slots.

3.3 Generating fault-tolerant mapping with divisible collision-free mappings

This section uses collision-free mapping $M(n, m, 1)$ to produce a fault-tolerant schedule. As mentioned in Section 3.2, a fault-tolerant schedule consisting of an identity mapping followed by a retransmission mapping must ensure that all messages are transmitted even if there are up to $f$ errors during the transmission regardless of which slots have the errors. We will now design retransmission mappings using collision-free mapping $M(n, m, 1)$. We first argue that a single arbitrary collision-free mapping $M(n, m, 1)$ as the retransmission mapping does not guarantee fault-tolerance. We then define an additional condition on collision-free mapping $M(n, m, 1)$, called divisibility, and argue that the identity mapping followed by a divisible 1-good mapping is sufficient to guarantee fault tolerance. We then argue that the modulo mapping [12] (Section 3.2, Definition 3) is in fact divisible. We
therefore achieve a retransmission mapping with length $O(f^2 \log^2 n)$ for a total schedule length of $n + O(f^2 \log^2 n)$.

First, consider a simple scenario to see why collision-free mapping $M(n, m, 1)$ is useful — say all $f$ errors occur during the identity mapping and none during the retransmission mapping.

**Observation 1.** If no error occurs during the retransmission mapping, then using a collision-free mapping $M(n, f, 1)$ (by setting $m = f$) as the retransmission mapping is sufficient to transmit all messages.

To see why this is true, recall that, by the definition of collision-free mapping, for any subset $S$ of $f$ messages (and therefore the subset of messages which experienced medium errors during the identity mapping), there will be at least one slot for each of these messages in the collision-free mapping $M(n, f, 1)$ where they do not collide with other messages in $S$. Therefore, all $f$ messages will transmit without collisions.

However, in the general case, when there may be errors during the retransmission phase, an arbitrary collision-free mapping for $\alpha = 1$ is not sufficient. To see this, observe that the identity mapping is a collision-free mapping for $\alpha = 1$. However, if we have an identity mapping, followed by another identity mapping for retransmission, we cannot guarantee fault tolerance. More generally, in order to guarantee fault tolerance, each message must be assigned to at least $f$ slots in the retransmission mapping ($f + 1$ slots when we include the initial identity mapping); otherwise, every slot containing the message may have a medium error and the message cannot transmit. Therefore, just relying on a mapping being collision-free for $\alpha = 1$ is not sufficient for fault tolerance. One option is to use multiple collision-free mappings for $\alpha = 1$, but that would increase the overhead (length of the retransmission phase). Instead, we argue that the modulo mapping has a much stronger property that makes a single modulo mapping sufficient.
3.3.1 Divisible collision-free mapping

We define divisible mappings specifically for the case of collision-free mappings for \( \alpha = 1 \). It turns out this definition generalizes for any \( \alpha \leq 1 \). However, since we only use this property for collision-free mappings for \( \alpha = 1 \), we only define it for \( \alpha = 1 \).

**Definition 5** (Divisible collision-free mapping \( M(n, m, 1) \)). Given \( n \) messages, a collision-free mapping \( M(n, m, 1) \) is divisible if it can be decomposed into \( m \) phases each containing a contiguous set of slots such that the following condition holds. For every positive integer \( k \leq m \), every grouping of \( k \) consecutive phases constitutes a collision-free mapping \( M(n, k, 1) \). In other words, for every size-\( k \) subset \( S \) of messages and any group of \( k \) consecutive phases, there is a slot in the phases to which \( x \) and no other message in \( S \) is assigned.

Divisibility is a counter-intuitive property. The idea is that a divisible mapping is made up of \( m \) phases and as we compose these phases, we get divisible mappings for larger and larger subsets. Each phase, in itself, is a collision-free mapping \( M(n, 1, 1) \) — that is, if we only have 1 message left, then one phase is sufficient to transmit this one message. Such mappings are trivial — a single slot with all \( n \) messages mapped to it is 1-good for all subsets of size 1. But then the definition gets stricter. It says that if we look at all the slots from any \( k = 2 \) contiguous phases together, they form a collision-free mapping \( M(n, 2, 1) \) — that is, for any pair of messages, there are slots within these phases where these messages do not conflict with each other. We continue in this vein for all values of \( k \).

Note that an arbitrary collision-free mapping for \( \alpha = 1 \) is not necessarily divisible. For example, the following identity mapping is collision-free for \( n = 4 \) and \( m = 4 \) and \( \alpha = 1 \):

\[
\langle \{X_1\}\{X_2\}\{X_3\}\{X_4\} \rangle ,
\]
since each message has an exclusive slot. However, the mapping is not divisible since we cannot split the mapping into smaller parts such that each part is collision-free for a smaller $m$.

\subsection{Divisible collision-free mapping $M(n, f, 1)$ guarantee fault-tolerance}

It is not at all clear how one would design such mappings and we will show that the modulo mapping is divisible in Section 3.3.3. This subsection argues that a divisible collision-free mapping $M(n, f, 1)$ provides fault-tolerance.

Before getting to the main result, we start by formalizing a generalization of Observation 1.

\textbf{Lemma 3.} Suppose that there are $\eta$ unsent messages, and a collision-free mapping $M(n, \eta, 1)$ is performed. Let $\phi$ be the number of medium errors suffered during the mapping. Then at most $\min(\eta, \phi)$ unsent messages remain at the completion of the mapping.

\textit{Proof.} Let $U$, with $\eta = |U|$, be the set of unsent messages at the beginning of the mapping. Clearly once a message has been sent, it cannot become unsent, so the number of unsent messages that remain at the end is at most $\eta$.

We focus here on showing that the number of unsent messages that remain at the end is also at most the number of errors $\phi$. Consider any particular message $u \in U$. By the definition of collision-free mapping, there exists at least one slot $s_u$ to which $u$ is assigned and to which no other message in $U$ is assigned, which we call a \textbf{collision-free slot for $u$}. Let $s_u$ be any one such collision-free slot for $u$. If no medium error occurs in slot $s_u$, then $u$ successfully transmits. If a medium error does occur in slot $s_u$, then we say that $u$ is blocked. Since the chosen collision-free slots are, by definition, distinct for each unsent message (i.e., $s_u \neq s_v$ for $u \neq v \in U$), each medium error blocks at most one message in $U$.  

55
Because only blocked messages may remain at the end, the number of messages that remain is thus at most the number of medium errors $\phi$.

**Theorem 4.** Consider a schedule comprising the identity mapping followed by a divisible collision-free mapping $M(n, f, 1)$. If at most $f$ medium errors occur in total, then all messages are successfully sent by the conclusion of the schedule.

**Proof.** Let $e_0 \leq f$ be the number of medium errors that occur during the identity mapping. The number of unsent messages that remain at the end of the identity mapping is $e_0$.

We now partition the divisible collision-free mapping $M(n, f, 1)$ into some number of rounds starting the numbering from round 1, and we call the identity mapping round 0. This partition is designed in an online fashion as follows. Let $n_i$ be the number of messages that remain at the end of round $i - 1$. If $n_i = 0$, then the process terminates. Otherwise, round $i$ corresponds to the next $n_i$ phases of the divisible collision-free mapping $M(n, f, 1)$. To be well-defined, we must ensure that enough phases remain in the divisible mapping, which we revisit in the next paragraph. Notice that by definition of divisibility, the round $i$ is a collision-free mapping $M(n, n_i, 1)$. Let $e_i$ be the number of medium errors occurring during the round. Then by Lemma 3, the number of unsent messages that remain at the end of round $i$ is at most $n_i \leq e_i$. (In fact, we have $n_i \leq \min(n_i, e_i)$, but we only need to leverage the weaker claim here.)

It remains to show that the process successfully terminates (i.e., we reach $n_i = 0$ for some $i$) before we run out of phases in the divisible mapping. Suppose for the sake of contradiction that the process does not successfully terminate, meaning that there exists some earliest round $r$ for which not enough phases remain in the divisible mapping. There are $f$ phases in the divisible mapping, so by the nontermination assumption we have that $\sum_{i=1}^{r} n_i > f$. Applying the fact that $n_i \leq e_{i-1}$ as proved in the previous paragraph, we have $f < \sum_{i=1}^{r} n_i \leq \sum_{i=1}^{r} e_{i-1} = \sum_{i=0}^{r-1} e_i$, i.e., the total number of medium errors incurred through rounds 0, 1, $\ldots$, $r - 1$ is strictly more than $f$. This contradicts the assumption that
a total of at most $f$ medium errors occur, and thus the assumption that the process does not terminate is false.

### 3.3.3 Modulo mapping is divisible

Now that we know a divisible mapping is useful, we will argue that the Modulo mapping described in Section 3.2 from Bender et al. [12] is an instance of a collision-free mapping for $\alpha = 1$. The first part of the proof is very similar to the argument in Bender et al. [12] where they prove that it is a collision-free mapping for $\alpha = 1$. We will then extend this argument to prove that the same property also implies divisibility.

**Theorem 5.** The Modulo mapping (definition 3) is a divisible collision-free mapping for $\alpha = 1$.

**Proof.** Consider a pair of messages $j$ and $\ell$, $j \neq \ell$, and suppose they collide in collection $p_i$. Then it must be the case that $j \equiv \ell \pmod{p_i}$, i.e., their difference is a multiple of $p_i$. Now consider multiple collections. In particular, let $C_0 = \lceil \log n / \log(m \log n) \rceil$. We argue that in total, $j$ and $\ell$ cannot collide in $C_0$ (or more) collections. Suppose for the sake of contradiction that $j$ and $\ell$ collide in at least $C_0$ collections. Then their difference must be the product of at least $C_0$ different primes, all with value at least $m \log n$. Thus $|j - \ell| \geq (m \log n)^{C_0} \geq n$. But the maximum difference between IDs is $n - 1$, which gives us a contradiction. Thus, $j$ and $\ell$ must collide fewer than $C_0$ times.

Bender et al. [12] used the same observation to prove the mapping collision-free for $\alpha = 1$. We use this property to prove divisibility. Let $C_0$ successive collections constitute a phase. Since the mapping contains $C = m[\log n / \log(m \log n)]$ collections, there are $C/C_0 = m$ phases. Given an integer $k$ ($k \leq m$), consider any size-$k$ subset $S$ of messages, and consider any particular message $j$ in this set. By the logic above, each other message collides with $j$ in fewer than $C_0$ collections. If we sum across the collisions with all $k - 1$ other messages in $S$, in total $j$ must experience fewer than $(k - 1)C_0$ collisions with other messages from...
Thus, for any $k$ contiguous phases and hence $kC_0$ contiguous collections, message $j$ must have at least one collection in which it does not experience a collision. Therefore, the slots of any $k$ phases (each phase contains $C_0$ collections) make a collision-free mapping $M(n, k, 1)$ for all $k < m$, which is the definition of divisibility.

Putting everything together, the retransmission phase consists of a modulo mapping, generated according to Definition 3 with parameter $m = f$. By Lemma 1, for $m = f$ this mapping has length $O(f^2 \log^2 n)$. By Theorem 5, this mapping is a divisible collision-free mapping $M(n, f, 1)$. By Theorem 4, a divisible collision-free mapping $M(n, f, 1)$ is sufficient. We thus conclude with the following:

**Corollary 6.** Consider the schedule consisting of the identity mapping followed by the modulo mapping for subset size $f$. If at most $f$ medium errors occur, then all messages are successfully sent by the conclusion of the schedule. Moreover, the retransmission length (i.e., the overhead) is $O(f^2 \log^2 n)$.

### 3.4 Generating fault-tolerant mapping with reducible collision-free mappings

This section applies multiple collision-free mappings to produce a shorter fault-tolerant schedule. In order to apply the method described in this section, we will define a property of collision-free mappings, namely **reducibility**. Once we define this property, we will show that the algorithm given here works for any reducible collision-free mapping, including the collision-free mapping for $\alpha = 1$. The main advantage is achieved when $\alpha < 1$, which allows for a shorter mapping length and hence better overall schedule. As a simple extension of Bender et al. [12], we show that there exists a reducible collision-free mapping $M(n, m, 1/2)$ with length $\Theta(m \log n)$. Coupled with the main algorithm in this section for producing a
good retransmission mapping from collision-free mappings, this gives us a retransmission mapping length of $O(f \log f \log n)$ for an overall schedule length of $n + O(f \log f \log n)$.

This bound is asymptotically better than the schedule produced using the modulo mapping. It should be noted, however, that we do not know of any efficient algorithm for producing the reducible collision-free mapping $M(n, m, 1/2)$ of length $\Theta(m \log n)$. Rather the simple random ball-bin method is likely to generate this reducible mapping, but verifying that the resulting mapping is actually a reducible collision-free mapping $M(n, m, 1/2)$ is computationally expensive. Nevertheless, a mapping for each set of parameters need only be generated once — therefore, we could use it to get small schedules.

### 3.4.1 Reducible collision-free mapping

Recall that a collision-free mapping $M(n, m, \alpha)$ only guarantees that for subsets of size $m$, at least $\alpha m$ of the message are assigned to time slots without collision. What if there are only $k \ll m$ unsent messages still trying to send? It turns out that the collision-free property as defined does not directly ensure progress.

To understand the issue, consider a specific subset $S$ of size $|S| = k$, $m = 2k$ and $\alpha = 1/2$. One valid (but not very effective) collision-free mapping for $\alpha = 1/2$ would be a mapping that assigns every element of $S$ to the same single slot, and all other $n - k$ elements to their own individual slots. This mapping meets the definition of collision-free mapping $M(n, 2k, 1/2)$, as all such subsets must include at least $k$ elements assigned to their own slots. However, if we only have $k$ unsent messages corresponding exactly to the subset $S$, then none of the messages may be collision free.

For example, consider the following collision-free mapping $M(8, 8, 1/2)$.

$$\langle \{X_1\}|\{X_2\}|\{X_3\}|\{X_4\}|\{X_5, X_6, X_7, X_8\} \rangle .$$
In the mapping, 4 messages — $X_1, X_2, X_3, X_4$ have their own slots. However, this mapping is not a collision-free mapping $M(8, 4, 1/2)$ since all the messages in the subset $X_5, X_6, X_7, X_8$ collide with each other. It is also not collision-free (for $\alpha = 1/2$) for many other values of $m$.

To overcome this issue, we extend the property to also be reducible, which simply means that the mapping needs to be collision-free for at most $m$, not just exactly equal to $m$. The preceding example is not reducible.

**Definition 6 (Reducible collision-free mapping $M(n, m, \alpha)$).** Given $n$ messages, a **reducible collision-free mapping** $M(n, m, \alpha)$ is a collision-free mapping $M(n, k, \alpha)$ for any $k \leq m$.

This property guarantees that if there are $k \leq f$ unsent messages, a reducible collision-free mapping $M(n, f, \alpha)$ will send at least $\alpha k$ messages without collisions (in the absence of medium errors). We note here that collision-free mappings for $\alpha = 1$ are always reducible.

**Lemma 7.** Any collision-free mapping for $\alpha = 1$ is reducible.

**Proof.** Consider a collision-free mapping $M(n, m, 1)$. We must show that it is also a collision-free mapping $M(n, k, 1)$ for $k < m$. In particular, for a subset $S$ with $|S| = k$, we must show that every element in the subset has a collision free slot. Let $S'$ be any subset with $|S' | = m$ and $S \subset S'$, i.e., augment $S$ by adding any $m - k$ elements. By definition of 1-good, every element in $S'$ has a collision-free slot, and hence so does every element in $S$.

However, since we already have a retransmission schedule which contains only one collision-free mapping for $\alpha = 1$, we will focus here on reducible mappings for $\alpha < 1$ in order to get shorter retransmission mappings overall.

### 3.4.2 Achieving fault-tolerance by repetitions

This section gives two methods that integrate multiple reducible collision-free mappings to generate a retransmission mapping. We will argue that the schedule consisting of an identity
mapping followed by this retransmission mapping guarantees fault tolerance. The overall schedule length depends on the sizes of the collision-free mappings. Section 3.4 proves that small reducible collision-free mappings for $\alpha = 1/2$ exist and analyzes the overall schedule length.

We begin by defining a geometric schedule (shown in Algorithm 1) — so called since the subset size for subsequent mappings decreases geometrically. For this schedule, we will prove that it achieves fault tolerance.

Algorithm 1 Constructing a retransmission mapping from reducible collision-free mappings using the geometric method

1: $\triangleright I(n)$ is the identity mapping for $n$ messages
2: $\triangleright M(n, m, \alpha)$ is a reducible collision-free mapping
3: function GEOMETRIC-INTEGRATION($n, f, \alpha$)
4: schedule = $I(n)$
5: for $i \leftarrow 0$ to $\log f$ do
6: append $\lceil((1/\alpha)(1 + 2^{i+1})\rceil$ copies of $M(n, f/2^i, \alpha)$ to schedule
7: end for
8: return schedule
9: end function

The schedule comprises the identity mapping followed by the retransmission mapping, which contains several reducible collision-free mappings for different values of $m$. Consider an instance on $n$ messages, $f$ total medium errors, and parameter $\alpha$. The retransmission mapping comprises roughly $\log f$ phases (logarithms are base 2), where each phase corresponds to an iteration of the main loop in Algorithm 1. The $i$th phase consists of $\lceil((1/\alpha)(1 + 2^{i+1})\rceil$ (roughly $\Theta(2^i)$) repetitions of a reducible collision-free mapping $M(n, f/2^i, \alpha)$. In each phase, the number of mappings doubles, but the length of each repetition of the mapping halves.

To prove correctness, the main goal is to prove the following invariant: at the start of phase $i$, there are at most $f/2^i$ unsent messages. This invariant is necessary — if more unsent messages exist, then a collision-free mapping $M(n, f/2^i, \alpha)$ has no guarantees on collisions.
for subsets of size larger than $m$. Moreover, if the invariant holds, then there are no unsent messages at the end of all the iterations. This invariant, or more precisely the inductive step of the invariant, is captured by the following lemma.

**Lemma 8.** Suppose that the $i$th phase (iteration $i$) begins with at most $f/2^i$ unsent messages. Suppose also that at most $f$ medium errors occur during the phase. Then at the end of the phase, there are at most $f/2^{i+1}$ unsent messages.

**Proof.** Suppose, for the sake of contradiction, that a phase fails, i.e., ends with at least $f/2^{i+1}$ unsent messages. Then there must be at least that many unsent messages at the start of every repetition of the collision-free mapping. We shall show that with only $f$ medium errors, keeping this many messages alive would require more than $f$ medium errors, hence generating the contradiction.

In the phase, each mapping is a collision-free mapping $M(n, m, \alpha)$ for $m \leq f/2^i$. Since by assumption there are at least $f/2^{i+1}$ unsent messages throughout, in each mapping repetition there must be at least $\alpha f/2^{i+1}$ messages that are assigned to at least one collision-free slots. For each message, consider just one such collision free slot. These messages would successfully transmit as long their slot does not suffer a medium error. That is to say, every such collision-free slot that does not incur a medium error results in a message being successfully transmitted. Note that this number of collision-free messages depends only on the assumption that there are still a lot of unsent messages — it does not depend on how medium errors are assigned in previous repetitions.

Summing across all repetitions, the total number of collision-free slots is at least $\alpha f/2^{i+1}$ times the number of repetitions, or at least $(1/\alpha)(1 + 2^{i+1}) \cdot \alpha f/2^{i+1} = f + f/2^{i+1}$ collision-free assignments. If $\hat{f}$ medium errors occur, the number of successful transmission is thus at least $f + f/2^{i+1} - \hat{f}$. To conclude the proof, we observe that the phase begins with at most $f/2^i$ unsent messages by assumption, and hence there must be fewer than $f/2^{i+1}$ successful transmissions to keep the surviving number of unsent messages above $f/2^{i+1}$. Thus we have
\[ f + f/2^{i+1} - \hat{f} < f/2^{i+1} \text{ or } \hat{f} > f, \] which contradicts the assumption that there are only \( f \) medium errors in the phase.

Note that the preceding lemma is stronger than necessary in that it allows for \( f \) medium errors in each phase, whereas there are only \( f \) medium errors in total.

**Theorem 9.** Consider the schedule generated by the algorithm described in Algorithm 1 for a particular settings of \( n, f, \) and \( \alpha > 0 \) for which the specified mappings exist. If at most \( f \) medium errors occur, then all \( n \) messages successfully transmit by the end of the schedule.

**Proof.** Via induction over phases, we can show that at the start (and end) of iteration \( i \), there are at most \( f/2^i \) (and \( f/2^{i+1} \)) unsent messages. For the base case, at the beginning of the retransmission mapping (start of phase \( i = 0 \)) there are at most \( f = f/2^0 \) unsent messages. For each subsequent phase, Lemma 8 shows the inductive step. Thus, after \( \log(f) \) phases, there is \( \leq 1/2 \) unsent message i.e., no unsent messages.

For the types of mappings we know, there are not many interesting values of \( \alpha \)—just \( \alpha = 1 \), which gives an easy constructive mapping, and \( \alpha = 1/2 \), which gives a much smaller mapping. Pushing \( \alpha \) smaller may result in the collision-free mapping being smaller, but this mapping-size improvement is offset by increasing the number of repetitions with \( (1/\alpha) \).

Nevertheless, it is possible to improve some constant factors in the integration algorithm itself, albeit at a cost of complicating the proof. We thus chose to lead with the algorithm with the clearer proof in this section. Algorithm 2 gives an alternative harmonic algorithm — so called since the subset size decreases harmonically in each iteration — specifically for \( \alpha = 1/2 \), which decreases the overall number of repetitions at each subset size. This is the version we use in our experiments. The main loop structure is different in this algorithm: there are now \( f \) phases, where phase \( i \) now corresponds to subset size \( m = f/i \), but the advantage is that each phase now has just a constant number of repetitions. Though quite different at first glance, it’s worth noting that the algorithms and proofs are conceptually...
similar: as with the geometric sequence, the harmonic sequence includes \( \Theta(2^i) \) numbers that are between \( 1/2^i \) and \( 1/2^{i-1} \).

### Algorithm 2

A modified (harmonic) algorithm for constructing a retransmission mapping using reducible collision-free mappings for \( \alpha = 1/2 \).

1:▷ \( I(n) \) is the identity mapping for \( n \) messages
2:▷ \( M(n, m, 1/2) \) is a reducible collision-free mapping for \( \alpha = 1/2 \)
3: function HARMONIC-INTEGRATION\((n, f)\)
4: schedule = \( I(n) \)
5: for \( i \leftarrow 1 \) to \( f \) do
6: append 3 copies of \( M(n, f/i, 1/2) \) to schedule
7: end for
8: return schedule
9: end function

The key to proving correctness of this harmonic mapping is to show that iteration \( i \) begins with at most \( f/i \), and ends with at most \( f/(i+1) \), unsent messages. This property is analogous to Lemma 8, and it is necessary and sufficient for similar reasons. That is, the reducible mapping \( M(n, m, 1/2) \) applied during iteration \( i \) is a collision-free mapping only for \( m \leq f/i \), so to argue anything about the number of collision-free messages we need to have at most \( f/i \) unsent messages. Moreover, at the end, having \( f/(f+1) < 1 \) unsent messages implies that there are no unsent messages.

Our proof of this key invariant follows a somewhat different structure from the analogous argument for the geometric version. To understand the setup, it is best to consider what would happen if there were no medium error — each invocation of a collision-free mapping for \( \alpha = 1/2 \) (and sufficiently large \( m \)) would reduce the number of unsent messages by 1/2, which would mean 1/8 per iteration. Thus, in the absence of medium errors, the number of unsent messages at the start of iteration \( i \) would be \( f/8^{i-1} \), which is far lower than the harmonic guarantee we set out to achieve. The point is that to keep the number of unsent
messages as high as \( f/i \) in every iteration, the number of medium errors must already be quite large.

Our analysis thus somewhat decouples the iteration number from the number of messages remaining. Our goal is to characterize each phase according to how many steps it takes in the harmonic sequence. For example, going from \( f/2 \) unsent messages down to \( f/5 \) takes three steps (the step from \( f/2 \) to \( f/3 \), the step from \( f/3 \) to \( f/4 \), and the step from \( f/4 \) to \( f/5 \)) in the harmonic sequence. To this end, we define the **harmonic rank** over positive integers \( 1, 2, \ldots, f \), denoted by \( hrank(x) \), to be the value \( j = hrank(x) \) such that \( f/(j+1) < x \leq f/j \).

For example, for \( f = 16 \), we have \( hrank(10) = 1 \), \( hrank(7) = 2 \), and \( hrank(4) = 4 \).

As before, we call iteration \( i \) of the main loop the **\( i \)th phase**, which now comprises three reducible collision-free mappings for \( \alpha = 1/2 \) and \( m = f/i \). As noted above, without medium errors the harmonic rank should increase by more than 1 in each phase. If there are some medium errors, the harmonic rank may increase by less. The following lemma flips this dependence around — the change to the harmonic rank gives us a bound on the number of medium errors.

**Lemma 10.** Let \( u \) and \( u' \) denote the number of unsent messages at the start and end of a particular phase, respectively. Suppose that \( u \leq f \) and that \( u' \geq 1 \), let \( h = hrank(u) \), and let \( h' = hrank(u') \). Suppose also that the phase number \( i \) satisfies \( i \leq h \), or equivalently \( u \leq f/i \). Then the number of medium errors that occurred during the phase is strictly greater than \( 2f/(h' + 1) - f/(2h) \).

**Proof.** Because by assumption \( u \leq f/i \), each copy of the reducible mapping during this phase is a collision-free mapping \( M(n, m, 1/2) \) for all \( m \geq u \). It follows that throughout the phase, for each copy of the mapping, at least half of the remaining unsent messages are each assigned to at least one collision-free slot.

We now consider each of the copies of the reducible mapping. Let \( u_k \), for \( k \in \{1, 2, 3\} \) denote the number of unsent messages that remain after performing the \( k \)th repetition of
the reducible mapping in this phase. Similarly, let \( x_k \) denote the number of medium errors occurring during copy \( k \). Our goal is to prove that \( x_1 + x_2 + x_3 > \frac{2f}{(h' + 1)} - \frac{f}{2h} \). To start, observe that

\[
\begin{align*}
  u_1 &\leq u/2 + x_1, \\
u_2 &\leq u_1/2 + x_2, \quad \text{and} \\
u_3 &\leq u_2/2 + x_3.
\end{align*}
\]

These inequalities follow from the fact that each unsent message assigned to a collision-free slot only remains unsent if it suffers a medium error.

Combining the three inequalities, we have

\[
u_1 + u_2 + u_3 \leq x_1 + x_2 + x_3 + u/2 + u_1/2 + u_2/2,
\]

or

\[
x_1 + x_2 + x_3 \geq u_3 + u_1/2 + u_2/2 - u/2
\]

Since the number of unsent messages is monotonically decreasing, we have \( u_i \geq u' > \frac{f}{(h' + 1)} \). We also have \( u \leq \frac{f}{h} \). Substituting these back into our inequality yields

\[
x_1 + x_2 + x_3 \geq 2u' - u/2 > \frac{2f}{h' + 1} - \frac{f}{2h}.
\]

Observe that if the harmonic rank increases by a lot during the phase (for instance, if \( h' \geq 4 \times h \)), then Lemma 10 does not say anything about the number of medium errors — in this case \( 2f/(h' + 1) - f/(2h) \) is negative. On the other hand, a slower increase to the harmonic rank can only occur in the presence of medium errors.
The next lemma shall be useful to bound the number of errors for a sequence of \( k \) phases starting with harmonic rank \( h_0 \) and ending with harmonic rank \( h_k \). Roughly speaking, the implication is that the lowest number of errors necessary\(^7\) to keep \( x \) messages alive across multiple phases is realized by allowing all but \( x \) messages to succeed immediately in the first phase, and then using all medium errors to maintain \( x \) messages in the subsequent phases.

Lemma 11. Consider a monotonically increasing sequence of \( 1 \leq h_0 \leq h_1 \leq \cdots \leq h_k \). Then we have

\[
\sum_{i=1}^{k} \left( \frac{2}{1 + h_i} - \frac{1}{2h_{i-1}} \right) \geq \frac{k + 1}{1 + h_k} - \frac{1}{2h_0}.
\]

Proof. We start by splitting the sum into

\[
\sum_{i=1}^{k} \left( \frac{2}{1 + h_i} - \frac{1}{2h_{i-1}} \right) = \sum_{i=1}^{k} \frac{1}{1 + h_i} + \sum_{i=1}^{k} \left( \frac{1}{1 + h_i} - \frac{1}{2h_{i-1}} \right) \geq \sum_{i=1}^{k} \frac{1}{1 + h_k} + \sum_{i=1}^{k} \left( \frac{1}{1 + h_i} - \frac{1}{2h_{i-1}} \right) = \frac{k}{1 + h_k} + \sum_{i=1}^{k} \left( \frac{1}{1 + h_i} - \frac{1}{2h_{i-1}} \right)
\]

To complete the proof, we simply observe that for \( h_i \geq 1 \), we have \( 1/(1 + h_i) \geq 1/(2h_i) \), or equivalently \( 1/(1 + h_i) - 1/(2h_i) \geq 0 \). We can thus cancel all consecutive intermediate terms in the sum (the sum telescopes), and we are left with

\[
\sum_{i=1}^{k} \left( \frac{1}{1 + h_i} - \frac{1}{2h_{i-1}} \right) \geq \frac{1}{1 + h_k} - \frac{1}{2h_0}.
\]

\(^7\)This statement is with respect to the lower bound of Lemma 10, which allows a negative number of errors. That lower bound is thus looser than the reality, where the number of medium errors cannot be negative.
We are now ready to prove the main correctness invariant: that each phase \( i \) starts and ends with at most \( f/i \) and \( f/(i+1) \) unsent messages, respectively, and hence the mapping is good for the number of remaining unsent messages.

**Lemma 12.** Let \( u_{i-1} \) and \( u_i \) denote the number of unsent messages at the start and end, respectively, of phase \( i \). If at most a total of \( f \) medium errors occur (over the whole schedule), then for all \( i \), \( 0 \leq i \leq f \), we have \( u_i \leq f/(i+1) \).

**Proof.** Suppose for the sake of contradiction that there exists an \( i \) such that \( u_i > f/(i + 1) \). Let \( x = \min\{ i \mid u_i > f/(i + 1) \} \) be the first such phase number.

Note that \( u_i \) is always an integer, and \( f/(i + 1) > 0 \), so \( u_i \geq 1 \) for all \( i \leq x \). Moreover, we also have \( u_i \leq f \) due to Lemma 3, which applies because the identity mapping is 1-good and suffers at most \( f \) medium errors. Thus for \( 0 \leq i \leq x \), we have \( 1 \leq u_i \leq f \), and the harmonic rank is well-defined for all \( i \) in this range. We thus let \( h_i = h_{\text{rank}}(u_i) \) for \( 0 \leq i \leq x \).

By choice of \( x \), we have \( u_{i-1} \leq f/i \) for all \( i \leq x \). Thus the conditions of Lemma 10 are met for all \( i \leq x \). (We map each \( u_{i-1} \) and \( u_i \) to \( u \) and \( u' \), respectively, in the statement of that lemma.) Following from Lemma 10, the total number of medium errors during phases 1 through \( x \) inclusive is strictly greater than

\[
\sum_{i=1}^{x} \left( \frac{2f}{1+h_i} - \frac{f}{2h_{i-1}} \right) = f \sum_{i=1}^{x} \left( \frac{2}{1+h_i} - \frac{1}{2h_{i-1}} \right)
\]

Applying Lemma 11,

\[
f \sum_{i=1}^{x} \left( \frac{2}{1+h_i} - \frac{1}{2h_{i-1}} \right) \geq f \left( \frac{x + 1}{1 + h_x} - \frac{1}{2h_0} \right)
\]

If \( u_0 \) messages survive the identity mapping, then the number of medium errors during the identity mapping is \( u_0 \) (Lemma 3). We have \( u_0 > f/(h_0 + 1) \) by definition of the harmonic rank. Adding all the errors together, the total number of medium errors is strictly greater
than
\[ f\left(\frac{x+1}{1+h_x} - \frac{1}{2h_0}\right) + \frac{f}{1+h_0} \geq \frac{f(x+1)}{1+h_x} , \]
which follows because \( h_0 \geq 1 \).

By choice of \( x \), \( u_x > \frac{f}{x+1} \) and hence \( h_x = hrank(u_x) < x+1 \), or \( h_x \leq x \). We thus have a strictly more than
\[ \frac{f(x+1)}{1+h_x} \geq \frac{f(x+1)}{1+x} = f \]
medium errors. This generates a contradiction as the number of medium errors is assumed to be at most \( f \) and cannot be strictly more than \( f \).

\[ \square \]

**Theorem 13.** Consider the schedule generated by the algorithm described in Algorithm 2 for a particular settings of \( n \) and \( f \). If at most \( f \) medium errors occur, then all \( n \) messages successfully transmit by the end of the schedule.

**Proof.** Lemma 12 implies that \( u_f \leq \frac{f}{f+1} < 1 \) — the number of unsent messages at the end of phase \( f \) is 0. Thus, all messages must successfully transmit by the schedule.

\[ \square \]

### 3.4.3 Applying the collision-free mapping for \( \alpha = 1/2 \) of small size

We have shown how to build fault-tolerant schedules via integrating many reducible collision-free mappings. In this subsection, we present a collision-free mapping for \( \alpha = 1/2 \) that is reducible and hence applies to achieve fault tolerance.

The mapping is same as the 1/2-good mapping described by Bender et al. [12]. We are not claiming the mapping itself as a contribution, but we present some of the analysis for completeness. Bender et al. [12] use the probabilistic method on random assignments to argue that the collision-free mapping for \( \alpha = 1/2 \) of modest size exist. That is, they show that a random process for generating a mapping has some chance of producing the mapping. Thus, one can obtain the mapping by repeating the random process and testing the result.
Bender et al. [12] omit several of the proofs from their paper, however, so we include those details here. Moreover, we show that the same mapping is also reducible.

The following lemma is similar to one given by Bender et al. [12], but no proof is provided in their paper. This balls-in-bins lemma is the main crux of the mapping; the full mapping comprises $\Theta(\log n)$ uniformly random (balls-in-bins) assignments of this form.

**Lemma 14.** Let $18m$ denote the number of bins and $k \leq m$ be the number of balls. There exists a constant $\delta \in (0,1)$ such that: if the $k$ balls are thrown uniformly at random into the $18m$ bins, then we have $\Pr(\text{fewer than } k/2 \text{ bins have exactly one ball}) < \delta^k$.

**Proof.** Let $cm$ be the number of bins (where we shall derive the $c \geq 18$ in the proof). Consider tossing $k$ balls, $k \leq m$, into those bins. Let $\rho$ denote the number of bins containing at least 1 ball. Let $x$ denote the number of bins having at least 2 balls. We want to bound the number $y = \rho - x$ of bins that contain exactly 1 ball. We have $(\rho - x) \times 1 + x \times 2 \leq k$, which leads $x \leq k - \rho$. Thus, we have

$$y \geq 2\rho - k$$

We want $y \geq 2\rho - k \geq k/2$. Observe that $2\rho - k \geq k/2$ is equivalent to $\rho \geq 3k/4$. It follows that if $\rho \geq 3k/4$, then $y \geq 2\rho - k \geq k/2$. Thus, we have $\Pr[y \geq k/2] \geq \Pr[\rho \geq 3k/4]$ which leads

$$\Pr[y < \frac{k}{2}] \leq \Pr[\rho < \frac{3k}{4}]$$

Now, we want to show $\Pr[\rho < 3k/4] < \delta^k$. For any subset of $[1, i]$ bins where $i \leq m$, we have the probability $p$ that all balls land in those bins is $p \leq \left(\frac{i}{cm}\right)^k$. The number of subsets of $[1, i]$ bins is $\binom{cm+i-1}{i} < e^{i\left(\frac{i}{(c+1)m}\right)^i}$. Apply a union bound over all the subsets, we have
Pr[\rho < i] < Pr[\rho \leq i] \leq e^i \left(\frac{i}{(c+1)m}\right)^{-i} \left(\frac{i}{cm}\right)^k. In particular, for i = 3k/4, we have

\[ Pr \left[ \rho < \frac{3k}{4} \right] < \left( \left( \frac{3c^3}{4c} \right) \left( \frac{c+1}{c} \right)^3 \right)^{\frac{i}{4}} \]  

(3.1)

Select \( \delta = \left( \left( \frac{3c^3}{4c} \right) \left( \frac{c+1}{c} \right)^3 \right)^{\frac{i}{4}} \), and notice that for \( c \geq 18 \), we have \( \delta < 1 \). Above all, we have

\[ Pr \left[ y < \frac{k}{2} \right] \leq Pr \left[ \rho < \frac{3k}{4} \right] < \delta^k. \]  

(3.2)

We now show that the balls-in-bins method has a non-zero chance of producing a reducible collision-free mapping for \( \alpha = 1/2 \). Therefore, the reducible collision-free mapping for \( \alpha = 1/2 \) must exist.

**Theorem 15.** The balls-in-bins method (definition 4) produces a reducible collision-free mapping \( M(n, m, 1/2) \) with at least a constant probability.

**Proof.** The method generates \( \Theta(\log n) \) independent collections. For each collection, we randomly (uniformly) allocate \( n \) messages to \( \Theta(m) \) slots. Due to Lemma 14, given \( k \) (\( k \leq m \)) messages, the probability \( P_1 \) that fewer than \( k/2 \) slots that each slot has exactly one of the \( k \) messages is \( P_1 < \delta^k \) for some \( \delta \in (0, 1) \). Since we have \( \theta(\log n) \) independent collections. Thus, we have the probability \( P_2 \) that every collection has less than \( k/2 \) slots that each slot has exactly one of the \( k \) messages is \( P_2 = P_1^{d \log n} < (\delta^k)^{d \log n} \) for \( d \geq 1 \). Since there are \( (\frac{n+k-1}{k}) \leq (\frac{2n}{k}) \leq (\frac{2en}{k})^k \leq e^{k^2 \log n} \) ways to pick \([1, k]\) messages out of \( n \) messages. We apply a union bound over all possible subsets that contain \([1, k]\) messages, we have the probability \( P_3 \) that the mapping is a reducible collision-free mapping \( M(n, k, 1/2) \) is \( P_3 < e^{k^2 \log n} \delta^{dk \log n} < 1 \) for sufficiently large \( d \). Thus, it is possible to get a reducible collision-free mapping \( M(n, m, 1/2) \) via the probability method.
Now that we have shown that the reducible collision-free mappings for $\alpha = 1/2$ of length $O(m \log n)$ exists, we can now compute the total length of the the retransmission mapping created using the method in Algorithm 1 or Algorithm 2.

**Lemma 16.** Assuming we generate reducible collision-free mappings for $\alpha = 1/2$ using balls-in-bins method and then use the above methods to generate a retransmission mapping, the size of the retransmission mapping in either case is $O(f \log f \log n)$

**Proof.** For the geometric method shown in Algorithm 1, we have $\log f$ phases. Phase $i$ contains $2^{i+1}$ copies of the reducible collision-free mapping $M(n, f/2^i, 1/2)$. Therefore, the length of phase $i$ is $O(2^{i+1} f/2^i \log n) = O(f \log n)$. Therefore, the total length is $O(f \log f \log n)$.

For the harmonic method shown in Algorithm 2, we have $f$ phases and each phase contains 3 copies of the reducible collision-free mapping $M(n, f/i, 1/2)$. Therefore, the length of each phase is $O(f/i \log n)$. If we add over all $i$, we get $O(\sum_{i=1}^{f} f/i \log n) = O(f \log f \log n)$. \qed

Putting everything together, by Theorem 15, reducible collision-free mappings for $\alpha = 1/2$ of length $\Theta(m \log n)$ exists. By Theorem 9, we can compose these mappings into a retransmission mapping which guarantees fault tolerance for $f$ errors and the length of this retransmission schedule is $O(f \log f \log n)$ (Lemma 16 for the total schedule length of $n + O(f \log f \log n)$).

### 3.5 Evaluation

This section describes our analytical evaluation of the overhead due to fault tolerance using various methods for generating the retransmission mapping. The methods described in this chapter are have asymptotically better overheads than previous methods — sublinear in $n$ instead of $O(nf)$. Until $n$ or $f$ grow large, however, the constants involved are of practical
concern. We now try to understand at what values of $n$ and $f$ these new methods begin to become effective.

We compare the lengths of the retransmission schedules. Our baseline is the dual mapping [3], henceforth denoted as DUAL. In DUAL, every message appears in $f$ slots and each slot contains at most 2 messages. Let $|M|$ denote the length of mapping $M$. From [3], we have $|\text{DUAL}| = nf/2$. We first compare DUAL with our approach based on the modulo mapping of Section 3.2.2, denoted by DIV for “divisible.” We then compare DUAL against our second approach in Section 3.4. We use RDC, for “reducible,” to denote the retransmission schedule that results from generating collision-free mappings with the ball-bin method and the harmonic integration.

3.5.1 Divisible collision-free mapping for $\alpha = 1$

The modulo mapping lends itself to a simple algorithm for constructing a divisible collision-free mapping for $\alpha = 1$. We build the DIV mapping following the specification in Definition 3 — that is, we simply identify the smallest $f \log n / \log f$ primes and then use these primes to construct each collection by applying the appropriate modulo operation to each message ID.

We begin by presenting the exact length of DIV for different $n$’s and $f$’s. We observe that DIV outperforms the baseline DUAL when the ratio $n/f$ is sufficiently large, as expected by the asymptotic bound. We then determine the minimum value of $n/f$ for which DIV outperforms the baseline.

Figure 3.1 shows the comparison of DIV against the baseline from different perspectives. For Figure 3.1a, we try different values of $n$ ranging between 1000 and 10000 in steps of 1000 and values of $f$ between 20 and 100 in steps of 20. For each pair of $n$ and $f$, we calculate the length of DIV divided by the length of the baseline (lower is better, and the value below 1 indicates DIV outperforms the baseline). To highlight some numbers, when $n = 5000$ and $f = 40$, DIV’s length is roughly $2/3$ the length of DUAL. When $n = 10000$
Figure 3.1: Numerical results of DIV. Figure 1a presents the ratio of DIV length to DUAL length for $n \in [1000, 10000]$ and $f \in [20, 100]$; Figure 1b keep $f$ constant and shows the sub-linear trend of DIV length as $n$ increases, comparing to the linear trend of DUAL; Figure 1c presents the trend of $r_1, r_2, r_3, r_4$ as $n$ increases, where $r_i$ denotes the minimal $n/f$ such that the length of DIV is shorter than DUAL length over $i$. DUAL mapping is the baseline.

and $f = 40$ or at $n = 5000$ and $f = 20$, the ratio is about $1/3$. Because the asymptotic ratio between the two retransmission mapping lengths is $O(|\text{DIV}|/|\text{DUAL}|) = O(f^2 \log^2 n/(nf)) = O((f/n) \log^2 n)$, these experiments confirm the expected behavior that the ratio between their lengths is roughly inversely proportional to $n/f$.

Figure 3.1b keeps $f$ constant and shows the dependence on $n$. Again, we see that as $n$ increases, DIV gains advantage — the size of the mapping increases very slowly with increasing $n$ compared to the baseline. Thus, in low medium-error environments, DIV scales very well as $n$ increases. In contrast, for fixed $f$ DUAL’s length is proportional to $n$.

Figure 3.1c tries to understand the performance in a slightly different way. In the above evaluation, we see that when $n/f$ is sufficient large, DIV outperforms the baseline, as is to be expected from the bound. In particular, the asymptotics indicate that DIV (with $-\text{DIV} -$ = $O(f^2 \log^2 n)$) should outperform DUAL (with $-\text{DUAL} -$ = $nf/2$) when $f = O(n/\log^2 n)$ or equivalently when $n/f = \Omega(\log^2 n)$ for sufficiently large $c$. We next try to understand how large $n/f$ needs to be to overcome the true constants in that $\Omega(\log^2 n)$ term. Let $r_1$ be the minimal $n/f$ s.t. $|\text{DIV}| < |\text{DUAL}|$. As $n$ increases, $r_1$ also increases, but slowly due to the dependence is on $\log^2 n$. 

74
Let \( r_i \) denote the minimal \( n/f \) s.t \(|\text{DIV}| < |\text{DUAL}|/i\). We call \( i \) the outperformance level and want to understand at what values of \( n \), we start to see a particular level of advantage from using a divisible mapping rather than the baseline. In the evaluation, we evenly select \( n \) in \([2000, 50000]\) (the step is 200). We calculate \( r_i \) for each \( n \) and the results are in Figure 3.1c. As \( n \) gets bigger, \( r_i \) increases with \( n \) as expected but very slowly. Therefore, above a certain value of \( n \) it might always be useful to use divisible mappings unless \( f \) is very large. We can also see that for \( n \in [2000, 50000] \), \( r_1 \leq 120 \), \( r_2 \leq 200 \), \( r_3 \leq 300 \), \( r_4 \leq 400 \). It indicates that DIV can be significantly better than the baseline in a low medium-error environments.

### 3.5.2 Reducible collision-free mapping for \( \alpha = 1/2 \)

We now consider the retransmission schedule RDC, which is built using the ball-bin method coupled with the harmonic technique in Section 3.4. These experiments are more complicated as the ball-bin method is not guaranteed to generate a reducible collision-free mapping for \( \alpha = 1/2 \). Instead, we must randomly construct candidate mappings and check if they are collision-free and reducible for \( \alpha = 1/2 \). This check is prohibitively expensive for large values of \( n \) and \( f \) since we must check for collisions for all \( f \)-size subsets of \( n \) messages and there are about \( n^f \) of them.

There are some interesting questions to be asked about reducible collision-free mappings for \( \alpha = 1/2 \), however, including the specific setting of constant parameters \( c \) and \( d \) in Definition 4. In this section, we will first explore which values of these constants for small values of \( n \) and \( f \) lead to producing mappings that are actually collision-free and reducible for \( \alpha = 1/2 \). We then compare the (potential) mapping length for various values of \( c \) and \( d \) to the baseline.

**Determining \( c \) and \( d \) in practice**  Recall that RDC is built up with reducible collision-free mappings for \( \alpha = 1/2 \) which generated by the ball-bin method. We see in Lemma 14
that a constant of $c = 18$ is sufficient. The constant $d$ appears in Theorem 15, and it is also large. (We do not solve for $d$, merely arguing that a sufficiently large $d$ exists.) The mapping size depends on $c \times d$. While the theoretical value of this quantity is large (since they are based on what can be proven using simple mathematical techniques, in practice, we can often get a valid mapping for smaller values.

For small values of $n$ and $f$, we can try small values of $c$ and $d$ and try to generate 1/2-good mappings by trying the balls-into-bin method several times. Table 3.1 shows the values of $c$ and $d$ that were sufficient to find valid mappings for $n = 200, 400$ and $f = 2, 3, 4$. Note, in the table, that relatively small values of $c$ and $d$ are sufficient to get valid mappings — $c \times d = 2$ is sufficient for all values that we tested. This indicates that although our mathematically proven upper bounds have high constants, the effective values of the true

<table>
<thead>
<tr>
<th>$n$</th>
<th>$f$</th>
<th>$c$</th>
<th>$d$</th>
<th>RDC</th>
<th>DUAL</th>
</tr>
</thead>
<tbody>
<tr>
<td>200</td>
<td>2</td>
<td>1</td>
<td>2</td>
<td>97</td>
<td>200</td>
</tr>
<tr>
<td>200</td>
<td>3</td>
<td>1</td>
<td>2</td>
<td>169</td>
<td>300</td>
</tr>
<tr>
<td>200</td>
<td>4</td>
<td>1</td>
<td>2</td>
<td>385</td>
<td>400</td>
</tr>
<tr>
<td>400</td>
<td>2</td>
<td>1</td>
<td>2</td>
<td>109</td>
<td>400</td>
</tr>
<tr>
<td>400</td>
<td>3</td>
<td>1</td>
<td>2</td>
<td>190</td>
<td>600</td>
</tr>
<tr>
<td>400</td>
<td>4</td>
<td>1</td>
<td>2</td>
<td>433</td>
<td>800</td>
</tr>
</tbody>
</table>

Table 3.1: Comparison of number of slots between RDC and DUAL (baseline) for small $n$’s and $f$’s.
constants may not be that large. The table also shows the length of the smallest correct schedule we were able to find using reducible mappings. We see that even for such small values of $f$ and $n$, RDC generally uses fewer slots than DUAL for its retransmission mapping, often by a significant amount. This advantage is only likely to increase for large $n$ and $f$ if we can find an efficient way to generate these mappings.

**Projection of performance of RDC for large $n$ and $f$** It is difficult to properly evaluate RDC for large $n$ and $f$ since checking whether a randomly generated mapping is correct is prohibitively expensive. Thus, we can only speculate here.

Figure 3.2 shows the comparison of length of the retransmission schedule constructed using reducible mappings to the baseline for various guesses for the correct value of $c \times d$. We saw earlier that small values of $c \times d$ is often sufficient, but we do not know if that is also true when $n$ and $f$ are large. Therefore, we show the comparison for various values.

Again, we use the values of $r_1, r_2, r_3$ and $r_4$ as defined for divisible mappings earlier. We see that different from the $r_i$ of DIV, the $r_i$ of RDC decreases to 1 when $n$ gets sufficiently large — implying that for large enough $n$, all values of $f \leq n$ benefit from using reducible mappings. This is also easy to see from the theoretical results — the length of both the baseline and reducible mapping method increase (almost) linearly with $f$, while divisible 1-good mapping had a quadratic dependence on $f$. While the divisible mapping was only good for small $f$ and (relatively) large $n$, this method is better than (at least as good as) the baseline for all values of $f$ for large enough $n$.

However, the correct values of $c$ and $d$ matters since constants matter in practice. For example, let us focus on $r_3$ in Figure 3.2 — $r_3 = 1$ shows the value of $n$ at which RDC is better than DUAL by a factor of 3. Roughly, when $c \times d = 2$, $r_3 = 1$ if $n \geq 4000$. When $c \times d = 4$, $r_3 = 1$ if $n \geq 8000$. When $c \times d = 18$, $r_3 = 1$ if $n \geq 60000$. Therefore, the smaller the value of $c \times d$, the smaller the value of $n$ at which we should prefer to use RDC over DUAL for all values of $f$. However, even when $c \times d$ is large, there still exists a sufficient
large \( n \) such that RDC always outperforms the baseline. It is also worth noting that RDC is significantly better than DIV even for large \( c \times d \) for reasonable values of \( n \).

3.6 Application implications

We have devised sublinear mappings for generating static schedules to transmit \( n \) messages over a shared medium. We now consider benefits of these mappings under diverse application scenarios. In particular, we will consider how the sublinear mappings can lead to better throughput under better fault models, how they can improve efficiency by putting messages from different applications together, and how they can be applied to mixed-criticality scenarios.

We will generally compare the dual mapping [3] (the best previously known result to the best of our knowledge) to the divisible collision-free mapping \( M(n, f, 1) \) that is generated by the modulo method. The reducible collision-free mapping for \( \alpha = 1/2 \) produced by the ball-bin method takes time exponential in \( f \) to generate and is less useful in practice, so we won’t consider it in this section — however, similar analysis applies even more strongly to that mapping.

3.6.1 Analysis under average fault model

Recall that the fault model in this chapter assumes that the total number of medium errors is always \( f \) no matter the schedule length. This is somewhat unrealistic — generally, we will encounter more faults for longer schedules.

We now consider the \textit{average fault model} where there are \( k \) faults per slot (on average) — so if our schedule takes \( X \) time to complete, it must be able to tolerate \( kX \) faults.\(^8\) Say we have a static mapping with the total schedule length \( L(n, f) \) given \( n \) messages and \( f \) total faults. In our average fault model, this \( f \) (the total number of faults over the schedule)

\(^8\)This model is also a simplification, but reasonable enough to make the observation we wish to make here.
is not known in advance and will depend on the length of the schedule itself. We must pick an $f$ such that $f \geq k \times L(n, f)$ when creating a schedule. We can use this model to compare the feasibility and the efficiency of the dual mapping and sublinear mappings.

Say $L_1(n, f)$ is the schedule length of the dual mapping. In this case, we must select $f$ such that $f \geq k \times L_1(n, f)$ and we have $L_1(n, f) = n + n \times f/2$. Thus, we have

$$L_1(n, f) \geq 2n/(2 - nk)$$  \hspace{1cm} (3.3)

Therefore, for any $n$, the fault rate must be $k \leq 2/n$ in order to find a feasible schedule.

Now to compare to the sublinear mapping: let $L_2(n, f)$ denote the schedule length of divisible collision-free mapping $M(n, f, 1)$. Again, given $n$ and $k$, we have $L_2(n, f) = n + f^2 \log^2 n$ and $f \geq k \times L_2$, then we have

$$k^2 \log^2 n L_2^2 - L_2 + n \leq 0$$  \hspace{1cm} (3.4)

This equation is only satisfiable if we have $1 - 4k^2 n \log^2 n \geq 0$. Therefore, given $n$, we must have $k \leq 1/(2\sqrt{n} \log n)$. For large values of $n$, sublinear mappings can support higher fault rates than linear mappings.

Apart from feasibility, we can also look at efficiency in this model. To do so, we define the concept of throughput.

**Definition 7** (Throughput). Given a static schedule of length $n + T$ that sends $n$ messages under at most $f$ medium errors, the throughput of the schedule is $n/(n + T)$.

The throughput of a schedule measures the average number of messages sent per slot over time. Clearly, the throughput of the naive schedule (sending each messages for $f + 1$ times) is $1/(f + 1)$. The throughput of the dual mapping is $2/(f + 2)$ which is almost twice as large as the naive schedule for large $f$. However, the throughput of dual mapping is at
most $1/2$ for $f \geq 2$ (maximized when $f = 2$). It means that if we want the dual mapping to tolerate at least 2 errors, for every 2 time slots, the schedule wastes at least 1 time slot, no matter how many messages are sent by the schedule.

Specifically, let $t_1(n, k)$ be the throughput of the dual mapping given $n$ messages and a fault rate of $k$. Since $t_1(n, k) = n/L_1(n, f)$, from equation 3.3, we have $t_1(n, k) \leq 1 - n \times k/2$. To find the throughput for divisible collision-free mapping $M(n, f, 1)$, we want to find the minimum value of $L_2(n, f)$ such that the equation 3.4 is satisfied (assuming feasibility). Solving this and computing $t_2$ gives us $t_2(n, k) = (1 + \sqrt{1 - 4k^2n \log^2 n})/2$. Therefore, for large values of $n$, the sublinear nature of the divisible mapping can allow for much higher throughput. To understand using an example, suppose the medium error rate of sending messages $k = 10^{-5}$ (e.g. the bit error rate of CAN is $10^{-11} \sim 10^{-7}$ [52]) and we want to send $n = 10^5$ messages in each period. Then, we have $t_1 = 0.5$ and $t_2 = 0.997$. In this case, the throughput of divisible 1-good mapping is twice as large as the throughput of dual mapping.

Therefore, if where we have to send messages periodically, sublinear mappings can support smaller periods and send messages more frequently than if we use dual mappings. On the other hand, if our messages do have a long enough periods — say that the period is long enough to allow for dual mappings, then the sublinear mappings may finish sending messages is half the time allowing the remaining time slots to be used for other (additional) messages.

3.6.2 Considering messages with different periods

In addition to the increased throughput, it turns out that sublinear overhead also has implications on how we should design our transmission protocols.

Say we have two periodic applications or two groups of messages (say $A$ and $B$) with different recurrent periods that are transmitted using the same medium. Say $T_A$ and $T_B$ are the periods of the two applications and $n_A$ and $n_B$ are the number of messages to be
transmitted in each periods respectively. Suppose \( T_A = 4T_B \), and \( n_A \approx 4n_B \). According to the average fault model, we must tolerate \( f \) faults within \( B \)'s period.

There are two strategies for transmitting these messages. First, within each period of \( B \), we can use \( T_B/2 \) time to send all of \( B \)'s messages (with fault tolerance) and then use \( T_B/2 \) time to send \( n_A/4 \) messages from \( A \). This would imply that \( T_B/2 \geq L(n_B, f) \) so everything is feasible. Second, we could interleave \( A \)'s and \( B \)'s messages so that we make a message bundle of \( n_C = n_B + n_A/4 \) messages which contains all of \( B \)'s and a quarter of \( A \)'s messages and transmit them during \( B \)'s period using fault tolerance. Then, we want \( T_B \geq L(n_C, f) \).

For linear mappings such as dual mappings, these two schedules are essentially the same. However for the collision-free mapping \( M(n, f, 1) \), the second (interleaved) mapping gives a better throughput and therefore can support more messages within the same period. This is due to the sublinear nature of the overhead — since \( \log^2(n_1 + n_2) \leq \log^2(n_1) + \log^2(n_2) \) (when \( n_1, n_2 \geq 8 \)). Therefore, by putting the messages of \( A \) and \( B \) together and then building the fault tolerant schedule together for the whole bundle of messages rather than one application at a time gives us better performance. This example indicates that sublinear mappings can motivate us to rethink modular design and may push for integration. In particular, instead of thinking of messages from the same applications and/or with similar deadlines together, sublinear overhead implies that we should strategically put messages (from different applications and possibly different periods) together into bundles since larger bundles lead to smaller overall overhead. Therefore, counter-intuitively, it might help performance to break modularity when designing distributed systems.

### 3.6.3 Retransmission mappings for dual-criticality scenarios

Previous work by Agrawal et al. [3] also generalized their dual-mapping method to a problem of transmitting messages of two criticalities. Formally, the problem they considered was that we have \( n_H \) high criticality messages and \( n_L \) low-criticality messages. The correctness
condition was: (1) If $\leq f_L$ medium errors occur, all messages must be successfully sent; (2) All high-criticality messages must be successfully sent if $\leq f_H$ medium errors occur. We now consider how we can solve this problem using sublinear overhead mappings.

Naively, we can simply have two retransmission mappings — one for the high criticality messages and one for the low-criticality messages. Given an algorithm for generating a retransmission mapping $RM(n, f)$ for any value of $n$ and $f$, we can generate two mappings $RM(n_H, f_H)$ for high-criticality messages and $RM(n_L, f_L)$ for low-criticality messages and concatenate them for a total schedule length of $I(n_L) + I(n_H) + RM(n_H, f_H) + RM(n_L, f_L)$ ($I$ is the identity mapping). Agrawal et al. [3] noticed that one can overlap the schedules of high-criticality and low-criticality messages if the retransmission schedule for single criticality satisfies a particular condition called adaptivity. In particular, a retransmission mapping $RM(n, f)$ for $n$ messages which tolerates $f$ faults is adaptive if, given any $f' < f$, $RM(n, f)$ can be decomposed two successive parts $RM_1(n, f, f')$ and $RM_2(n, f, f')$ such that $RM_1$ tolerates $f'$ faults for $n$ messages.

If a mapping $RM(n, f)$ is adaptive, Agrawal et al. [3] argue that it allows us to create shorter schedules than naively concatenating mappings. As before, we generate $RM(n_H, f_H)$ and $RM(n_L, f_L)$ and note that $RM(n_H, f_H)$ can be decomposed. We first send the identity mapping $I(n_H)$ followed by $RM_1(n_H, f_H, f_L)$ mapping. After this we overlap the transmission of the remaining $RM_2(n_H, f_H, f_L)$ with the entire schedule schedule for low-criticality messages — that is the identity mapping $I(n_L)$ followed by $RM(n_L, f_L)$. Agrawal et al. argue that this works, and we omit this argument. Therefore, the total length of the schedule is $n_H + |RM_1(n_H, f_H, f_L)| + \max\{|RM_2(n_H, f_H, f_L)|, n_L + |RM(n_L, f_L)|\}$.

We now argue that retransmission mappings described in Sections 3.2.2 are adaptive. (The argument for reducible mappings from Section 3.4 is similar.)

Claim 1. Any modulo mapping is adaptive. The modulo mapping (Definition 3) has $|RM_1(n, f, f')| = O(ff' \log^2 n)$ and $|RM_2(n, f, f')| = O(f(f - f') \log^2 n)$
Proof. Recall that the modulo mapping is a divisible collision-free mapping $M(n, f, 1)$, which would contain $f$ phases such that any $k$ consecutive phases are collision-free and divisible for any $k$ messages. Therefore, the first $f'$ phases consist of a divisible collision-free mapping $M(n, f', 1)$ by definition. Therefore, the size of $RM_1(n, f, f')$ is $O(ff' \log^2 n)$. $I(n) + RM_1$ can successfully send $n$ messages under $f'$ medium errors.

**Corollary 17.** Consider a dual-criticality scenario with $n_H$ high-criticality messages which must tolerate $f_H$ faults and $n_L$ low criticality messages which must tolerate $n_L$ faults. Using the modulo method, we can generate a fault-tolerant mapping with length $n_H + O(f_H f_L \log^2 n_H) + \max\{O(f_H (f_H - f_L) \log^2 n_H), n_L + O(f_L^2 \log^2 n_L)\}$.

When $f_H$ is substantially bigger than $f_L$ (and for reasonable values of $n_L$), this can represent significant savings over simple concatenation since the low-criticality transmission can entirely overlap with $M_2$ making it essentially free.

In this case, the asymptotic advantage of overlap is not very large since the retransmission mapping is itself small. However, if $f_H$ is very large compared to both $f_L$, then one might see some advantage.

### 3.7 Related works

Contention resolution technologies such as randomized backoff protocols are widely used to resolve contention for simple multiple access channels. In a randomized backoff protocol, when multiple jobs try to access a simple channel at the same time, they collide with each other and both back off for a random amount of time before retrying. Jobs can arrive at the simple channel following the statistical queuing-theory model [112] or in bursty (all jobs arrive at time 0) [11]. Jobs can have unit-sizes [11], or heterogeneous sizes [12]. Jamming, which is similar to our notion of medium errors has also been studied in this context [13,113,132]. In contrast to the type of fault-tolerant scheduling studied in this chapter, however, backoff protocols are not typically static.
Fault tolerance has also been studied in the context of real-time scheduling. There are two kinds of fault-tolerant techniques: the replication approach (also called the primary-backup approach) [33, 47, 135] and the re-execution approach [23, 67]. The re-execution approach is similar in flavor to the idea here where we might transmit the messages again.

There is a large body of works related to collision-free mappings. Some mappings are called "selector" (e.g. [12, 34, 45, 68, 78]). Some mappings are called "$\alpha$-good mappings". Selectors and $\alpha$-good mappings are conceptually equivalent to the collision-free mapping (which is proposed in this chapter). In particular, $(n, k, r)$-selector [34] is a $r/k$-good mapping for subset size of $k$ given $n$ messages. Note that our work propose "divisible" property and "reducible" property for the collision-free mapping and leverage them to achieve fault-tolerate transmissions. It is possible to find a mapping with a smaller size that has the reducible property and can be generated by a deterministic method. Then, our harmonic schedule could be quickly built in practice.
Chapter 4

Provably-good strategies for data placement in distributed Key-Value stores

4.1 Introduction

Motivation  Distributed key-value stores are used extensively on modern platforms, particularly in cloud applications [7, 39, 44, 61, 62], but also in large-scale databases [46, 66, 83, 123, 136] and distributed file systems [59, 77, 129]. In these applications, there is a large corpus of data items (e.g., key-value pairs, objects, files) stored across many servers. Clients of the system send requests, which access some particular data. Each request is routed to the server that has the items, and the server is responsible for processing this request and responding to the client. If the server is busy, the request may be queued.

For such systems, an important optimization criterion is throughput — the number of requests that the system can process on average per time period. If most requests end up accessing a small number of servers, most system capacity might be idle while a few servers’ queues grow unboundedly. The client may care about latency — the amount of time they wait between sending a request and receiving a reply. Since long queues impact latency, distributed storage systems implement bounded queues for servers such that a server with a full queue rejects future requests.
Problem The goal of the system is to *accept* or consume as many requests as possible, or inversely to *reject* as few requests as possible while keeping the queue size small — this is equivalent to maximizing throughput while keeping latency small. To avoid rejecting many requests (thereby reducing the system throughput), distributed storage systems try to balance load across servers. However, they have no direct control over the load, since requests are generated by clients and the request access pattern may change over time. Therefore, if all the “hot items” — data items that are being accessed frequently — are on the same server, that server will experience a high load. Avoiding overload may require load balancing via moving these items to other servers dynamically.

Load balancing algorithms for distributed storage systems generally have the following steps: (1) initially distribute data across servers in some manner; (2) each server has a queue with bounded size (decided by the algorithm) and rejects incoming requests if its queue is full; (3) do dynamic data distribution by choosing some data to move from an “overloaded” servers to other servers. The question considered in this work is how one should do these steps. This problem has been studied empirically using various heuristics based on past system behaviors or theoretically using queuing theoretic assumptions on request arrivals [42,107].

Contribution In this chapter, we consider this problem from a theoretical perspective. More formally, assume that the data is divided into \( n \) chunks that must be housed on \( m \) servers. At every time step, each server can process one request. We assume that the requests are generated by an adversarial process that generates \( m \) requests per time step and maximizes the number of rejected requests. Our goal is to understand what strategies with small queues may work against various adversarial assumptions. We have the following results:

**Simple deterministic strategies work against random clients, but no deterministic strategy is good against an adversarial client:** If the client is not adversarial and picks a random chunk to request in every time step, then any simple strategy that divides the
chunks evenly across servers works well; however, if the client is adversarial, then, for any deterministic algorithm, there exists a request sequence such that the algorithm rejects most requests.

**Simple randomized strategy works reasonably well against oblivious adversary:** A simple randomized strategy that places chunks on servers uniformly at random and then never moves them performs does ok — that is, it accepts at least a constant fraction of the requests in expectation. In addition, this strategy provides a strong bound of rejecting only $O(1/m)$ fraction of the requests if either of the following is true: (1) if the adversary is weakened so that there is a bound on how frequently it can access the same chunk; or (2) servers have speed augmentation and each server consumes $O(\log m)$ requests per time step instead of 1.

**Data transfer is useful:** We show that no strategy without data transfer and constant speed can consume $1 - 1/m$ fraction of requests with an adversarial client. We also design an algorithm that starts with a randomized allocation and then moves chunks out of heavily loaded servers to balance the load. Given constant speedups of processing and data transfer, we show that this strategy consumes $1 - 1/m$ fraction of the requests even against a fully adversarial client.

### 4.2 System model and performance metrics

In distributed data stores, we have $m$ servers that store a large collection of items, such as key-value pairs, objects, or files. We can partition the data into $n$ ($n > m^2$) data chunks, where a chunk is a collection of keys or a contiguous range of keys. The system distributes chunks to servers according to its load balancing policies. In this chapter, we are going to abstract away the details of exactly how the data partition is done and assume that $n$ and $m$ are given and do not change over time. We also leave other problems such as data replication
(and other database mechanisms, such as write-ahead log and multi-version control) as future work.

Clients send requests to the cluster online and we assume that each request accesses a single data chunk. When a request arrives, it is delivered to the server that hosts the appropriate data chunk. For analysis purposes, we divide the timeline into identical time slots, which have a length equal to the time to process a request. For simplicity, we do not distinguish the read, write, and read-range operations. Each server has a FIFO queue (first-in-first-out) with the maximum length of \( q \) to store the unserved client requests. When requests arrive, they are stored in the respective server’s queue unless there is no space in which case the request is rejected. Since a server can consume one request in a slot, the cluster can consume at most \( m \) requests in the ideal case where the \( m \) requests access chunks at \( m \) different servers. Therefore, to make the workload feasible, we assume that at most \( m \) requests arrive at the cluster in a time slot and each of these requests accesses a different chunk.

During system initialization, the system allocates chunks to servers according to some load balancing algorithms, after which client requests are served. For the first few results in this chapter, we assume that the allocation is fixed after initialization. However, some servers may be overloaded, and a load-balancing algorithm may want to move data from one server to another; therefore, later sections of the chapter analyze algorithms that move data. We formally model the process of the data movement, named data transfer, as follows. At any time, a server can be involved in one data transfer. In each transfer, a server can send at most \( s \) chunks to one other server, and this takes \( s \) time slots. In other words, transferring \( s \) or fewer chunks from one server to another takes \( s \) time; therefore, the transfer time is a stepped function, not linear. This models the fact that the latency of moving data from one server to another is often large, but the bandwidth is also large — therefore, sending or receiving 1 chunk vs. several chunks up to the bandwidth takes the same amount of time.
On the other hand, moving two sets of chunks from or to two different servers takes 2s time slots, since the server must prepare the transfer and initiate the transfer. We use the single parameter $s$ for modeling both the maximum number of chunks in a data transfer (with a total size below the bandwidth) and the latency of a data transfer.

As noted in Section 4.1, the goal is to minimize the number of rejected requests, which is a measure of throughput.

**Definition 8 (Throughput).** Given a sequence $\sigma$ of requests arriving over time, $A(\sigma)$ denotes the number of requests that are accepted by algorithm $A$ on sequence $\sigma$.

We assume the input sequence of client requests is generated by an oblivious adversary defined as follows.

**Definition 9 (Oblivious adversary).** The oblivious adversary knows how the online algorithm works, but it does not know the random choices made by the algorithm.

For deterministic data allocation, the oblivious adversary always knows the exact locations of chunks at any point in time; therefore, unsurprisingly, we were able to show that no deterministic algorithm performs well. Most of this chapter analyzes the performance of various randomized algorithms.

The theoretical performance of an algorithm can be analyzed by comparing its throughput with optimal throughput.

**Definition 10 (Constant competitive).** An online algorithm $A$ is (constant) $c$-competitive, if $A(\sigma) \geq c \cdot OPT(\sigma)$ for any finite input sequence $\sigma$, where $OPT$ is the offline optimal.

An algorithm’s performance is better when $c$ gets closer to 1. While constant competitiveness is nice, we would prefer not to reject a constant fraction of the requests. Thus, we define the following, stronger performance criterion.
Definition 11 (Almost optimal). An online algorithm $A$ is almost optimal, if $A(\sigma) \geq (1 - O(1/m))OPT(\sigma)$ for any finite input sequence $\sigma$, where $OPT$ is the optimal offline scheduler.

For randomized algorithms, similar definitions apply except that we compare the expected throughput of the algorithm with the throughput of $OPT$. Note that in the best case for an offline optimal where all requests arriving in a time slot access different servers and are processed in this slot, all the $|\sigma|$ requests in a finite input sequence $\sigma$ can be accepted by the optimal. Hence, an online algorithm is near optimal if it can accept $(1 - 1/m)|\sigma|$ requests for all input sequences.

Finally, some of our algorithms will require resource augmentation or speedup. Resource augmentation implies that we allow the algorithm to perform certain operations faster than the optimal algorithm can. We consider two types of resource augmentation: (1) speedup of request processing on servers and (2) speedup of data transfer. Often resource augmentation is necessary to achieve nontrivial results. However, the smaller the resource augmentation required, the better the algorithm. We generally want the resource augmentation factor to be no more than a constant.

4.3 Deterministic policies

This section analyzes deterministic policies for allocating data chunks to servers. For warm up, we will first show that if the client requests access to random chunks, then any policy that evenly balances the number of chunks on each server performs well. On the other hand, no deterministic policy performs well against an oblivious adversary — for any deterministic policy, there exists an access pattern that causes the system to accept only a small number of requests.
4.3.1 Simple policies work well against random clients

We now do a simple analysis showing that all reasonable allocations work for random clients where each request picks a chunk to access uniformly at random and independently. Without loss of generality, we relax the assumption and allow multiple requests to access the same chunk even in the same time slot. For this random client, we consider a simple even allocation, where $n/m$ chunks are allocated to each server. The allocation is generated once and need not change. The following lemma is easy to prove using Chernoff bounds.

**Lemma 18.** For a random client and an even allocation, in any $\log m$ consecutive time slots, the probability that more than $3 \log m$ requests arrive at a particular server is less than $1/m^2$.

*Proof.* In $\log m$ consecutive time steps, $m \log m$ total chunks are picked uniformly at random. Let $X_i$ denote the random variable that request $i$ goes to a server (say server a). $X_i = 1$ with the probability of $1/m$. All $X_i$’s are independent. Let $X = \sum_{i=1}^k X_i$ denote the number of requests out of the $m \log m$ requests that go to the server a. Chernoff bounds gives us the result. \hfill \Box

Lemma 18 leads to the following theorem.

**Theorem 19.** Given a random client, an even allocation with speed 3 and queue size $q = 6 \log m$ accepts at least $(1 - 1/m)|\sigma|$ requests in expectation for any sequence of $|\sigma|$ requests.

*Proof.* Divide the execution into rounds, where each round has $m \log m$ requests — therefore, each round has at least $\log m$ time steps, so a server can process $3 \log m$ requests with speed 3 from its queue. From Lemma 18, the probability that a server receives more than $3 \log m$ requests in a round is at most $1/m^2$. By union bound, in a particular round $r$, the probability that at least one server receives more than $3 \log m$ requests is at most $1/m$.\hfill 91
For a particular round $r$, we will prove via induction that the number of leftover requests from the previous round in any server’s queue is at most $3 \log m$ for all servers at the start of every round. We have two cases. First, in round $r$, no server gets more than $3 \log m$ requests. In this case, the queue size of server $a$ at the end of the round does not increase, since server $a$ can process $3 \log m$ requests during a round if available. During this round, there can be at most $6 \log m$ requests in any queue and no requests are rejected. Second, in round $r$, some server get more than $3 \log m$ requests. We can pessimistically assume that all the $3 \log m$ requests that arrived in $r$ are rejected maintaining the inductive hypothesis. Since requests are rejected only in the second type of rounds, which occur with a probability at most $1/m$, we reject at most $1/m$ fraction of requests in expectation.

4.3.2 No deterministic policy works against an adversarial client

We now consider an oblivious adversary. Intuitively, since the algorithm is deterministic and the adversary knows the allocation, it can always overload some server. Here we argue that no deterministic policy, even if it moves data to load balance, can work well with small queues.

**Theorem 20.** Consider a deterministic algorithm $D$ running on a system with $m$ servers, $n$ chunks, max queue length $q$, data transfer time $s$, and constant speedup for both request processing and data transfer, where $n > m^2$. There exists a request sequence $\sigma$ such that $\text{OPT}(\sigma)/D(\sigma) = \Omega(ms/(q+cs))$ where $\text{OPT}$ is the offline optimal.

**Proof.** Since $n > m^2$, at any time instant, there exists a server with more than $m$ chunks. Since the algorithm is deterministic, the adversary always knows which server has more than $m$ chunks and can send all $m$ requests to these chunks. Since the server can process only $c$ chunks per time step, the queue will soon fill up causing most requests to be rejected.

Now, say that the $D$ moves chunks. We divide the timeline into rounds. Each round has $ms$ time slots, where $s$ is the number of slots to perform a data transfer. Here is the
adversary’s strategy. At the start of round \( i \), pick \( m \) chunks that are all in a particular server for \( D \), say \( a \), and send requests to these chunks only for the next \( s \) time steps. By the end of \( s \) steps, \( D \) can move \( s \) of these chunks away from \( a \) with \( q \) requests. With speed \( c \) for request processing, \( D \) can process at most \( cs \) requests in the \( s \) time steps and store at most \( q \) requests in the queue. Therefore, it accepts at most \( cs + 2q \) of these \( ms \) requests. After this, for the rest of the round (the remaining \( ms - s \) time steps), the adversary does not send any requests.

Since OPT knows the sequence, in the previous round \( i - 1 \), it will set up to ensure that all these \( m \) chunks were on different servers so it can accept all requests. In the remaining \((m - 1)s\) time steps of round \( i \), OPT (in collaboration with the adversary) sets up for the next round. It knows which server will have at least \( m \) chunks at the start of the next round for \( D \), so it distributes these \( m \) chunks across \( m \) servers for OPT so that OPT can answer all \( ms \) requests in the next round. The adversary can then repeat for the next round.

4.4 Randomized policies with no transfers

In this section, we consider a simple randomized strategy that randomly allocates chunks to servers, which is similar to the game of throwing randomized balls into bins. We will show that it is constant competitive. We will also prove that no strategy that doesn’t move data can be almost optimal against an oblivious adversary. In the next section, we will see a strategy that is almost optimal.

**Definition 12** \((M_{\text{ball-bin}})\). For \( n \) chunks and \( m \) servers, the algorithm picks a server for each chunk uniformly at random and independently. A \( M_{\text{ball-bin}} \) mapping denotes the mapping between chunks and servers.

4.4.1 Upper Bounds for Balls into Bins

We will now prove a theorem against an adversarial client.
Theorem 21. Given a system with $m$ servers and $n$ requests. For any sequence $\sigma$, say $E[B_{\sigma}]$ is the expected number of requests accepted by balls into bins allocation $B$. Then, $E[B_{\sigma}]/|\sigma| \geq 1 - 1/e$ even with queue size $q = 1$ and no resource augmentation.

Proof. Consider the balls into bins allocation and consider a particular time step $t$ when the client sends requests to $k \leq m$ distinct chunks. These $k$ chunks were randomly thrown on $m$ servers. Now consider a particular server $a$. The probability that exactly $i$ of these $k$ requests hit $a$ is:

$$\binom{k}{i} \left( \frac{1}{m} \right)^i \left( \frac{m-1}{m} \right)^{k-i}$$

Therefore, the expected number of servers that get at least 1 request is at least $x = m - m \left( \frac{m-1}{m} \right)^k \geq (1 - \frac{1}{e})k$. Any server that gets at least one request processes at least one request — therefore, at least $x$ requests out of $k$ total requests were consumed at this time step. Therefore, by adding over all time steps, we get the result. 

We see that the balls into bins allocation is constant competitive with no speed augmentation and with very small queues. What if we give it speed augmentation? We now show that with sufficient ($\Theta(\log m)$) speed augmentation, it is almost optimal.

Theorem 22. Given a system with $m$ servers and $n$ requests. For any sequence $\sigma$, say $E[B_{\sigma}]$ is the expected number of requests accepted by the balls into bins allocation $B$. Then, $E[B_{\sigma}]/|\sigma| \geq 1 - O(1/m)$ with $\Theta(\log m)$ speedup on request processing and queue size $q = \Theta(\log m)$.

Proof. Consider a particular time step $t$ when the client sends requests to $k \leq m$ distinct chunks. These $k$ chunks were randomly thrown on $m$ servers when the allocation was done. Now consider a particular server $a$. The probability that at least $x$ requests access this server at this time step is at most $\binom{k}{x} \left( \frac{1}{m} \right)^x \leq \frac{k^x}{x!} \left( \frac{1}{m} \right)^x = \frac{1}{x!} < 1/m^2$ for $x = c \log m$ with large enough $c$. At any time step, with $\Theta(\log m)$ speed, $x = \Theta(\log m)$ requests can be processed.
by a server. Therefore, the balls into bins allocation can process all the \( k \) requests that arrive on that time step with a probability at least \( 1 - 1/m \). Even assuming pessimistically that all requests at other times are rejected, summing over time still gives us the result. 

### 4.4.2 Lower bound on algorithms with no data transfer

We saw that the balls into bins algorithm is almost optimal with \( \Theta(\log m) \) speed; however, what about constant speed? We now show a lower bound, saying that any algorithm without moving data cannot be almost optimal with constant speed.

**Theorem 23.** Given any policy \( D \) for allocating chunks, for any queue with length \( q \), a constant speedup of processing, if \( n > m^2 \), then there exists an input sequence, such that 

\[
\frac{E[D]}{E[Opt]} \leq p
\]

where \( p < 1 \) is a constant.

Here is the adversary’s strategy: It simply selects \( m \) different chunks at random and repeatedly requests these chunks in every time slot. The challenge is to show that there is no randomized strategy that can provide an almost optimal acceptance ratio in expectation for this sequence. To do this, we will define a concept of a **group**, which will allow us to extract a common property of all distributions.

**Definition 13** (group). Let a \( T \)-group consists of exactly \( T \) chunks that are in the same server, and different groups do not have any chunks in common.

We want all groups to have the same number of chunks in order to compute probabilities. If a server has at least \( T \) chunks, we create groups with \( T \) chunks each and leave the leftover chunks **ungrouped**. Repeatedly put \( T \) ungrouped chunks into groups until we have fewer than \( T \) ungrouped chunks. The following observation just says that there are a large number of groups.

**Observation 2.** For any position of \( n \) chunks in \( m \) servers, the number of \( (n/2m) \)-groups is at least \( m \).
We now only consider grouped chunks and we will show that a large number of requests sent to grouped chunks will be rejected. In particular, we will first show that the probability that a group gets a large number of requests is not too small.

**Lemma 24.** Given a specific distribution of the chunks, and suppose the adversary randomly selects $m$ chunks. Then for $k = o(m)$ and $n > m^2$, the probability that an $(n/2m)$-group has at least $k$ requests is greater than $p = \frac{1}{8^{k!}} \cdot e^{-\frac{3}{2}}$. We can say that these groups are overloaded.

**Proof.** Since the adversary randomly chooses $m$ distinct chunks, the number of chunks in each group follows a *Hypergeometric Distribution*. In particular, consider $k = o(m)$ and group size $T = n/2m$. We can assume that $m - k > m/2$ and $T - k > T/2$ and $m - k < n/m$. We can then bound the probability that a particular group has $k$ chunks among the $m$ chunks by the following:

\[
\frac{\binom{T}{k} \binom{n-T}{m-k}}{\binom{n}{m}} = \frac{\frac{T!}{k!(T-k)!} \cdot \frac{(n-T)!}{(m-k)!(n-T-(m-k))!}}{\frac{n!}{m!(n-m)!}} \cdot \frac{\frac{n!}{m!(n-m)!}}{\frac{m!}{(n-m)!}} 
\]

\[
> \frac{1}{k!} \cdot \frac{T!}{(T-k)!} \cdot \frac{(n-T)!}{(n-T-(m-k))!} \cdot \frac{m!}{(m-k)!} > \frac{1}{k!} \cdot (T-k)^k \cdot (n-T-(m-k))^{m-k} \cdot (m-k)^k
\]

\[
> \frac{1}{k!} \cdot \left(\frac{m}{2}\right)^k \cdot \left(\frac{T}{2}\right)^k \cdot \left(n \left(1 - \frac{1}{2m} - \frac{1}{m}\right)\right)^{m-k}
\]

\[
= \frac{1}{8^{k!}} \cdot \left(1 - \frac{3}{2} \cdot \frac{1}{m}\right)^{m-k}
\]

\[
> \frac{1}{8^{k!}} \cdot e^{-\frac{3}{2}}
\]

This proves the lemma. □
Now we are able to give a proof for Theorem 23. Since the adversary repeatedly requests the same $m$ chunks, overloaded groups remain overloaded. Therefore, regardless of the queue size, servers with overloaded groups will eventually reject many of their requests.

Proof. According to lemma 24, each $(n/2m)$-group is overloaded with probability $p$ — they get at least $k$ requests on every time step. Therefore, with any speedup smaller than $k$, these groups remain overloaded forever since their total capacity is smaller than their average load. Therefore, these chunks will reject at least one request per time step. Since the expected number of overloaded groups is $mp = \frac{m}{s^k} \cdot e^{-\frac{3}{2}}$, with speed $< k$, at least $mp$ requests are rejected on every time step within expectation while the optimal algorithm can handle all $m$ requests. Therefore, the fraction of the requests that any algorithm accepts is $p$, and $p$ is a constant for any constant $k$. 

4.4.3 Constraining the request sequence.

The argument in the previous section indicates that the worst-case workload for any randomized strategy that does not move data is an adversary which requests the same $m$ chunks repeatedly. However, this is often not what happens in the real world. In this section, we show that we can do better if the adversary can not request the same chunks repeatedly — we assume that the client issues at most one request to a particular chunk in any consecutive $\log m$ time steps and show that balls into bins strategy is almost optimal against this adversary.

We divide time into phases of size $\log m$. The following Lemma is similar to Lemma 18, except that the randomness comes from the randomized allocation rather than the client.

Lemma 25. Against a constraint adversary which can not send more than one request to the same server in one phase, the probability that any server gets more than $3 \log m$ requests in a phase is at most $1/m^2$. 

With Lemma 25, we can prove Theorem 26 in a manner very similar to the proof of Theorem 19.

**Theorem 26.** Given a queue with size \( q = O(\log m) \) and a constrained adversarial input \( \sigma \), the expected number of requests consumed by balls into bins policy is \( E[B] \geq (1 - 1/m)\sigma \).

The result indicates that a workload is great if requests are not successively and repeatedly access to the same chunk.

### 4.5 Randomized policies with transfers

In this section, our goal is to design a policy that is almost optimal (consumes \( 1 - O(1/m) \) fraction of the requests) with constant speedup and without placing any restrictions on the input sequence. In particular, we propose an algorithm, say \( Y \) which satisfies the following strong guarantee.

**Theorem 27.** Say we have a constant speedup of data movement and a constant speedup of processing on the system \( Y \) with the queue length \( q = \Theta(s \log^2 m) \), where \( s \) is the number of time slots to complete a data transfer. For any input sequence \( \sigma \), the expected number of requests consumed by \( Y \) is at least \( (1 - O(1/m^2))\sigma \).

#### 4.5.1 An overview of the system \( Y \)

In prior sections, we have seen that while deterministic policies can not be shown to be constant competitive (even allowing data movement) (Theorem 20), simple randomized policies such as allocating chunks uniformly at random can achieve constant competitiveness (Theorem 21). In particular, with balls into bins randomization, a constant fraction of the requests can be consumed on each time step. However, since some servers get more than a constant number of requests (some servers get \( \Omega(\log m) \) requests) on each time step, these servers are overloaded and the adversary can keep them overloaded — therefore, with at most constant...
speed and with no restrictions on the input sequence, no randomized policy without data movement can be almost optimal (Theorem 23).

To address this issue, we will design a system, we call it, \( Y \) which moves data to get a good load balance. In particular, we start with a randomized balls into bins allocation just like the previous sections. Therefore, all chunks have a home server where they were allocated through balls into bins. However, when the queue of the server contains a request that has been in the system for \( \Theta(s \log m) \) time, this indicates that this server is overloaded and can not handle all the requests being sent to it. At this point, a batch movement process is triggered on this server and the chunks in the server that have pending requests are distributed to other servers so that these old pending requests can be handled efficiently by other servers. Once these pending requests are handled, these chunks are moved back to their home server so that the original randomized allocation is restored.

**Modeling assumptions** For simplicity in analysis, we will make some modeling assumptions that do not impact the overall result. In particular, we will assume that each server has two processors: a primary processor which consumes the requests that arrive at this server from the client and a secondary processor which consumes the requests that are sent to it from other servers for load balancing purposes from other processors. Each server has its own queues, where the primary and secondary queues are both of size \( \Theta(s \log^2 m) \). Note that this does not impact the theorem statement since we allow for constant speedup — if we allow for processor speed of \( \rho^p \), then each of the processors can run at half the speed. Similarly, the queue length can be split among the two queues while only impacting constant factors. In this analysis, we will not try to optimize constant factors; therefore, the constant factors computed will be large. In the evaluation section, we will see that the algorithm requires a quite small constant in practice to perform very well.

To restate the algorithm using these terms: when a request arrives at a server from a client, it is put in the primary queue. If the primary queue is full, then the request is rejected.
If the age of the oldest request in the primary queue is $6s \log m$, the server triggers a batch movement process, and any chunk that has a request in the primary queue is moved to other servers and their corresponding requests are moved to those servers’ secondary queues \textit{(move out phase)}. These moved requests are processed by the target processor’s secondary server. Once a server’s secondary queue is empty, the chunks that have been moved there are sent back to their home servers \textit{(move back phase)}. We will define the precise policy of movement momentarily, but let us first consider what the challenges are in designing this policy.

**Challenges** Intuitively, the system $Y$ should work since (1) it moves the requests out of the primary queue in a timely manner to avoid filling the primary queue; (2) it spreads the requests among the secondary queues, which efficiently consumes the requests and avoids filling secondary queue. However, there are a few challenges. First, a batch movement process might take a long time if the batches are large. When a server triggers a batch movement, many chunks in that server may have pending requests in the primary queue and all these chunks must be moved. Recall, from Section 4.2, that it takes $s$ time to move at most $s$ chunks between two particular servers — we call this one operation a \textit{transfer}. Therefore, depending on how many transfers are required, a full batch movement may take time. This can cause the queue of the server to get longer while the batch process is executing as well as blocking other batch processes from starting. Second, we may get unlucky and a server may get a very large number of requests from the client at the same time step. When this happens, the primary queue can become overloaded very quickly, potentially causing downstream effects. Third, the batch process is deterministic — therefore, in principle, the adversary can guess the location of chunks when they are being processed away from their home server and can overload the servers where these chunks are located by sending too many requests there.
Algorithm Description

We can now describe $Y$ and how it handles these challenges. The queue size of each server is $\Theta(s \log^2 m)$ for a sufficiently large constant hidden in the $\Theta$-notation.

(I) **Batch trigger:** A batch process at a particular server $a$ is triggered when the oldest request at this server that is not already a part of an older batch process is $6s \log m$ old.

(II) **Batch start:** In $Y$, batch processes may not start as soon as they are triggered; they are executed in order. Say, the batch process $i$ is triggered before the batch process $j$ is triggered. If the processes are triggered on the same server, then $j$ executes after $i$ completes since the same chunks may be involved in both. If they are on different servers, they can execute in parallel. Our distribution algorithm ensures that a server is involved in at most transfer at a time.

(III) **Data distribution during moveout:** Once a batch is triggered, we know which chunks are part of the batch — that is which chunks have pending requests which are part of this batch. These chunks are divided into $\log m$ packages for transfer so that each bucket has $O(s)$ chunks and at most $O(s \log m)$ requests. We later show that this is possible with high probability. Therefore, each of these packages can be moved to destination servers using one transfer each. When the batch starts, these buckets are moved to $\log m$ different servers and the corresponding requests are moved to those server’s secondary queues.

(IV) **Process the chunks and move back:** The secondary processor of the target server processes the requests from the secondary queue in order. Once all requests of a particular transfer have been processed, the chunks are moved back to their home server.

(V) **Request handling during batch:** Even when a chunk has been moved to a different server for processing its pending requests, any new requests to this chunk are still sent to its home server’s primary queue and these are processed once the chunk has moved back to its home server.
(VI) Flow control and batch cut-off: In order to avoid some boundary conditions, we perform some control (1) If a single server receives more than $2 \log m$ requests in a single time step, then some of the requests are immediately rejected to ensure that only $2 \log m$ requests are added to any primary queue on any single time step. (2) If the server has more than $24s \log m$ chunks with requests in the primary queue, the server only moves $24s \log m$ chunks with the most requests, and the server rejects the requests of unmoved chunks.

Causes of rejection We want to show that this process rejects at most $O(1/m^2)$ fraction of the requests in expectation. Note that requests can be rejected due to the following reasons: (1) flow control and batch cut; (2) rejection from the primary queue if it becomes full; and (3) rejection from the secondary queue if it becomes full. We will show that the first two reasons cause few rejects in expectation and the last reason causes no rejections for a good load balancing policy. However, in order to show this, we must first bound the time it needs to complete a batch process.

4.5.2 Bounding the execution time of a batch process

We first bound the execution time of a batch process — the time between the triggering and completion of a batch process. We will divide time into phases of size $3s \log m$. We say that $X_i$ is the set of batch processes that were triggered during phase $i$. We will prove the following key Lemma.

Lemma 28. Any batch process that was triggered during phase $i$ will be completed by the end of phase $i + 1$ with constant speedup on both processing speed and transfer speed.

In order to prove this lemma, we first prove some supporting lemmas. First, we recall that a batch process is triggered when the oldest request in the queue which is not already part of the batch process has been in the queue for $6s \log m$ time. Therefore, a batch process can contain requests from at most $6s \log m$ time steps. We can bound the number of requests that are part of a batch process as follows:
Lemma 29. Given the balls into bins allocation, constant $c \geq 1$ and $s < m/(c \log m)$. In $6s \log m$ consecutive time slots, the probability that a particular server receives $12s \log^2 m$ or more requests is less than $1/m^2$.

Proof. Note that the client can send requests to the same chunk in different time steps, but in one time slot it must send $m$ different requests. In each time step, for a particular server $a$, say $X_a^i$ is the random variable representing the number of requests that hit this server at time step $i$. For $x = 2 \log m$, we have

$$\Pr(X_a^i \geq x) \leq \binom{m}{x} \left( \frac{1}{m} \right)^x \leq \frac{m^x}{x!} \left( \frac{1}{m} \right)^x = \frac{1}{x!} < \frac{1}{m^3}$$

The last equation works when $m > 4$. Therefore, we can sum up $X_a^i$ through all $cs \log m$ steps and use Bernoulli’s inequality to get the bound we need.

We can also bound the number of chunks involved in a batch process by a method very similar to Lemma 18.

Lemma 30. Given a balls into bins allocation, the total number of chunks from a particular server that can be requested within $6s \log m$ time is $24s \log m$ with probability of at least $1 - 1/m^2$. Therefore, the total number of chunks involved in a batch process is $24s \log m$ with probability of $1 - 1/m^2$. Hence, each batch process causes $\log m$ transfers to different targets.

Given these lemmas, we can define a transfer packaging policy. Recall that each data transfer can transfer up to $s$ chunks to a particular server. Once we have defined a batch with at most $O(s \log^2 m)$ requests and $O(s \log m)$ chunks, we package these into $\log m$ transfers greedily. We keep adding chunks to a transfer until either the number of requests in the transfer is greater than $c_1 s \log m$ for some constant $c_1$ or the number of chunks in the transfer is greater than $c_2 s$. At this point, this transfer is complete and we start a new transfer. Due to the previous two lemmas, the total number of transfers for a particular
batch process is at most \( \log m \) for suitable choices of \( c_1 \) and \( c_2 \). Therefore, we have the following corollary:

**Corollary 31.** *The amount of time it takes to move all the chunks that are part of the same batch process to \( \log m \) target servers is \( s \log m \) assuming that we have data movement speed-up of 24 — that, we can move 24\( s \) chunks in \( s \) time.*

We can also bound the number of requests in the secondary queue under certain conditions. Consider batch processes in \( X_i \cup X_{i+1} \) — the batch processes triggered during two consecutive phases. We want to bound the number of requests that end up in the same server’s secondary queue due to these batch processes.

**Lemma 32.** *Considering only requests which are part of batches from \( X_i \cup X_{i+1} \), the number of these requests that end up in a particular secondary server’s queue is at most \( c_3 s \log m \) for some constant \( c_3 \).*

*Proof.* All the requests that are part of batch processes in \( X_i \cup X_{i+1} \) arrive within a time interval of \( 12s \log m \) since any request which is in batch from \( X_i \) must have arrived at most \( 6s \log m \) time before the beginning of \( X_i \). Therefore, these batches can contain a maximum of \( 12sm \log m \) requests. Since requests are evenly balanced across servers during the batch movements and we have \( m \) servers, no server has more than \( c_3 s \log m \) requests for a suitably large \( c_3 \).

We now observe the following invariant about consecutive phases since a batch is triggered when the oldest request which is not part of a batch arrived 6\( s \) \( \log m \) time ago.

**Observation 3.** *Consider any two consecutive phases \( i \) and \( i+1 \), and consider the set \( X_i \cup X_{i+1} \) — the set of batch processes triggered during these phases. No two batch processes in this set can be triggered by the same server.*

Now we can prove Lemma 28 via induction.
Proof. The base case is trivial since the first phase doesn’t trigger any batch processes. For Inductive Hypothesis, assume that all batch processes triggered during phase $i$ completed by the end of phase $i + 1$. Therefore, all batch processes triggered during phase $i + 1$ are ready to start at the beginning of phase $i + 2$ since all prior batch processes at their respective servers have completed.

None of the batch processes triggered during phase $i + 1$ have the same source (Observation 3). In addition, they trigger log $m$ transfers each (Corollary 30). Therefore, there are a total of at most $m$ sources and at most $m \log m$ targets of the transfers — these can be scheduled in $s \log m$ time without source or destination conflicts with sufficient large constant speedup in data transfer speed.

Now consider the processing of these requests by the secondary server. The secondary server can only contain requests due to batch processes triggered during phase $i + 1$ and phase $i + 2$ (since phase $i + 2$ has started, some of the batch processes in this phase may have also started). By Lemma 32, the total number of requests in any secondary queue due to requests from these two phases is at most $c_3 s \log m$ for some constant $c_3$. Therefore, again, with sufficient constant speedup on processing, all these requests can be processed by the secondary server in the next $s \log m$ steps. Finally, all the chunks from batch processes in phase $i + 1$ must move back and again, which can be done in $s \log m$ time by using a reverse schedule from the move-out schedule. Therefore, in at most $3s \log m$ time after the start of phase $i + 1$ — that is, at the end of phase $i + 2$ — all these batches have completed. □

4.5.3 Bounding the number of rejected requests

We can now show that the number of rejected requests is small. Below is the proof of Theorem 27.

Proof. Recall that requests can be rejected due to the two control strategies. First note that Lemma 30 implies that the probability that a batch has more than $24s \log m$ chunks involved
is very small. The expected number of requests rejected due to this is small. Now consider the case where $2 \log m$ requests arrive at the same server. For a particular server $a$, say $X_a$ is the random variable representing the number of requests that hit this server at this time step. Denote $x = 2 \log m$, we have

$$
\Pr(X \geq x) \leq \binom{k}{x} \left( \frac{1}{m} \right)^x \leq \frac{k^x}{x!} \left( \frac{1}{m} \right)^x = \frac{1}{x!}
$$

For $m > 4$ we have $\frac{1}{x!} < 1/m^3$. According to the union bound through all the servers, the probability that it is rejected due to the policy is less than $1/m^2$.

In addition to the control strategies, a request may be rejected if the primary queue is full. However, recall that each batch process completes in time at most $O(s \log m)$ and due to Lemma 29 the total number of requests that arrive in this time period is at most $O(s \log^2 m)$ with probability $1 - 1/m^2$. Since the primary queue has this capacity, the probability of a request being rejected due to this is small. Finally, Lemma 32 indicates that requests can never be rejected from the secondary queue since the number of items in it are at most $O(s \log m)$.

\[\square\]

4.6 Related work

Both the problem and the solution are related to many topics studied by researchers. Here we provide a brief overview of some of the related work.

Distributed key-value stores have been designed both for academia and commercial use [46, 59, 79, 83, 123, 136]. Most of these systems are evaluated empirically and most of this work finds that handling online requests that might be skewed towards certain keys is often an important challenge [9, 43]. Most of the theoretical analysis of these systems is done by making stochastic assumptions on the arrival pattern of requests [98,99,114]. In this chapter, we take a different tack and analyze these systems under adversarial inputs.
Another related area of research is distributed hash tables, which have received extensive theoretical and empirical investigation. For instance, consistent hashing [76, 97] is widely used to partition the data in distributed hash maps [46, 83, 120]. Randomization is widely used to build and maintain the distributed hash tables [48, 85, 125]. In addition, some distributed hash tables consider network topology in the design of the hash table itself [10, 117]. The problem we consider in this chapter is also related to general load balancing problems where tasks can be dispatched to various servers. The techniques used in this chapter such as balls-into-bins analysis were originally developed in the load balancing context. These games have been extensively analyzed for various metrics [15, 60, 111]. In addition, variations of these games such as power of two choices [57, 96, 99] have also been analyzed extensively and sometimes used in the design of distributed hash tables. The problem we considered in this chapter is different but related, to the distributed hash table problem as well as the load balancing problem. In particular, the model for queuing and the model for moving data from one location to another is generally different in distributed hash tables which leads to different algorithmic and analytical challenges.

Since most online systems are dynamic where the system load changes and the keys which are accessed frequently change over time, data migration is an important tool in this system design. There has been extensive empirical work for designing data migration schemes and evaluating them in a diverse range of systems [28, 32, 65, 79, 121, 123, 124, 130, 136] for various distributed workloads in key-value stores and distributed databases. Most of this work is empirical or uses stochastic assumptions on data arrivals for analysis.
Chapter 5

Simulation of data-placement policies for distributed key-value stores

5.1 Introduction

Motivation  In Chapter 4, we theoretically analyzed various data-placement strategies for distributed key-value stores to achieve high throughput. In this chapter, we evaluate our theoretical assumptions in real-world systems, demonstrate the effectiveness of data transfer, and evaluate the speedup to consume almost all requests under practical workloads.

Although we have theoretically shown that the strategies achieve high throughput in Chapter 4, there needs to be more in understanding the performance of the strategies in practice. Recall that in Chapter 4, we made assumptions on request processing and data transfer, and we assumed that input sequences of requests are adversarial, and we showed that data transfers are essential to achieve almost optimal throughput with constant speedup. However, whether the assumptions are reasonable in practice is unclear. Further, the real-world workload is rarely adversarial. It is unclear whether data transfers are still essential against practical workloads. Moreover, we showed that the batch strategy achieves almost optimal with constant speedup. It is unknown whether the speedup is large in practice.

Problems  We are thrilled to ask the following questions to understand the performance of the data-placement strategies in practice:

108
• Are the assumption of request processing and the data-transfer model in Chapter 4 reasonable in practice? What are the reasonable values of data-transfer latency and bandwidth in real-world systems?

• How do the strategies in Chapter 4 perform against practical workloads? In particular, is data transfer essential to consume almost all requests under practical workloads?

• How large the processing speedup is required for the strategies to consume all requests under practical settings?

Furthermore, recall that Chapter 4 shows that the batch strategy (the randomized policy with data transfer) includes a move-back step against the oblivious adversary. When input sequence is not adversarial, how does the batch strategy perform without the move-back step? In addition, recall that the batch strategy uses two queues: One queue is used for queuing the requests issued by the client; The other queue is used for queuing the requests moved in by the data transfer. Which queue should be allocated more resources to process when both queues contain requests? Additionally, when the batch strategy selects a server to move in chunks, which server should be selected? How do those options impact throughput? Besides, is data transfer essential to achieve a small latency in practice?

**Contribution** In this chapter, we conduct simulations to answer those questions. The simulation simplifies real-world systems, but the simulation captures key features of key-value stores. We have the following contributions:

• We measure request service time, data-transfer bandwidth, and data-transfer latency in a real-world system. We show that our request-processing and data-transfer models are reasonable in real-world systems. In our simulations, we set parameters to align with the parameters we measured in the real-world system.
• We build three different workloads for our simulations: (1) Adversarial workload, which repeatedly sends requests to a small group of chunks; (2) Zipfian workload, which follows the Zipfian distribution. The Zipfian distribution is widely used in real-world benchmarks for key-value stores, such as YCSB [43]; (3) Cloud workload, which is a synthesis workload of real-world applications.

• We compare the throughput of the randomized strategy and the batch strategy under the three workloads. Our results show that the batch strategy accepts at least 97% requests for all three workloads while the randomized strategy accepts 55%, 80%, and 92% requests for the adversarial workload, the Zipfian workload, and the Cloud workload respectively. Our results indicate that data transfer is essential to achieve high throughput in practice.

• We compare the throughput of the randomized strategy and the batch strategy under different speedups. When running the batch strategy against all workloads, 1 speedup is sufficient to accept at least 97% requests, and 2 speedup is sufficient to accept all requests. Our results show that the speedup for the batch strategy to consume all requests against practical workloads is between 1 and 2, which is much smaller than the speedup we use in our analysis in Chapter 4.

• We compare the batch strategy and the batch strategy without the move-back step for different design options, considering: (1) Destination selection of data transfer; (2) Consuming priority of the primary and secondary queue. The simulation result indicates: (1) Keeping the secondary queue short is critical for both strategies; (2) If data transfer changes the ownership of the chunk (and upcoming requests go to the destination server when the data transfer starts), a server with a long primary queue should not be the destination; (3) The batch strategy (with the move-back step)
achieves a similar throughput comparing to the batch strategy without the move-back step with appropriate design choices.

- We compare the latency of requests between the randomized strategy and the batch strategy without the move-back step. We see that the batch strategy achieves a smaller latency than the randomized strategy. We observe that the batch strategy sacrifices (increases) the medium latency to reduce the maximum latency when there is no speedup. Once the batch strategy has 2 speedup, both the medium and maximum latency significantly decrease.

5.2 Evaluating assumptions and estimate parameters

In this section, we aim to evaluate the assumption of request processing and the data transfer model (used by the theoretical analysis in Chapter 4) and estimate data-transfer latency and bandwidth in real-world systems. We conduct experiments on a real-world key-value store — FoundationDB [136]. FoundationDB is a transactional and distributed key-value store widely used in cloud services (e.g., iCloud and Snowflake). We use the Yahoo! Cloud Serving Benchmark (YCSB) [43] to generate the workload.

5.2.1 Evaluating request processing

Recall that in Chapter 4, we divide the timeline into slots. Each time slot represents the time of servicing one request. In each slot, a server consumes at most 1 request per time slot (m is the number of servers). In this section, we estimate the request service time and the number of requests a server can process within a single service time. Recall that our theoretical analysis in Chapter 4 does not distinguish read and write requests. In the evaluation, we estimate the service time of four operations: read, scan (read a key range), insert, and update. In the simulation, we do not distinguish read and write requests, but we set reasonable parameters for both read and write requests.
<table>
<thead>
<tr>
<th></th>
<th>WorkloadA</th>
<th>WorkloadB</th>
<th>WorkloadC</th>
<th>WorkloadD</th>
<th>WorkloadE</th>
<th>WorkloadF</th>
</tr>
</thead>
<tbody>
<tr>
<td>Read</td>
<td>0.554 ms</td>
<td>0.466 ms</td>
<td>0.458 ms</td>
<td>0.457 ms</td>
<td>-</td>
<td>0.504 ms</td>
</tr>
<tr>
<td>Scan</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>0.492 ms</td>
<td>-</td>
</tr>
<tr>
<td>Insert</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>3.822 ms</td>
<td>4.26 ms</td>
<td>-</td>
</tr>
<tr>
<td>Update</td>
<td>4.232 ms</td>
<td>4.324 ms</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>3.72 ms</td>
</tr>
</tbody>
</table>

Table 5.1: The minimal time span (milliseconds) of processing a single request when running the default YCSB workloads. The request includes read, scan, insert, and update.

**Experimental design**

In the experiment, we run a single FoundationDB process as a proxy to estimate the service time of each storage server in the cluster. We estimate the service time by measuring the minimal time span of processing requests. We use default YCSB workloads to generate requests. In each workload, read, scan, insert, and update operations are mixes in a specific ratio and requests are generated following a specific distribution. We conduct experiments for all six default YCSB workloads to estimate the service time in different scenarios comprehensively. For each workload, we issue 100,000 requests.

Further, we evaluate the number of requests that can be processed within the service time of a single request. We conduct experiments for the six default YCSB workloads. For each workload, we gradually increase the load (increasing the number of requests sent by YCSB clients per second and increasing the number of YCSB clients) until the observed throughput reaches the maximum. When we observe the throughput, we run the workload for a sufficiently long time until the throughput reaches stability. For each test, we record the observed maximum throughput. We multiply the maximum throughput and the time of processing a single request to estimate the number of requests a single server can complete within a single service time.
#reqs/s  980   7322  14826  13872  3628  966

Table 5.2: Throughput of the default YCSB workloads. Each value is the number of requests processed per second in a particular workload.

**Experimental results**

In Table 5.1, we see that the minimal time for request processing is 0.5 milliseconds (ms) for read requests and 5 ms for write requests. We estimate that the service time of requests is 0.5 ms for read requests and 5 ms for write requests.

Further, we notice that multiple requests can be processed during the service time of a single request by a single process of FoundationDB. Table 5.2 shows that the maximum throughput of the read-heavy workload (WorkloadB and WorkloadC and WorkloadD and WorkloadE) is about 10000 requests per second and the maximum throughput of the write-heavy workload (WorkloadA and WorkloadF) is about 1000 requests per second. According to Table 5.1, the service time of the read request is about 0.5 ms, and the service time of the write request is about 5 ms. Therefore, we estimate that a single FoundationDB process can complete approximately 5 requests during a single service time.

A single FoundationDB process can complete multiple requests in the service time of processing a single request, mainly because the storage server of FoundationDB services the requests in batch to support the multi-version concurrency control [17] efficiently. Therefore, we introduce a new parameter in addition to our theoretical model: *concurrency p*. A single server can process *p* requests when servicing a single request, aka those *p* requests are processed concurrently. Note that in Chapter 4, we assume *p* = 1, but the theoretical results hold for any constant value of *p*. In the evaluation, we estimate the single-server concurrency *p* = 5. We will use this value of *p* in our simulation.
5.2.2 Evaluating data-transfer model

Recall that in Chapter 5, we assume a data-transfer model, where each server can be involved in 1 data transfer at a time, and a data transfer has a specific bandwidth to move data — moving $s$ chunks takes the same time as moving 1 chunk while moving $s + 1$ chunks takes twice the time as moving $s$ chunks. This subsection evaluates the data-transfer model. Our experimental platform is a FoundationDB cluster running on Amazon Kubernetes Service. We develop experimental components in FoundationDB to measure the runtime statistics and customize workloads to mimic real-world applications.

Experimental design

We set up a FoundationDB cluster on the Amazon Kubernetes Service with 70 storage servers and one data distributor managed by the FoundationDB Operator. Each storage server runs in a dedicated Kubernetes pod with Amazon EBS storage. The RocksDB [26] is used as the storage engine.

The storage layer of the FoundationDB implementation that supports snapshots, including RocksDB, works as follows. The keys (values) are persistent in RocksDB checkpoints. Each checkpoint is stored as a physical file on the disk. Each key is persistent in one checkpoint file. A data transfer from server $i$ to server $j$ is implemented as fetching and transmitting checkpoint files from server $i$ to $j$.

Our case study runs the realistic workload using the Yahoo! Cloud Serving Benchmark (YCSB) [43], with an extension that client requests have a 90/10 Read/Write ratio, which mimics the pattern in real-world applications. Since the data distributor issues a data transfer and monitors it until the data chunks complete moving, we measure the data transfer latency inside the data distributor. Hence, the latency measurement is accurate without needing clock synchronization between storage servers. When recording a data transfer, we collect the number of bytes in the moved data chunks, its end-to-end latency, the source
storage server, and the destination storage server. We run the workload for hours to collect sufficient data points.

**Experimental results**

(a) One standalone data transfer with increasing total data sizes  
(b) Increasing numbers of data transfers from multiple servers to the same destination  
(c) Increasing numbers of data transfers from the same source to multiple servers

Figure 5.1: 50%, 75%, 90%, and maximum latencies of data transfer under different settings.

We compare the data-transfer latencies under different settings: (1) moving chunks with different total data sizes from one server to another server, (2) receiving chunks in multiple data transfers from different servers at the same destination server, and (3) sending chunks in multiple data transfers from the same source server to different servers.

Figure 5.1a presents the latency of standalone data transfers with different total sizes. We see that the p90 latency does not increase within the data size of 500MB, mainly because the network bandwidth between storage servers is approximately above 1GB per second, which is larger than the total size. Thus, the latency is dominated by the CPU overhead of the storage server to prepare for the data transfer.

For multiple data transfers to the same destination server or from the same source server in Figures 5.1b and 5.1c, we can see that the latency increases roughly linearly with an increasing number of data transfers. The storage server can only process the data transfers sequentially, which is consistent with our data-transfer model in Chapter 4.
We observe the standalone data-transfer latency, which occurs when no other data transfer is on the same source server and the same destination server. Figure 5.2 presents the normalized frequency of latency of standalone data transfers. We observe that most data transfers are completed within 0.5 seconds. Thus, we estimate that each data transfer takes 0.5 seconds to complete.

Above all, we estimate that a data transfer can move at most 500MB data, and a data transfer takes 0.5 seconds to complete.

### 5.2.3 Estimating data-transfer latency and bandwidth

In the simulation, the data-transfer latency is the number of iterations that the data transfer requires. Since an iteration corresponds to the period of processing one request on a storage server, we estimate the data-transfer latency of the simulation by

\[
\text{Data-transfer latency (iterations)} = \frac{\text{Data-transfer time span}}{\text{Service time}}
\]

Since the estimated service time of the request is between 0.5 ms and 5 ms. Since we estimate that each data transfer takes 500 ms to carry at most 500 MB data, we decide
\( s \in [100, 1000] \) as the data-transfer latency in our simulation. Since a data transfer can move at most \( s \) chunks, the chunk size is between 0.5 MB and 5 MB, which is reasonable.

### 5.3 Simulating workloads

In this section, we develop three workloads for the simulation. Our simulation is a series of iterations. Each iteration corresponds to the period of processing a single request, as the time slot defined in Chapter 4. Recall that the previous section introduces the single-server concurrency \( p \), which indicates that \( p \) requests can be processed concurrently by a single server in real-world systems. Thus, a cluster can consume at most \( mp \) requests per iteration in simulation, where \( m \) is the number of servers. Our workloads aim at fully utilizing the capacity of the cluster. Thus, the workload issues \( mp \) requests in each iteration, and each chunk is requested for at most \( p \) times. In this section, we develop three workloads: *Adversary workload*, *Zipfian workload*, and *Cloud workload*. In particular, the Cloud workload is generated by tracing the request sequence in a real-world application.

#### 5.3.1 Adversary workload

In the Adversary workload, the client repeatedly issues requests to the same group of chunks. In particular, the Adversary arbitrarily selects \( m \) chunks as a group at the beginning of the simulation. The client repeatedly sends requests to the same group of chunks throughout the simulation. The adversary workload is the most skewed workload of the three developed workloads — a small number of chunks are accessed by all requests.

#### 5.3.2 Zipfian workload

In the Zipfian workload, the number of requests to chunks follows the Zipfian distribution. The probability density for the \( k \)-th most frequently accessed chunk is

\[
P(k) = \frac{k^{-a}}{\zeta(a)}
\]
for integer $k$ and $k \geq 1$ and $\zeta$ is the Riemann Zeta function. The Zipfian workload is often considered an input in the research of tolerating the skewed workload in distributed key-value stores [9, 16, 22, 29, 43, 79, 87, 133]. We use an existing tool of NumPy to generate the population of chunks. We select $a = 2$ and the total number of chunks is 1000. The client sends the requests in iterations, satisfying: (1) each iteration has $mp$ requests; (2) each chunk is requested for at most $p$ times per iteration. Once all requests are sent, the client sends the same set of requests again.

![Figure 5.3: Request frequency of chunks in the Zipfian workload](image)

Figure 5.3 demonstrates the request frequency of chunks in the Zipfian workload. In the Zipfian workload, a small group of chunks has most of the requests.

### 5.3.3 Cloud workload

The Cloud workload is generated by tracing requests in real-world cloud applications. The application has a scan-heavy workload mixed with write requests. A scan is a request to read a range of keys. We run the trace for a sufficiently long time. We record the key range of scan requests and the key accessed by write requests.
In the trace, we collect the keys (ranges) of requests. To get the access frequency of chunks, we need to map the keys (ranges) to chunks. In practice, a key-value store often uses a prefix tree (e.g., LSM-trie [131]) to partition the key space into chunks. Since we are interested in data-placement strategies rather than data-partitioning methods, for simplicity, we assign a request to a chunk according to the prefix of the requested key (range). If a request is a write, we map it to a chunk according to the prefix of the key. For each scan, we observe that the prefix of the begin key and the end key are the same. Thus, we assign a scan request according to the prefix of the begin key of the range.

We first choose a prefix length and then group the requests into chunks. For each chunk, we count the number of requests. Since all traced keys with the same prefix belong to the same chunk, the number of chunks accessed in the trace is impacted by the prefix length. A larger prefix length leads to a larger number of chunks accessed in the trace. In our traced requests, the number of chunks is about 1000 if we select the prefix length as 6. When we select 8 as the prefix length, the number of chunks becomes about 10000. Finally, the client sends the requests in iterations, satisfying: (1) each iteration has \( mp \) requests; (2) each chunk is requested for at most \( p \) times per iteration.

Figure 5.4: Request frequency of chunks in the Cloud workload with different numbers of requested chunks
Figure 5.4a shows the request frequency by the Cloud workload which accesses 1000 chunks, and Figure 5.4b presents the request frequency by the Cloud workload with 10000 accessed chunks. We see that for both cases, the distributions are similar to the Zipfian distribution — most requests access to a small portion of keys. We see that the distribution of 1000 chunks is much more skewed than the distribution of 10000 chunks. In the distribution of 1000 chunks, there is a cliff between the 200 most frequently accessed chunks and other chunks, which is mainly because the requests for the case of 1000 chunks are more concentrated than that of 10000 chunks. We will use the Cloud workload with 1000 accessed chunks in the evaluation section (Section 5.5).

5.4 Simulation design

In this section, we introduce a framework for simulating distributed key-value stores. Further, we build the data-placement strategies based on the framework, which will be used in the evaluation section (Section 5.5).

5.4.1 Simulation framework of distributed key-value store

This subsection introduces a set of rules which capture key features of distributed key-value stores. The simulation must obey the rules when running different data-placement strategies.

In the simulation, the timeline is divided into iterations. Each iteration corresponds to the period of processing a single request, as the time slot defined in Chapter 4. Initially, chunks are assigned to servers, and each server owns some chunks. In each iteration, a client issues requests to the chunks.

**Request processing** There are $m$ servers in the simulated cluster. When a request arrives at the cluster, the request immediately appears on the server that owns the requested chunk. Each server has a queue that can buffer at most $q$ requests. When a request arrives at a server, the request gets immediately dropped if the queue is full. According to Section 5.2.1,
a single storage server consumes at most $p$ requests per iteration. In addition, Chapter 4 used the speedup $x$ as the resource augmentation in the competitive analysis. So, in the simulation, a server can consume at most $xp$ requests per iteration with $x$ speedup.

**Request process order** In real-world key-value stores, the processing order of requests often follows the arrival order of the requests to guarantee correctness and consistency. In the simulation, the server processes the requests to the same chunk following the arrival order of the requests. This rule keeps when the chunk is moving. While a chunk is moving, existing requests to the chunk move along with the chunk, and these requests cannot be processed. Any new request to the chunk must wait for those existing requests of the same chunk to be complete. Thus, any request to a moving chunk cannot be processed until the chunk completes moving.

### 5.4.2 Implementations of data-placement policies

In this subsection, we build the randomized and batch strategies introduced in Chapter 4.

**The randomized strategy (Rand)** is a static data placement. Initially, the simulator randomly and independently allocates chunks to servers at the uniform. Once a chunk is allocated to a server, the server owns the chunk. During the runtime, any request to the chunk goes to the owner of the chunk. The ownership of each chunk does not change throughout the simulation.

**The batch policy (Batch)** combines the randomized policy and the data transfer. The batch policy uses the randomized policy to initialize the ownership of chunks. The ownership does not change throughout the simulation. Each server has two queues — a primary queue and a secondary queue. The primary queue handles any new request from the client. The secondary queue is used for the batch process. At any moment, the total length of the primary and the secondary queues should be at most $q$. When a request arrives
at either the primary or the secondary queue, the request gets immediately dropped if the total queue length is larger than $q$.

In the batch policy, on each server, if the primary queue has a request that has arrived for at least $6s \log_2 m$ iterations ($s$ is data-move latency and $m$ is the number of servers), the server triggers a batch process. The batch process starts when no ongoing batch process is triggered by the same server, and the number of requests in the primary queue is at least $s \log_2 m$. When a batch process starts, the source server puts all chunks that have requests in the primary queue into at most $\log_2 m$ batches, and each batch has at most $s$ chunks. If the number of requested chunks is more than $s \log_2 m$, the $s \log_2 m$ chunks with the most requests are selected to put in the batches. In the primary queue, the requests out of the batches are dropped. When making the batches, we walk through the chunks to add the chunks to batches. For each chunk, we select the batch with minimal requests to add the chunk. For each batch, a destination server is selected. The requests are added to the secondary queue when the chunks are moved to the destination server. When adding a request to the secondary queue, the request gets immediately dropped if the total queue length is $q$. Once the destination server completes all requests of a batch, the batch is sent back to the source server. Once all groups are back on the home server, the batch process ends.

The batch process without the move-back step ($Batch^*$) is similar to $Batch$, except (1) $Batch^*$ policy does not have the move-back step; (2) the chunk ownership changes by the data transfer of $Batch^*$ policy. When a chunk starts moving to the destination server, the owner of the chunk becomes the destination server, and upcoming requests to the chunk go to the destination server. Once all moved requests in a batch process are completed by the destination server(s), the batch process ends.
5.5 Evaluation

In this section, we conduct simulation experiments with different workloads to evaluate our proposed strategies in Chapter 4. We use the simulation of the strategies developed in Section 5.4. We set the simulation parameter according to the estimation from real-world systems in Section 5.2. We use the workloads built in Section 5.3 to generate input requests to the simulated strategies. Then, we conduct three experiments: (1) We compare the throughput between the randomized strategy and the batch strategy; (2) We compare the batch strategy and the batch strategy without the move-back step for various design options. (3) We compare the latency between the randomized strategy and the batch strategy without the move-back step.

Experimental design  In Section 5.4, we implemented three strategies: (1) Rand; (2) Batch; (3) Batch*. The Rand and Batch strategies are proposed in Chapter 4. The Rand strategy is the randomized strategy, and the Batch strategy is the randomized strategy with the data transfer. The Batch* strategy is a modified version of the Batch strategy, where Batch* does not include the move-back step.

In Section 5.3, we developed three workloads: (1) Adversary; (2) Zipfian; (3) Cloud. In the Adversary workload, all requests access to a small group of chunks throughout the experiment. In the Zipfian workload, the number of requests to chunks follows the Zipfian distribution. The Cloud workload is generated by tracing the workload from real-world cloud applications.

In our experiments, we run the three strategies against the three workloads. In each experiment, a workload generates requests in each iteration until the end of the experiment. When adding a request to a server, the request is rejected if the number of requests in the server queue is equal to $q = 30sp \log^2 m$, where $s$ is the data-move latency, $p$ is the single-server concurrency of request processing, and $m$ is the number of servers. In the experiment,
we set $m = 30$, $s \in [100, 1000]$, and $p = 5$, according to the parameter estimation in a real-world system in Section 5.2. We have $q \in [361163, 3611636]$. Note that the Batch strategy and the Batch* strategy use two queues. In this case, a request is rejected if the total length of both queues is equal to $q$. Each experiment runs for at least $10 \times q$ iterations, where the workload issues more than 21 million requests. For each experiment, we repeatedly run for 80 times.

### 5.5.1 Comparing throughput between the Rand strategy and the Batch strategy

In this subsection, we compare the throughput between the Rand strategy and the Batch strategy under the three workloads. We compare the Rand strategy and the Batch strategy under different speedups of request processing. We use the reject ratio as the proxy to
compare the throughput between the strategies. The reject ratio is defined as:

\[ \text{Reject ratio} = \frac{N_{\text{reject}}}{N_{\text{reject}} + N_{\text{accept}}} \]

where \( N_{\text{reject}} \) is the number of rejected requests and \( N_{\text{accept}} \) is the number of accepted requests in the experiment. We use the same queue size for the strategies in each comparison. We run our experiments by setting \( s = 100 \) and \( s = 500 \) respectively. For each strategy with a specific setting, we run 80 times and record the reject ratios of the 80 runs.

Figure 5.5 presents the mean, maximum, 90%, and 50% reject ratios of the 80 runs for each strategy with different speedups and values of \( s \) under the Adversary, Zipfian, and Cloud workloads. \textit{RandX} denotes the Rand strategy with a processing speedup of \( X \). \textit{BatchX} denotes the Batch strategy with a processing speedup of \( X \).

First, we see that the reject ratio is similar for \( s = 100 \) and \( s = 500 \) because the \( q = 30sp\log_2 m \) increases linearly along with \( s \). When the data transfer takes longer to complete, the available queue size gets larger accordingly. As a result, the reject ratio is not sensitive to \( s \) in our experiments.

Second, the result shows that the Batch strategy consumes almost all requests with a small speedup. Even with the Adversary workload, the Batch strategy with one speedup accepts at least 97% requests (0.03 reject ratio) for all 80 runs of the simulation. When we run the Batch strategy with 2 speedup, no reject appears in all 80 runs for both \( s = 100 \) and \( s = 500 \).

Third, the Batch strategy significantly outperforms the Rand strategy for all workloads. Under the Adversary workload, the Rand strategy with 4 speedup has a 0.1 reject ratio among the 80 runs. For the Zipfian workload, the Rand strategy with 3 speedup has a 0.03 reject ratio among the 80 runs. For the Cloud workload, the Rand strategy with one speedup has a 0.08 reject ratio among the 80 runs. On the other hand, the Batch strategy uses a
smaller speedup to consume all requests in all runs, compared to the Rand strategy for all
workloads.

In general, our experimental results indicate that data transfers are essential to consume
almost all requests under practical workloads, and the speedup for the Batch strategy to
consume all requests is small.

5.5.2 Comparing throughput between the Batch strategy and the
Batch* strategy under different design choices

This subsection compares the throughput between the Batch strategy and the Batch* strat-
egy (the strategy without the move-back step). During the comparison, it turns out that
different design choices (our theoretical analysis does not cover in Chapter 4) impact the
throughput, which causes different results for different design choices. Thus, we compare the
two strategies with different design choices and show that the Batch strategy and the Batch*
strategy achieve similar throughput when choosing appropriate design options, respectively.
For each strategy with a specific option, we run 80 times and record the reject ratios of the
80 runs.

Priority of consuming requests

Recall that the batch process has two queues. One is the primary queue which buffers new
requests sent by clients. The other is the secondary queue which buffers the requests moved
by data transfer. Which queue should be processed with higher priority to minimize the reject
ratio when both queues have requests? Figure 5.6 compares four options for consuming
requests: In each round, (1) SecondaryOnly uses all processing time to consume the
secondary queue; (2) SecondaryFirst consumes the primary queue only if the secondary
queue is empty; (3) Isolated option spends equal time on consuming the primary queue
and the secondary queue; (4) PrimaryFirst option consumes the secondary queue only if
the primary queue is empty.
Figure 5.6: Mean, maximum, 90%, and 50% reject ratios of the Batch strategy and the Batch* strategy under the Adversary workload, the Zipfian workload, and the Cloud workload for different options of request consuming. The SecondaryOnly option denotes the strategy that only consumes the request in the secondary queue. The SecondaryFirst option consumes the primary queue only if the secondary queue is empty. The Isolated option spends a half busy time consuming the primary queue and a half busy time consuming the secondary queue. The PrimaryFirst option consumes the secondary queue only if the primary queue is empty. \( s = 100 \).

As shown in Figure 5.6, for both the Batch strategy and the Batch* strategy, the SecondaryFirst option achieves the lowest reject ratio for all workloads, except for the case of the Batch* strategy with the Adversary workload. On the other hand, the PrimaryFirst option achieves the largest reject ratio among all cases. The simulation result indicates that
Figure 5.7: Mean, maximum, 90%, and 50% reject ratios of the Batch strategy and the Batch* strategy under the Adversary workload and the Zipfian workload for different design options of destination selection for data transfers. The 2Q option denotes the strategy which selects the destination server with the minimal length of the secondary queue when moving data. The Q option denotes the strategy which selects the destination server with the minimal total length of both queues when moving data. $s = 100$.

Consuming the secondary queue in time is critical. With the PrimaryFirst option the server almost dedicates to processing the request in the primary queue. As a result, the request in the secondary queue is not processed in time, which blocks the following batch processes. Then, primary queues of hot servers get long quickly, eventually leading to many rejections. For example, for the Batch strategy with the PrimaryFirst option under the Adversary workload, all rejections are due to a long primary queue.

As a result, we use the SecondaryFirst option by default for both the Batch strategy and the Batch* strategy in our experiments.

**Destination selection of data transfer**

When conducting a data transfer, a server is selected as the destination server to move in the data. How to choose the destination server? Figure 5.7 compares two options: (1) $2Q$ chooses the server with the minimal secondary queue length; (2) $Q$ selects the server with the minimal total queue length.
Figure 5.7 compares the options under the Adversary and Zipfian workloads. Figure 5.7 does not contain the result of the Cloud workload since both options have no rejection. For the Batch strategy, selecting the destination with the minimal secondary queue length to move chunks has a lower reject ratio than selecting the minimal total queue length to move chunks. We suspect the reason is that the data transfer of the Batch strategy does not change the ownership of chunks. A hot server may still be hot after the server conducts the batch process. Therefore, a hot server may repeatedly trigger batch processes. In a batch process, when selecting the destination server by choosing the minimal total queue length, the data transfer can choose a server with a longer secondary queue length than that of selecting the minimal secondary queue length. As a result, the batch process takes a longer time to complete. Then, the following batch process triggered at the same server may be blocked, which leads to more rejections on the primary queue. For example, when running the Batch strategy against the Adversary workload, we observe that the number of rejections by the primary queue for the option Q is more than 10 times larger than that of the option 2Q. Moreover, we see that more than 87% of the rejections are due to a long primary queue (i.e., the primary queue is longer than the secondary queue when the reject happens).

On the other hand, for the Batch* strategy, selecting the destination with the minimal total queue length to move chunks has a lower reject ratio than selecting the minimal secondary queue length to move chunks. We suspect the reason is that the data transfer of the Batch* strategy changes the ownership of chunks. When a chunk starts moving to the destination server, upcoming requests of the chunk go to the destination server. If the data transfer chooses the destination without considering the primary queue length, the primary queue of the destination server may get long quickly, which causes many rejections. For example, when running the Batch* strategy with the option Q under the Adversary workload, 10.5% rejections are due to the long primary queue. On the other hand, the portion becomes 88.4% for the option 2Q, which is significantly larger than the option Q.
Figure 5.8: Mean, maximum, 90%, and 50% latencies of the Batch* strategy and the Rand strategy under the Adversary workload, the Zipfian workload, and the Cloud workload with different speedups and different values of $s$. RandX denotes the Rand strategy with a processing speedup of $X$. Batch*X denotes the Batch* strategy with a processing speedup of $X$. $s = 100$ and $s = 500$.

The simulation result shows that the Batch strategy with the option 2Q and the Batch* strategy with the option Q achieve a relatively small reject ratio. Further, the reject ratios for the two cases are close. As a result, we use the option 2Q and the option Q as the default options for the Batch strategy and the Batch* strategy in our experiments, respectively.

5.5.3 Comparing latency between the Rand strategy and the Batch* strategy

Section 5.5.1 showed the significance of data transfers to consume almost all requests. Is the data transfer essential to achieve a small response time (i.e., request latency) when using the infinite queue size? In this subsection, we compares the Rand strategy and the Batch* strategy with the infinite queue size under the Adversary, Zipfian, and Cloud workloads. For
each experiment, we do not reject any request. We run our experiments by setting $s = 100$ and $s = 500$ respectively. We run each strategy with a specific setting 80 times and record the reject ratios of the 80 runs.

Figure 5.8 presents the mean, maximum, 90%, and 50% latencies of the 80 runs for each strategy with different speedups and values of $s$ under the Adversary, Zipfian, and Cloud workloads. $\text{Rand}^X$ denotes the Rand strategy with a processing speedup of $X$. $\text{Batch}^X$ denotes the Batch* strategy with a processing speedup of $X$.

First, our experimental results show that the Batch* strategy with 1 speedup has a larger medium latency, but the tail of latency is significantly shorter compared to the Rand strategy running at 1 speedup. For example, for the Adversary workload with $s = 100$, the median latency (p50) of the Rand strategy with 1 speedup is 0, which is significantly smaller than that of the Batch* strategy with 1 speedup. However, the 90% percentile latency (p90) of the Rand strategy with 1 speedup is 347984, much larger than that of the Batch* strategy with 1 speedup (48358). Even for the Rand strategy with 4 speedup, the maximum latency is 144466, significantly larger than that of the Batch* strategy with 1 speedup (76940).

Second, the latency of the Batch* strategy gets significantly shortened when it has 2 speedup, where the maximum latency reduces to 3246, and the p90 latency reduces to 0. On the other hand, the Rand strategy with 2 speedup achieves 361165 maximum latency and 213921 p90 latency. We have similar trends for all three workloads and for both $s = 100$ and $s = 500$.

Overall, our experimental results indicate that the Batch* strategy sacrifices (increases) the medium latency to shorten the latency tail. This trade-off takes more advantage of processing speedup, resulting in significantly smaller latency compared to the Rand strategy.
Chapter 6

Conclusion

In this dissertation, we developed scheduling solutions that provide high throughput with small or bounded latency for various parallel and distributed systems.

First, we developed AMCilk, as an efficient runtime system to execute multiprogrammed parallel workloads. We showed that the importance of scheduler implementation to the application throughput and latency. Given various provably-good scheduling policies for different performance goals, applications achieve significant throughput boosts and latency reductions purely due to the responsive processor-reallocation implementation between parallel jobs in AMCilk.

Second, we built static schedules for real-time transmission on shared media. By carefully designing and composing collision-free mappings [12], the overhead of our schedules is asymptotically smaller than the state-of-the-art schedule [3]. As a result, our static schedules significantly shorten the schedule length, allowing applications to transmit messages at high throughput on noisy shared media.

Third, we theoretically compared various data-placement strategies against oblivious adversaries and showed that a well-designed strategy with a small queue size could consume almost all requests with constant speedup in the worst-case setting. We demonstrated the capacities and limits of various choices, thereby providing guidance for thinking about trade-offs. Further, we conducted a simulation to evaluate our theoretical model and showed that the strategies achieve high throughput with small speedup under realistic settings.
Above all, we presented scheduling algorithms, policies, and mechanisms to achieve high throughput with small latency in different parallel and distributed systems, including computing, network, and storage. Those results push the boundary toward high throughput in latency-sensitive infrastructures.
References


[27] Rohit Chandra, Leo Dagum, David Kohr, Ramesh Menon, Dror Maydan, and Jeff McDonald. Parallel programming in OpenMP. Morgan kaufmann, 2001.


[124] Li Wang, Yiming Zhang, Jiawei Xu, and Guangtao Xue. {MAPX}: Controlled data migration in the expansion of decentralized {Object-Based} storage systems. In 18th USENIX Conference on File and Storage Technologies (FAST 20), pages 1–11, 2020.


