Provably efficient adaptive scheduling of parallel jobs on multiprocessors

He, Yuxiong

2008

He, Y. (2008). Provably efficient adaptive scheduling of parallel jobs on multiprocessors.
Doctoral thesis, Nanyang Technological University, Singapore.

<https://hdl.handle.net/10356/60519>

<https://doi.org/10.32657/10356/60519>

2/1

Provably Efficient Adaptive Scheduling of Parallel Jobs on Multiprocessors

by
Yuxiong He

Submitted to Computer Science Program, Singapore-MIT Alliance
School of Computer Engineering, Nanyang Technological University

in partial fulfillment of the requirements for the
degree of

Doctor of Philosophy

at the

Singapore-MIT Alliance

January 2008

Signature of Author .....

Yuxiong He

Singapore-MIT Alliance

Nanyang Technological University

Certified by .....

Assoc. Prof. Wen Jing Hsu

SMA Fellow, NTU

Thesis Supervisor

Certified by .....

Prof. Charles E. Leiserson

SMA Fellow, MIT

Thesis Supervisor

Accepted by .....

Prof. Tomas Lozano-Perez

SMA-CS Program Co-Chair, MIT

Accepted by .....

Assoc. Prof. Hwee Tou NG

SMA-CS Program Co-Chair, NUS

Abstract

This thesis presents feedback-driven adaptive algorithms for efficient scheduling of parallel jobs on multiprogrammed multiprocessors.

Multiprocessor scheduling is often structured in two levels, where a kernel-level OS allocator allots processors to jobs and a user-level thread scheduler schedules the ready threads of a job on the allotted processors. In the context of adaptive scheduling, the number of processors allotted to a particular job may vary during the job's execution, and the thread scheduler must adapt to these changes in processor resources. For overall system efficiency, the thread scheduler should provide parallelism feedback to the OS allocator to avoid the situation where a job is either allotted too many processors to use productively or it is allotted too few processors to complete without undue delay. Since the future parallelism of a job is generally unknown, however, how to provide a suitable parallelism feedback and allocate processors efficiently is a challenge. This thesis investigates adaptive scheduling algorithms, including adaptive thread schedulers, OS allocators, and the two-level schedulers that combine them. The schedulers operate in an online nonclairvoyant manner, oblivious to the future characteristics of the job.

For adaptive thread scheduling, I introduce A-GREEDY and A-STEAL, which provide history-based feedback about the job's parallelism without assuming the knowledge of the job's future parallelism. These thread schedulers provably complete the jobs in near-optimal time while guaranteeing low waste.

For OS allocator, I present and analyze RAD, which combines the space-sharing OS allocator Dynamic Equipartitioning with the time-sharing round-robin algorithm. When using instantaneous parallelism as feedback, RAD achieves 3-competitiveness with mean response time for batched parallel jobs. It offers the best competitive ratio to date.

For two-level scheduling, I present G-RAD and W-RAD. Both achieve $O(1)$ -competitiveness with respect to makespan and mean response time for non-batched jobs and batched jobs respectively. They are the first nonclairvoyant schedulers to guarantee provable efficiency, fairness, and minimal overhead.

The performance of my schedulers has been tested via simulation. For individual parallel jobs, A-STEAL provides almost perfect linear speedup across a variety of processor availability profiles for jobs with sufficient parallelism. Based on my experiments, for arbitrary job sets, the makespan produced by G-RAD is no more than 1.39 times the optimal on average; while for batched job sets, the mean response time is no more than 2.37 times the optimal on average.

In addition to specific algorithms, I present general techniques for developing makespan-efficient two-level scheduling algorithms, and I elaborate analytical methods for the proof of the mean response time.

In addition to scheduling homogeneous processing resources, I also introduce a new job model and an adaptive scheduler for systems with functionally heterogeneous resources. The scheduler yields the best deterministic online nonclairvoyant algorithm with respect to makespan. In batched setting, it is $(4K+1-4K/(|\mathcal{J}|+1))$ -competitive for mean response time, where K denotes the number of categories of heterogeneous resources and $\mathcal{J}$ denotes the job set.

Acknowledgments

First and foremost I would like to thank my advisors, Prof. Hsu Wen Jing and Prof. Charles E. Leiserson for their guidance and support over these four years.

Prof. Hsu is a wonderful advisor. He always gives me freedom to explore in my interested areas, and provides me guidance during the process. He offers me numerous support in my PhD research — ideas, knowledge, time, resources, financial assistance, and about everything I need. He not only teaches me how to do research, but also shares many of his valuable experiences in life.

Prof. Leiserson shows me a great model of a computer scientist. He has a terrific sense of grand research directions. At the same time he knows every single detail on how to use LaTeX, Power Points, etc. I asked him the secret of keeping such a boundless enthusiasm on research. Now, I believe what he believes: “One should have the passion and belief that you can change the world”.

Much of the work in this thesis could not be done without the collaboration of the research teams at both NTU and MIT. The team members include Fang Hui, Hongyang, Jianyang, Shin Yee, Yahong in NTU, and Angelina, Jeremy, Jim, John, Kunal, Vicky at MIT.

I would like to specially thank Kunal Agrawal of CSAIL MIT. Kunal initiated the original work on adaptive thread scheduling, which has led to fruitful collaborations with her. During my visit to MIT, we had many enjoyable conversations and discussions on research and probably everything in life. Without her, I could never have adapted to the new environment so quickly and joyfully.

I would also like to specially thank Hongyang Sun of NTU. Hongyang is a junior PhD student in my research team, and he is a wonderful collaborator. His effort and dedication inspired several important discoveries and improvements on our joint work. He is also a good friend and a real gentleman. I always steal his boiled water, and he just keeps filling the water cooker.

I would like to thank my family for their support. Even though my parents are rather far away from my universities, their care, love and encouragement is always around me. I'd like to say many thanks to my boyfriend Huaning for his love and

tolerance. When I am in trouble, he is the one backing me up. When I get lost, he is the one helping me sort things out. When I occasionally feel bored, he is the one bringing me all kinds of fun and surprise. Taking PhD is exciting but never easy. He is the one making my PhD life much enjoyable and colorful.

Many thanks to my good friends, some of whom I shall mention in random order Shaoying, Wang Miao, Cuiyang, Chen Qing, Helen, and Lan. We have wonderful times together sharing our happiness, sadness, excitement and disappointment. They have taught me many things I can't learn from textbooks. Their friendship, care, and encouragement always surround me and help me to move forward.

Finally, but not least importantly, I'd like to thank the Singapore-MIT Alliance for funding my PhD study.

Contents

Abstract	i
Acknowledgments	iii
List of Figures	ix
Table of Notations	1
1 Introduction	1
1.1 Adaptive Scheduling	1
1.2 Formulation of Scheduling Problems	7
1.3 Statement of Results	10
1.4 Overview of Thesis	14
2 Adaptive Thread Schedulers	15
2.1 Assumptions and Results	16
2.2 Adaptive Greedy Algorithm	18
2.3 A-GREEDY Trim Analysis for Unit Quanta	19
2.4 A-GREEDY Trim Analysis of the General Case	27
2.5 Adaptive Data-Parallel Scheduling	31
2.6 Adaptive Work-stealing Algorithm	33
2.7 A-STEAL Time Analysis	36
2.8 A-STEAL Waste Analysis	43
2.9 Interpretation of the Bounds	44
3 The RAD OS Allocator	47
3.1 The RAD Algorithm	47
3.2 Properties of Squashed Sum	48
3.3 Mean Response Time Lower Bounds	51
3.4 Mean Response Time of I-RAD	53

4	Two-Level Adaptive Scheduling with History-based Feedback	60
4.1	Makespan of G-RAD	61
4.2	Mean Response Time of G-RAD for Batched Jobs	63
4.3	W-RAD Performance	78
4.4	Two-level Scheduling System with RAD	80
5	Empirical Results	85
5.1	Simulation Setup	87
5.2	Time Experiments	90
5.3	Waste Experiments	92
5.4	Time-Waste Experiments	94
5.5	Utilization Experiments	97
5.6	Makespan Experiments	98
5.7	Mean Response Time Experiments	101
5.8	Load Experiments	101
5.9	Proactive RAD Experiments	102
6	Techniques for Analyzing Two-level Adaptive Scheduling Algorithms	106
6.1	Desirable Properties of Thread Schedulers	106
6.2	Properties of OS Allocators	109
6.3	Makespan-Efficient Two-level Schedulers	111
6.4	Techniques for Response Time Analysis	113
7	Scheduling on Heterogeneous Resources	121
7.1	The K -Resource Scheduling Model and Algorithm	122
7.2	The K-RAD Algorithm	124
7.3	K-DAG Makespan Lower Bounds	125
7.4	K-RAD Makespan Analysis	129
7.5	K-DAG Mean Response Time Lower Bounds	132
7.6	K-RAD Mean Response Time Analysis	132
8	Literature Review	139
8.1	Common Techniques for Scheduling	139
8.2	Scheduling Serial Jobs for Makespan and Mean Response Time	142
8.3	Thread Scheduling for Parallel Jobs	143
8.4	Adaptive Job Scheduling	145

9 Discussions and Conclusion 148

9.1 Summary 148

9.2 Future Work 149

9.3 Concluding Remarks 151

List of Publications 152

Bibliography 154

List of Figures

1-1	An example that compares adaptive scheduling algorithm with and without parallelism feedback.	4
2-1	Pseudocode of the adaptive greedy thread scheduler A-GREEDY. . . .	19
2-2	Pseudocode of the adaptive work-stealing thread scheduler A-STEAL. . . .	35
5-1	The DAG of a square-wave job used in the simulation.	88
5-2	Comparing the (true) mean availability $\bar{P}$ with the trimmed availability $\widetilde{P}$	91
5-3	Comparing the theoretical and practical waste of A-STEAL for various values of the utilization parameter δ	93
5-4	Measuring waste with varying parallelism.	93
5-5	Comparing the time and waste of A-STEAL against ABP when $P = 512$ and $\bar{P} = 30, 60$	95
5-6	Comparing the time and waste of A-STEAL against ABP when $P = 128$ and $\bar{P} = 30, 60$	96
5-7	Comparing the utilization of W-RAD and ABP+EQ.	99
5-8	Comparing makespan produced by G-RAD against the optimal offline algorithm.	100
5-9	Comparing mean response time produced by G-RAD against the optimal offline algorithm.	100
5-10	Measuring makespan with varying load.	103
5-11	Measuring mean response time with varying load.	103
5-12	Comparing makespan of Proactive RAD against the original RAD. . . .	104
5-13	Comparing mean response time of Proactive RAD against the original RAD.	104
6-1	Pseudocode of the Proactive A-GREEDY algorithm.	108
7-1	An example of K -DAG job model.	123

7-2 Pseudocode for the variant of the RAD OS allocator used in heteroge-
neous resource scheduling. 126

7-3 An example that shows any deterministic online scheduler is at best
 $(K + 1 - K/P_{max})$ -competitive with respect to makespan. 128

Table of Notations

Terminology	Notation
terms that are related to a job J	
work of J	$T_1(J)$
span of J	$T_\infty(J)$
release time of J	$r(J)$
completion time of J	$T(J)$
waste of J	$w(J)$
non-overflow waste of J	$\varpi(J)$
mean processor availability of J	$\bar{P}(J)$
trimmed mean processor availability of J (trimmed availability)	$\tilde{P}(J)$
number of satisfied time steps of J	$sat(J)$
allotment of J at quantum q	$a(J, q)$
desire of J at quantum q	$d(J, q)$
terms that are related to a job set $\mathcal{J}$	
total number of jobs in $\mathcal{J}$	$ \mathcal{J} $
total work of $\mathcal{J}$	$T_1(\mathcal{J})$
aggregate span of $\mathcal{J}$	$T_\infty(\mathcal{J})$
total satisfied time steps of $\mathcal{J}$	$sat(\mathcal{J})$
squashed work of $\mathcal{J}$	$sqw(\mathcal{J})$
squashed allotment of $\mathcal{J}$	$sqa(\mathcal{J})$
makespan of $\mathcal{J}$	$T(\mathcal{J})$
makespan of $\mathcal{J}$ produced by optimal offline scheduler	$T^*(\mathcal{J})$
total response time of $\mathcal{J}$	$R(\mathcal{J})$
total response time of $\mathcal{J}$ produced by optimal offline scheduler	$R^*(\mathcal{J})$
mean response time of $\mathcal{J}$	$\bar{R}(\mathcal{J})$
mean response time of $\mathcal{J}$ produced by optimal offline scheduler	$\bar{R}^*(\mathcal{J})$
other terms	
total number of processors	P
utilization parameter	δ
responsiveness parameter	ρ

Chapter 1

Introduction

As computers with multiple processors or multicore chips have proliferated, it has become more important than ever to ensure efficient execution of parallel jobs. Various degrees of success have been achieved on parallel job scheduling over the years. Few existing scheduling algorithms, however, offer both efficiency and fairness over a wide range of work loads. Moreover, in order to obtain analytical results, many of them require prior information about jobs, which may be difficult to obtain in practice. This thesis provides algorithmic solutions for the efficient scheduling of parallel jobs on multiprocessors without requiring any prior information about the jobs. Both theoretical and empirical evidences demonstrate that my scheduling algorithms not only execute jobs fast without wasting many processor cycles, but also guarantee to share resources fairly among competing jobs. My algorithms can be applied to schedule applications written in a wide range of parallel languages, which include data-parallel languages such as High Performance Fortran (HPF) [62], *Lisp [93], C* [115], NESL [16, 18], and ZPL [34], as well as control-parallel languages for high performance computing such as Cilk [20], Intel Thread Building Blocks (TBB) [42], Sun Fortress [106], Microsoft Task Parallel Library (TPL) [95], and Multilisp [71].

1.1 Adaptive Scheduling

Multiprocessors are often used for multiprogrammed workloads where many parallel jobs share the same machine. Since jobs can enter and exit the multiprocessor system at any time, the amount of resources available to a job may change during the job's execution. The intrinsic parallelism of a parallel job may be different over its execution, so the amount of resources required for a job can also vary. Intuitively, the

number of processors allotted to a job should also change according to the resource availability and requirement in order to make efficient use of a multiprocessor system. An **adaptive** scheduler can change the number of processors allotted to a job while the job is executing. This thesis focuses on the design and analysis of adaptive scheduling strategies for parallel jobs on multiprocessors. In this section, I present the background and potential benefits of adaptive scheduling and elaborate the main challenges for developing efficient adaptive scheduling algorithms.

Uniprocessor scheduling focuses on deciding which thread to run on the processor at any time step. Multiprocessor scheduling adds an additional dimension, not only deciding when to run a thread, but also where to run it. Thus, scheduling in multiprocessors implies a two-dimensional division of resources among competing jobs, both in time and in space. If jobs occupy disjoint processor resources, the schedulers are **space-sharing**; alternatively, if different jobs share the same processor resources at different times, the schedulers are **time-sharing**.

Scheduling parallel jobs on multiprocessors engenders yet another source of complexity, since each individual job can have concurrent threads. A scheduler must not only decide how to efficiently share multiprocessors among a number of competing jobs, but also decide how to coordinate threads in each individual job. Therefore, a scheduler for parallel jobs has two main responsibilities: to allocate processors to jobs, and to schedule the threads of a job to its allotted processors.

These scheduling functions can be achieved by either a single-level or a two-level approach [57]. The single-level approach combines the issues of processor allocation and thread scheduling, while the two-level approach decouples the two issues. A two-level scheduler includes a kernel-level **OS allocator**, which allots processors to jobs, and a user-level **thread scheduler** which maps the threads of a given job to the allotted processors. Compared with the single-level approach, the two-level approach removes the operating system overhead from each thread scheduling decision, and it allows for application-specific optimizations. In this thesis, I study two-level scheduling of parallel jobs.

Like earlier research on parallel job scheduling [15, 18, 54, 80, 111], I model the execution of a parallel job J as a dynamically unfolding directed acyclic graph (DAG). Each vertex represents a unit-time thread, while each edge represents a serial dependency between threads. Figure 1-1(a) shows two DAGs, which represent two parallel jobs J_1 and J_2 , respectively. A thread of a job becomes **ready** to be executed when all its parents have been executed. The **release time** $r(J)$ of the job J is the time

at which J becomes first available for processing.

Most prior work on thread scheduling for multithreaded jobs deals with **nonadaptive** scheduling [17, 18, 23, 31, 66, 111], where the OS allocator allots a fixed number of processors to the job for its entire lifetime. For jobs whose parallelism is unknown in advance and which may change during execution, this strategy may waste processor cycles [127], because a job with low parallelism may be allotted more processors than it can productively use. Moreover, in a multiprogrammed environment, nonadaptive scheduling may not allow a new job to start, because existing jobs may already be using most of the processors.

With adaptive scheduling [5] (called “dynamic” scheduling in many papers), the OS allocator can change the number of processors allotted to a job while the job is executing. To schedule a job adaptively, the parallel job needs to be **malleable**, allowing the number of the allotted processors to vary during its execution. The malleability of a parallel application is usually decided by the way it is written. With adaptive scheduling, new jobs can enter the system, because the OS allocator can simply recruit processors from the already executing jobs and allot them to the new job. Unfortunately, without adequate feedback, this strategy may cause waste, because a job with low parallelism may still be allotted more processors than it can productively use.

The solution we present is an adaptive scheduling strategy where the thread scheduler provides **parallelism feedback** to the OS allocator so that when a job cannot use many processors, those processors can be reallocated to jobs with ample need. Based on this parallelism feedback, the OS allocator adaptively changes the allotment of processors according to the availability of processors in the current system environment and the OS allocator’s administrative policy. Between regular intervals, called **scheduling quanta**, a thread scheduler estimates its job’s **desire**, which is the number of processors the job wants for next quantum, and provides it to the OS allocator. Based on the desire of all running jobs, the OS allocator follows its processor-allocation policy to determine the **allotment** of the job. We call the feedback mechanism a **request-allotment** protocol.¹

To compare and elaborate the adaptive scheduling algorithms with/without paral-

---

¹Formally, a request-allotment protocol comprises of two functions f and g . The function f represents the desire estimation algorithm of the thread schedulers. For each job J_i , the function f takes the past and current status of J_i as inputs, and returns the desire of J_i for the next quantum as output. The function g represents the scheduling policy of OS allocator. It takes jobs’ desire as inputs and returns jobs’ allotment in next quantum as outputs.

Figure 1-1: Comparing adaptive scheduling algorithm with and without parallelism feedback. Figure 1-1(a) shows DAGs of job J_1 and J_2 , which are released at time 0 and 1, respectively. Figure 1-1(b) shows a schedule generated by EQ-Simple-Greedy when executing J_1 and J_2 on 4 processors. Figure 1-1(c) shows a schedule generated by Adapt-Inst-Greedy when executing the same jobs on the same set of processors. For both Figures 1-1(b) and 1-1(c), the horizontal axis represents the scheduling time and the vertical axis represents the 4 processors on the system. The vertex at row p and column t represents the thread executed by processor p at the time step t . The shaded area with horizontal lines indicates the allotment of job J_1 , while the shaded area with grids indicates the allotment of job J_2 .

lelism feedback, I describe a sample algorithm for each of them, and demonstrate their performance in Figure 1-1. The algorithm without parallelism feedback, called EQ-

1.1. Adaptive Scheduling

5

Simple-Greedy, applies equipartitioning OS allocator and a simple adaptive greedy thread scheduler. equipartitioning evenly shares its processors among the jobs. The adaptive greedy scheduler simply maps ready threads of its job to the allotted processors in greedy manner. At any time step t where the job is allotted with α processors, if the number of ready threads is greater than α , the thread scheduler executes α of them; otherwise, the thread scheduler executes all the them.² The algorithm with parallelism feedback, called Adapt-Inst-Greedy, uses **instantaneous parallelism**, which is the number of processors the job can effectively use at the present moment, as parallelism feedback to the OS allocator. The OS allocator, called Adapt, simply provides a job whatever it requests, *i.e.*, a job's allotment is always equal to its desire.

Figure 1-1 compares EQ-Simple-Greedy with Adapt-Inst-Greedy and demonstrates the benefits of using parallelism feedback. Consider two malleable jobs J_1 and J_2 whose DAGs are shown in Figure 1-1(a). Figure 1-1(b) shows a schedule generated by EQ-Simple-Greedy during the execution of J_1 and J_2 on 4 processors. At time 0, job J_1 is released, and allotted with all 4 processors. At time 1, job J_2 is released, and equipartitioning allots each of J_1 and J_2 with 2 processors. This partition remains unchanged until time 9 where job J_2 completes. Let's look at time step 3. Job J_1 that has only one ready thread (labeled as 3) is allotted 2 processors, while job J_2 that has 3 ready threads (labeled as b , c , and d) is also allotted 2 processors. The processor 2 is therefore wasted. Apparently, with appropriate parallelism feedback, processor 2 can be better used by redistributing it from job J_1 to J_2 . This reveals the problem of an adaptive scheduler without parallelism feedback where a job with low parallelism may be allotted more processors than it can productively use.

Figure 1-1(c) presents a schedule of J_1 and J_2 produced by Adapt-Inst-Greedy with quantum length equal to 1. Let's look at time step 3. The instantaneous parallelism of job J_1 is 1 since it only has one ready thread with label 3. Similarly, the instantaneous parallelism of job J_2 is 3 since it has three ready threads labeled as b , c and d . Therefore, the desired numbers of processors for J_1 and J_2 are 1 and 3, respectively. Since the Adapt OS allocator makes a job's allotment equal to its desire, it allots 1 and 3 processors to job J_1 and J_2 respectively. We can see that the processor 2 that is wasted in Figure 1-1(b) at time step 3 is making useful work because the parallelism feedback provides extra information for OS allocator to make a better decision on partitioning processors among jobs. Regarding the effect on the overall performance, without parallelism feedback, the scheduler in Figure 1-

---

²A time step t denotes the time interval $[t-1, t)$.

1(b) completes two jobs using 10 time steps, while with parallelism feedback, the scheduler in Figure 1-1(c) completes two jobs using only 8 time steps. Figure 1-1 demonstrates that appropriate parallelism feedback can reduce waste of individual jobs and therefore benefit overall system performance.

After exploring the potential benefits of feedback-driven adaptive scheduling, I will now discuss the challenges on its request-allotment protocol. As shown in Figure 1-1, when the parallelism of jobs does not change as frequently as processor reallocation, simple and accurate ways exist based on measuring the instantaneous parallelism [45, 46, 68, 103, 134]. In this case that a job's parallelism does not change fast, instantaneous parallelism can predict the parallelism of a job for the entire upcoming quantum, and precisely capture the job's desired number of processors. Thus, the request-allotment protocol reduces to designing an OS allocator that allocates processors effectively.

Unfortunately, using instantaneous parallelism as feedback may cause gross misallocation of processor resources [119] when jobs' parallelism can change quickly. For instance, the parallelism of a job within a scheduling quantum may alternate between parallel and serial phases. The sampling of instantaneous parallelism at a scheduling event between quanta may lead the thread scheduler to request either too many or too few processors depending on which phase is active at the time of sampling, whereas a suitable request may be substantially different. Consequently, the job may either waste processor cycles or take too long to complete. Moreover, with a distributed thread scheduler, it can be costly just collecting a job's instantaneous parallelism. Therefore, a more robust desire-estimation strategy is required.

Another question on designing a two-level adaptive scheduling algorithm is how the OS allocator should partition the processors among the various jobs. Intuitively, an OS allocator can just provide whatever a job requests as Adapt does. The total desires for all jobs can be far larger than the total number of processors in the system, however. In such a case, OS allocator simply cannot satisfy the desire of every job. Moreover, for a two-level scheduler to possess other properties such as fairness among jobs, it must have an OS allocator with more sophisticated processor allocation policies.

This thesis focuses on solving the problems of how a thread scheduler estimates the desire of its job and how an OS allocator partitions the processors among jobs in order to guarantee fair and efficient scheduling of parallel jobs.

1.2 Formulation of Scheduling Problems

This section formalizes the job model, defines the scheduling model, and presents the optimization criteria for thread schedulers and two-level schedulers.

Our initial scheduling problem consists of a collection of independent jobs $\mathcal{J} = \{J_1, J_2, \dots, J_{|\mathcal{J}|}\}$ to be scheduled on a collection of P identical processors. If all jobs in the job set $\mathcal{J}$ are released at the same time, $\mathcal{J}$ is a **batched** job set, otherwise, it is a **nonbatched** job set. This thesis studies the scheduling problems on both batched and nonbatched job sets.

I model the execution of a parallel job J_i as a DAG such that $J_i = (V_i, E_i)$ where V_i and E_i represent the sets of J_i 's vertices and edges respectively. The **work** $T_1(J_i)$ of the job J_i corresponds to the total number of vertices in the DAG, *i.e.*, $T_1(J_i) = |V_i|$. Recall that Figure 1-1(a) shows DAGs of job J_1 and J_2 . The work of job J_1 and J_2 is equal to 14 and 11, respectively. Each edge $e \in E_i$ from vertex u to v represents a dependency between the two vertices. The precedence relationship $u \prec v$ holds if and only if there exists a path from vertex u to v in E_i . The **span** (or **critical-path length**) $T_\infty(J_i)$ corresponds to the number of vertices on the longest chain of the precedence dependencies. In Figure 1-1(a), the span of J_1 is equal to 8, and the span of J_2 is 7.

Our scheduling model assumes that time is broken into a sequence of equal-sized scheduling quanta $1, 2, \dots$, each of length L , where each quantum q includes the interval $[L \cdot q, L \cdot q + 1, \dots, L \cdot (q + 1) - 1]$ of time steps. The quantum length L is a system configuration parameter chosen to be long enough to amortize the time to reallocate processors among the various jobs and the time to perform various other bookkeeping for scheduling, including the time for the thread scheduler to communicate with the OS allocator, which typically involves a system call.

The OS allocator and the thread schedulers interact as follows. The OS allocator may reallocate processors between quanta. Between quantum $q - 1$ and quantum q , the thread scheduler of a given job J_i determines the job's desire $d(J_i, q)$. Based on the desire of all running jobs, the OS allocator follows its processor-allocation policy to determine the allotment $a(J_i, q)$ of the job under the requirement that $a(J_i, q) \leq d(J_i, q)$. Once a job is allotted its processors, the allotment does not change during the quantum.

A schedule $\chi = (\tau, \pi)$ of a job set $\mathcal{J}$ is defined by two mappings

$$\tau : \bigcup_{J_i \in \mathcal{J}} V_i \rightarrow \{1, 2, \dots, \infty\} ,$$

and

$$\pi : \bigcup_{J_i \in \mathcal{J}} V_i \rightarrow \{1, 2, \dots, P\} ,$$

which map the vertices of the jobs in the job set $\mathcal{J}$ to the set of time steps, and the set of processors on the machine respectively. A valid mapping must preserve the precedence relationships between tasks. For any two vertices $u, v \in V_i$ of the job J_i , if $u \prec v$, then $\tau(u) < \tau(v)$, *i.e.*, the vertex u must be executed before the vertex v . A valid mapping must also ensure that one processor can be assigned to only one job at any given time. For any two vertices u and v , both $\tau(u) = \tau(v)$ and $\pi(u) = \pi(v)$ are true if and only if $u = v$.

The scheduling algorithms can be categorized into three types — offline, online clairvoyant, and online nonclairvoyant algorithms. The **offline** algorithms [87, 94] possess complete information of its scheduled job set. The **online clairvoyant** algorithms [33, 37, 70, 79, 90, 98, 124] know the resource requirements of a job once it is released, but they must base their decisions only on jobs that have been released. This thesis investigates **online nonclairvoyant** scheduling, where future information on jobs' resource requirements and release times is unknown to the scheduling algorithm. The thread scheduler of a job operates in an online manner, oblivious to the future characteristics of the dynamically unfolding DAG.

The performance of a thread scheduler is measured by using job's completion time and waste, which are defined as follows.

Definition 1.1 The **completion time** $T_\chi(J_i)$ of a job J_i under a schedule χ is the time at which the schedule completes the execution of the job, *i.e.*, $T_\chi(J_i) = \max_{v \in V_i} \tau(v)$.

Definition 1.2 The **waste** $w(J_i)$ of a job J_i under a schedule χ is the total number of processor cycles allotted to but not used by the job, *i.e.*, $w(J_i) = \sum_{t=0}^{\infty} a(J_i, t) - T_1(J_i)$.

We need both completion time and waste to measure the performance of a thread scheduler. Minimizing only one of the metrics is trivial. For example, a scheduler that always requests one processor incurs no waste. Similarly, to minimize completion time

1.2. Formulation of Scheduling Problems

9

alone, a scheduler can always request a maximum number of processors. Achieving both short completion time and small waste implies that a thread scheduler not only perform well in terms of scheduling its own job, but that the unneeded processors can be repurposed to other needy jobs for better system utilization.

The performance of a two-level scheduler is measured by using makespan and mean response time, which are defined as follows.

Definition 1.3 The **makespan** of a given job set $\mathcal{J}$ under the schedule χ is the time taken to complete all jobs in the job set $\mathcal{J}$, i.e., $T_\chi(\mathcal{J}) = \max_{J_i \in \mathcal{J}} T_\chi(J_i)$.

In Figure 1-1(a), the completion time of job J_1 is 10 and that of J_2 is 9. The makespan of the job set $\mathcal{J} = \{J_1, J_2\}$ under this schedule χ_1 is $T_{\chi_1}(\mathcal{J}) = \max\{9, 10\} = 10$. Similarly, in Figure 1-1(b), the makespan of $\mathcal{J}$ under this schedule χ_2 is $T_{\chi_2}(\mathcal{J}) = \max\{8, 8\} = 8$.

Definition 1.4 The **response time** of a job J_i under a schedule χ is $T_\chi(J_i) - r(J_i)$, which is the duration between its release time $r(J_i)$ and the completion time $T_\chi(J_i)$. The **total response time** of a job set $\mathcal{J}$ under a schedule χ is given by $R_\chi(\mathcal{J}) = \sum_{J_i \in \mathcal{J}} (T_\chi(J_i) - r(J_i))$ and the **mean response time** is $\bar{R}_\chi(\mathcal{J}) = R_\chi(\mathcal{J}) / |\mathcal{J}|$.

In Figure 1-1(a), the response time of job J_1 is 10, which is the duration between its release time 0 and completion time 10, and the response time of J_2 is 8. The total response time of $\mathcal{J} = \{J_1, J_2\}$ under this schedule χ_1 is given by $R_{\chi_1}(\mathcal{J}) = 10 + 8 = 18$, and mean response time is $\bar{R}_{\chi_1}(\mathcal{J}) = 9$. Similarly, in Figure 1-1(b), the total response time of $\mathcal{J}$ under this schedule χ_2 is given by $R_{\chi_2}(\mathcal{J}) = 8 + 7 = 15$, and mean response time is $\bar{R}_{\chi_2}(\mathcal{J}) = 7.5$.

For most scheduling problems, no online nonclairvoyant algorithm can produce an optimal solution for all input instances. Arguably the most common method for evaluating the performance of an online algorithm is the worst-case ratio between the quality of the computed solution and that of the corresponding optimal solution. This method is called **competitive analysis** [29, 88, 125]. Let $T^*(\mathcal{J})$ denote the makespan of the job set $\mathcal{J}$ scheduled by an optimal offline scheduler, and $\chi(A)$ denote the schedule produced by an algorithm A for the job set $\mathcal{J}$. A deterministic algorithm A is said to be **c-competitive** if there exists a constant b such that

$$T_{\chi(A)}(\mathcal{J}) \leq c \cdot T^*(\mathcal{J}) + b$$

holds for the schedule $\chi(A)$ of any job set. A randomized algorithm S is said to be c -competitive if there exists a constant b such that

$$E[T_{\chi(S)}(\mathcal{J})] \leq c \cdot T^*(\mathcal{J}) + b$$

holds for every job set $\mathcal{J}$ where $E[T_{\chi(S)}(\mathcal{J})]$ represents the expected makespan of $\mathcal{J}$ produced by S . Thus, for each job set $\mathcal{J}$, a c -competitive algorithm is guaranteed to have the makespan (or the expected makespan) within a factor c of that incurred in the optimal clairvoyant algorithm (up to the additive constant b). Similarly, the concept of competitiveness can be applied to mean response time. Our goal is to find an algorithm with a competitive ratio as small as possible. Ideally, this competitive ratio should be a constant independent of any parameter of the input instance such as the total number of jobs and their individual characteristics.

1.3 Statement of Results

My PhD work investigates techniques and strategies for designing and analyzing adaptive scheduling algorithms for parallel jobs on multiprocessors. I explore new history-based feedback mechanisms in the request-allotment protocol for resource allocation. I also develop various adaptive scheduling schemes that offer both fairness and provable efficiency without entailing large overhead or requiring prior job information. Here, I summarize the main contributions of this thesis:

- Two adaptive thread schedulers, Adaptive-Greedy (A-GREEDY) and Adaptive-Steal (A-STEAL), which provide provably good history-based feedback about the job's parallelism. These thread schedulers complete job in near-optimal time while guaranteeing low waste. We introduce a new analytical technique called *trim analysis*.
- An online nonclairvoyant OS allocator, called RAD.³ When using instantaneous parallelism as feedback, RAD achieves 3-competitiveness for batched parallel jobs with respect to mean response time. It offers the best competitive ratio among all existing algorithms to date.
- Two online nonclairvoyant adaptive scheduling systems, G-RAD and W-RAD, can be derived by integrating RAD OS allocator with the A-GREEDY and A-STEAL thread schedulers, respectively. Both G-RAD and W-RAD achieve

---

³RAD stands for Round robin And Dynamic equi-partitioning, as will be defined shortly in this section.

$O(1)$ -competitiveness with respect to makespan and mean response time for nonbatched jobs and batched jobs, respectively. They are the first nonclairvoyant schedulers that ensure provable efficiency, fairness and minimal overhead.

- Simulation results that provide strong evidence that A-STEAL, G-RAD, and W-RAD will perform well in practice.
- Principles and techniques to design and analyze adaptive scheduling algorithms that ensure good performance on makespan and mean response time.
- A novel analytical model and an adaptive scheduler, called K-RAD, for heterogeneous resources with various functionalities. The scheduler is proven to be the best deterministic online nonclairvoyant algorithm with respect to makespan. In batched setting, K-RAD is $(4K + 1 - 4K/(|\mathcal{J}| + 1))$ -competitive for mean response time, where K denotes the number of categories of heterogeneous resources.

I will now elaborate the main results in detail.

Adaptive thread schedulers

The thesis presents two adaptive thread schedulers, A-GREEDY and A-STEAL, which provide history-based parallelism feedback to an OS allocator. A-GREEDY is a greedy thread scheduler suitable for centralized scheduling, where each job's thread scheduler can dispatch all the ready threads to the allotted processors in a centralized manner, as with data-parallel jobs. A-STEAL is a distributed thread scheduler, where each job is executed by decentralized *work stealing* [23, 32, 71, 116]. A-GREEDY and A-STEAL are joint work with Kunal Agrawal and Charles E. Leiserson of MIT and Hsu Wen Jing of NTU. They were originally presented in our papers [2–4].

Both A-GREEDY and A-STEAL guarantee not to waste many processor cycles while simultaneously ensuring that the job completes quickly. Instead of using instantaneous parallelism, A-GREEDY and A-STEAL provide parallelism feedback to the OS allocator based on the job's behavior in the previous quantum. We have analyzed these thread schedulers under stringent adversarial conditions, showing that they are robust under various system environments and allocation policies. To analyze the thread schedulers under our adversarial model, we have developed a new technique, called trim analysis, which can be used to show that the thread scheduler behaves well in the vast majority of time steps and only performs poorly in a few. Even though A-GREEDY and A-STEAL provide parallelism feedback using the past behavior of the job and I do not assume that job's future parallelism is correlated with its history of parallelism, my analysis shows that these algorithms schedule the

job well with respect to both waste and completion time.

RAD OS allocator

This thesis introduces an adaptive OS allocator RAD which combines the space-sharing job scheduling algorithm “Dynamic Equipartitioning” (DEQ) [103, 130] with the time-sharing round-robin algorithm. When the total number of jobs is no greater than the total number of processors, it applies DEQ as the OS allocator, allotting each job an equal number of processors unless a job requests less. When the total number of jobs is greater than the total number of processors, RAD applies the round-robin algorithm, ensuring that each job receives an equal slice of processing time.

The first two-level scheduler presented in this thesis, called I-RAD, applies RAD as OS allocator and instantaneous parallelism as feedback. The best previous mean-response-time bound for batched jobs by an online nonclairvoyant algorithm was $2 + \sqrt{3} \approx 3.73$, proved by Edmonds *et al.* [52]. I show that I-RAD is 3-competitive for batched jobs with respect to the mean response time, providing the best competitive ratio to date. This joint work with Hongyang Sun and Hsu Wen Jing was originally presented in [78].

Two-level adaptive scheduling with history-based feedback

This thesis presents two adaptive two-level schedulers, G-RAD and W-RAD, which use history-based feedback and RAD as the OS allocator. G-RAD, which couples RAD with A-GREEDY, is suitable for centralized thread scheduling, while W-RAD, which couples RAD with A-STEAL, is more suitable for scheduling in a distributed manner. Intuitively, if each job provides good parallelism feedback and makes productive use of allotted processors, a good OS allocator can ensure that *all* jobs make good progress. The analysis of G-RAD and W-RAD affirms the intuition.

Based on the “equalized allotment” scheme for processor allocation, and by using the utilization of the past quantum as feedback, I show that G-RAD and W-RAD are provably efficient with respect to both makespan and mean response time. More precisely, both of them achieve $O(1)$ -competitiveness with respect to the makespan for job sets with arbitrary release times and $O(1)$ -competitiveness with respect to the mean response time for batched job sets where all jobs are released simultaneously. As mentioned in Section 1.1, using instantaneous parallelism as feedback has its limitation when the parallelism of a job changes faster than the quantum length. G-RAD and W-RAD, which use history-based feedback, are robust to any parallelism profile and workload. Unlike many previous results [38, 84, 85, 102, 110, 118, 131], G-RAD and W-RAD do not use any prior information of jobs. Moreover, because

the quantum length can be adjusted to amortize the cost of context-switching during processor reallocation, they provide effective control over the scheduling overhead and ensure efficient utilization of processors. G-RAD and W-RAD were originally presented in [76, 77].

Empirical results

In addition to theoretical analysis, I have tested the efficiency of these schedulers empirically. I built a discrete-time simulator to evaluate the performance of A-STEAL, W-RAD, and G-RAD. With respect to the performance of thread schedulers working on individual parallel jobs, the experiments confirm that A-STEAL provides almost perfect linear speedup across a variety of processor availability profiles for jobs with sufficient parallelism. I compared A-STEAL with the ABP algorithm, an adaptive work-stealing thread scheduler developed by Arora, Blumofe, and Plaxton [5] which does not employ parallelism feedback. On moderately to heavily loaded machines with large numbers of processors, A-STEAL typically completed jobs more than twice as quickly than ABP, despite being allotted the same or fewer processors on every step, while wasting only 10% of the processor cycles wasted by ABP. With respect to the performance of two-level schedulers, for job sets with arbitrary release times, the makespan achieved by G-RAD is no more than 1.39 times the optimal on average and it never exceeds 4.5 times. For batched job sets, the mean response time achieved by G-RAD is no more than 2.37 times the optimal on average, and it never exceeds 5.5 times.

Principles and techniques

In addition to illustrating the individual adaptive scheduling algorithms, this thesis presents principles and techniques for adaptive scheduling. I prove two theorems that show how the properties of an OS allocator and the performance of a thread scheduler affect the makespan competitiveness of the combined two-level scheduler. I also introduce a systematic approach for analyzing the mean response time of adaptive schedulers.

Scheduling on heterogeneous resources

A practical parallel application usually interleaves access to heterogeneous resources, such as computation, I/O and communication. This thesis explores adaptive scheduling on heterogeneous systems that incorporate special purpose processors such as vector units, floating-point coprocessors, and I/O processors. It presents and analyzes the adaptive OS allocator K-RAD for heterogeneous resources. K-RAD addresses the problem where heterogeneous resources have various functionalities. Accordingly,

the resources and the tasks can be categorized into different types, and a task of a given category can only be executed on a processor of the matching category. Let K denote the number of categories of heterogeneous resources. I call the corresponding scheduling problem as ***K-resource*** scheduling. I show that, for any set of jobs with arbitrary release times, K-RAD is $(K + 1 - 1/P_{max})$ -competitive with respect to makespan, where P_{max} denotes the maximum number of processors among all categories. This competitive ratio is provably the best possible for any nonclairvoyant deterministic algorithm for K -resource scheduling. I also show that for any batched job set $\mathcal{J}$, K-RAD is $(4K + 1 - 4K/(|\mathcal{J}| + 1))$ -competitive with respect to the mean response time. K-RAD is joint work with Hongyang Sun and Hsu Wen Jing originally presented in [78].

1.4 Overview of Thesis

The remainder of the thesis is organized as follows. Chapter 2 presents and analyzes the two adaptive thread schedulers, A-GREEDY and A-STEAL, which operate using history-based parallelism feedback. Chapter 3 presents the OS allocator RAD and analyzes its efficiency by using instantaneous parallelism as feedback. Combining OS allocator RAD with thread schedulers A-GREEDY and A-STEAL provides adaptive two-level schedulers G-RAD and W-RAD, respectively. Chapter 4 shows their performance with respect to provable efficiency, fairness, and minimal overhead. Chapter 5 presents the empirical results of our schedulers. Chapter 6 presents general principles and techniques in designing and analyzing efficient adaptive schedulers. Chapter 7 extends the adaptive scheduling framework to heterogeneous resources. Chapter 8 reviews the related work in the literature. Finally, Chapter 9 sums up the thesis and discusses some open problems.

Chapter 2

Adaptive Thread Schedulers

Without knowing the future progress of a job, can a thread scheduler provide effective parallelism feedback to the OS allocator? In this chapter, I answer this question affirmatively by presenting two adaptive thread schedulers, namely A-GREEDY and A-STEAL, which provide parallelism feedback. A-GREEDY is a greedy thread scheduler suitable for centralized scheduling, where each job's thread scheduler can dispatch all the ready threads to the allotted processors in a centralized manner, such as the scheduling of data-parallel jobs. A-STEAL is a distributed thread scheduler, where each job is executed by decentralized work-stealing [23, 32, 71, 116]. Instead of using instantaneous parallelism, A-GREEDY and A-STEAL provide parallelism feedback to the OS allocator based on a single summary statistic and the job's behavior in the previous quantum. Even though they provide parallelism feedback using only the past behavior of the job, and the job's future parallelism may not be correlated with its history of parallelism, my analysis shows that they schedule the job well with respect to both waste and completion time.

This chapter is organized as follows. Section 2.1 states the assumptions and key results of A-GREEDY and A-STEAL. Section 2.2 presents the adaptive greedy thread scheduler A-GREEDY. Section 2.3 provides a trim analysis for the special case of A-GREEDY when the scheduling quantum has length 1, *i.e.*, thread schedulers request the processors from the OS allocator at each time step, and Section 2.4 extends this trim analysis to the general case. Section 2.5 discusses an application of A-GREEDY to schedule data-parallel programs with bounded time, space, and waste. Section 2.6 describes the A-STEAL algorithm, Section 2.7 provides a trim analysis of its completion time, and Section 2.8 provides the waste analysis. Section 2.9 explains trade-offs among the various parameters of A-STEAL.

Since a thread scheduler is only responsible for the schedule of an individual job, this chapter simplifies the notations presented in Section 1.2 by removing the job identifier. For example, the work of job J is denoted as T_1 instead of $T_1(J)$, and the desire of job J at quantum q is denoted as d_q instead of $d(J, q)$. The other notations in this chapter follow the same convention of omitting the job identifier unless specified otherwise.

2.1 Assumptions and Results

The performance of a thread scheduler is measured by the completion time and waste of the job it scheduled. Since a thread scheduler always works together with the OS allocator in the request-allotment protocol, its performance usually depends on the decision of OS allocator. *To make thread schedulers robust to different OS allocators, system environments and administrative policies, our analysis of A-GREEDY and A-STEAL assumes the worst case. The OS allocator is an adversary in deciding the availability of processors.* In this section, I refine the scheduling model to focus on thread scheduling, and introduce the main results of A-GREEDY and A-STEAL.

An adaptive thread scheduler performs two major tasks. The first task is to estimate the desire of the job in the next quantum, and the second task is to schedule the ready threads of the job to its allotted processors. The adaptive thread scheduler interacts with the OS allocator as follows. Between quanta $q - 1$ and q , the thread scheduler estimates its job's desire d_q , which is the number of processors the job needs for quantum q . It provides the desire d_q to the OS allocator as its parallelism feedback. The OS allocator follows some processor allocation policy to determine the **processor availability** p_q , which is the number of processors to which the job is entitled for quantum q .

With a **conservative** allocator, the number of processors the job receives for quantum q is the job's allotment $a_q = \min\{d_q, p_q\}$, which is the smaller of its desire and the processor availability. For example, suppose that a job requests $d_q = 5$ processors and the OS allocator decides that the availability is $p_q = 10$. Then, the job is allotted $a_q = \min\{d_q, p_q\} = \min\{5, 10\} = 5$ processors. If the availability is only $p_q = 3$, however, the job's allotment is $a_q = \min\{5, 3\} = 3$. Once a job is allotted its processors, the allotment does not change during the quantum. Consequently, the thread scheduler must do a good job before a quantum begins of estimating how many processors it will need for all L time steps of the quantum. It also must use the allotted processors efficiently, *i.e.*, scheduling threads efficiently.

In an adaptive setting where the number of processors allotted to a job can change during execution, both $T_1/\bar{P}$ and T_∞ are lower bounds on the running time, where $\bar{P}$ is the mean processor availability during the computation. An adversarial OS allocator, however, can prevent any thread scheduler from offering good speedup with respect to the mean availability $\bar{P}$. For example, if the adversary chooses a huge number of processors for the job's processor availability just when the job has little instantaneous parallelism, no adaptive scheduling algorithm can effectively utilize the available processors during that quantum.

We introduce trim analysis to analyze the time bound of adaptive thread schedulers under these adversarial conditions. From the field of statistics, trim analysis borrows the idea of ignoring a few “outliers.” A **trimmed mean**, for example, is calculated by discarding a certain number of lowest and highest values and then computing the mean of those that remain. For our purposes, it suffices to trim the availability from just the high side. For a given value R , we define the ***R-high-trimmed mean availability*** as the mean availability after ignoring the R steps with the highest availability. A good thread scheduler should provide linear speedup with respect to an R -trimmed availability, where R is as small as possible.

A-GREEDY algorithm uses a greedy scheduler [22, 31, 66], which, when coupled with A-GREEDY's parallelism-feedback strategy, is provably effective. We prove that A-GREEDY guarantees linear speedup with respect to the $O(T_\infty + L \lg P)$ -trimmed availability. Specifically, consider a job with work T_1 and span T_∞ running on a machine with P processors and a scheduling quantum of length L . A-GREEDY completes the job in $O(T_1/\tilde{P} + T_\infty + L \lg P)$ time steps, where $\tilde{P}$ denotes the $O(T_\infty + L \lg P)$ -trimmed availability. Thus, the job achieves linear speed up with respect to $\tilde{P}$ when $T_1/T_\infty \gg \tilde{P}$, that is, when the job's parallelism dominates the $O(T_\infty + L \lg P)$ -trimmed availability. In addition, we prove that the total number of processor cycles wasted by the job is $O(T_1)$, representing at most a constant factor overhead.

A-STEAL [3, 4] is a thread scheduler that works in a decentralized fashion, using randomized work-stealing [5, 23, 63] to schedule the threads on allotted processors. Unlike A-GREEDY, it does not require a global view of all the available work to schedule. A-STEAL applies the same desire-estimation algorithm as A-GREEDY to calculate its job's desire. We also analyze the job completion time produced by A-STEAL by trim analysis. For a job, A-STEAL guarantees linear speedup with respect to $O(T_\infty + L \lg P)$ -trimmed availability. In addition, A-STEAL wastes at most $O(T_1)$ processor cycles.

2.2 Adaptive Greedy Algorithm

This section presents the adaptive greedy thread scheduler — A-GREEDY. Before each quantum, A-GREEDY provides parallelism feedback to the OS allocator based on the job’s history of utilization using a simple multiplicative-increase, multiplicative-decrease algorithm. A-GREEDY classifies quanta as “satisfied” versus “deprived” and “efficient” versus “inefficient.” Of the four possibilities of classification, however, A-GREEDY only uses three: inefficient, efficient-and-satisfied, and efficient-but-deprived. Using this three-way classification and the job’s desire for the previous quantum, it computes the desire for the next quantum.

To classify a quantum q as satisfied versus deprived, A-GREEDY compares the job’s allotment a_q with its desire d_q . The quantum q is **satisfied** if $a_q = d_q$, that is, the job receives as many processors as A-GREEDY requested from the OS allocator. Otherwise, if $a_q < d_q$, the quantum is **deprived**, because the job did not receive as many processors as A-GREEDY requested.

Classifying a quantum as either efficient or inefficient is more complicated. We define the **usage** u_q of a quantum q as the amount of work completed by the job during the quantum, which is to say, the total number of unit-time threads in the dag that were completed during the quantum. The maximum possible usage for a quantum q is La_q , where L is the length of quanta and a_q is the job’s allotment for quantum q . A-GREEDY uses a **utilization parameter** δ , where $0 < \delta \leq 1$, as a threshold to differentiate between efficient and inefficient quanta. Typical values for δ might be 80–95%. A quantum q is **efficient** if $u_q \geq \delta La_q$, that is, the usage is at least a δ fraction of the maximum possible usage, in which case the job wastes few (at most $(1 - \delta)La_q$) processor cycles. A quantum q is **inefficient** otherwise.

A-GREEDY calculates the desire d_q of the current quantum q based on the previous desire d_{q-1} and the three-way classification of quantum $q - 1$ as inefficient, efficient-and-satisfied, and efficient-but-deprived. The initial desire is $d_1 = 1$. A-GREEDY uses a **responsiveness parameter** $\rho > 1$ to determine how quickly the scheduler responds to changes in parallelism. Typical values of ρ might range between 1.2 and 2.0. Figure 2-1 shows the pseudocode of A-GREEDY for one quantum. The algorithm takes as input the quantum q , the utilization parameter δ , and the responsiveness parameter ρ . Intuitively, it operates as follows:

- If quantum $q - 1$ was inefficient, A-GREEDY overestimated the desire. In this case, A-GREEDY disregards whether the quantum is satisfied or deprived, and it decreases the desire (line 4) in quantum q .

```

A-GREEDY( $q, \delta, \rho$ )
1  if  $q = 1$ 
2      then  $d_q \leftarrow 1$                                  $\triangleright$  base case
3  elseif  $u_{q-1} < L\delta a_{q-1}$ 
4      then  $d_q \leftarrow d_{q-1}/\rho$                          $\triangleright$  inefficient
5  elseif  $a_{q-1} = d_{q-1}$ 
6      then  $d_q \leftarrow \rho d_{q-1}$                          $\triangleright$  efficient-and-satisfied
7  else  $d_q \leftarrow d_{q-1}$                                  $\triangleright$  efficient-but-deprived
8  Report desire  $d_q$  to the OS allocator.
9  Receive allotment  $a_q$  from the OS allocator.
10 Greedily schedule on  $a_q$  processors for  $L$  time steps.

```

Figure 2-1: Pseudocode of the adaptive greedy thread scheduler A-GREEDY, which provides parallelism feedback to an OS allocator in the form of a desire for processors. Before quantum q , A-GREEDY uses the previous quantum's desire d_{q-1} , allotment a_{q-1} , and usage u_{q-1} to compute the current quantum's desire d_q based on the utilization parameter δ and the responsiveness parameter ρ .

- If quantum $q - 1$ was efficient-and-satisfied, the job has effectively utilized the allotted processors. Thus, A-GREEDY speculates that the job can use more processors and increases the desire (line 6) in quantum q .
- If quantum $q - 1$ was efficient but deprived, the job used all the processors it was allotted, but A-GREEDY had requested more processors for the job than the job actually received from the OS allocator. Since A-GREEDY has no evidence whether the job could have used all the processors requested, it maintains the same desire (line 7) in quantum q .

Remarkably, this simple algorithm provides strong guarantees on waste and performance.

2.3 A-Greedy Trim Analysis for Unit Quanta

This section uses a trim analysis to analyze A-GREEDY for the special case where $L = 1$, that is, where each quantum consists of only a single time step. For unit quanta, adaptive scheduling can be done efficiently using instantaneous parallelism as feedback. Surprisingly, A-GREEDY's algorithm for desire estimation, which only uses historical information, provides nearly as good time bounds. Moreover, as we shall see in Section 2.4, these bounds can be extended to the case when $L \gg 1$. For a

thread scheduler based on instantaneous parallelism, a straightforward extension to $L \gg 1$ would require the information about the future parallelism of the job for all L time steps of the quantum. The analysis for unit quanta explains the effectiveness of A-GREEDY's strategy for desire estimation.

For unit quanta, we will show that A-GREEDY with utilization parameter $\delta = 1$ completes a job with work T_1 and span T_∞ in at most $T \leq T_1/\tilde{P} + 2T_\infty + \log_\rho P + 1$ time steps, where P denotes the number of processors in the machine and $\tilde{P}$ is the $(2T_\infty + \log_\rho P + 1)$ -trimmed availability. In contrast, a greedy thread scheduler that uses instantaneous parallelism as feedback completes the job in at most $T \leq T_1/\tilde{P} + T_\infty$ time steps, where $\tilde{P}$ is the T_∞ -trimmed availability. Thus, even without instantaneous parallelism, A-GREEDY operates nearly as efficiently. Moreover, the total number of processor cycles wasted by A-GREEDY in the course of computation is bounded by ρT_1 .

To prove the completion-time bounds, we use a trim analysis. We label each quantum as either **accounted** or **deductible**. Accounted quanta are those where $u_q = p_q$, that is, the usage equals the processor availability. The deductible quanta are those where $u_q < p_q$. Our trim analysis will show that when we ignore the relatively few deductible quanta, we obtain linear speedup on the more numerous accounted quanta.

We first relate the labeling of accounted and deductible to the three-way classification of quanta as inefficient, efficient-and-satisfied, and efficient-but-deprived.

Inefficient: In an inefficient quantum q , we have $u_q < a_q \leq p_q$, that is, the job uses fewer processors than it was allotted, and therefore it uses fewer processors than what are available. Thus, inefficient quanta are deductible quanta, irrespective of whether they were satisfied or deprived.

Efficient-and-satisfied: On an efficient quantum q , we have $u_q = a_q$. Since $a_q = \min\{p_q, d_q\}$ by definition, on a satisfied quantum, we have $a_q = d_q \leq p_q$. Thus, we have $u_q \leq p_q$. Since we cannot guarantee that $u_q = p_q$, we assume pessimistically that quantum q is deductible.

Efficient-but-deprived: As before, on an efficient quantum q , we have $u_q = a_q$. On a deprived quantum, we have by definition that $a_q < d_q$, and since $a_q = \min\{p_q, d_q\}$, we have $a_q = p_q$. Thus, we have $u_q = a_q = p_q$, and quantum q is accounted.

2.3. A-GREEDY Trim Analysis for Unit Quanta

21

Time Analysis

We prove the completion time bound of A-GREEDY by separately bounding the number of deductible quanta and the number of accounted quanta. We first apply a potential function argument to prove that the number of deductible quanta is at most $2T_\infty + \log_\rho P + 1$. We then show that the number of accounted quanta is at most T_1/P_A , where T_1 is the total work and P_A is the mean availability over all accounted quanta. Thus, the total time to complete the job is at most $T_1/P_A + 2T_\infty + \log_\rho P + 1$, i.e., the sum of the two parts. Finally, we show that $P_A \geq \tilde{P}$, where $\tilde{P}$ is the $(2T_\infty + L \log_\rho P + 1)$ -trimmed availability, which yields the desired result.

Our analysis uses a characterization of greedy scheduling based on whether the job uses all its allotted processors in a given step. We define a step to be **complete** if the job uses all the allotted processors in the step and **incomplete** if the job does not use all the available processors. In the special case of unit quanta, an inefficient quantum consists of a single incomplete step, while an efficient quantum consists of a single complete step. The following lemma from [19, 23, 50] shows that whenever a greedy scheduler (including A-GREEDY) schedules an incomplete step, the job makes progress on its total span.

Lemma 2.1 *Any greedy scheduler reduces a job's remaining span by 1 after every incomplete step.* $\square$

We next bound the maximum desire during the course of the computation.

Lemma 2.2 *Suppose that A-GREEDY schedules a job on a machine with P processors. If ρ is A-GREEDY's responsiveness parameter, then for every quantum q , the job's desire satisfies $d_q \leq \rho P$.*

Proof. We use induction on the number of quanta. The base case $d_1 = 1$ holds trivially. If a given quantum $q - 1$ was inefficient, the desire d_q decreases, and thus $d_q < d_{q-1} \leq \rho P$ by induction. If quantum $q - 1$ was efficient-and-satisfied, then $d_q = \rho d_{q-1} = \rho a_{q-1} \leq \rho P$. If quantum $q - 1$ was efficient-but-deprived, then $d_q = d_{q-1} \leq \rho P$ by induction. $\square$

The deductible quanta for A-GREEDY are either inefficient or efficient-and-satisfied. The next lemma bounds their number.

Lemma 2.3 *Suppose that A-GREEDY schedules a job with span T_∞ on a machine with P processors. If ρ is A-GREEDY's responsiveness parameter, $\delta = 1$ is its uti-*

lization parameter, and $L = 1$ is the quantum length, then the schedule produces at most $2T_\infty + \log_\rho P + 1$ deductible quanta.

Proof. We use a potential-function argument based on the job's desire d_q before quantum q . Define the potential before quantum q to be

$$\Phi(q) = 2T_\infty^q - \log_\rho d_q ,$$

where T_∞^q denotes the remaining span before quantum q is executed, that is, the length of the longest path in the unexecuted DAG. The initial potential is

$$\begin{aligned} \Phi(1) &= 2T_\infty - \log_\rho d_1 \\ &= 2T_\infty , \end{aligned}$$

since the desire in the first quantum is $d_1 = 1$. If the job executes for Q quanta, the final potential is

$$\begin{aligned} \Phi(Q+1) &= 2T_\infty^{Q+1} - \log_\rho d_{Q+1} \\ &\geq 0 - \log_\rho(\rho P) \\ &= -\log_\rho P - 1 , \end{aligned}$$

by Lemma 2.2. Since the potential starts at $2T_\infty$ and is at least $-\log_\rho P - 1$ at the end of the computation, the total decrease of the potential is $\Phi(1) - \Phi(Q+1) \leq 2T_\infty + \log_\rho P + 1$.

We now compute the decrease in potential during each quantum based on the three-way classification. Each case will use the fact that the decrease in potential during any quantum q is

$$\begin{aligned} \Delta\Phi &= \Phi(q) - \Phi(q+1) \\ &= (2T_\infty^q - \log_\rho d_q) - (2T_\infty^{q+1} - \log_\rho d_{q+1}) \\ &= 2(T_\infty^q - T_\infty^{q+1}) - (\log_\rho d_q - \log_\rho d_{q+1}) . \end{aligned}$$

Inefficient: An inefficient quantum q consists of a single incomplete step. After an incomplete step, the remaining span reduces by 1 (Lemma 2.1). Moreover, we have $d_{q+1} = d_q/\rho$, since A-GREEDY reduces the desire after an inefficient quantum. Thus,

2.3. A-GREEDY Trim Analysis for Unit Quanta

23

the decrease in potential after an inefficient quantum is

$$\begin{aligned}
 \Delta\Phi &= 2(T_\infty^q - T_\infty^{q+1}) - (\log_\rho d_q - \log_\rho d_{q+1}) \\
 &= 2(T_\infty^q - (T_\infty^q - 1)) - (\log_\rho d_q - \log_\rho(d_q/\rho)) \\
 &= 2(1) - (1) \\
 &= 1.
 \end{aligned}$$

Efficient-and-satisfied: A-GREEDY increases the desire after every efficient-and-satisfied quantum ($d_{q+1} = \rho d_q$). The remaining span never increases. Thus, the decrease in potential is

$$\begin{aligned}
 \Delta\Phi &= 2(T_\infty^q - T_\infty^{q+1}) - (\log_\rho d_q - \log_\rho d_{q+1}) \\
 &\geq 0 - (\log_\rho d_q - \log_\rho(\rho d_q)) \\
 &= 1.
 \end{aligned}$$

Efficient-but-deprived: After efficient-and-satisfied quanta, A-GREEDY maintains the previous desire ($d_{q+1} = d_q$), and, as before, the span never increases. Thus, the decrease in potential is

$$\begin{aligned}
 \Delta\Phi &= 2(T_\infty^q - T_\infty^{q+1}) - (\log_\rho d_q - \log_\rho d_{q+1}) \\
 &\geq 0.
 \end{aligned}$$

Thus, the potential never increases, and it decreases by at least 1 after every deductible quantum. Thus, the number of deductible quanta is at most $2T_\infty + \log_\rho P + 1$, the total decrease in potential. $\square$

We now bound the number of accounted quanta.

Lemma 2.4 *Suppose that A-GREEDY schedules a job with work T_1 . If $\delta = 1$ is A-GREEDY's utilization parameter and $L = 1$ is the quantum length, then the schedule produces at most T_1/P_A accounted quanta, where P_A is the mean availability on accounted quanta.*

Proof. Let A be the set of accounted quanta, and D be the set of deductible quanta. The mean availability on accounted quanta is $P_A = (1/|A|) \sum_{q \in A} p_q$. The total number of threads executed over the course of the computation is $T_1 = \sum_{q \in A \cup D} u_q$,

since each of the T_1 threads is executed exactly once in either an accounted or a deductible quantum. Since accounted quanta are those for which $u_q = p_q$, we have

$$\begin{aligned} T_1 &= \sum_{q \in A \cup D} u_q \\ &\geq \sum_{q \in A} u_q \\ &= \sum_{q \in A} p_q \\ &= |A| P_A \end{aligned}$$

Thus, the number of accounted quanta is $|A| \leq T_1/P_A$. $\square$

We can now bound the completion time of a job scheduled by A-GREEDY with unit quanta.

Theorem 2.5 *Suppose that A-GREEDY schedules a job with work T_1 and span T_∞ on a machine with P processors. If ρ is A-GREEDY's responsiveness parameter, $\delta = 1$ is its utilization parameter, and $L = 1$ is the quantum length, then A-GREEDY completes the job in*

$$T \leq T_1/\tilde{P} + 2T_\infty + \log_\rho P + 1$$

time steps, where $\tilde{P}$ is the $(2T_\infty + \log_\rho P + 1)$ -trimmed availability.

Proof. The proof is a trim analysis. Let A be the set of accounted quanta, and D be the set of deductible quanta. Lemma 2.3 shows that there are $|D| \leq 2T_\infty + \lg P + 1$ deductible time steps, since each quantum consists of a single time step.

Since we trim the $2T_\infty + \lg P + 1$ steps that have the highest availability, the mean availability on the accounted time steps must be at least the $(2T_\infty + \lg P + 1)$ -trimmed availability, *i.e.*, $P_A \geq \tilde{P}$. From Lemma 2.4, the number of accounted quanta is $|A| \leq T_1/P_A \leq T_1/\tilde{P}$, and since $T = |A| + |D|$, the desired time bound follows. $\square$

Waste Analysis

We now prove the waste bound for A-GREEDY with unit quanta. Let $w_q = a_q - u_q$ be the waste of quantum q . In efficient quanta, the usage is $u_q = a_q$, and the waste is $w_q = 0$. Therefore, the job wastes processor cycles only on inefficient quanta. The next theorem shows that the waste on inefficient quanta can be amortized against the work done on efficient quanta.

2.3. A-GREEDY Trim Analysis for Unit Quanta

25

Theorem 2.6 *Suppose that A-GREEDY schedules a job with work T_1 on a machine. If ρ is A-GREEDY's responsiveness parameter, $\delta = 1$ is its utilization parameter, and $L = 1$ is the quantum length, then A-GREEDY wastes at most ρT_1 processor cycles in the course of its computation.*

Proof. We use a potential-function argument based on the job's desire d_q before quantum q . Define the potential $\Psi(q)$ before quantum q as

$$\Psi(q) = \rho T_1^q + \frac{\rho}{\rho - 1} d_q ,$$

where T_1^q is the total number of unexecuted threads in the computation before quantum q . Thus, the initial potential is

$$\begin{aligned} \Psi(1) &= \rho T_1^1 + \frac{\rho}{\rho - 1} d_1 \\ &= \rho T_1 + \rho/(\rho - 1) , \end{aligned}$$

since $d_1 = 1$. If the job executes for Q quanta, the final potential is

$$\begin{aligned} \Psi(Q + 1) &= \rho T_1^{Q+1} + \frac{\rho}{\rho - 1} d_{Q+1} \\ &\geq 0 + \rho/(\rho - 1) , \\ &= \rho/(\rho - 1) , \end{aligned}$$

since the desire d_q of any quantum q is at least 1. Therefore the total decrease in potential is $\Psi(1) - \Psi(Q + 1) \leq \rho T_1$.

Based on the three-way classification, we shall show that if the waste on quantum q is $w_q = a_q - u_q$, then the potential decreases by at least w_q during the quantum. Each way will use the fact that the decrease in potential during any quantum q is

$$\begin{aligned} \Delta \Psi_q &= \Psi(q) - \Psi(q + 1) \\ &= \left(\rho T_1^q + \frac{\rho}{\rho - 1} d_q \right) - \left(\rho T_1^{q+1} + \frac{\rho}{\rho - 1} d_{q+1} \right) \\ &= \rho(T_1^q - T_1^{q+1}) + \frac{\rho}{\rho - 1} (d_q - d_{q+1}) . \end{aligned}$$

Inefficient: For any quantum q , $w_q < a_q$, which is to say, the number of processor cycles wasted is less than the total number of processor cycles allotted. Since the allotment is $a_q \leq d_q$, we have $w_q < d_q$. After an inefficient quantum q , A-GREEDY

reduces the desire to $d_{q+1} = d_q/\rho$. Thus, the decrease in potential is

$$\begin{aligned}\Delta\Psi_q &= \rho(T_1^q - T_1^{q+1}) + \frac{\rho}{\rho-1}(d_q - d_{q+1}) \\ &> \frac{\rho}{\rho-1}(d_q - d_q/\rho) \\ &= d_q \\ &> w_q.\end{aligned}$$

Efficient-and-satisfied: Since no processor cycles are wasted on any efficient quantum q , we have $w_q = 0$ and the remaining work reduces by $u_q = a_q$. On an efficient-and-satisfied quantum q , the allotment is the same as the desire ($a_q = d_q$) and A-GREEDY increases the desire ($d_{q+1} = \rho d_q$) after the quantum. Thus, the decrease in potential is

$$\begin{aligned}\Delta\Psi_q &= \rho(T_1^q - T_1^{q+1}) + \frac{\rho}{\rho-1}(d_q - d_{q+1}) \\ &= \rho a_q + \frac{\rho}{\rho-1}(d_q - \rho d_q) \\ &= \rho d_q - \rho d_q \\ &= 0 \\ &= w_q.\end{aligned}$$

Efficient-but-deprived: On any efficient quantum q , we have $w_q = 0$ and the amount of remaining work reduces by $u_q = a_q$. Since the quantum q is efficient-but-deprived, we have $d_{q+1} = d_q$, because A-GREEDY maintains the previous desire. Therefore, the decrease in potential is

$$\begin{aligned}\Delta\Psi &= \rho(T_1^q - T_1^{q+1}) + \frac{\rho}{\rho-1}(d_q - d_{q+1}) \\ &= \rho a_q + 0 \\ &> 0 \\ &= w_q.\end{aligned}$$

In all three cases, if the job wastes w_q processors in quantum q , the potential decreases by at least w_q . Consequently, the total waste during the course of the computation is at most ρT_1 , the total decrease in potential. $\square$

2.4 A-Greedy Trim Analysis of the General Case

We now use the trim analysis to analyze the general case of A-GREEDY when each scheduling quantum has L time steps, δ is the utilization parameter, ρ is the responsiveness parameter, and P is the number of processors in the machine. For a job with work T_1 and span T_∞ , A-GREEDY achieves the following bounds on running time and waste, where $\tilde{P}$ is the $(2T_\infty/(1-\delta) + L \log_\rho P + L)$ -trimmed availability:

$$\begin{aligned} T &\leq \frac{T_1}{\delta \tilde{P}} + \frac{2T_\infty}{1-\delta} + L \log_\rho P + L, \\ W &\leq \frac{1+\rho-\delta}{\delta} T_1. \end{aligned}$$

As in Section 2.3, we label each quantum as either accounted or deductible. Recall that a quantum q of length L and processor availability p_q has a total of Lp_q processor cycles available. Accounted quanta are those for which $u_q \geq \delta Lp_q$, that is, the job uses at least a δ fraction of all available processor cycles. The deductible quanta are those for which $u_q < \delta Lp_q$. By the same logic as in Section 2.3, inefficient quanta or efficient-and-satisfied quanta are labeled as deductible. efficient-but-deprived quanta, on the other hand, are labeled accounted.

Time Analysis

We bound the accounted and deductible quanta separately. We first show how inefficient quanta affect the remaining span of the job.

Lemma 2.7 *A-GREEDY reduces a job's remaining span by at least $(1-\delta)L$ after every inefficient quantum, where δ is A-GREEDY's utilization parameter and L is the quantum length.*

Proof. The total number of threads completed in an inefficient quantum q is less than δLa_q . Therefore, there can be at most δL complete steps in an inefficient quantum, since on a complete step, the job uses all the allotted processors, completing a_q threads. Since there are L time steps in a quantum, there are at least $(1-\delta)L$ incomplete steps. Thus, the span reduces by at least $(1-\delta)L$, since Lemma 2.1 shows that every incomplete step reduces the span by 1. $\square$

The next lemma bounds the number of deductible quanta.

Lemma 2.8 *Suppose that A-GREEDY schedules a job with span T_∞ on a machine with P processors. If ρ is A-GREEDY's responsiveness parameter, δ is its utilization parameter, and L is the quantum length, then the schedule produces at most $2T_\infty/((1-\delta)L) + \log_\rho P + 1$ deductible quanta.*

Proof. We use a potential-function argument as in Lemma 2.3. Define the potential before quantum q as

$$\Phi(q) = 2T_\infty^q/((1-\delta)L) - \log_\rho d_q ,$$

where T_∞^q is the remaining span before quantum q . If the job completes in Q quanta, the total decrease in potential is

$$\begin{aligned} \Phi_1 - \Phi_{Q+1} &= \frac{2T_\infty - 0}{(1-\delta)L} - (\log_\rho 1 - \log_\rho d_{Q+1}) \\ &\leq \frac{2T_\infty}{(1-\delta)L} + \log_\rho P + 1 , \end{aligned}$$

since $d_{Q+1} \leq \rho P$ by Lemma 2.2.

We can compute the decrease in potential during each quantum based on the three-way classification. Applying argument similar to those in Lemma 2.3, we can show that the potential decreases by at least 1 after every deductible quantum and that it never increases. Therefore, the total number of deductible quanta is at most $2T_\infty/((1-\delta)L) + \log_\rho P + 1$, the total decrease in potential. $\square$

We now bound the number of accounted quanta.

Lemma 2.9 *Suppose that A-GREEDY schedules a job with work T_1 . If δ is A-GREEDY's utilization parameter and L is the quantum length, then the schedule produces at most $T_1/(\delta L P_A)$ accounted quanta, where P_A is the mean availability on accounted quanta.*

Proof. Let A be the set of accounted quanta, and let D be the set of deductible quanta. The mean availability on accounted quanta is $P_A = (1/|A|) \sum_{q \in A} p_q$. The total number of threads executed in the course of the computation is $T_1 = \sum_{q \in A \cup D} u_q$. Since the accounted quanta are those for which $u_q \geq \delta L p_q$, we have

$$\begin{aligned} T_1 &= \sum_{q \in A \cup D} u_q \\ &\geq \sum_{q \in A} u_q \end{aligned}$$

2.4. A-GREEDY Trim Analysis of the General Case

29

$$\begin{aligned}
&\geq \sum_{q \in A} \delta L p_q \\
&= \delta L |A| P_A .
\end{aligned}$$

Therefore, the total number of accounted quanta is at most $|A| \leq T_1/(\delta L P_A)$. $\square$

The next theorem provides the time bound for A-GREEDY.

Theorem 2.10 *Suppose that A-GREEDY schedules a job with work T_1 and span T_∞ on a machine with P processors. If ρ is A-GREEDY's responsiveness parameter, δ is its utilization parameter, and L is the quantum length, then A-GREEDY completes the job in*

$$T \leq T_1/(\delta \tilde{P}) + 2T_\infty/(1 - \delta) + L \log_\rho P + L$$

time steps, where $\tilde{P}$ is the $(2T_\infty/(1 - \delta) + L \log_\rho P + L)$ -trimmed availability.

Proof. The proof is a trim analysis. Let A be the set of accounted quanta, and D be the set of deductible quanta. Lemma 2.8 shows that there are $|D| \leq 2T_\infty/((1 - \delta)L) + \log_\rho P + 1$ deductible quanta, and hence at most $L|D| = 2T_\infty/(1 - \delta) + L \log_\rho P + L$ time steps belong to deductible quanta. We have that $P_A \geq \tilde{P}$, since the mean availability on the accounted time steps (we trim the $L|D|$ deductible steps) must be at least the $(2T_\infty/(1 - \delta) + L \log_\rho P + L)$ -trimmed availability (we trim the $2T_\infty/(1 - \delta) + L \log_\rho P + L$ steps that have the highest availability). From Lemma 2.9, the number of accounted quanta is $|A| \leq T_1/\delta P_A \leq T_1/\delta \tilde{P}$, and since $T = L(|A| + |D|)$, the desired time bound follows. $\square$

Waste Analysis

We now prove the waste bound for A-GREEDY.

Theorem 2.11 *Suppose that A-GREEDY schedules a job with work T_1 on a machine. If ρ is A-GREEDY's responsiveness parameter, δ is its utilization parameter, and L is the quantum length, then A-GREEDY wastes at most $(1 + \rho - \delta)T_1/\delta$ processor cycles in the course of its computation.*

Proof. We prove the bound using a potential-function argument similar to the one presented in Theorem 2.6. In this case the potential before quantum q is defined as

$$\Psi(q) = \frac{1 + \rho - \delta}{\delta} T_1^q + \frac{\rho}{\rho - 1} L d_q ,$$

where T_1^q is the total number of unexecuted threads in the computation before quantum q and d_q is the desire for quantum q . By arguments similar to those presented in Theorem 2.6, one can show that if the job wastes w_q processors in quantum q , then the potential decreases by at least w_q in quantum q . If the computation completes in Q quanta, the total decrease in potential in the course of the computation is

$$\begin{aligned} \Psi(1) - \Psi(Q+1) &= \frac{1 + \rho - \delta}{\delta} (T_1 - T_{Q+1}) \\ &\quad + \frac{\rho}{\rho - 1} L (d_1 - d_{Q+1}) \\ &\leq \frac{1 + \rho - \delta}{\delta} T_1, \end{aligned}$$

since $d_1 = 1$ and $d_{Q+1} \geq 1$. Therefore, the total waste is at most $(1 + \rho - \delta)T_1/\delta$, the total decrease in potential. $\square$

We can decompose the bounds of Theorems 2.10 and 2.11 into separate bounds for accounted and deductible quanta.

Corollary 2.12 *Suppose that A-GREEDY schedules a job with work T_1 and span T_∞ on a machine with P processors, and suppose that ρ is A-GREEDY's responsiveness parameter, δ is its utilization parameter, and L is the quantum length. Let T_a and T_d be the number of time steps in accounted and deductible quanta, respectively, and let W_a and W_d be the waste on accounted and deductible quanta, respectively. Then, A-GREEDY achieves the following bounds:*

$$\begin{aligned} T_a &\leq (1/\delta)T_1/\tilde{P}, \\ T_d &\leq (2 \min\{L, 1/(1 - \delta)\})T_\infty + L \log_\rho P + L, \\ W_a &\leq (1/\delta - 1)T_1, \\ W_d &\leq (\rho/\delta)T_1. \end{aligned}$$

$\square$

As can be seen from these inequalities, the bounds for accounted quanta are stronger than those for deductible quanta. The reason is that the OS allocator in our model is adversarial. In practice, however, it seems unlikely that the OS allocator would actually act as an adversary. Thus, A-GREEDY's behavior on the deductible

quanta is likely to be much better than predicted by these worst-case bounds. Moreover, since the adversary's bad behavior is limited to relatively few deductible quanta, we conjecture that in practice the overall time and waste of a real scheduler based on A-GREEDY more closely follows the bounds for accounted quanta. This conjecture is proved to be true both theoretically and empirically when we combine A-GREEDY with our OS allocator RAD in Chapter 4 and Chapter 5.

2.5 Adaptive Data-Parallel Scheduling

In this section, we discuss a practical application of A-GREEDY to schedule programs written in data-parallel languages, such as High Performance Fortran (HPF) [62], *Lisp [93], C* [115], NESL [16,18], and ZPL [34]. Indeed, data-parallel job scheduling algorithms in the literature often model a job as a dag of tasks [15,18,54,80,128]. Of particular interest is the work by Blueloch and his coauthors [15,18,111] which provides various nonadaptive thread schedulers for a generalized class of data-parallel jobs, called *nested* data-parallel jobs. Specifically, their “prioritized” thread schedulers are provably efficient with respect to both time and space. This section applies the desire-estimation strategy of A-GREEDY to data-parallel scheduling. In particular, A-GREEDY can be combined with prioritized thread schedulers to produce adaptive thread schedulers that are provably efficient with respect to time, space, and waste.

Data-parallel languages present the abstraction of operations on vectors (or matrices), rather than on single scalar values. The total number of vector operations corresponds to the span of the computation, and the total number of scalar operations corresponds to the work. The time to perform a vector operation on a given number of processors may vary, because vectors may have different lengths from operation to operation. The time may also vary due to vector operations requiring different amounts of work. For example, one vector operation might be an element-wise addition operation, taking time proportional to the length of the vectors, and another vector operation might be an outer-product, taking time proportional to the product of vector lengths.

In a multiprogramming setting, several data-parallel programs might share a single parallel machine, and the OS allocator changes the allotment of processors to various jobs based on their parallelism feedback and its administrative policy. A typical data-parallel thread scheduler might map the individual scalar operations to the allotted processors before each vector operation, perhaps using central control. The instantaneous parallelism of the job is simply the work in the next vector oper-

ation, which is typically known to the thread scheduler, because it knows the vector lengths. If the thread scheduler can communicate with the OS allocator before every vector operation, then using instantaneous parallelism as feedback works fine. This strategy may induce high scheduling overheads, however, since it may not be possible to amortize communication with the OS allocator over a single vector operation. Since the thread scheduler only knows the parallelism of the next vector operation, not of subsequent ones, if the thread scheduler executes multiple vector operations in a single scheduling quantum, an A-GREEDY-like adaptive strategy for parallelism feedback should outperform a strategy based on instantaneous parallelism.

Blelloch, Gibbons, and Matias’s prioritized thread scheduler [15], called PDF,¹ provides good bounds for data-parallel scheduling. The machine model used in this work is a synchronous P -processor EREW-PRAM [75] augmented with a “scan” primitive [64,65]. Suppose that a job has T_1 work and a span of T_∞ , and suppose that executing the job in a serial, depth-first fashion uses S_1 space. Then, PDF completes the job in at most $O(T_1/P + T_\infty)$ time steps and requires less than $S_1 + O(PT_\infty)$ space. Blelloch and Greiner [18] extend the PDF algorithms to schedule programs written in the nested data-parallel language NESL with only a small increase in the running time and space.

Combining PDF with A-GREEDY produces an algorithm A-PDF that can schedule data-parallel jobs efficiently in an adaptive setting. A-PDF uses the parallelism feedback mechanisms of A-GREEDY to interact with the OS allocator. At the beginning of each quantum q , A-PDF calculates the desire d_q based on the three-way classification of the previous quantum and reports the desire to the OS allocator according to the algorithm in Figure 2-1. the OS allocator allots $a_q = \min(p_q, d_q)$ processors to the job. Then, A-PDF uses the prioritized depth-first-like techniques described in [15] for thread synchronization and execution on a_q processors in the quantum q . For each time step in quantum q , if there are more than a_q processors in the ready pool, the a_q ready threads with highest priorities are scheduled. Otherwise, all the ready threads in the pool are scheduled.

The following theorem — which can be proved in a straightforward fashion by combining our analysis of A-GREEDY with that of [15] — bounds the time, space, and waste of A-PDF.

Theorem 2.13 *Suppose that A-PDF schedules a job with work T_1 and span T_∞ on a P -processor EREW-PRAM augmented with a scan primitive, where L is the*

---

¹PDF stands for P-schedule Depth First scheduler

2.6. Adaptive Work-stealing Algorithm

33

quantum size, $\tilde{P}$ is the $O(T_\infty + L \lg P)$ -trimmed availability, and S_1 is the space taken for a serial schedule. Then, A-PDF completes the job in T steps, takes S space, and wastes W processor cycles, where

$$\begin{aligned} T &= O(T_1/\tilde{P} + T_\infty + L \lg P) \\ S &\leq S_1 + O(PT_\infty) , \\ W &= O(T_1) . \end{aligned}$$

□

Narlikar and Blelloch [111] present an asynchronous algorithm which can be used to schedule data-parallel jobs. Their algorithm obtains the bounds $T = O(T_1/P + T_\infty \lg P)$ and $S = S_1 + O(PT_\infty \lg P)$. A-GREEDY can be combined with this scheduler as well, producing the following bounds:

$$\begin{aligned} T &= O(T_1/\tilde{P} + T_\infty \lg P + L \lg P) , \\ S &= S_1 + O(PT_\infty \lg P) , \\ W &= O(T_1) , \end{aligned}$$

where $\tilde{P}$ is the $O(T_\infty \lg P + L \lg P)$ -trimmed availability.

2.6 Adaptive Work-stealing Algorithm

Randomized work-stealing [5, 23, 63] proves to be an effective way of designing a thread scheduler, both in theory and in practice. The decentralized thread scheduler is unaware of all the available threads to execute at a given moment. Whenever a processor runs out of work, it “steals” work from another processor chosen at random. To date, however, no work-stealing thread schedulers have been designed that provide provably effective parallelism feedback to an OS allocator. This section presents the adaptive work-stealing thread scheduler — A-STEAL. Before the start of a quantum, A-STEAL estimates processor desire based on the job’s history of utilization. It uses this estimate as its parallelism feedback to the OS allocator, which it provides in the form of a request for processors. In this section, we describe A-STEAL and its desire-estimation heuristic.

During a quantum, A-STEAL uses *work-stealing* [5, 23, 107] to schedule the job’s threads on the allotted processors. A-STEAL can use any provably good work-stealing algorithm, such as that of Blumofe and Leiserson [23] or the nonblocking one

presented by Arora, Blumofe, and Plaxton [5]. In a work-stealing thread scheduler, every processor allotted to the job maintains a queue of ready threads for the job. When the ready queue of a processor becomes empty, the processor becomes a **thief**, randomly picking a **victim** processor and **stealing** work from the victim's ready queue. If the victim has no available work, then the steal is **unsuccessful**, and the thief continues to steal at random from the other processors until it successfully finds work or the job completes. At all times, every processor is either working or stealing.

This basic work-stealing algorithm must be modified to deal with dynamic changes in processor allotment to the job between quanta. Two simple modifications make the work-stealing algorithm adaptive.

Allotment gain: When the allotment increases from quantum $q - 1$ to q , the thread scheduler obtains $a_q - a_{q-1}$ additional processors. Since the ready queues of these new processors start out empty, all these processors immediately start stealing to get work from the other processors.

Allotment loss: When the allotment decreases from quantum $q - 1$ to q , the OS allocator deallocates $a_{q-1} - a_q$ processors, whose ready queues may be nonempty. To deal with these queues, we use the concept of “mugging” [24]. When a processor runs out of work, instead of stealing immediately, it looks for a **muggable** queue, a nonempty queue that has no associated processor working on it. Upon finding a muggable queue, the thief **mugs** the queue by taking over the entire queue as its own. Thereafter, it works on the queue as if it were its own. If there are no muggable queues, the thief steals as per normal.

At all time steps during the execution of A-STEAL, every processor is either working, stealing, or mugging. We call the cycles a processor spends on working, stealing, and mugging as **work-cycles**, **steal-cycles**, and **mug-cycles**, respectively. Cycles spent stealing and mugging are **wasted**.

The salient part of A-STEAL is its desire-estimation algorithm, which is extended from the desire-estimation heuristic for the A-GREEDY. To estimate the desire for the next quantum $q+1$, A-STEAL classifies the previous quantum q as either “satisfied” or “deprived” and either “efficient” or “inefficient.” The classification of satisfied versus deprived is the same as that described in A-GREEDY. The quantum q is satisfied if $a_q = d_q$, that is, the job receives as many processors as A-STEAL requested for it from the OS allocator. Otherwise, if $a_q < d_q$, the quantum is deprived. The classification of efficient and inefficient quanta is somewhat different, however.

A-STEAL classifies a quantum as efficient versus inefficient by comparing the usage

2.6. Adaptive Work-stealing Algorithm

35

```

A-STEAL ( $q, \delta, \rho$ )
1  if  $q = 1$ 
2    then  $d_q \leftarrow 1$                                  $\triangleright$  base case
3  elseif  $n_{q-1} < L\delta a_{q-1}$ 
4    then  $d_q \leftarrow d_{q-1}/\rho$                        $\triangleright$  inefficient
5  elseif  $a_{q-1} = d_{q-1}$ 
6    then  $d_q \leftarrow \rho d_{q-1}$                        $\triangleright$  efficient-and-satisfied
7  else  $d_q \leftarrow d_{q-1}$                              $\triangleright$  efficient-but-deprived
8  Report  $d_q$  to the OS allocator.
9  Receive allotment  $a_q$  from the OS allocator.
10 Schedule on  $a_q$  processors using randomized work
    stealing for  $L$  time steps.

```

Figure 2-2: Pseudocode of the adaptive work-stealing thread scheduler A-STEAL, which provides parallelism feedback to an OS allocator in the form of processor desire. Before quantum q , A-STEAL uses the previous quantum's desire d_{q-1} , allotment a_{q-1} , and nonsteal usage n_{q-1} to compute the current quantum's desire d_q based on the utilization parameter δ and the responsiveness parameter ρ .

with the allotment. Recall that the usage u_q of quantum q as the total number of work-cycles in q . Let's define the **nonsteal usage** n_q as the sum of the number of work-cycles and mug-cycles. We call a quantum q efficient if $n_q \geq \delta L a_q$, that is, the nonsteal usage is at least a δ fraction of the total processor cycles allotted. A quantum is inefficient otherwise. Inefficient quanta contain at least $(1 - \delta)L a_q$ steal-cycles. Like A-GREEDY, the utilization parameter δ works as the threshold to differentiate between efficient and inefficient quanta.

It might seem counterintuitive for the definition of "efficient" to include mug-cycles. After all, mug-cycles are wasted. The rationale is that mug-cycles arise as a result of an allotment loss. Thus, they do not generally indicate that the job has a surplus of processors.

A-STEAL calculates the desire d_q of the current quantum q based on the previous desire d_{q-1} and the three-way classification of quantum $q - 1$ as inefficient, efficient-and-satisfied, and efficient-but-deprived. The initial desire is $d_1 = 1$. Like A-GREEDY, A-STEAL uses a responsiveness parameter $\rho > 1$ to determine how quickly the scheduler responds to changes in parallelism.

Figure 2-2 shows the pseudocode of A-STEAL for one quantum. A-STEAL takes as

input the quantum q , the utilization parameter δ , and the responsiveness parameter ρ . For the first quantum, it requests 1 processor. Thereafter, it operates as follows:

- **Inefficient:** If quantum $q - 1$ was inefficient, it contained many steal-cycles, which indicates that most of the processors had insufficient work to do. Therefore, A-STEAL overestimated the desire for quantum $q - 1$. In this case, A-STEAL does not care whether quantum $q - 1$ was satisfied or deprived. It simply decreases the desire (line 4) for quantum q .
- **Efficient-and-satisfied:** If quantum $q - 1$ was efficient-and-satisfied, the job effectively utilized the processors that A-STEAL requested on its behalf. In this case, A-STEAL speculates that the job can use more processors. It increases the desire (line 6) for quantum q .
- **Efficient-but-deprived:** If quantum $q - 1$ was efficient-but-deprived, the job used all the processors it was allotted, but A-STEAL had requested more processors for the job than the job actually received from the OS allocator. Since A-STEAL has no evidence whether the job could have used all the processors requested, it maintains the same desire (line 7) for quantum q .

After determining the job's desire, A-STEAL requests that many processors from the OS allocator, receives its allotment, and then schedules the job on the allotted processors for the L time steps of the quantum.

2.7 A-Steal Time Analysis

This section uses a trim analysis to analyze A-STEAL with respect to the completion time. Suppose that A-STEAL schedules a job with work T_1 and span T_∞ on a machine with P processors. Let ρ denote A-STEAL's responsiveness parameter, δ its utilization parameter, and L the quantum length. For any constant $0 < \epsilon < 1$, A-STEAL completes the job in time $T = O\left(T_1/\tilde{P} + T_\infty + L \log_\rho P + L \ln(1/\epsilon)\right)$, with probability at least $1 - \epsilon$, where $\tilde{P}$ is the $O(T_\infty + L \log_\rho P + L \ln(1/\epsilon))$ -trimmed availability. This bound implies that A-STEAL achieves linear speedup on all the time steps excluding $O(T_\infty + L \log_\rho P + L \ln(1/\epsilon))$ time steps with the highest processor availability.

We make two assumptions to simplify the presentation of the analysis. First, we assume that there is no contention for steals and that a steal can be completed on a single time step. Second, we assume that a mug can be completed in one time step as well. That is, if there is a muggable queue, then a thief processor can find it instantly

and mug it. If there is no muggable queue, thief processor does know instantly that it should start stealing. We shall relax these assumptions at the end of this section.

We prove our completion-time bound using a trim analysis, which calculates the performance by discarding a few outliers and measures that for the remaining majority. Like the analysis of A-GREEDY, we label each quantum as either accounted or deductible. Accounted quanta are those for which $n_q \geq L\delta p_q$, where n_q is the nonsteal usage. That is, the job works or mugs for at least a δ fraction of the Lp_q processor cycles available during the quantum. Conversely, the deductible quanta are those for which $n_q < L\delta p_q$. We can relate this labeling of quanta as accounted versus deductible to a three-way classification of quanta. According to their definitions, it is not hard to see that inefficient and efficient-and-satisfied quanta are deductible, and efficient-but-deprived quanta are accounted. Our trim analysis will show that when we ignore the relatively few deductible quanta, we obtain linear speedup on the more numerous accounted quanta.

We now analyze the execution time of A-STEAL by bounding the number of deductible and accounted quanta separately. Two observations provide intuition for the proof. First, each inefficient quantum contains a large number of steal-cycles, which we can expect to reduce the remaining span. This observation will help us to bound the number of deductible quanta. Second, most of the processor cycles on an efficient quantum are spent either working or mugging. We shall show that there cannot be too many mug-cycles during the job's execution, and thus most of the processor cycles on efficient quanta are spent doing useful work. This observation will help us to bound the number of accounted quanta.

The following lemma, proved in [23], shows how steal-cycles reduce the job's span.

Lemma 2.14 *If a job has r ready queues, then $3r$ steal-cycles suffice to reduce the job's remaining span by at least 1 with probability at least $1 - 1/e$, where e is the base of the natural logarithm. $\square$*

The next lemma shows that an inefficient quantum reduces the job's span, which we shall later use to bound the total number of inefficient quanta.

Lemma 2.15 *If ρ is A-STEAL's responsiveness parameter and L is the quantum length, on an inefficient quantum, A-STEAL reduces a job's remaining span by at least $(1 - \delta)L/6$ with probability greater than $1/4$.*

Proof. Let q be an inefficient quantum. A processor with an empty ready queue steals only when it cannot mug a queue, and hence, all the steal-cycles on quantum q occur when the number of nonempty queues is at most the allotment a_q . Therefore, Lemma 2.14 dictates that $3a_q$ steal-cycles suffice to reduce the span by 1 with probability at least $1 - 1/e$. Since the quantum q is inefficient, it contains at least $(1 - \delta)La_q$ steal-cycles. Divide the time steps of the quantum into **rounds** such that each round contains $3a_q$ steal-cycles. Thus, there are at least $j = (1 - \delta)La_q/(3a_q) = (1 - \delta)L/3$ rounds.² We call a round **good** if it reduces the span by at least 1; otherwise, the round is **bad**. For each round i on quantum q , we define the indicator random variable X_i to be 1 if round i is a bad round and 0 otherwise, and let $X = \sum_{i=1}^j X_i$. Since $\Pr\{X_i = 1\} < 1/e$, linearity of expectation dictates that $E[X] < j/e$. We now apply Markov's inequality [41, p. 1111], which says that for a nonnegative random variable, we have $\Pr\{X \geq t\} \leq E[X]/t$ for all $t > 0$. Substituting $t = j/2$, we obtain $\Pr\{X > j/2\} \leq E[X]/(j/2) \leq (j/e)/(j/2) = 2/e < 3/4$. Thus, with probability greater than $1/4$, the quantum q contains at least $j/2$ good rounds. Since each good round reduces the span by at least 1, with probability greater than $1/4$, the span reduces by least $j/2 = ((1 - \delta)L/3)/2 = (1 - \delta)L/6$ during quantum q . $\square$

Lemma 2.16 *Suppose that A-STEAL schedules a job with span T_∞ . If L is the quantum length, then for any $\epsilon > 0$, the schedule produces at most $48T_\infty/(L(1 - \delta)) + 16\ln(1/\epsilon)$ inefficient quanta with probability at least $1 - \epsilon$.*

Proof. Let I be the set of inefficient quanta. Define an inefficient quantum q as **productive** if it reduces the span by at least $(1 - \delta)L/6$ and **unproductive** otherwise. For each quantum $q \in I$, define the indicator random variable Y_q to be 1 if q is productive and 0 otherwise. By Lemma 2.15, we have $\Pr\{Y_q = 1\} > 1/4$. Let the total number of unproductive quanta be $Y = \sum_{q \in I} Y_q$. To simplify, let $A = 6T_\infty/((1 - \delta)L)$. If the job's execution contains $|I| \geq 48T_\infty/((1 - \delta)L) + 16\ln(1/\epsilon)$ inefficient quanta, then we have $E[Y] > |I|/4 \geq 12T_\infty/((1 - \delta)L) + 4\ln(1/\epsilon) = 2A + 4\ln(1/\epsilon)$. Using the Chernoff bound $\Pr\{Y < (1 - \lambda)E[Y]\} < \exp(-\lambda^2 E[Y]/2)$ [109, p. 70] and choosing $\lambda = (A + 4\ln(1/\epsilon)) / (2A + 4\ln(1/\epsilon))$, we obtain

$$\Pr\{Y < A\}$$

²Actually, the number of rounds is $j = \lfloor (1 - \delta)L/3 \rfloor$, but we shall ignore the roundoff for simplicity. A more detailed analysis can nevertheless produce the same constants in the bounds for Lemmas 2.16 and 2.19.

$$\begin{aligned}
&= \Pr \left\{ Y < \left(1 - \frac{A + 4 \ln(1/\epsilon)}{2A + 4 \ln(1/\epsilon)} \right) (2A + 4 \ln(1/\epsilon)) \right\} \\
&= \Pr \{ Y < (1 - \lambda) (2A + 4 \ln(1/\epsilon)) \} \\
&\leq \exp \left(-\frac{\lambda^2}{2} (2A + 4 \ln(1/\epsilon)) \right) \\
&= \exp \left(-\frac{1}{2} \cdot \frac{(A + 4 \ln(1/\epsilon))^2}{2A + 4 \ln(1/\epsilon)} \right) \\
&< \exp \left(-\frac{1}{2} \cdot 4 \ln(1/\epsilon) \cdot \frac{1}{2} \right) \\
&= \epsilon .
\end{aligned}$$

Therefore, if the number $|I|$ of inefficient quanta is at least $48T_\infty/((1 - \delta)L) + 16 \ln(1/\epsilon)$, the number of productive quanta is at least $A = 6T_\infty/((1 - \delta)L)$ with probability at least $1 - \epsilon$. By Lemma 2.15 each productive quantum reduces the span by at least $(1 - \delta)L/6$, and therefore at most $A = 6T_\infty/((1 - \delta)L)$ productive quanta occur during job's execution. Consequently, with probability at least $1 - \epsilon$, the number of inefficient quanta is $|I| \leq 48T_\infty/((1 - \delta)L) + 16 \ln(1/\epsilon)$. $\square$

Since A-STEAL applies the same desire-estimation algorithm as A-GREEDY, the following technical lemma that bounds the maximum value of the desire can be shown similarly as Lemma 2.2 proved in Chapter 2.

Lemma 2.17 *Suppose that A-STEAL schedules a job on a machine with P processors. If ρ is A-STEAL's responsiveness parameter, then before any quantum q , the desire d_q of the job is bounded by $d_q < \rho P$.* $\square$

The next lemma reveals a relationship between inefficient quanta and efficient-and-satisfied quanta.

Lemma 2.18 *Suppose that A-STEAL schedules a job on a machine with P processors. If ρ is A-STEAL's responsiveness parameter and the schedule produces m inefficient quanta, then it produces at most $m + \log_\rho P + 1$ efficient-and-satisfied quanta.*

Proof. Assume for the purpose of contradiction that a job's execution has m inefficient quanta, but $k > m + \log_\rho P + 1$ efficient-and-satisfied quanta. Recall that desire increases by ρ after every efficient-and-satisfied quantum, decreases by ρ after every inefficient quantum, and does not change otherwise. Thus, the total increase

in desire is ρ^k , and the total decrease in desire is ρ^m . Since the desire starts at 1, the desire at the end of the job is $\rho^{k-m} > \rho^{\log_\rho P+1} = \rho P$, contradicting Lemma 2.17. $\square$

The following lemma bounds the number of efficient-and-satisfied quanta.

Lemma 2.19 *Suppose that A-STEAL schedules a job with span T_∞ on a machine with P processors. If ρ is A-STEAL's responsiveness parameter, δ is its utilization parameter, and L is the quantum length, then the schedule produces at most $48T_\infty/((1-\delta)L) + \log_\rho P + 16 \ln(1/\epsilon) + 1$ efficient-and-satisfied quanta with probability at least $1 - \epsilon$ for any $\epsilon > 0$.*

Proof. The lemma follows directly from Lemmas 2.16 and 2.18. $\square$

The next lemma exhibits the relationship between inefficient quanta and efficient-and-satisfied quanta.

Lemma 2.20 *Suppose that A-STEAL schedules a job, and let I and C denote the sets of inefficient and efficient-and-satisfied quanta, respectively, produced by the schedule. If ρ is A-STEAL's responsiveness parameter, then there exists an injective mapping $f : I \rightarrow C$ such that for all $q \in I$, we have $f(q) < q$ and $d_{f(q)} = d_q/\rho$.*

Proof. For every inefficient quantum $q \in I$, define $r = f(q)$ to be the latest efficient-and-satisfied quantum such that $r < q$ and $d_r = d_q/\rho$. Such a quantum always exists, because the initial desire is 1 and the desire increases only after an efficient-and-satisfied quantum. We must prove that f does not map two inefficient quanta to the same efficient-and-satisfied quantum. Assume for the sake of contradiction that there exist two inefficient quanta $q < q'$ such that $f(q) = f(q') = r$. By definition of f , the quantum r is efficient-and-satisfied, $r < q < q'$, and $d_q = d_{q'} = \rho d_r$. After the inefficient quantum q , A-STEAL reduces the desire to d_q/ρ . Since the desire later increases again to $d_{q'} = d_q$ and the desire increases only after efficient-and-satisfied quanta, there must be an efficient-and-satisfied quantum r' in the range $q < r' < q'$ such that $d(r') = d(q')/\rho$. But then, by the definition of f , we would have $f(q') = r'$. Contradiction. $\square$

We can now bound the total number of mug-cycles executed by processors.

Lemma 2.21 *Suppose that A-STEAL schedules a job with work T_1 on a machine with P processors. If ρ is A-STEAL's responsiveness parameter, δ is its utilization parameter, and L is the quantum length, the schedule produces at most $((1+\rho)/(L\delta - 1 - \rho))T_1$ mug-cycles.*

2.7. A-STEAL Time Analysis

41

Proof. When the allotment decreases, some processors are deallocated, and their ready queues are declared muggable. The total number M of mug-cycles is at most the number of muggable queues during the job's execution. Since the allotment reduces by at most $a_q - 1$ from quantum q to quantum $q+1$, there are $M \leq \sum_q (a_q - 1) < \sum_q a_q$ mug-cycles during the execution of the job.

By Lemma 2.20, for each inefficient quantum q , there is a distinct corresponding efficient-and-satisfied quantum $r = f(q)$ that satisfies $d_q = \rho d_r$. By definition, each efficient-and-satisfied quantum r has a nonsteal usage $n_r \geq L\delta a_r$ and allotment $a_r = d_r$. Thus, we have $n_r + n_q \geq L\delta a_r = (L\delta/(1+\rho))(a_r + \rho a_r) = (L\delta/(1+\rho))(a_r + \rho d_r) \geq (L\delta/(1+\rho))(a_r + a_q)$, since $a_q \leq d_q$ and $d_q = \rho d_r$. Except for these inefficient quanta and their corresponding efficient-and-satisfied quanta, every other quantum q is efficient, and hence $n_q \geq L\delta a_q$ for these quanta. Let $N = \sum_q n_q$ be the total number of nonsteal-cycles during the job's execution. We have $N = \sum_q n_q \geq (L\delta/(1+\rho)) \sum_q a_q \geq (L\delta/(1+\rho))M$. Since the total number of nonsteal-cycles is the sum of work-cycles and mug-cycles and the total number of work-cycles is T_1 , we have $N = T_1 + M$, and hence, $T_1 = N - M \geq (L\delta/(1+\rho))M - M = ((L\delta - 1 - \rho)/(1+\rho))M$, which yields $M \leq ((1+\rho)/(L\delta - 1 - \rho))T_1$. $\square$

Lemma 2.22 *Suppose that A-STEAL schedules a job with work T_1 on a machine with P processors. If ρ is A-STEAL's responsiveness parameter, δ is its utilization parameter, and L is the quantum length, the schedule produces at most $(T_1/(L\delta P_A))(1 + (1+\rho)/(L\delta - 1 - \rho))$ accounted quanta, where P_A is the mean availability on the accounted quanta.*

Proof. Let A and D denote the set of accounted and deductible quanta, respectively. The mean availability on accounted quanta is $P_A = (1/|A|) \sum_{q \in A} p_q$. Let N be the total number of nonsteal-cycles. By definition of accounted quanta, the nonsteal usage satisfies $n_q \geq L\delta a_q$. Thus, we have $N = \sum_{q \in A \cup D} n_q \geq \sum_{q \in A} n_q \geq \sum_{q \in A} \delta L p_q = \delta L |A| P_A$, and hence, we obtain

$$|A| \leq N/(L\delta P_A) . \quad (2.1)$$

But, the total number of nonsteal-cycles is the sum of the number of work-cycles and mug-cycles. Since there are at most T_1 work-cycles on accounted quanta and Lemma 2.21 shows that there are at most $M \leq ((1+\rho)/(L\delta - 1 - \rho))T_1$ mug-cycles, we have $N \leq T_1 + M \leq T_1(1 + (1+\rho)/(L\delta - 1 - \rho))$. Substituting this bound on N

into Inequality (2.1) completes the proof. $\square$

We are now ready to bound the running time of jobs scheduled with A-STEAL.

Theorem 2.23 *Suppose that A-STEAL schedules a job with work T_1 and span T_∞ on a machine with P processors. If ρ is A-STEAL's responsiveness parameter, δ is its utilization parameter, and L is the quantum length, then for any $\epsilon > 0$, with probability at least $1 - \epsilon$, A-STEAL completes the job in*

$$T \leq \frac{T_1}{\delta \tilde{P}} \left(1 + \frac{1 + \rho}{L\delta - 1 - \rho} \right) + O \left(\frac{T_\infty}{1 - \delta} + L \log_\rho P + L \ln(1/\epsilon) \right) \quad (2.2)$$

time steps, where $\tilde{P}$ is the $O(T_\infty/(1 - \delta) + L \log_\rho P + L \ln(1/\epsilon))$ -trimmed availability.

Proof. The proof uses trim analysis. Let A be the set of accounted quanta, and let D be the set of the deductible quanta. Lemmas 2.16 and 2.19 show that there are at most $|D| = O(T_\infty/((1 - \delta)L) + \log_\rho P + \ln(1/\epsilon))$ deductible quanta, and hence at most $L|D| = O(T_\infty/(1 - \delta) + L \log_\rho P + L \ln(1/\epsilon))$ time steps belong to the deductible quanta. We have that $P_A \geq \tilde{P}$, since the mean availability on the accounted time steps (we trim the $L|D|$ deductible steps) must be at least the $O(T_\infty/(1 - \delta) + L \log_\rho P + L \ln(1/\epsilon))$ -trimmed availability (we trim the $O(T_\infty/(1 - \delta) + L \log_\rho P + L \ln(1/\epsilon))$ steps that have the highest availability). From Lemma 2.22, the number of accounted quanta is $(T_1/(L\delta P_A))(1 + (1 + \rho)/(L\delta - 1 - \rho))$, and since $T = L(|A| + |D|)$, the desired time bound follows. $\square$

Corollary 2.24 *Suppose that A-STEAL schedules a job with work T_1 and span T_∞ on a machine with P processors. If ρ is A-STEAL's responsiveness parameter, δ is its utilization parameter, and L is the quantum length, then A-STEAL completes the job in expected time $E[T] = O(T_1/\tilde{P} + T_\infty + L \lg P)$, where $\tilde{P}$ is the $O(T_\infty + L \lg P)$ -trimmed availability.*

Proof. Straightforward conversion of high-probability bound to expectation, together with setting δ and ρ to suitable constants. $\square$

Our analysis made two assumptions to ease the presentation. First, we assumed that there is no contention for steals. Second, we assumed that a thief processor can find a muggable queue and mug it in unit time. Now, let us relax these assumptions.

The first issue dealing with the contention on steals has been addressed by Blumofe and Leiserson in [23]. A balls-and-bins argument can be used to prove that taking the

contention of steals into account would increase the running time by at most $O(\lg P)$, which is tiny compared to the other terms in our running time.

Mugging requires more data-structure support. When a processor runs out of work, it needs to find out if there are any muggable queues for the job. As a practical matter, these muggable queues can be placed in a set (using any synchronous queue or set implementations as in [117, 122, 123, 133]). This strategy could increase the number of mug-cycles by a factor of P in the worst case. If $P \ll L$, however, this change does not affect the running time bound by much. Moreover, in practice, the number of muggings is so small that the time spent on muggings is insignificant compared to the total running time of the job. Alternatively, to obtain a better theoretical bound, we could use a counting network [6] with width P to implement the list of muggable queues, in which case each mugging operation would consume $O(\lg^2 P)$ processor cycles. The number of accounted steps in the time bound from Lemma 2.22 would increase slightly to $(T_1/\delta\tilde{P}) (1 + (1 + \rho) \lg^2 P / (L\delta - 1 - \rho))$, but the number of deductible steps would not change.

2.8 A-Steal Waste Analysis

This section proves that when a job is scheduled by A-STEAL, the total number of processor cycles wasted during the job's execution is $W = O(T_1)$ in the worst case.

Theorem 2.25 *Suppose that A-STEAL schedules a job with work T_1 on a machine with P processors. If ρ is A-STEAL's responsiveness parameter, δ is its utilization parameter, and L is the quantum length, then A-STEAL wastes at most*

$$W \leq \left(\frac{1 + \rho - \delta}{\delta} + \frac{(1 + \rho)^2}{\delta(L\delta - 1 - \rho)} \right) T_1 \quad (2.3)$$

processor cycles during the course of the computation.

Proof. Let M be the total number of mug-cycles, and let S be the total number of steal-cycles. Hence, we have $W = S + M$. Since Lemma 2.21 provides an upper bound of M , we only need to bound S . We use an accounting argument to calculate S based on whether a quanta is inefficient or efficient. Let S_{ineff} and S_{eff} , where $S = S_{\text{ineff}} + S_{\text{eff}}$, be the numbers of steal-cycles on inefficient and efficient quanta, respectively.

Inefficient quanta: Lemma 2.20 shows that every inefficient quantum q with desire d_q corresponds to a distinct efficient-and-satisfied quantum $r = f(q)$ with desire

$d_r = d_q/\rho$. Thus, the steal-cycles on quantum q can be amortized against the nonsteal-cycles on quantum r . Since quantum r is efficient-and-satisfied, its nonsteal usage satisfies $n_r \geq L\delta a_q/\rho$, and its allocation is $a_r = d_r$. Therefore, we have $n_r \geq L\delta a_r = L\delta d_r = L\delta d_q/\rho \geq L\delta a_q/\rho$. Let s_q be the number of steal-cycles on quantum q . Since the quantum contains at most La_q total processor cycles, we have $s_q \leq La_q \leq \rho n_r/\delta$, that is, the number of steal-cycles in the inefficient quantum q is at most a ρ/δ fraction of the nonsteal-cycles in its corresponding efficient-and-satisfied quantum r . Therefore, the total number of steal-cycles in all inefficient quanta satisfies $S_{\text{ineff}} \leq (\rho/\delta)(T_1 + M)$.

Efficient quanta: On any efficient quantum q , the job performs at least $L\delta a_q$ work- and mug-cycles and at most $L(1 - \delta)a_q$ steal-cycles. Summing over all efficient quanta, the number of steal-cycles on efficient quanta is $S_{\text{eff}} \leq ((1 - \delta)/\delta)(T_1 + M)$.

Using the bound $M \leq ((1 + \rho)/(L\delta - 1 - \rho))T_1$ from Lemma 2.21, we obtain

$$\begin{aligned}
W &= S + M \\
&= S_{\text{ineff}} + S_{\text{eff}} + M \\
&\leq (\rho/\delta)(T_1 + M) + ((1 - \delta)/\delta)(T_1 + M) + M \\
&= (T_1 + M)\frac{1 + \rho - \delta}{\delta} + M \\
&\leq \left(T_1 + T_1\frac{1 + \rho}{L\delta - 1 - \rho}\right)\frac{1 + \rho - \delta}{\delta} + T_1\frac{1 + \rho}{L\delta - 1 - \rho} \\
&= T_1\left(\left(1 + \frac{1 + \rho}{L\delta - 1 - \rho}\right)\frac{1 + \rho - \delta}{\delta} + \frac{1 + \rho}{L\delta - 1 - \rho}\right) \\
&= T_1\left(\frac{1 + \rho - \delta}{\delta} + \frac{(1 + \rho)^2}{\delta(L\delta - 1 - \rho)}\right),
\end{aligned}$$

which proves the theorem. $\square$

2.9 Interpretation of the Bounds

In this section, we simplify the bounds on A-STEAL to understand the tradeoffs involved between completion time and waste. Although our bounds are good asymptotically, they contain large constants. One might wonder whether these large constants might adversely affect A-STEAL's practical utility. We will argue that most of the large constants are due to the assumption of the adversarial behavior of the OS allocator, and thus, we should expect the practical performance of A-STEAL to be better than that indicated by our bounds. Moreover, we have implemented A-STEAL

2.9. Interpretation of the Bounds

45

in simulation, which provides strong evidence that A-STEAL should be efficient in practice. The empirical results of A-STEAL are presented in Chapter 5.

If the utilization parameter δ and responsiveness parameter ρ are constants, the bounds in Inequalities (2.2) and (2.3) can be simplified as follows:

$$\begin{aligned} T &\leq \frac{T_1}{\delta \tilde{P}}(1 + O(1/L)) + O\left(\frac{T_\infty}{1 - \delta} + L \log_\rho P + L \ln(1/\epsilon)\right), \\ W &\leq \left(\frac{1 + \rho - \delta}{\delta} + O(1/L)\right) T_1. \end{aligned} \quad (2.4)$$

This reformulation allows us to see the trade-offs due to the setting of the parameters δ and ρ more easily. As δ increases and ρ decreases, the completion time increases and the waste decreases. Typical values for the utilization parameter δ might lie between 80% and 95%, and the responsiveness parameter ρ might be set between 1.2 and 2.0. The quantum length L is a system configuration parameter, which might have values in the range 10^3 to 10^5 .

For the time bound in Inequality (2.4), as the values of δ approaches toward 1, the coefficient of $T_1/\tilde{P}$ decreases toward 1, and the job comes closer to perfect linear speedup on accounted steps, but the number of deductible steps increases as well. The large number of deductible steps is due to the adversarial OS allocator. Our simulation results detailed in Chapter 5 indicate that the jobs usually achieve nearly perfect linear speedup with respect to $\tilde{P}$ for a variety of availability profiles.

To see how these settings affect the waste bound, consider the waste bound in Inequality (2.4) as consisting of two parts, where the waste due to steal-cycles is $S \leq ((1 + \rho - \delta)T_1)/\delta$, and the waste due to the mug-cycles is only $M = O(1/L)T_1$. Since the waste on mug-cycles is just a tiny fraction compared to the work T_1 , an implementation of A-STEAL need not overly concern itself with the bookkeeping overhead of adding and removing processors from jobs.

The major part of waste comes from steal-cycles, where S is generally less than $2T_1$ for typical parameter values. The analysis of Theorem 2.25 shows that the number of steal-cycles on efficient steps is bounded by $((1 - \delta)/\delta)T_1$, which is a small fraction of S . Thus, most of the waste that occurs in the bound can be attributed to the steal-cycles on the inefficient quanta. To ensure the robustness of A-STEAL, our analysis assumes that the OS allocator is an adversary, which creates as many inefficient quanta as possible. Since OS allocators are generally not adversarial, we should not expect these large overheads to materialize in practice. The simulation results in

Chapter 5 give strong evidence that the waste is a much smaller fraction of T_1 than that suggested by our theoretical bound. In those experiments, A-STEAL typically wastes less than 20% of the allotted processor cycles.

Chapter 3

The RAD OS Allocator

This chapter presents an adaptive OS allocator, called RAD, which receives feedback from thread schedulers and allots processors to jobs before each scheduling quantum. RAD combines the space-sharing OS allocator “Dynamic Equipartitioning” (DEQ) [103, 130] with the time-sharing round-robin (RR) algorithm.

In this chapter, I analyze the performance of I-RAD, which couples RAD with a greedy scheduler that uses instantaneous parallelism as feedback. The best previous mean-response-time bound for online nonclairvoyant algorithm is $2 + \sqrt{3} \approx 3.73$ proved by Edmonds *et al.* [52]. I show that I-RAD is 3-competitive for batched jobs with respect to the mean response time, which offers the best competitive ratio known.

The remainder of this chapter is organized as follows. Section 3.1 describes RAD algorithm. Section 3.2 illustrates an important concept called squashed sum, and presents its two important properties. Section 3.3 presents lower bounds for mean response time. Finally, in Section 3.4, I show that I-RAD is 3-competitive for batched jobs with respect to mean response time.

3.1 The RAD Algorithm

RAD unifies the space-sharing job scheduling algorithm DEQ with the time-sharing round-robin algorithm. When the number of jobs exceeds that of processors, RAD schedules the jobs in a batched round-robin fashion, which allocates one processor to each job with an equal share of time. Otherwise, when the number of jobs is not more than the number of processors, RAD uses DEQ as OS allocator. DEQ gives each job an equal share of processor allotments unless the job requests less.

When a batch of jobs are scheduled in the round-robin fashion, RAD maintains a queue of jobs. At the beginning of each quantum, if there are more than P jobs, it pops P jobs from the top of the queue, and allots one processor to each of them during the quantum. At the end of the quantum, RAD pushes the P jobs back to the bottom of the queue if they are uncompleted. The new jobs can be put into the queue once they are released.

When the number of jobs is not more than the number of processors, RAD applies DEQ. DEQ attempts to give each job a fair share of processors. If a job requires less than its fair share, however, DEQ distributes the excess processors to the other jobs. More precisely, upon receiving the desires $\{d(J_i, q)\}$ from the thread schedulers of all jobs $J_i \in \mathcal{J}$, DEQ executes the following processor-allocation algorithm:

1. Set $n = |\mathcal{J}|$. If $n = 0$, return.
2. If the desire of every job $J_i \in \mathcal{J}$ satisfies $d(J_i, q) \geq P/n$, assign each job $a(J_i, q) = P/n$ processors.¹
3. Otherwise, let $\mathcal{J}' = \{J_i \in \mathcal{J} : d(J_i, q) < P/n\}$, i.e., the set of jobs that require less than the fair share of processors. Assign $a(J_i, q) = d(J_i, q)$ processors to each $J_i \in \mathcal{J}'$. Update $\mathcal{J} = \mathcal{J} - \mathcal{J}'$, and $P = P - \sum_{J_i \in \mathcal{J}'} d(J_i, q)$. Go to Step 1.

The desire of jobs that DEQ uses can be instantaneous parallelism, history-based feedback provided by A-GREEDY or A-STEAL, or other types of parallelism feedback.

According to the scheduling policy of DEQ, for a given quantum all jobs receive the same number of processors to within 1, unless their desire is less. To simplify the analysis in the thesis, we shall assume that all deprived jobs receive exactly the same number of processors, which we term the *mean deprived allotment* for the quantum. Relaxing this assumption may double the execution-time bound of a job, but our algorithms remain $O(1)$ -competitive. A tighter but messier analysis retains the constants of the simpler analysis presented here. At any quantum where the number of jobs is equal to the number of processors, DEQ and round-robin give exactly the same processor allotment, allocating each of P jobs 1 processor.

3.2 Properties of Squashed Sum

Squashed sum is an important concept for assisting in the analysis of the total response time. This section defines squashed sum and presents two key properties.

---

¹If P is not dividable by n , $P\%n$ jobs receive $\lceil P/n \rceil$ processors each, and every other job receives $\lfloor P/n \rfloor$ processors.

3.2. Properties of Squashed Sum

49

We first introduce two equivalent definitions for squashed sum, which help to establish a lower bound on total response time.

Definition 3.1 Given a list $\langle a_i \rangle$ of m nonnegative numbers, let f be a permutation on $\{1, 2, \dots, m\}$ that satisfies $a_{f(1)} \leq a_{f(2)} \leq \dots \leq a_{f(m)}$. The **squashed sum** of list $\langle a_i \rangle$ is defined as

$$\text{sq-sum}(\langle a_i \rangle) = \sum_{i=1}^m (m - i + 1) a_{f(i)} . \quad (3.1)$$

By observing that the permutation f on the list $\langle a_i \rangle$ gives the minimum value for the squashed sum formulation described by Equation (3.1), we obtain the following equivalent definition for the squashed sum of list $\langle a_i \rangle$

$$\text{sq-sum}(\langle a_i \rangle) = \min_{g \in \Upsilon} \sum_{i=1}^m (m - i + 1) a_{g(i)} , \quad (3.2)$$

where $\Upsilon = \{g \mid g : \{1, 2, \dots, m\} \rightarrow \{1, 2, \dots, m\}\}$ denotes the set of all permutations on $\{1, 2, \dots, m\}$.

The following lemma presents the first property of the squashed sum. It says that if the elements in a list are item wise larger than their corresponding terms in a second list, the squashed sum of the first list is also larger than that of the second.

Lemma 3.1 Let $\langle \alpha_i \rangle$ and $\langle \beta_i \rangle$ be two lists of nonnegative integers with m elements each, and suppose that $\alpha_i \leq \beta_i$ for $i = 1, 2, \dots, m$. Then, we have $\text{sq-sum}(\langle \alpha_i \rangle) \leq \text{sq-sum}(\langle \beta_i \rangle)$.

Proof. Let f be a permutation on $\{1, 2, \dots, m\}$ satisfying $\alpha_{f(1)} \leq \alpha_{f(2)} \leq \dots \leq \alpha_{f(m)}$, and let g be a permutation on $\{1, 2, \dots, m\}$ satisfying $\beta_{g(1)} \leq \beta_{g(2)} \leq \dots \leq \beta_{g(m)}$.

We first show that $\alpha_{f(i)} \leq \beta_{g(i)}$ for $i = 1, 2, \dots, m$. Suppose for the purpose of contradiction that there exists a $j \in \{1, 2, \dots, m\}$ such that $\alpha_{f(j)} > \beta_{g(j)}$. Then, there must be at least j integers smaller than $\alpha_{f(j)}$ in $\langle \beta_i \rangle$, namely $\beta_{g(1)}, \beta_{g(2)}, \dots, \beta_{g(j)}$. Since $\alpha_i \leq \beta_i$ for $i = 1, 2, \dots, m$, we have $\alpha_{g(i)} \leq \beta_{g(i)} < \alpha_{f(j)}$ for $i = 1, 2, \dots, j$. Thus, there are at least j elements smaller than $\alpha_{f(j)}$ in $\langle \alpha_i \rangle$, namely $\alpha_{g(1)}, \alpha_{g(2)}, \dots, \alpha_{g(j)}$. By definition of the permutation f , however, at most $j - 1$ integers are smaller than $\alpha_{f(j)}$ in $\langle \alpha_i \rangle$. A contradiction. Therefore, we have $\alpha_{f(i)} \leq \beta_{g(i)}$ for $i = 1, 2, \dots, m$.

Consequently, by Definition 3.1, we have

$$\begin{aligned} \text{sq-sum}(\langle \alpha_i \rangle) &= \sum_{i=1}^m (m-i+1) \alpha_{f(i)} \\ &\leq \sum_{i=1}^m (m-i+1) \beta_{g(i)} \\ &= \text{sq-sum}(\langle \beta_i \rangle) . \end{aligned}$$

□

The following technical lemma shows the second property of the squashed sum. It tells us the minimum increase in the squashed sum of a list when the algebraic sum of the elements in the list increases by a given amount. Intuitively, the increase on small elements results in a larger increase of the squashed sum than that on large elements. Thus, the minimum increase is achieved when we increase the largest elements of the list only.

Lemma 3.2 *Let $\langle a_i \rangle$ and $\langle b_i \rangle$ be two lists of m nonnegative numbers that satisfy $b_i = a_i + s_i$, where $0 \leq s_i \leq h$ for $i = 1, 2, \dots, m$, and h is a positive number. Let $S = \sum_{i=1}^m s_i$ represent the algebraic sum of $\{s_i\}$. Let $l = |\{s_i \mid s_i = h\}|$ denote the number of instances of s_i that have value h . If $l > 0$, then we have*

$$\text{sq-sum}(\langle b_i \rangle) \geq \text{sq-sum}(\langle a_i \rangle) + S(l+1)/2 .$$

Proof. Let $\Upsilon = \{g \mid g : \{1, 2, \dots, m\} \rightarrow \{1, 2, \dots, m\}\}$ denote the set of all permutations on $\{1, 2, \dots, m\}$. Let $f \in \Upsilon$ denote a permutation that satisfies $b_{f(1)} \leq b_{f(2)} \leq \dots \leq b_{f(m)}$. Then, according to Equation (3.1) in Definition 3.1, we have

$$\begin{aligned} \text{sq-sum}(\langle b_i \rangle) &= \sum_{i=1}^m (m-i+1) b_{f(i)} \\ &= \sum_{i=1}^m (m-i+1) (a_{f(i)} + s_{f(i)}) \\ &= \sum_{i=1}^m (m-i+1) a_{f(i)} + \sum_{i=1}^m (m-i+1) s_{f(i)} \\ &\geq \min_{g \in \Upsilon} \sum_{i=1}^m (m-i+1) a_{g(i)} + \min_{k \in \Upsilon} \sum_{i=1}^m (m-i+1) s_{k(i)} \\ &= \text{sq-sum}(\langle a_i \rangle) + \text{sq-sum}(\langle s_i \rangle) . \end{aligned}$$

3.3. Mean Response Time Lower Bounds

51

Now, we show that $\text{sq-sum}(\langle s_i \rangle) \geq S(l+1)/2$. To simplify the notation, rename the elements of the list $\langle s_i \rangle$ such that $s_1 \leq s_2 \leq \dots \leq s_{m-l} < s_{m-l+1} = s_{m-l+2} = \dots = s_m$. Since $\sum_{i=1}^m s_i = S$, we have $s_{m-l+1} = s_{m-l+2} = \dots = s_m = h = (S - \sum_{i=1}^{m-l} s_i)/l$. Then, the squashed sum of $\langle s_i \rangle$ is

$$\begin{aligned}
 \text{sq-sum}(\langle s_i \rangle) &= \sum_{i=1}^m (m-i+1)s_i \\
 &= \sum_{i=1}^{m-l} (m-i+1)s_i + \sum_{i=m-l+1}^m (m-i+1)s_i \\
 &= \sum_{i=1}^{m-l} (m-i+1)s_i + \frac{(S - \sum_{i=1}^{m-l} s_i)}{l} \sum_{i=1}^l i \\
 &= \sum_{i=1}^{m-l} (m-i+1)s_i + \frac{(S - \sum_{i=1}^{m-l} s_i) l(l+1)}{2} \\
 &= \frac{\sum_{i=1}^{m-l} (2m-2i+2)s_i}{2} + \frac{(S - \sum_{i=1}^{m-l} s_i)(l+1)}{2} \\
 &= \frac{S(l+1)}{2} + \frac{\sum_{i=1}^{m-l} (2m-2i-l+1)s_i}{2} \\
 &\geq \frac{S(l+1)}{2} .
 \end{aligned}$$

The last inequality holds because $s_i \geq 0$ and $2m-2i-l+1 > 0$ for $i = 1, 2, \dots, m-l$. $\square$

3.3 Mean Response Time Lower Bounds

This section describes the mean response time lower bounds for batched jobs. Then, we introduce two notations that are used to simplify the presentation of the analysis in Section 3.4.

Lower Bounds

The lower bounds of mean response time for batched jobs make use of the following concepts.

Definition 3.2 *The **squashed work area** of a job set $\mathcal{J}$ on P processors is*

$$\text{sqw}(\mathcal{J}) = \frac{1}{P} \text{sq-sum}(\langle T_1(J_i) \rangle) ,$$

where $T_1(J_i)$ is the work of job $J_i \in \mathcal{J}$.

Definition 3.3 The *aggregate span* of $\mathcal{J}$ is

$$T_\infty(\mathcal{J}) = \sum_{J_i \in \mathcal{J}} T_\infty(J_i) ,$$

where $T_\infty(J_i)$ is the span of job $J_i \in \mathcal{J}$.

The research in [46, 131, 132] establishes two lower bounds for the mean response time:

$$\overline{R^*}(\mathcal{J}) \geq T_\infty(\mathcal{J}) / |\mathcal{J}| , \quad (3.3)$$

$$\overline{R^*}(\mathcal{J}) \geq sqw(\mathcal{J}) / |\mathcal{J}| , \quad (3.4)$$

where $\overline{R^*}(\mathcal{J})$ denotes the mean response time of $\mathcal{J}$ scheduled by an optimal clairvoyant scheduler. Both the aggregate span $T_\infty(\mathcal{J})$ and the squashed work area $sqw(\mathcal{J})$ are lower bounds for the total response time $R^*(\mathcal{J})$ under an optimal clairvoyant scheduler.

Notation

We introduce two notations — *t-suffix* and *t-prefix*. For any time step t , the *t-suffix*, denoted as $\vec{t}$, represents the set of time steps from t to the completion of $\mathcal{J}$, i.e., $\vec{t} = \{t, t+1, \dots, T(\mathcal{J})\}$, while the *t-prefix*, denoted as $\overleftarrow{t}$, represents set of time steps from 1 to t , i.e., $\overleftarrow{t} = \{1, 2, \dots, t\}$. We will be interested in the suffixes of jobs, namely, the portion of jobs that remain after some number of steps have been executed. To that end, define the *t-suffix* of a job $J_i \in \mathcal{J}$ to be the job $J_i(\vec{t})$ induced by those vertices in $V(J_i)$ that execute on or after time t , that is,

$$J_i(\vec{t}) = (V(J_i(\vec{t})), E(J_i(\vec{t}))) ,$$

where $v \in V(J_i(\vec{t}))$ if $v \in V(J_i)$ and $\tau(v) \geq t$, and $(u, v) \in E(J_i(\vec{t}))$ if $(u, v) \in E(J_i)$ and $u, v \in V(J_i(\vec{t}))$. The *t-suffix* of the job set $\mathcal{J}$ is

$$\mathcal{J}(\vec{t}) = \{J_i(\vec{t}) : J_i \in \mathcal{J} \text{ and } V(J_i(\vec{t})) \neq \emptyset\} .$$

Thus, we have $\mathcal{J} = \mathcal{J}(\vec{1})$, and the number of uncompleted jobs at time step t is the number $|\mathcal{J}(\vec{t})|$ of nonempty jobs in $\mathcal{J}(\vec{t})$. Similarly, we can define the *t-prefix* of

a job J_i as $J_i(\overleftarrow{t})$ and the t -prefix of a job set $\mathcal{J}$ as $\mathcal{J}(\overleftarrow{t})$.

3.4 Mean Response Time of I-RAD

I-RAD combines RAD with a greedy thread scheduler using instantaneous parallelism feedback. Recall that instantaneous parallelism is the number of processors the job can effectively use at the present moment.

To analyze the mean response time of I-RAD, we consider two cases. The first case corresponds to **light load**, where $|\mathcal{J}| \leq P$. In this case, I-RAD employs only DEQ. The second case corresponds to **heavy load**, where $|\mathcal{J}| > P$, and in this case I-RAD employs both DEQ and round-robin. Since we consider batched jobs, the number of uncompleted jobs decreases monotonically. When the number of uncompleted jobs drops to P , I-RAD switches its OS allocator from round-robin to DEQ. Therefore, we prove the second case based on the properties of round-robin scheduling and the results of the first case. We will show that under both light and heavy workload, I-RAD has a competitive ratio of 3 as stated in the following theorem.

Theorem 3.3 *I-RAD is 3-competitive with respect to mean response time for any batched job set $\mathcal{J}$.*

Proof. The analysis of this theorem is presented in Sections 3.4.1 and 3.4.2 respectively. $\square$

3.4.1 Analysis of I-RAD under Light Load

When the number of jobs is no more than the number of processors, I-RAD uses DEQ to assign processors to jobs. DEQ never allots more processors to a job than the job's desire. At any time step, if a job's allotment is equal to its desire, the job is said to be **satisfied**; if its allotment is less than the desire, the job is called **deprived**. Recall that at any time step, all deprived jobs scheduled by DEQ receive the same number of processors, called the mean deprived allotment. The following lemma analyzes the behavior of satisfied and deprived jobs at each time step under light load.

Lemma 3.4 *Suppose that a batched job set $\mathcal{J}$ with $|\mathcal{J}| \leq P$ is scheduled by I-RAD on P processors. Its total response time is bounded by*

$$R(\mathcal{J}) \leq \left(2 - \frac{2}{|\mathcal{J}| + 1}\right) sqw(\mathcal{J}) + T_\infty(\mathcal{J}) . \quad (3.5)$$

Proof. We will prove Inequality (3.5) by induction on the remaining execution time of the job set $\mathcal{J}(\vec{t})$.

Basis: $t = T(\mathcal{J}) + 1$ where $T(\mathcal{J})$ denotes the completion time of job set $\mathcal{J}$. When $t = T(\mathcal{J}) + 1$, we have $\mathcal{J}(\vec{t}) = \emptyset$. It follows that $R(\mathcal{J}(\vec{t})) = 0$, $sqw(\mathcal{J}(\vec{t})) = 0$, and $T_\infty(\mathcal{J}(\vec{t})) = 0$. Thus, the claim holds trivially.

Induction: $1 \leq t \leq T(\mathcal{J})$. Let $n = |\mathcal{J}(\vec{t})|$ denote the number of uncompleted jobs at time step t . We have $n \geq |\mathcal{J}(\vec{t+1})|$ since the number of uncompleted jobs decreases monotonically for a batched job set.

Suppose by induction that Inequality (3.5) holds at time step $t + 1$, i.e.,

$$\begin{aligned} R(\mathcal{J}(\vec{t+1})) &\leq \left(2 - \frac{2}{|\mathcal{J}(\vec{t+1})| + 1}\right) sqw(\mathcal{J}(\vec{t+1})) + T_\infty(\mathcal{J}(\vec{t+1})) \\ &\leq \left(2 - \frac{2}{n+1}\right) sqw(\mathcal{J}(\vec{t+1})) + T_\infty(\mathcal{J}(\vec{t+1})) . \end{aligned} \quad (3.6)$$

We will show that it still holds at time step t , i.e.,

$$R(\mathcal{J}(\vec{t})) \leq \left(2 - \frac{2}{n+1}\right) sqw(\mathcal{J}(\vec{t})) + T_\infty(\mathcal{J}(\vec{t})) . \quad (3.7)$$

The following notations denote the changes in, respectively, the total response time, the squashed work area, and the aggregate span from time t to $t + 1$:

$$\begin{aligned} \Delta rt &= R(\mathcal{J}(\vec{t})) - R(\mathcal{J}(\vec{t+1})) , \\ \Delta sqw &= sqw(\mathcal{J}(\vec{t})) - sqw(\mathcal{J}(\vec{t+1})) , \\ \Delta T_\infty &= T_\infty(\mathcal{J}(\vec{t})) - T_\infty(\mathcal{J}(\vec{t+1})) . \end{aligned}$$

Given the induction hypothesis (Inequality (3.6)), we need only show that the following inequality holds for our claim (Inequality (3.7)) to be true.

$$\left(2 - \frac{2}{n+1}\right) \Delta sqw + \Delta T_\infty \geq \Delta rt . \quad (3.8)$$

3.4. Mean Response Time of I-RAD

55

We divide the proof of Inequality (3.8) into four steps.

(1) To bound Δrt : At any time t , the total number of uncompleted jobs is $|\mathcal{J}(\vec{t})| \leq n$. Since each uncompleted job in $\mathcal{J}(\vec{t})$ adds one time step to the total response time during step t , we have

$$\Delta rt \leq n. \quad (3.9)$$

(2) To bound ΔT_∞ : At time step t , an uncompleted job J_i is either satisfied or deprived. Thus, the uncompleted jobs $\mathcal{J}(\vec{t})$ can be partitioned into satisfied jobs $\mathcal{JS}(t)$ and deprived jobs $\mathcal{JD}(t)$, i.e., $\mathcal{J}(\vec{t}) = \mathcal{JS}(t) \cup \mathcal{JD}(t)$. If $J_i \in \mathcal{JS}(t)$, the span of J_i must reduce by 1 at time step t , i.e., $T_\infty(J_i(\vec{t})) = T_\infty(J_i(\vec{t+1})) + 1$. If $J_i \in \mathcal{JD}(t)$, the span of J_i never increases at any time step t , i.e., $T_\infty(J_i(\vec{t})) \geq T_\infty(J_i(\vec{t+1}))$. Therefore, the aggregate span of $\mathcal{J}$ must reduce by at least $|\mathcal{JS}(t)|$ at time step t , and we have

$$\Delta T_\infty \geq |\mathcal{JS}(t)|. \quad (3.10)$$

(3) To bound Δsqw : Since $\mathcal{JD}(t)$ denotes the set of deprived jobs at time t , we consider two cases depending on whether $|\mathcal{JD}(t)| = 0$.

Case 1. $|\mathcal{JD}(t)| = 0$: There exist no deprived jobs at time t . In this case, it is obvious that

$$\Delta sqw \geq 0. \quad (3.11)$$

Case 2. $|\mathcal{JD}(t)| > 0$: There is at least one deprived job at time t . In this case, all of the P processors must have been allotted to the jobs (otherwise, there would not be any deprived jobs). Therefore, the algebraic sum of the total work of the uncompleted jobs decrease by P . Moreover, all deprived jobs have the same allotment, and each deprived job always receives fewer processors than any of the satisfied jobs. Applying Lemma 3.2 with $S = P$ and $l = |\mathcal{JD}(t)|$, we have

$$\begin{aligned} \Delta sqw &= \frac{\text{sq-sum}(T_1(\mathcal{J}(\vec{t}))) - \text{sq-sum}(T_1(\mathcal{J}(\vec{t+1})))}{P} \\ &\geq \frac{P(|\mathcal{JD}(t)| + 1)/2}{P} \\ &= \frac{|\mathcal{JD}(t)| + 1}{2}. \end{aligned} \quad (3.12)$$

(4) **To derive Inequality (3.8):** At any time step t , an uncompleted job is either satisfied or deprived. Therefore, we have

$$|\mathcal{JS}(t)| + |\mathcal{JD}(t)| = n . \quad (3.13)$$

We will now show Inequality (3.8) by combining the results of Δrt , ΔT_∞ and Δsqw . In the case of $|\mathcal{JD}(t)| = 0$, all n jobs are satisfied, *i.e.*, $|\mathcal{JS}(t)| = n$. Inequalities (3.9), (3.10), and (3.11) indicate that Inequality (3.8) holds. In the case of $|\mathcal{JD}(t)| > 0$, the following derivation shows that Inequality (3.8) also holds:

$$\begin{aligned} \left(2 - \frac{2}{n+1}\right) \Delta sqw + \Delta T_\infty &\geq \left(2 - \frac{2}{n+1}\right) \frac{|\mathcal{JD}(t)| + 1}{2} + |\mathcal{JS}(t)| \\ &= \frac{n}{n+1} (n+1 - |\mathcal{JS}(t)|) + |\mathcal{JS}(t)| \\ &= n + \frac{|\mathcal{JS}(t)|}{n+1} \\ &\geq n \\ &\geq \Delta rt . \end{aligned}$$

Hence, the induction hypothesis holds. $\square$

3.4.2 Analysis of I-RAD under Heavy Load

We now derive the mean response time of I-RAD for batched jobs for the second case where $|\mathcal{J}| > P$. Since all jobs in the job set $\mathcal{J}$ arrive at time step 0, the number of uncompleted jobs decreases monotonically. When the number of uncompleted jobs drops down to P or below, I-RAD switches its OS allocator from round-robin to DEQ. We divide the analysis into two steps. First, we analyze the completion time of the jobs which are scheduled by round-robin during their entire execution. Second, we incorporate the response time bound in Lemma 3.4 to prove a bound on the response time of I-RAD in general.

A batched job set $\mathcal{J}$ can be divided into two subsets — **RR set** and **DEQ set**. The RR set, denoted as $\mathcal{J}_{RR}$, includes all the jobs in $\mathcal{J}$ that are entirely scheduled by RR for their execution. The DEQ set, denoted as $\mathcal{J}_{DEQ}$, includes all the jobs in $\mathcal{J}$ that are scheduled by round-robin at the beginning, and by DEQ eventually. There exists a unique quantum q called the **final RR quantum** such that q is the last quantum scheduled by round-robin, while the quanta from $q + 1$ onwards are all scheduled by DEQ. According to I-RAD, there must be more than P uncompleted jobs at the

3.4. Mean Response Time of I-RAD

57

beginning of q , and at most P uncompleted jobs immediately after q . Let σ denote the total number of uncompleted jobs immediately after q . By definition $\sigma = |\mathcal{J}_{DEQ}|$ and $\sigma \leq P$. Moreover, since round-robin always gives each job an equal share of computation time, jobs with less work complete earlier than the jobs with more work. Relabel the jobs according to their work such that $T_1(J_1) \leq T_1(J_2) \leq \dots \leq T_1(J_{|\mathcal{J}|})$. We have $\mathcal{J}_{RR} = \{J_i \mid 1 \leq i \leq |\mathcal{J}| - \sigma\}$ and $\mathcal{J}_{DEQ} = \{J_i \mid i > |\mathcal{J}| - \sigma\}$, i.e., $\mathcal{J}_{DEQ}$ includes the σ jobs that have the largest amount of work, and $\mathcal{J}_{RR}$ includes the other $|\mathcal{J}| - \sigma$ jobs.

The following lemma bounds the completion time of the jobs in $\mathcal{J}_{RR}$.

Lemma 3.5 *Consider the jobs according to their work such that $T_1(J_1) \leq T_1(J_2) \leq \dots \leq T_1(J_{|\mathcal{J}|})$. For $1 \leq i \leq |\mathcal{J}| - \sigma$, the completion time $T(J_i)$ of a job J_i is $T(J_i) = ((|\mathcal{J}| - i)T_1(J_i) + \sum_{1 \leq j \leq i} T_1(J_j)) / P$.*

Proof. Because all P processors are always busy at all time under the schedule of round-robin, we get the time to complete J_i by dividing the total work done by P . Since round-robin gives each job an equal amount of time to process, at the time of completing job J_i , any other job J_x with less work than J_i should have completed all of its work, and any other job J_y with more work than J_i should have completed the same amount of work as J_i . Therefore, the total work that has been done in $\mathcal{J}$ up to the time where J_i completes its work is given by $(|\mathcal{J}| - i)T_1(J_i) + \sum_{1 \leq j \leq i} T_1(J_j)$. $\square$

The following theorem bounds the total response time of I-RAD. The analysis focuses on the case where the number of jobs is greater than the number of processors.

Theorem 3.6 *Suppose that a job set $\mathcal{J}$ is scheduled by RAD on a machine with P processors. The response time $R(\mathcal{J})$ of $\mathcal{J}$ is bounded by $R(\mathcal{J}) \leq 2sqw(\mathcal{J}) + T_\infty(\mathcal{J})$.*

Proof. Since the case where $|\mathcal{J}| \leq P$ is already resolved in Lemma 3.4, we focus on the case where $|\mathcal{J}| > P$. The analysis is divided into three parts: (1) we calculate the total response time $R(\mathcal{J}_{RR})$ of the RR set; (2) we derive the total response time $R(\mathcal{J}_{DEQ})$ of the DEQ set; (3) we sum them together.

(1) Calculate $R(\mathcal{J}_{RR})$: According to Lemma 3.5, for any job $J_i \in \mathcal{J}_{RR}$, its completion time is $T(J_i) = (1/P)((n - i)T_1(J_i) + \sum_{1 \leq j \leq i} T_1(J_j))$. Thus, the total response time of the jobs in $\mathcal{J}_{RR}$ is

$$R(\mathcal{J}_{RR}) = \frac{1}{P} \left(\sum_{1 \leq i \leq n - \sigma} (n - i)T_1(J_i) + \sum_{1 \leq i \leq n - \sigma} \sum_{1 \leq j \leq i} T_1(J_j) \right)$$

$$\begin{aligned}
&= \frac{1}{P} \left(\sum_{1 \leq i \leq n-\sigma} (n-i)T_1(J_i) + \sum_{1 \leq i \leq n-\sigma} (n-\sigma-i+1)T_1(J_i) \right) \\
&= \frac{1}{P} \sum_{1 \leq i \leq n-\sigma} (2n-2i+1)T_1(J_i) - \frac{\sigma}{P} \sum_{1 \leq i \leq n-\sigma} T_1(J_i) . \quad (3.14)
\end{aligned}$$

(2) Calculate $R(\mathcal{J}_{DEQ})$: The σ jobs in $\mathcal{J}_{DEQ}$ are scheduled by round-robin until the time step $t = T(J_{n-\sigma})$ at which the job $J_{n-\sigma}$ completes, and scheduled by DEQ afterwards. The total response time of $\mathcal{J}_{DEQ}$ is

$$R(\mathcal{J}_{DEQ}) = R\left(\mathcal{J}_{DEQ}\left(\overline{t+1}\right)\right) + \sigma \cdot T(J_{n-\sigma}) . \quad (3.15)$$

From Lemma 3.5, we know that the completion time of the job $J_{n-\sigma}$. To get $R(\mathcal{J}_{DEQ})$, we only need to calculate $R\left(\mathcal{J}_{DEQ}\left(\overline{t+1}\right)\right)$.

Since the job set $\mathcal{J}_{DEQ}$ is scheduled by DEQ as the OS allocator from time step t onwards, we can apply the total response time bound in Lemma 3.4 to calculate $R\left(\mathcal{J}_{DEQ}\left(\overline{t+1}\right)\right)$. During the interval $\overline{t}$, each job in $\mathcal{J}_{DEQ}$ completes work $T_1(J_{n-\sigma})$, and has remaining work $T_1\left(J_i\left(\overline{t+1}\right)\right) = T_1(J_i) - T_1(J_{n-\sigma})$. The squashed work of $\mathcal{J}_{DEQ}\left(\overline{t+1}\right)$ is

$$\begin{aligned}
&sqw\left(\mathcal{J}_{DEQ}\left(\overline{t+1}\right)\right) \\
&= \frac{1}{P} \text{sq-sum} \left(\left\langle T_1\left(J_i\left(\overline{t+1}\right)\right) \mid n-\sigma+1 \leq i \leq n \right\rangle \right) \\
&= \frac{1}{P} \sum_{n-\sigma+1 \leq i \leq n} (n-i+1) (T_1(J_i) - T_1(J_{n-\sigma})) \\
&= \frac{1}{P} \sum_{n-\sigma+1 \leq i \leq n} (n-i+1)T_1(J_i) - \frac{\sigma(\sigma+1)T_1(J_{n-\sigma})}{2P} . \quad (3.16)
\end{aligned}$$

According to Lemma 3.4 and Lemma 3.5, we have

$$\begin{aligned}
R(\mathcal{J}_{DEQ}) &= R\left(\mathcal{J}_{DEQ}\left(\overline{t+1}\right)\right) + \sigma \cdot T(J_{n-\sigma}) \\
&\leq 2sqw\left(\mathcal{J}_{DEQ}\left(\overline{t+1}\right)\right) + T_\infty\left(\mathcal{J}_{DEQ}\left(\overline{t+1}\right)\right) + \sigma \cdot T(J_{n-\sigma}) \\
&\leq 2 \left(\frac{1}{P} \sum_{n-\sigma+1 \leq i \leq n} (n-i+1)T_1(J_i) - \frac{\sigma(\sigma+1)T_1(J_{n-\sigma})}{2P} \right)
\end{aligned}$$

3.4. Mean Response Time of I-RAD

59

$$\begin{aligned}
& +T_{\infty} \left(\mathcal{J}_{DEQ} \left(\overline{t+1} \right) \right) + \frac{\sigma}{P} \left(\sum_{1 \leq i \leq n-\sigma} T_1(J_i) + \sigma \cdot T_1(J_{n-\sigma}) \right) \\
& \leq \frac{2}{P} \sum_{n-\sigma+1 \leq i \leq n} (n-i+1)T_1(J_i) + T_{\infty} \left(\mathcal{J}_{DEQ} \left(\overline{t+1} \right) \right) \\
& \quad + \frac{\sigma}{P} \sum_{1 \leq i \leq n-\sigma} T_1(J_i) . \tag{3.17}
\end{aligned}$$

(3) Calculate $R(\mathcal{J})$: Summing up $R(\mathcal{J}_{RR})$ in Inequality (3.14) and $R(\mathcal{J}_{DEQ})$ in Inequality (3.17), the total response time of $\mathcal{J}$ is as follows:

$$\begin{aligned}
R(\mathcal{J}) &= R(\mathcal{J}_{RR}) + R(\mathcal{J}_{DEQ}) \\
&\leq \frac{2}{P} \sum_{1 \leq i \leq n} (n-i+1)T_1(J_i) + T_{\infty} \left(\mathcal{J}_{DEQ} \left(\overline{t+1} \right) \right) \\
&\leq 2sqw(\mathcal{J}) + T_{\infty}(\mathcal{J}) ,
\end{aligned}$$

which completes the proof. $\square$

Since both $sqw(\mathcal{J})$ and $T_{\infty}(\mathcal{J})$ are lower bounds of the total response time, Theorem 3.6 indicates that I-RAD is 3-competitive for batched jobs with respect to mean response time.

Chapter 4

Two-Level Adaptive Scheduling with History-based Feedback

Parallel job scheduling on multiprocessors is often structured in two-levels. Intuitively, if each thread scheduler provides good parallelism feedback and makes productive use of available processors, a good OS allocator can ensure that *all* the jobs perform well. To affirm this intuition, this chapter presents and analyzes two adaptive two-level schedulers — G-RAD and W-RAD, both of which employ history-based feedback. G-RAD, which couples RAD with A-GREEDY, is suitable for centralized thread scheduling such as might be used to schedule data-parallel jobs, where each job's thread scheduler can dispatch all the ready threads to the allotted processors in a centralized manner. W-RAD, which couples RAD with A-STEAL, is suitable for when each job is executed by decentralized work-stealing [23, 32, 71, 116].

In this chapter, I show that both G-RAD and W-RAD ensure fair allocation for any workload, and they offer provable efficiency without requiring prior information about job parallelism. Moreover, they minimize scheduling overhead and ensure efficient utilization of processors. Specifically, I show that both of schedulers are $O(1)$ -competitive against an optimal offline scheduler with respect to makespan for jobs with arbitrary release times, and they are $O(1)$ -competitive for batched jobs with respect to mean response time. G-RAD and W-RAD are the first nonclairvoyant scheduling algorithms that offer such guarantees.

The remainder of this chapter is organized as follows. Section 4.1 analyzes the competitiveness of G-RAD with respect to makespan. Section 4.2 examines the competitiveness of G-RAD for batched jobs with respect to mean response time. Section 4.3 presents the W-RAD algorithm and analyzes its performance. Finally,

Section 4.4 introduces some general formulas to characterize the efficiency of X-RAD, which is the two-level scheduling system coupling RAD with an arbitrary thread scheduler X .

4.1 Makespan of G-RAD

This section shows that G-RAD is c -competitive with respect to makespan for a constant $c \geq 1$. The exact value of c is related to the choice of the utilization parameter and responsiveness parameter in A-GREEDY. In this section, we first review lower bounds for makespan. Then, we analyze the competitiveness of G-RAD with respect to makespan.

Lower Bounds

Given a job set $\mathcal{J}$ and P processors, lower bounds on the makespan of any OS allocator can be obtained based on release time, work, and span. Recall that for a job $J_i \in \mathcal{J}$, the quantities $r(J_i)$, $T_1(J_i)$, and $T_\infty(J_i)$ represent the release time, work, and span of J_i , respectively. Let $T^*(\mathcal{J})$ denote the makespan produced by an optimal scheduler on a job set $\mathcal{J}$ scheduled on P processors. Let $T_1(\mathcal{J}) = \sum_{J_i \in \mathcal{J}} T_1(J_i)$ denote the total work of the job set. The following two inequalities give two lower bounds on the makespan [30]:

$$T^*(\mathcal{J}) \geq \max_{J_i \in \mathcal{J}} \{r(J_i) + T_\infty(J_i)\} , \quad (4.1)$$

$$T^*(\mathcal{J}) \geq T_1(\mathcal{J}) / P . \quad (4.2)$$

Analysis

We first study the makespan of G-RAD for job sets with arbitrary job release times. We state a lemma that describes the number of satisfied steps of a single job scheduled by A-GREEDY.

Lemma 4.1 *Let ρ denote A-GREEDY's responsiveness parameter, δ its utilization parameter, and L the quantum length. For a job J_i with work $T_1(J_i)$ and span $T_\infty(J_i)$ on a machine with P processors. A-GREEDY produces at most $2T_\infty(J_i)/(1-\delta) + L \log_\rho P + L$ satisfied steps.*

Proof. From Lemma 2.8, we know that the number of deductible quanta for job J_k scheduled by A-GREEDY is at most $2T_\infty/(1-\delta)L + \log_\rho P + 1$. Since deductible quanta are either efficient-and-satisfied quanta or inefficient quanta, all satisfied quanta are

deductible quanta. Thus, the total number of satisfied quanta is bounded by $2T_\infty/(1-\delta)L + \log_\rho P + 1$, and the total number of satisfied steps is bounded by $2T_\infty(J_i)/(1-\delta) + L\log_\rho P + L$. $\square$

The following theorem analyzes the makespan of any job set $\mathcal{J}$ with arbitrary release times. The makespan bound is based on the release time $r(J_i)$, span $T_\infty(J_i)$, and work $T_1(J_i)$ of individual job J_i , as well as the total work $T_1(\mathcal{J})$ of the job set $\mathcal{J}$.

Theorem 4.2 *Suppose that G-RAD schedules a job set $\mathcal{J}$ on a machine with P processors. Let ρ denote A-GREEDY's responsiveness parameter, δ its utilization parameter, and L the quantum length. Then, G-RAD completes the job set in*

$$T(\mathcal{J}) \leq \frac{\rho+1}{\delta} \frac{T_1(\mathcal{J})}{P} + \frac{2}{1-\delta} \max_{J_i \in \mathcal{J}} \{T_\infty(J_i) + r(J_i)\} + L\log_\rho P + 2L \quad (4.3)$$

time steps.

Proof. Suppose job J_k is the last job completed among the jobs in $\mathcal{J}$. Let $S(J_k)$ denote the set of satisfied steps for J_k , and let $D(J_k)$ denote its set of deprived steps. The job J_k is scheduled to start its execution at the beginning of the quantum q where $L \cdot q < r(J_k) \leq L(q+1)$, which is the quantum immediately after J_k 's release. Therefore, we have $T(\mathcal{J}) \leq r(J_k) + L + |S(J_k)| + |D(J_k)|$. We now bound $|S(J_k)|$ and $|D(J_k)|$ respectively.

From Lemma 4.1, we know that the number of satisfied steps attributed to J_k is at most $|S(J_k)| \leq 2T_\infty(J_k)/(1-\delta) + L\log_\rho P + L$.

To bound the total number of deprived steps $D(J_k)$ of job J_k , observe that for each step $t \in D(J_k)$, G-RAD applies either DEQ or round-robin as the OS allocator. Round-robin always allots all processors to jobs. By definition, DEQ must allot all processors to jobs whenever J_k is deprived. Thus, the total allotment on such a step t is always equal to the total number P of processors. Moreover, the total allotment of $\mathcal{J}$ over J_k 's deprived steps $D(J_k)$ is $a(\mathcal{J}, D(J_k)) = \sum_{t \in D(J_k)} \sum_{J_i \in \mathcal{J}} a(J_i, t) = P|D(J_k)|$. Since any allotted processor is either working or wasted, the total allotment for any job J_i is bounded by the sum of its total work $T_1(J_i)$ and total waste $w(J_i)$. By Theorem 2.11, the waste for the job J_i is at most $(\rho-\delta+1)/\delta$ times its work. Thus, the total number of allotted processor cycles for job i is at most $T_1(J_i) + w(J_i) \leq (\rho+1)T_1(J_i)/\delta$. The total number of allotted processor cycles for all jobs is at most $\sum_{J_i \in \mathcal{J}} (\rho+1)T_1(J_i)/\delta = ((\rho+1)/\delta)T_1(\mathcal{J})$. Given $a(\mathcal{J}, D(J_k)) \leq ((\rho+1)/\delta)T_1(\mathcal{J})$

and $a(\mathcal{J}, D(J_k)) = P |D(J_k)|$, we have

$$|D(J_k)| \leq \frac{\rho + 1}{\delta} \frac{T_1(\mathcal{J})}{P}.$$

Therefore, we obtain

$$\begin{aligned} T(\mathcal{J}) &< r(J_k) + L + |D(J_k)| + |S(J_k)| \\ &\leq r(J_k) + L + \frac{\rho + 1}{\delta} \frac{T_1(\mathcal{J})}{P} + \frac{2}{1 - \delta} T_\infty(J_k) + L \log_\rho P + L \\ &\leq \frac{\rho + 1}{\delta} \frac{T_1(\mathcal{J})}{P} + \frac{2}{1 - \delta} (r(J_k) + T_\infty(J_k)) + L \log_\rho P + 2L \\ &\leq \frac{\rho + 1}{\delta} \frac{T_1(\mathcal{J})}{P} + \frac{2}{1 - \delta} \max_{J_i \in \mathcal{J}} (T_\infty(J_i) + r(J_i)) + L \log_\rho P + 2L. \end{aligned}$$

□

Since both $T_1(\mathcal{J})/P$ and $\max_{J_i \in \mathcal{J}} \{T_\infty(J_i) + r(J_i)\}$ are lower bounds on $T^*(\mathcal{J})$, we obtain the following corollary.

Corollary 4.3 *Suppose that G-RAD schedules a job set $\mathcal{J}$ on a machine with P processors. Let ρ denote A-GREEDY's responsiveness parameter, δ its utilization parameter, and L the quantum length. Then, G-RAD completes the job set in*

$$T(\mathcal{J}) \leq \left(\frac{\rho + 1}{\delta} + \frac{2}{1 - \delta} \right) T^*(\mathcal{J}) + L \log_\rho P + 2L$$

time steps, where $T^*(\mathcal{J})$ is the makespan of $\mathcal{J}$ produced by an optimal clairvoyant scheduler. □

When $\delta = 0.5$ and ρ approaches 1, the competitiveness ratio $(\rho + 1)/\delta + 2/(1 - \delta)$ approaches its minimum value 8. Thus, G-RAD is $(8 + \epsilon)$ -competitive with respect to makespan for any constant $\epsilon > 0$.

4.2 Mean Response Time of G-RAD for Batched Jobs

Mean response time is an important measure for multiuser environments where we desire as many users as possible to receive fast response from the system. We shall show that G-RAD is $O(1)$ -competitive for batched jobs with respect to the mean response time. The proof of the competitiveness of mean response time is divided

into two parts, which are described in Section 4.2.1 and Section 4.2.2 respectively. Recall that, when $|\mathcal{J}| \leq P$, *i.e.*, the number of jobs in $\mathcal{J}$ is less than or equal to the number of processors P , G-RAD always uses DEQ as its OS allocator. In Section 4.2.1, we show that G-RAD is $O(1)$ -competitive in this case. For the second case, when $|\mathcal{J}| > P$, G-RAD uses both round-robin and DEQ. For a batch of jobs, the number of uncompleted jobs decreases monotonically as time progresses. When the number of uncompleted jobs drops to P , G-RAD switches from round-robin to DEQ for job scheduling. In Section 4.2.2, we prove the second case based on the properties of round-robin scheduling and the results of the first case.

Let $\chi = (\tau, \pi)$ be the schedule of a job set $\mathcal{J}$ produced by G-RAD. For simplicity we shall use the notations $R(\mathcal{J}) = R_\chi(\mathcal{J})$ and $\bar{R}(\mathcal{J}) = \bar{R}_\chi(\mathcal{J})$ in the remainder of this section.

4.2.1 Mean Response Time Analysis under Light Load

In this section, we bound the total response time of G-RAD when $|\mathcal{J}| \leq P$. The bound uses the squashed work area $sqw(\mathcal{J})$, and the aggregate span $T_\infty(\mathcal{J})$ of the job set $\mathcal{J}$. Specifically, we shall show that

$$R(\mathcal{J}) \leq \left(2 - \frac{2}{|\mathcal{J}| + 1}\right) \frac{\rho + 1}{\delta} sqw(\mathcal{J}) + \frac{2}{1 - \delta} T_\infty(\mathcal{J}) + |\mathcal{J}| L(\log_p P + 1) .$$

We begin by defining several auxiliary concepts.

Definition 4.1 Suppose that a job set $\mathcal{J}$ is scheduled by G-RAD on P processors. For any job $J_i \in \mathcal{J}$, let $S(J_i)$ and $D(J_i)$ denote the sets of satisfied and deprived time steps, respectively. The **satisfied count** of a job J_i is $sat(J_i) = |S(J_i)|$. The **total satisfied count** of $\mathcal{J}$ is

$$sat(\mathcal{J}) = \sum_{J_i \in \mathcal{J}} |S(J_i)| .$$

The **accumulated allotment** of J_i is

$$a(J_i) = \sum_{t=1}^{\infty} a(J_i, t) .$$

The **squashed allotment area** of $\mathcal{J}$ is

$$sqa(\mathcal{J}) = \frac{1}{P} \text{sq-sum}(\langle a(J_i) \rangle) .$$

Thus, $sat(\mathcal{J})$ is the total number of satisfied steps of all jobs in $\mathcal{J}$, $a(J_i)$ is the job J_i 's total allotment on all time steps, and $sqa(\mathcal{J})$ is $1/P$ of the squashed sum of the accumulated allotments for all jobs in $\mathcal{J}$.

The analysis of Lemma 4.7 comprises three major steps:

- **Step 1:** We prove that

$$R(\mathcal{J}) \leq \left(2 - \frac{2}{|\mathcal{J}| + 1}\right) squa(\mathcal{J}) + sat(\mathcal{J}) ,$$

thereby relating the total response time $R(\mathcal{J})$ to the squashed allotment area $sqa(\mathcal{J})$ and the total satisfied count $sat(\mathcal{J})$.

- **Step 2:** We bound the squashed allotment area $sqa(\mathcal{J})$ in terms of the squashed work area $sqw(\mathcal{J})$.
- **Step 3:** We bound the total satisfied count $sat(\mathcal{J})$ in terms of the aggregate span $T_\infty(\mathcal{J})$. Since both $sqw(\mathcal{J})$ and $T_\infty(\mathcal{J})$ are lower bounds on the total response time, we can derive an upper bound of the mean response time against the optimal.

Step 1

We bound the total response time $R(\mathcal{J})$ of G-RAD in terms of the squashed allotment area $sqa(\mathcal{J})$ and total satisfied count $sat(\mathcal{J})$. This proof is similar to the analysis of Lemma 3.4 which relates total response time with squashed work and aggregated span when using instantaneous parallelism as feedback. Lemma 3.4 is indeed a special case of the following lemma. Their relationship will be illustrated further after the proof.

Lemma 4.4 *Suppose that a job set $\mathcal{J}$ is scheduled by G-RAD on a machine with P processors where $|\mathcal{J}| \leq P$. The total response time of $\mathcal{J}$ can be bounded as*

$$R(\mathcal{J}) \leq \left(2 - \frac{2}{|\mathcal{J}| + 1}\right) squa(\mathcal{J}) + sat(\mathcal{J}) . \quad (4.4)$$

Proof. Suppose that G-RAD produces a schedule $\chi = (\tau, \pi)$ for $\mathcal{J}$. Since $|\mathcal{J}| \leq P$, RAD always uses DEQ to allocate processors to jobs.

This proof uses the t-suffix notation $\vec{t}$ as defined in Section 3.3. Like the proof of Lemma 3.4, we shall prove this lemma by induction on the remaining execution time steps of the job set $\mathcal{J}(\vec{t})$.

Basis: $t = T(\mathcal{J}) + 1$. Since we have $\mathcal{J}(\vec{t}) = \emptyset$, it follows that $R(\mathcal{J}(\vec{t})) = 0$, $sqa(\mathcal{J}(\vec{t})) = 0$, and $sat(\mathcal{J}(\vec{t})) = 0$. Thus, the claim holds trivially.

Induction: $1 \leq t \leq T(\mathcal{J})$. Let $n = |\mathcal{J}(\vec{t})|$ denote the number of uncompleted jobs at time step t . We have $n \geq |\mathcal{J}(\vec{t+1})|$ since the number of uncompleted jobs decreases monotonically for a batched job set.

For the induction hypothesis, suppose that Inequality (4.4) holds at time step $t + 1$, *i.e.*,

$$\begin{aligned} R(\mathcal{J}(\vec{t+1})) &\leq \left(2 - \frac{2}{|\mathcal{J}(\vec{t+1})| + 1}\right) sq_a(\mathcal{J}(\vec{t+1})) + sat(\mathcal{J}(\vec{t+1})) \\ &\leq \left(2 - \frac{2}{n+1}\right) sq_a(\mathcal{J}(\vec{t+1})) + sat(\mathcal{J}(\vec{t+1})) . \end{aligned} \quad (4.5)$$

We prove that it still holds at time step t , *i.e.*,

$$R(\mathcal{J}(\vec{t})) \leq \left(2 - \frac{2}{n+1}\right) sq_a(\mathcal{J}(\vec{t})) + sat(\mathcal{J}(\vec{t})) . \quad (4.6)$$

The following notations denote the changes in, respectively, the total response time, the squashed work area, and the aggregate span from time step t and $t + 1$:

$$\begin{aligned} \Delta rt &= R(\mathcal{J}(\vec{t})) - R(\mathcal{J}(\vec{t+1})) , \\ \Delta sq_a &= sq_a(\mathcal{J}(\vec{t})) - sq_a(\mathcal{J}(\vec{t+1})) , \\ \Delta sat &= sat(\mathcal{J}(\vec{t})) - sat(\mathcal{J}(\vec{t+1})) . \end{aligned}$$

Given the induction hypothesis (Inequality (4.5)), we need only prove that

$$\left(2 - \frac{2}{n+1}\right) \Delta sq_a + \Delta sat \geq \Delta rt \quad (4.7)$$

in order to prove our claim (Inequality (4.6)). We divide the proof of Inequality (4.7) into four steps.

(1) To bound Δrt : At any time t , the total number of uncompleted jobs is $|\mathcal{J}(\vec{t})| \leq n$. Since each uncompleted job in $\mathcal{J}(\vec{t})$ adds one time step to the total

4.2. Mean Response Time of G-RAD for Batched Jobs

67

response time during step t , we have

$$\Delta rt \leq n . \quad (4.8)$$

(2) To bound Δsat : At time step t , an uncompleted job J_i is either satisfied or deprived. Thus, the uncompleted jobs can be partitioned as $\mathcal{J}(\vec{t}) = \mathcal{JS}(t) \cup \mathcal{JD}(t)$, representing the set of satisfied and deprived jobs at time t , respectively. If $J_i \in \mathcal{JS}(t)$, the satisfied count of J_i must reduce by 1 at time step t , i.e., $sat(J_i(\vec{t})) = sat(J_i(\vec{t+1})) + 1$. If $J_i \in \mathcal{JD}(t)$, the satisfied count of J_i never increases at any time step t , i.e., $sat(J_i(\vec{t})) \geq sat(J_i(\vec{t+1}))$. Therefore, the total satisfied count of $\mathcal{J}$ must reduce by at least $|\mathcal{JS}(t)|$ at time step t , and we have

$$\Delta sat \geq |\mathcal{JS}(t)| . \quad (4.9)$$

(3) To bound Δsqa :

Since $\mathcal{JD}(t)$ denotes the set of deprived jobs at time t , we consider two cases depending on whether $|\mathcal{JD}(t)| = 0$.

Case 1. $|\mathcal{JD}(t)| = 0$: There exist no deprived jobs at time t . In this case, we have

$$\Delta sqa \geq 0. \quad (4.10)$$

Case 2. $|\mathcal{JD}(t)| > 0$: There is at least one deprived job at time t . In this case, all of the P processors must be allotted to the jobs (otherwise, there would not be any deprived jobs). Therefore, the algebraic sum of the total allotment of the uncompleted jobs increases by P . Applying Lemma 3.2 by replacing $S = P$ and $l = |\mathcal{JD}(t)|$, we have

$$\begin{aligned} \Delta sqa &= \frac{\text{sq-sum}(a(\mathcal{J}, \vec{t})) - \text{sq-sum}(a(\mathcal{J}, \vec{t+1}))}{P} \\ &\geq \frac{P(|\mathcal{JD}(t)| + 1)/2}{P} \\ &= \frac{|\mathcal{JD}(t)| + 1}{2} . \end{aligned} \quad (4.11)$$

(4) **To derive Inequality (4.7):** At any time step t , an uncompleted job is either satisfied or deprived. Therefore, we have

$$|\mathcal{JS}(t)| + |\mathcal{JD}(t)| = n . \quad (4.12)$$

We now prove Inequality (4.7) by combining the bounds on Δrt , Δsat , and Δsqa . In the case where $|\mathcal{JD}(t)| = 0$, all n jobs are satisfied, *i.e.*, $|\mathcal{JS}(t)| = n$. Inequalities (4.8), (4.9), and (4.10) indicate that Inequality (4.7) holds. In the case that $|\mathcal{JD}(t)| > 0$, the following derivation shows that Inequality (4.7) also holds:

$$\begin{aligned} \left(2 - \frac{2}{n+1}\right) \Delta sqa + \Delta sat &\geq \left(2 - \frac{2}{n+1}\right) \frac{|\mathcal{JD}(t)| + 1}{2} + |\mathcal{JS}(t)| \\ &= \frac{n}{n+1} (n+1 - |\mathcal{JS}(t)|) + |\mathcal{JS}(t)| \\ &= n + \frac{|\mathcal{JS}(t)|}{n+1} \\ &\geq n \\ &\geq \Delta rt . \end{aligned}$$

Hence, the proof of the induction is complete, which proves the lemma. $\square$

Lemma 4.4 relates the total response time of a job set scheduled by RAD to the squashed allotment area and the total satisfied count. The relationship holds regardless of the behavior of the thread schedulers, as long as they can send their desires to OS allocator at the beginning of each scheduling quantum.

Lemma 3.4 is a special case of Lemma 4.4, where a greedy thread scheduler with instantaneous parallelism is applied, which can be seen as follows. With instantaneous parallelism, a job's allotment at each time step is equal to its work done at the time step, so we have $squ(\mathcal{J}) = sqa(\mathcal{J})$; a satisfied time step always reduces the span of a job by 1, and so we have $sat(J_i) \leq T_\infty(J_i)$ and $sat(\mathcal{J}) \leq T_\infty(\mathcal{J})$. Then, from Lemma 4.4, we can derive Lemma 3.4 easily.

Step 2

We bound the squashed allotment area $sqa(\mathcal{J})$ in terms of the squashed work area $squ(\mathcal{J})$.

Lemma 4.5 *Suppose that a job set $\mathcal{J}$ is scheduled by G-RAD. Let ρ denote A-GREEDY's responsiveness parameter, and δ its utilization parameter. The squashed*

4.2. Mean Response Time of G-RAD for Batched Jobs

69

allotment area of $\mathcal{J}$ can be bounded as

$$sqa(\mathcal{J}) \leq \frac{\rho+1}{\delta} sqw(\mathcal{J}) , \quad (4.13)$$

where $sqw(\mathcal{J})$ is the squashed work area of the job set $\mathcal{J}$.

Proof. Let $c = (\rho+1)/\delta$. We first show that $a(J_i) \leq cT_1(J_i)$ for every job $J_i \in \mathcal{J}$. According to Lemma 4.1, any job J_i wastes at most $w(J_i) = ((\rho+1-\delta)/\delta)T_1(J_i)$ processor cycles. For each job J_i , we have

$$\begin{aligned} a(J_i) &= T_1(J_i) + w(J_i) \\ &\leq ((\rho+1-\delta)/\delta)T_1(J_i) + T_1(J_i) \\ &= cT_1(J_i) . \end{aligned}$$

To complete the proof, we use Definition 3.1 and apply the squashed sum property proved in Lemma 3.1:

$$\begin{aligned} sqa(\mathcal{J}) &= (1/P) \text{sq-sum}(\langle a(J_i) \rangle) \\ &\leq (1/P) \text{sq-sum}(\langle cT_1(J_i) \rangle) \\ &= c \cdot (1/P) \text{sq-sum}(\langle T_1(J_i) \rangle) \\ &= c \cdot sqw(\mathcal{J}) . \end{aligned}$$

□

Step 3

We bound the total satisfied count $sat(\mathcal{J})$ in terms of the aggregate span $T_\infty(\mathcal{J})$.

Lemma 4.6 *Suppose that a job set $\mathcal{J}$ is scheduled by G-RAD. Let ρ denote A-GREEDY's responsiveness parameter, and δ its utilization parameter. The total satisfied count of $\mathcal{J}$ can be bounded as*

$$sat(\mathcal{J}) \leq \frac{2}{1-\delta} T_\infty(\mathcal{J}) + |\mathcal{J}| (L \log_\rho P + L) . \quad (4.14)$$

Proof. We bound the total satisfied count using Lemma 4.1:

$$sat(\mathcal{J}) = \sum_{J_i \in \mathcal{J}} |S(J_i)|$$

$$\begin{aligned}
&\leq \sum_{J_i \in \mathcal{J}} \left(\frac{2T_\infty(J_i)}{1-\delta} + L \log_\rho P + L \right) \\
&= \frac{2}{1-\delta} T_\infty(\mathcal{J}) + |\mathcal{J}| (L \log_\rho P + L) .
\end{aligned}$$

□

By applying the results of Lemmas 4.4, 4.5, and 4.6, we can easily show Lemma 4.7, which gives a bound on total response time when $|\mathcal{J}| \leq P$.

Lemma 4.7 *Suppose that a job set $\mathcal{J}$ is scheduled by G-RAD on P processors where $|\mathcal{J}| \leq P$. Let ρ denote A-GREEDY's responsiveness parameter, δ its utilization parameter, and L the quantum length. The total response time $R(\mathcal{J})$ of the schedule is at most*

$$R(\mathcal{J}) \leq \left(2 - \frac{2}{|\mathcal{J}|+1} \right) \frac{\rho+1}{\delta} sqw(\mathcal{J}) + \frac{2}{1-\delta} T_\infty(\mathcal{J}) + |\mathcal{J}| L(\log_\rho P + 1) .$$

Proof. From Inequalities (4.4), (4.13), and (4.14), we obtain

$$\begin{aligned}
R(\mathcal{J}) &\leq \left(2 - \frac{2}{|\mathcal{J}|+1} \right) sqw(\mathcal{J}) + sat(\mathcal{J}) \\
&\leq \left(2 - \frac{2}{|\mathcal{J}|+1} \right) \frac{\rho+1}{\delta} sqw(\mathcal{J}) + \frac{2}{1-\delta} T_\infty(\mathcal{J}) + |\mathcal{J}| L(\log_\rho P + 1) .
\end{aligned}$$

□

4.2.2 Mean Response Time Analysis under Heavy Load

We now derive the mean response time of G-RAD for batched jobs when $|\mathcal{J}| > P$. Since all jobs in the job set $\mathcal{J}$ arrive at time step 0, the number of uncompleted jobs decreases monotonically. When the number of uncompleted jobs drops to P or below, G-RAD switches its OS allocator from round-robin to DEQ.

In the analysis, we first prove two technical lemmas which show the properties of round-robin as an OS allocator. Then, we analyze the completion time of the jobs which are scheduled by round-robin during their entire execution. Finally, we apply the response time bound in Lemma 4.7, and bound the response time of G-RAD in general. This analysis on round-robin part is similar to the proof of Theorem 3.6. The difference is that the analysis of G-RAD is applicable to round-robin with any quantum size L while I-RAD sets the quantum size $L = 1$ to include only one time step.

Recall that a batched job set $\mathcal{J}$ can be divided into two subsets, an RR set and a DEQ set. The RR set $\mathcal{J}_{RR}$ includes all the jobs in $\mathcal{J}$ which are entirely scheduled by round-robin for their execution. The DEQ set $\mathcal{J}_{DEQ}$ includes all the jobs in $\mathcal{J}$ which are scheduled by round-robin at the beginning and by DEQ eventually. There exists a unique final RR quantum q such that q is the last quantum scheduled by round-robin, while the quanta from $q + 1$ onwards are all scheduled by DEQ. According to RAD, there must be greater than P uncompleted jobs at the beginning of q , and less than or equal to P uncompleted jobs immediately after the execution of q . Let σ denote the total number of uncompleted jobs immediately after the execution of the final RR quantum. We know that $\sigma = |\mathcal{J}_{DEQ}|$, and $\sigma \leq P$. Let π denote a permutation that lists the jobs according to the nondecreasing order of their completion time, *i.e.*, $T(J_{\pi(1)}) \leq T(J_{\pi(2)}) \leq \dots \leq T(J_{\pi(|\mathcal{J}|)})$. We have $\mathcal{J}_{RR} = \{J_{\pi(i)} \mid 1 \leq i \leq |\mathcal{J}| - \sigma\}$ and $\mathcal{J}_{DEQ} = \{J_{\pi(i)} \mid i > |\mathcal{J}| - \sigma\}$, *i.e.*, $\mathcal{J}_{DEQ}$ includes the σ jobs that are completed last, and $\mathcal{J}_{RR}$ includes the other $|\mathcal{J}| - \sigma$ jobs.

The following two technical lemmas present the properties of round-robin as an OS allocator. The first lemma shows that jobs make almost the same progress on the execution of their work when they are scheduled by round-robin. The second lemma relates the work of jobs to their completion time.

Lemma 4.8 *Suppose that a batched job set $\mathcal{J}$ is scheduled by G-RAD on a machine with P processors where $|\mathcal{J}| > P$. At any time step t scheduled by round-robin, for any two uncompleted jobs J_i and J_j , we have $|T_1(J_i(\overleftarrow{t})) - T_1(J_j(\overleftarrow{t}))| \leq L$, where L is the length of the scheduling quantum.*

Proof. Under the schedule of round-robin, the allotted processors are used efficiently. At any time step, by definition of round-robin, each job only gets at most one processor. Since an uncompleted job has at least one ready thread at any time step, the allotted processors are always making progress on useful work.

Since round-robin gives an equal share of processors to all uncompleted jobs, for any two jobs that arrive at the same time, the allotments differ by at most L at any time. Because the allotted processors are always making progress on useful work, the work done for any two uncompleted jobs differs by at most L at any time before their completion. This completes the proof. $\square$

Lemma 4.9 *Suppose that a batched job set $\mathcal{J}$ is scheduled by G-RAD on a machine with P processors, where $|\mathcal{J}| > P$. The following two statements hold:*

4.9.1. If $J_i \in \mathcal{J}_{RR}$, $J_j \in \mathcal{J}_{RR}$, and $T_1(J_i) < T_1(J_j)$, then $T(J_i) \leq T(J_j)$.

4.9.2. If $J_i \in \mathcal{J}_{RR}$ and $J_j \in \mathcal{J}_{DEQ}$, then $T_1(J_i) \leq T_1(J_j)$.

Proof. We prove the Statement 4.9.1 by showing that its contrapositive is true. If $J_i \in \mathcal{J}_{RR}$, $J_j \in \mathcal{J}_{RR}$, and $T(J_i) < T(J_j)$, we shall show that $T_1(J_i) \leq T_1(J_j)$. Let $t = T(J_i)$. At time step t , job J_i completes work $T_1(J_i)$. From Lemma 4.8, we know that $T_1(J_j(\overleftarrow{t})) \geq T_1(J_i(\overleftarrow{t})) - L = T_1(J_i) - L$. Since job J_j completes after job J_i , job J_j takes at least one more scheduling quantum than J_i to complete its execution. Thus the work done for J_j during the period from t to $T(J_j)$ is at least L . Therefore, we have $T_1(J_j) = T_1(J_i(\overleftarrow{T(J_j)})) \geq T_1(J_i(\overleftarrow{t})) + L \geq T_1(J_i)$.

For any two jobs $J_i \in \mathcal{J}_{RR}$, and $J_j \in \mathcal{J}_{DEQ}$, we have $T(J_i) < T(J_j)$. By using similar analysis, we can prove the Statement 4.9.2. $\square$

Lemma 4.9 relates the work of jobs to their completion time. Statement 4.9.2 tells us that only the σ jobs with largest work are scheduled by DEQ eventually, and the other $|\mathcal{J}| - \sigma$ jobs are scheduled by round-robin for their overall execution. Moreover, according to Statement 4.9.1, under the schedule of round-robin, the jobs with less work are completed more quickly than those with more work. Consider the jobs according to their work such that $T_1(J_1) \leq T_1(J_2) \leq \dots \leq T_1(J_{|\mathcal{J}|})$. From Lemma 4.9, we have $\mathcal{J}_{RR} = \{J_i \mid 1 \leq i \leq |\mathcal{J}| - \sigma\}$ and $\mathcal{J}_{DEQ} = \{J_i \mid i > |\mathcal{J}| - \sigma\}$.

The following lemma bounds the completion time of the jobs.

Lemma 4.10 *Suppose that G-RAD schedules a batched job set $\mathcal{J}$ on a machine with P processors where $|\mathcal{J}| > P$, and let $T_1(J_i)$ denote the work of a job $J_i \in \mathcal{J}$. Consider the jobs according to their work such that $T_1(J_1) \leq T_1(J_2) \leq \dots \leq T_1(J_{|\mathcal{J}|})$. For $1 \leq i \leq |\mathcal{J}| - \sigma$, the completion time $T(J_i)$ of a job J_i satisfies*

$$T(J_i) \leq \left((|\mathcal{J}| - i + 1) T_1(J_i) + \sum_{1 \leq j < i} T_1(J_j) \right) / P + L.$$

Proof. Since we consider the jobs according to their work, from Lemma 4.9, we have $J_i \in \mathcal{J}_{RR}$ where $1 \leq i \leq |\mathcal{J}| - \sigma$. Such a job J_i completes its overall execution under the schedule of round-robin as the OS allocator.

We first evaluate $T_1(J_i(\overleftarrow{t}))$, which is the work done for J_i up to a time step t . Suppose that the job J_i terminates at the end of a quantum q where $T(J_i) = q(L+1) - 1$. Let $t = qL - 1$ be the end of quantum $q - 1$, which is L steps before the completion of J_i . The work done for J_i in interval $\overleftarrow{t}$ is $T_1(J_i(\overleftarrow{t})) = T_1(J_i) - L$.

4.2. Mean Response Time of G-RAD for Batched Jobs

73

According to Lemma 4.8, no job completes more than $T_1(J_i(\overleftarrow{t})) + L$ amount of work in interval $\overleftarrow{t}$. Therefore, for any job J_j with $j > i$, we have

$$\begin{aligned} T_1(J_j(\overleftarrow{t})) &\leq T_1(J_i(\overleftarrow{t})) + L \\ &= T_1(J_i) . \end{aligned} \quad (4.15)$$

For each job J_j where $j < i$, by definition, we always have

$$T_1(J_j(\overleftarrow{t})) \leq T_1(J_j) . \quad (4.16)$$

Thus, at time step t , from Inequalities (4.15) and (4.16), the total work done for the job set $\mathcal{J}$ is

$$\begin{aligned} T_1(\mathcal{J}(\overleftarrow{t})) &= \sum_{J_j \in \mathcal{J}} T_1(J_j(\overleftarrow{t})) \\ &= \sum_{1 \leq j < i} T_1(J_j(\overleftarrow{t})) + T_1(J_i(\overleftarrow{t})) + \sum_{i < j \leq |\mathcal{J}|} T_1(J_j(\overleftarrow{t})) \\ &\leq \sum_{1 \leq j < i} T_1(J_j) + T_1(J_i) + \sum_{i < j \leq |\mathcal{J}|} T_1(J_j) \\ &= (|\mathcal{J}| - i + 1)T_1(J_i) + \sum_{1 \leq j < i} T_1(J_j) . \end{aligned} \quad (4.17)$$

Since round-robin always allots all processors to jobs, and all allotted processors are doing useful work, round-robin executes P ready threads at any time step. Thus, the total work done for job set $\mathcal{J}$ increases by P at each time step. From Inequality (4.17), we have

$$\begin{aligned} t &= T_1(\mathcal{J}(\overleftarrow{t})) / P \\ &\leq \left((|\mathcal{J}| - i + 1)T_1(J_i) + \sum_{1 \leq j < i} T_1(J_j) \right) / P . \end{aligned} \quad (4.18)$$

Since $T(J_i) = t + L$, given Inequality (4.18), we complete the proof. $\square$

The following lemma bounds the total response time of job sets scheduled by G-RAD when $|\mathcal{J}| > P$, where $sqw(\mathcal{J})$ is squashed work area, and $T_\infty(\mathcal{J})$ is the aggregate span.

Lemma 4.11 *Suppose that a batched job set $\mathcal{J}$ is scheduled by G-RAD on a ma-*

chine with P processors where $|\mathcal{J}| > P$. Let ρ denote A-GREEDY's responsiveness parameter, and δ its utilization parameter. The response time $R(\mathcal{J})$ of $\mathcal{J}$ is at most

$$R(\mathcal{J}) \leq \left(2 - \frac{2}{|\mathcal{J}| + 1}\right) \frac{\rho + 1}{\delta} sqw(\mathcal{J}) + \frac{2}{1 - \delta} T_\infty(\mathcal{J}) + |\mathcal{J}|L + O(LP \log_\rho P). \quad (4.19)$$

Proof. The jobs in $\mathcal{J}$ can be divided into RR set $\mathcal{J}_{RR}$ and DEQ set $\mathcal{J}_{DEQ}$. Let $n = |\mathcal{J}|$ denote the number of jobs in $\mathcal{J}$. Recall that σ denotes the number of jobs in $\mathcal{J}_{DEQ}$, i.e., $\sigma \leq P$. Consider the jobs in the ascending order of their completion time such that $T(J_1) \leq T(J_2) \leq \dots \leq T(J_n)$. From Lemma 4.9, we have $\mathcal{J}_{RR} = \{J_i \mid 1 \leq i \leq n - \sigma\}$ and $\mathcal{J}_{DEQ} = \{J_i \mid i > n - \sigma\}$. We will calculate the total response time of the jobs in $\mathcal{J}_{RR}$ and $\mathcal{J}_{DEQ}$ respectively.

(1) Calculate $R(\mathcal{J}_{RR})$. According to Lemma 4.10, the completion time of a job $J_i \in \mathcal{J}_{RR}$ is $T(J_i) \leq (1/P)((n - i + 1)T_1(J_i) + \sum_{1 \leq j < i} T_1(J_j)) + L$. Thus, the total response time of the jobs in $\mathcal{J}_{RR}$ is

$$\begin{aligned} R(\mathcal{J}_{RR}) &= \sum_{1 \leq i \leq n - \sigma} T(J_i) \\ &\leq \sum_{1 \leq i \leq n - \sigma} \left(\frac{(n - i + 1)T_1(J_i) + \sum_{1 \leq j < i} T_1(J_j)}{P} + L \right) \\ &= \frac{1}{P} \left(\sum_{1 \leq i \leq n - \sigma} (n - i + 1)T_1(J_i) + \sum_{1 \leq i \leq n - \sigma} \sum_{1 \leq j < i} T_1(J_j) \right) + L(n - \sigma) \\ &= \frac{1}{P} \left(\sum_{1 \leq i \leq n - \sigma} (n - i + 1)T_1(J_i) + \sum_{1 \leq i \leq n - \sigma} (n - \sigma - i)T_1(J_i) \right) + L(n - \sigma) \\ &= \frac{1}{P} \sum_{1 \leq i \leq n - \sigma} (2n - \sigma - 2i + 1)T_1(J_i) + L(n - \sigma) \\ &< \frac{1}{P} \sum_{1 \leq i \leq n - \sigma} (2n - \sigma - 2i + 1)T_1(J_i) + Ln. \end{aligned} \quad (4.20)$$

(2) Calculate $R(\mathcal{J}_{DEQ})$. The σ jobs in $\mathcal{J}_{DEQ}$ are scheduled by round-robin until the time step $t = T(J_{n - \sigma})$, which is the completion of the job $J_{n - \sigma}$, and scheduled by DEQ afterward. The total response time of $\mathcal{J}_{DEQ}$ is

$$R(\mathcal{J}_{DEQ}) = R\left(\mathcal{J}_{DEQ}\left(\overline{t+1}\right)\right) + \sigma \cdot t. \quad (4.21)$$

4.2. Mean Response Time of G-RAD for Batched Jobs

75

From Lemma 4.10, we know that the completion time of the job $J_{n-\sigma}$ is

$$t \leq \left((\sigma + 1)T_1(J_{n-\sigma}) + \sum_{1 \leq i < n-\sigma} T_1(J_i) \right) / P + L. \quad (4.22)$$

To get $R(\mathcal{J}_{DEQ})$, we only need to calculate $R(\mathcal{J}_{DEQ}(\overline{t+1}))$.

Since the job set $\mathcal{J}_{DEQ}$ is scheduled by DEQ as the OS allocator from time step t onwards, we can apply the total response time bound in Lemma 4.7 to calculate $R(\mathcal{J}_{DEQ}(\overline{t+1}))$. During the interval $\overleftarrow{t}$, job $J_{n-\sigma}$ completes work $T_1(J_{n-\sigma})$. From Lemma 4.8, we know that each job J_i with $i > n - \sigma$ has completed its work by at least $T_1(J_{n-\sigma}) - L$. Thus, such a job J_i has remaining work $T_1(J_i(\overline{t+1})) \leq T_1(J_i) - T_1(J_{n-\sigma}) + L$. The squashed work of $\mathcal{J}_{DEQ}(\overline{t+1})$ is

$$\begin{aligned} & sqw(\mathcal{J}_{DEQ}(\overline{t+1})) \\ &= \frac{1}{P} \text{sq-sum}(\langle T_1(J_i(\overline{t+1})) \mid n - \sigma + 1 \leq i \leq n \rangle) \\ &\leq \frac{1}{P} \text{sq-sum}(\langle T_1(J_i) - T_1(J_{n-\sigma}) + L \mid n - \sigma + 1 \leq i \leq n \rangle) \\ &= \frac{1}{P} \sum_{n-\sigma+1 \leq i \leq n} (n - i + 1)(T_1(J_i) - T_1(J_{n-\sigma}) + L) \\ &= \frac{1}{P} \sum_{n-\sigma+1 \leq i \leq n} (n - i + 1)T_1(J_i) - \frac{(1 + \sigma)\sigma}{2P}T_1(J_{n-\sigma}) + \frac{(1 + \sigma)\sigma}{2P}L \quad (4.23) \end{aligned}$$

$$\leq \frac{1}{P} \sum_{n-\sigma+1 \leq i \leq n} (n - i + 1)T_1(J_i) - \frac{(1 + \sigma)\sigma}{2P}T_1(J_{n-\sigma}) + PL. \quad (4.24)$$

Let the constant $c = 2 - 2/(1 + P) < 2$. According to Lemma 4.7, we have

$$\begin{aligned} & R(\mathcal{J}_{DEQ}(\overline{t+1})) \\ &\leq c \cdot \frac{\rho + 1}{\delta} sqw(\mathcal{J}_{DEQ}(\overline{t+1})) + \frac{2}{1 - \delta} T_\infty(\mathcal{J}_{DEQ}(\overline{t+1})) + \sigma(L \log_\rho \sigma + L) \\ &\leq c \cdot \frac{\rho + 1}{\delta} sqw(\mathcal{J}_{DEQ}(\overline{t+1})) + \frac{2}{1 - \delta} T_\infty(\mathcal{J}) + PL(\log_\rho P + 1) \\ &= c \cdot \frac{\rho + 1}{\delta} sqw(\mathcal{J}_{DEQ}(\overline{t+1})) + E_1, \quad (4.25) \end{aligned}$$

where

$$E_1 = \frac{2}{1-\delta} T_\infty(\mathcal{J}) + PL(\log_p P + 1). \quad (4.26)$$

We now calculate the response time of $\mathcal{J}_{DEQ}$. Since we know that $c = 2 - 2/(1 + P) > 1$, the responsiveness parameter $\rho > 1$, and the utilization parameter $\delta \leq 1$, we have $c(\rho + 1)/\delta > 2$. Given Equations (4.21) and (4.26) and Inequalities (4.22), (4.23), and (4.25), the response time of $\mathcal{J}_{DEQ}$ is

$$\begin{aligned} R(\mathcal{J}_{DEQ}) &= R(\mathcal{J}_{DEQ}(\overline{t+1})) + \sigma \cdot t \\ &\leq c \cdot \frac{\rho+1}{\delta} sqw(\mathcal{J}_{DEQ}(\overline{t+1})) + E_1 + \sigma \cdot t \\ &\leq c \cdot \frac{\rho+1}{\delta} \left(\frac{1}{P} \sum_{n-\sigma+1 \leq i \leq n} (n-i+1)T_1(J_i) - \frac{(\sigma+1)\sigma}{2P} T_1(J_{n-\sigma}) + PL \right) + E_1 \\ &\quad + \frac{(\sigma+1)\sigma}{P} T_1(J_{n-\sigma}) + \frac{\sigma}{P} \sum_{1 \leq i < n-\sigma} T_1(J_i) + PL \\ &\leq c \cdot \frac{\rho+1}{\delta P} \sum_{n-\sigma+1 \leq i \leq n} (n-i+1)T_1(J_i) + \frac{\sigma}{P} \sum_{1 \leq i < n-\sigma} T_1(J_i) + E_2, \end{aligned} \quad (4.27)$$

where

$$E_2 = E_1 + \left(c \cdot \frac{\rho+1}{\delta} + 1 \right) PL. \quad (4.28)$$

(3) Calculate $R(\mathcal{J})$. Given $R(\mathcal{J}_{RR})$ in Inequality (4.20), $R(\mathcal{J}_{DEQ})$ in Inequality (4.27), and $c(\rho + 1)/\delta > 2$, the response time of $\mathcal{J}$ is the sum of them as follows:

$$\begin{aligned} R(\mathcal{J}) &= R(\mathcal{J}_{RR}) + R(\mathcal{J}_{DEQ}) \\ &< \frac{1}{P} \sum_{1 \leq i \leq n-\sigma} (2n - \sigma - 2i + 1)T_1(J_i) + Ln \\ &\quad + c \cdot \frac{\rho+1}{\delta P} \sum_{n-\sigma+1 \leq i \leq n} (n-i+1)T_1(J_i) + \frac{\sigma}{P} \sum_{1 \leq i < n-\sigma} T_1(J_i) + E_2 \\ &= \frac{1}{P} \sum_{1 \leq i \leq n-\sigma} (2n - 2i + 1)T_1(J_i) + c \cdot \frac{\rho+1}{\delta P} \sum_{n-\sigma+1 \leq i \leq n} (n-i+1)T_1(J_i) \\ &\quad + \frac{\sigma}{P} \sum_{1 \leq i < n-\sigma} T_1(J_i) - \frac{\sigma}{P} \sum_{1 \leq i \leq n-\sigma} T_1(J_i) + E_2 + Ln \end{aligned}$$

4.2. Mean Response Time of G-RAD for Batched Jobs

77

$$\begin{aligned}
&\leq c \cdot \frac{\rho+1}{\delta P} \sum_{J_i \in \mathcal{J}} (n-i+1) T_1(J_i) + Ln + E_2 \\
&= \left(2 - \frac{2}{n+1}\right) \frac{\rho+1}{\delta} sqw(\mathcal{J}) + \frac{2}{1-\delta} T_\infty(\mathcal{J}) + Ln + O(PL \log_p P)
\end{aligned}$$

□

Combining Lemma 4.7 and Lemma 4.11, we obtain the following theorem, which bounds the total response time of a batched job set scheduled by G-RAD.

Theorem 4.12 *Suppose that a job set $\mathcal{J}$ is scheduled by G-RAD on P processors. Let ρ be A-GREEDY's responsiveness parameter, δ its utilization parameter, and L the quantum length. The total response time $R(\mathcal{J})$ of the schedule is at most*

$$R(\mathcal{J}) = \left(2 - \frac{2}{|\mathcal{J}|+1}\right) \frac{\rho+1}{\delta} sqw(\mathcal{J}) + \frac{2}{1-\delta} T_\infty(\mathcal{J}) + O(|\mathcal{J}| L \log_p P), \quad (4.29)$$

where $sqw(\mathcal{J})$ is the squashed work area of $\mathcal{J}$ and $T_\infty(\mathcal{J})$ is the aggregate span of $\mathcal{J}$.

Since both $sqw(\mathcal{J})/|\mathcal{J}|$ and $T_\infty(\mathcal{J})/|\mathcal{J}|$ are lower bounds on $\bar{R}(\mathcal{J})$, we obtain the following corollary.

Corollary 4.13 *Suppose that a batched job set $\mathcal{J}$ is scheduled by G-RAD. Let ρ denote A-GREEDY's responsiveness parameter, and δ its utilization parameter. The mean response time $\bar{R}(\mathcal{J})$ of the schedule satisfies*

$$\bar{R}(\mathcal{J}) = \left(\left(2 - \frac{2}{|\mathcal{J}|+1}\right) \frac{\rho+1}{\delta} + \frac{2}{1-\delta} \right) \bar{R}^*(\mathcal{J}) + O(L \log_p P),$$

where $\bar{R}^*(\mathcal{J})$ denotes the mean response time of $\mathcal{J}$ scheduled by an optimal clairvoyant scheduler.

Proof. Combine Theorem 4.12 with Inequalities (3.3) and (3.4). □

Since both the quantum length L and the number of processors P are independent variables with respect to any job set $\mathcal{J}$, Corollary 4.13 shows that G-RAD is $O(1)$ -competitive with respect to mean response time for batched jobs. Specifically, when $\delta = 2/3$ and ρ approaches 1, G-RAD's competitiveness ratio approaches the minimum value 12. Thus, G-RAD is $(12 + \epsilon)$ -competitive with respect to mean response time for any constant $\epsilon > 0$.

The competitive ratio of 12 for G-RAD is a pessimistic bound. We expect that in practice, G-RAD should perform closer to optimal. In particular, when the job set $\mathcal{J}$ exhibits reasonably large total parallelism, we have $sqw(\mathcal{J}) \gg T_\infty(\mathcal{J})$, and thus, the term involving $sqw(\mathcal{J})$ in Theorem 7.4 dominates the total response time. More importantly, the OS allocator RAD is not actually an adversary of A-GREEDY, and simulations of A-STEAL [3] suggest that in practice A-GREEDY should produce waste closer to $(1/\delta - 1)T_1(J_i)$. From the proof of Lemma 4.5, one can determine that the coefficient on the term $sqw(\mathcal{J})$ becomes $(2 - 2/(|\mathcal{J}| + 1))/\delta$ when a job's waste is no more than $1/\delta - 1$ times its work. That is to say, in this scenario, the mean response time of a job set scheduled by G-RAD is about $(2/\delta)sqw(\mathcal{J})$. Since δ is typically in the range of 0.5 to 1, if the job set has reasonably large total parallelism, G-RAD is likely to achieve the mean response time of less than 4 times the optimal. This conjecture is affirmed by our simulation results presented in Chapter 5.

4.3 W-RAD Performance

W-RAD is a distributed two-level adaptive scheduler, which schedules and executes individual jobs without central knowledge of all available threads. In this section, we show that W-RAD is $O(1)$ -competitive with respect to makespan for jobs with arbitrary release times and $O(1)$ -competitive with respect to mean response time for batched jobs.

The methods used to analyze W-RAD are similar to those for G-RAD. Since W-RAD is a randomized scheduling algorithm, however, we show that its makespan (or its expected mean response time) is within a factor c of that incurred in an optimal clairvoyant algorithm in expectation, not in the worst case. Let $\chi = (\tau, \pi)$ be the schedule of a job set $\mathcal{J}$ produced by W-RAD. For simplicity we shall use the notations $T(\mathcal{J}) = T_\chi(\mathcal{J})$ and $R(\mathcal{J}) = R_\chi(\mathcal{J})$.

The following lemma bounds the expected satisfied steps of a job scheduled by A-STEAL. It provide a starting point for the analysis.

Lemma 4.14 *Suppose that A-STEAL schedules a job J_i with span $T_\infty(J_i)$ on a machine with P processors. Let ρ denote A-STEAL's responsiveness parameter, δ its utilization parameter, and L the quantum length. Then, it produces at most $48T_\infty(J_i)/(1 - \delta) + L \log_\rho P + L$ satisfied steps in expectation.*

Proof. This bound can be derived from Lemmas 2.16 and 2.19, which bound the

4.3. W-RAD Performance

79

total inefficient quanta and efficient-and-satisfied quanta, respectively. $\square$

According to the analysis of Theorem 4.2, as long as each job J_i 's thread scheduler guarantees to have $O(T_\infty(J_i))$ total satisfied steps and waste only $O(T_1(J_i))$ processor cycles, the OS allocator RAD can ensure $O(1)$ -competitiveness with respect to makespan. Therefore, according to Lemma 4.14 and Theorem 2.25, we can bound the makespan produced by W-RAD as follows in terms of the release time $r(J_i)$, span $T_\infty(J_i)$, and work $T_1(J_i)$ of an individual job $J_i \in \mathcal{J}$, as well as on the total work $T_1(\mathcal{J})$ of the job set $\mathcal{J}$.

Theorem 4.15 *Suppose that W-RAD schedules a job set $\mathcal{J}$ on a machine with P processors. Let ρ denote A-STEAL's responsiveness parameter, δ its utilization parameter, and L the quantum size. Then, we expect W-RAD to complete $\mathcal{J}$ in*

$$\begin{aligned} E[T(\mathcal{J})] = & \left(\frac{\rho+1}{\delta} + \frac{(1+\rho)^2}{\delta(L\delta-1-\rho)} \right) \frac{T_1(\mathcal{J})}{P} \\ & + O\left(\frac{\max_{J_i \in \mathcal{J}} \{r(J_i) + T_\infty(J_i)\}}{1-\delta} \right) + L \log_\rho P + 2L \end{aligned} \quad (4.30)$$

time steps. $\square$

According to the analysis of Theorem 4.12, as long as each job J_i 's thread scheduler guarantees to have $O(T_\infty(J_i))$ total satisfied steps and waste only $O(T_1(J_i))$ processor cycles, the OS allocator RAD can ensure $O(1)$ -competitiveness for batched job sets with respect to mean response time. Therefore, according to Lemma 4.14 and Theorem 2.25, we can bound the mean response time produced by W-RAD as follows.

Theorem 4.16 *Suppose that a batched job set $\mathcal{J}$ is scheduled by W-RAD. Let ρ denote A-STEAL's responsiveness parameter, δ its utilization parameter, and L the quantum length. Then, the expected response time of the schedule satisfies*

$$\begin{aligned} E[R(\mathcal{J})] = & \left(2 - \frac{2}{|\mathcal{J}|+1} \right) \left(\frac{\rho+1}{\delta} + \frac{(1+\rho)^2}{\delta(L\delta-1-\rho)} \right) sqw(\mathcal{J}) \\ & + O\left(\frac{T_\infty(\mathcal{J})}{1-\delta} + |\mathcal{J}| L \log_\rho P \right), \end{aligned}$$

where $sqw(\mathcal{J})$ is the squashed work area, and $T_\infty(\mathcal{J})$ is the aggregate span. $\square$

Remark Theorems 4.15 and 4.16 show that W-RAD is $O(1)$ -competitive for both makespan and, in the batch setting, mean response time. We anticipate that W-RAD's competitive ratios should be better in practice, especially when the total work is much larger than span and the machine is moderately or highly loaded. In this case, the term on $T_1(\mathcal{J})/P$ in Inequality (4.30) is much larger than the term $\max_{J_i \in \mathcal{J}} \{T_\infty(J_i) + r(J_i)\}$, i.e., the term on $T_1(\mathcal{J})/P$ dominates the makespan bound. The proof of Theorem 4.15 calculates the coefficient of $T_1(\mathcal{J})/P$ in Inequality (4.30) as the ratio of the total allotment (total work plus total waste) and the total work. When the OS allocator is RAD, which is not a true adversary, empirical results in Chapter 5 indicate that each job J_i only wastes about $(1/\delta - 1)T_1(J_i)$ processor cycles, which is not as large as the worst-case waste in Theorem 2.25. Therefore, when we use RAD as the OS allocator, the coefficient of $T_1(\mathcal{J})/P$ seems more likely to approach $1/\delta$. In other words, the makespan of a job set $\mathcal{J}$ scheduled by W-RAD might more typically be about $T_1(\mathcal{J})/\delta P$. Since δ is typically in the range of 0.5 to 1, W-RAD may exhibit makespan that is only about 2 times the optimal when the jobs have reasonably large parallelism and the machine is moderately or heavily loaded. Similarly, W-RAD may exhibit only 4 times optimal with respect to the mean response time for batched jobs under the same conditions.

4.4 Two-level Scheduling System with RAD

Using RAD as the OS allocator, this section investigates sufficient conditions of the thread schedulers to make the coupled two-level system efficient with respect to both makespan and mean response time. Given a thread scheduler X , let's name the two-level scheduling system coupling X with RAD as X -RAD. If X can ensure bounded waste and satisfied count, Theorem 4.17 provides a makespan bound on X -RAD.

Theorem 4.17 *Suppose that a job J_i is scheduled by an adaptive thread scheduler X with waste no more than $w(J_i)$, and with satisfied count not more than $\text{sat}(J_i)$. Let P denote the total number of processors on the machine. The makespan of the job set $\mathcal{J}$ produced by the coupled two-level scheduler X -RAD is bounded by*

$$T(\mathcal{J}) \leq \frac{w(\mathcal{J}) + T_1(\mathcal{J})}{P} + \max_{J_i \in \mathcal{J}} (\text{sat}(J_i) + r(J_i)) + L, \quad (4.31)$$

where $w(\mathcal{J})$ denotes the total waste of $\mathcal{J}$ and $\text{sat}(J_i)$ and $r(J_i)$ denote the satisfied count and release time of job J_i , respectively.

4.4. Two-level Scheduling System with RAD

81

Proof. This proof is a generalization of the proof in Theorem 4.2. Suppose that job J_k is the last job completed among the jobs in $\mathcal{J}$. Let $S(J_k)$ denote the set of satisfied steps for J_k and $D(J_k)$ denote its set of deprived steps. We have $T(\mathcal{J}) \leq r(J_k) + L + |S(J_k)| + |D(J_k)|$. According to Definition 4.1, the satisfied count is $\text{sat}(J_i) = |S(J_i)|$. Since we have that $T(\mathcal{J}) \leq |D(J_k)| + \max_{J_i \in \mathcal{J}} (\text{sat}(J_i) + r(J_i)) + L$, we only need to bound $|D(J_k)|$.

To bound the total number of deprived steps $D(J_k)$ of job J_k , observe that for each step $t \in D(J_k)$, RAD applies either DEQ or round-robin as the OS allocator. Round-robin always allots all processors to jobs. By definition, DEQ must have allotted all processors to jobs whenever J_k is deprived. Thus, the total allotment on such a step t is always equal to the total number of processors P . Moreover, the total allotment of $\mathcal{J}$ over J_k 's deprived steps $D(J_k)$ is $a(\mathcal{J}, D(J_k)) = \sum_{t \in D(J_k)} \sum_{J_i \in \mathcal{J}} a(J_i, t) = P |D(J_k)|$. Since any allotted processor is either working or wasted, the total allotment for any job J_i is bounded by the sum of its total work $T_1(J_i)$ and total waste $w(J_i)$. The total number of allotted processor cycles for all jobs in $\mathcal{J}$ is at most $T_1(\mathcal{J}) + w(\mathcal{J})$. Thus, we have $P |D(J_k)| \leq T_1(\mathcal{J}) + w(\mathcal{J})$, which implies that $|D(J_k)| \leq (T_1(\mathcal{J}) + w(\mathcal{J}))/P$, completing the proof. $\square$

Theorem 4.18 provides a mean response time bound in a batched setting.

Theorem 4.18 *Suppose that a job J_i is scheduled by an adaptive thread scheduler X with waste not more than $c \cdot T_1(J_i)$ where c is a constant, and with satisfied count no more than $\text{sat}(J_i)$. Let P denote the total number of processors on the machine, and L denote the length of the scheduling quantum. The total response time of a batched job set $\mathcal{J}$ produced by the coupled two-level scheduler X-RAD is bounded by*

$$R(\mathcal{J}) \leq 2(c+1)\text{sqw}(\mathcal{J}) + \text{sat}(\mathcal{J}) + O(L|\mathcal{J}|), \quad (4.32)$$

where $\text{sqw}(\mathcal{J})$ denotes the squashed work of $\mathcal{J}$ and $\text{sat}(\mathcal{J})$ denotes the total satisfied count of $\mathcal{J}$.

Proof. As with the proof of Theorem 4.12, we consider two cases — light load and heavy load.

Under light load, we know that the total response time produced by X-RAD is bounded by

$$R(\mathcal{J}) \leq 2\text{sqa}(\mathcal{J}) + \text{sat}(\mathcal{J}), \quad (4.33)$$

since Inequality (4.33) depends only on the behavior of RAD and is irrelevant to the thread schedulers. Given the waste of thread scheduler X to be at most $c \cdot T_1(J_i)$, according to Lemma 3.1, we obtain

$$\begin{aligned}
 sq_a(\mathcal{J}) &= (1/P) \text{sq-sum}(\langle a(J_i) \rangle) \\
 &= (1/P) \text{sq-sum}(\langle T_1(J_i) + w(J_i) \rangle) \\
 &\leq (1/P) \text{sq-sum}(\langle (c+1)T_1(J_i) \rangle) \\
 &= (c+1) \cdot (1/P) \text{sq-sum}(\langle T_1(J_i) \rangle) \\
 &= (c+1) \cdot sqw(\mathcal{J}) .
 \end{aligned} \tag{4.34}$$

Combining Inequalities (4.33) and (4.34), we obtain

$$R(\mathcal{J}) \leq 2(c+1)sqw(\mathcal{J}) + sat(\mathcal{J}) , \tag{4.35}$$

under light load.

Under heavy load, we can parrot the analysis in Lemma 4.11. The jobs in $\mathcal{J}$ can be divided into RR set $\mathcal{J}_{RR}$ and DEQ set $\mathcal{J}_{DEQ}$. Let $n = |\mathcal{J}|$ denote the number of jobs in $\mathcal{J}$, and let σ denote the number of jobs in $\mathcal{J}_{DEQ}$, *i.e.*, $\sigma \leq P$. Consider the jobs in the ascending order of their completion time such that $T(J_1) \leq T(J_2) \leq \dots \leq T(J_n)$. From Lemma 4.9, we have $\mathcal{J}_{RR} = \{J_i \mid 1 \leq i \leq n - \sigma\}$ and $\mathcal{J}_{DEQ} = \{J_i \mid i > n - \sigma\}$. We will calculate the total response time of the jobs in $\mathcal{J}_{RR}$ and $\mathcal{J}_{DEQ}$ respectively.

(1) Calculate $R(\mathcal{J}_{RR})$. A reasonable thread scheduler should not waste any processor cycles under the initial sequential execution where the processor allotment of the job is 1. Therefore, any thread scheduler would behave the same in terms of completing the jobs in $R(\mathcal{J}_{RR})$ when using RAD as the OS allocator. According to Lemma 4.10, the completion time of a job $J_i \in \mathcal{J}_{RR}$ is $T(J_i) \leq (1/P)((n - i + 1)T_1(J_i) + \sum_{1 \leq j < i} T_1(J_j)) + L$. Thus, the total response time of the jobs in $\mathcal{J}_{RR}$ is

$$\begin{aligned}
 R(\mathcal{J}_{RR}) &= \sum_{1 \leq i \leq n - \sigma} T(J_i) \\
 &< \frac{1}{P} \sum_{1 \leq i \leq n - \sigma} (2n - \sigma - 2i + 1)T_1(J_i) + Ln.
 \end{aligned} \tag{4.36}$$

(2) Calculate $R(\mathcal{J}_{DEQ})$. The σ jobs in $\mathcal{J}_{DEQ}$ are scheduled by round-robin until the time step $t = T(J_{n-\sigma})$, which is the completion of the job $J_{n-\sigma}$, and scheduled

4.4. Two-level Scheduling System with RAD

83

by DEQ afterward. The total response time of $\mathcal{J}_{DEQ}$ is

$$R(\mathcal{J}_{DEQ}) = R(\mathcal{J}_{DEQ}(\overline{t+1})) + \sigma \cdot t. \quad (4.37)$$

From Lemma 4.10, we know that the completion time of the job $J_{n-\sigma}$ is

$$t \leq \left((\sigma + 1)T_1(J_{n-\sigma}) + \sum_{1 \leq i < n-\sigma} T_1(J_i) \right) / P + L. \quad (4.38)$$

To get $R(\mathcal{J}_{DEQ})$, we only need to calculate $R(\mathcal{J}_{DEQ}(\overline{t+1}))$.

Since the job set $\mathcal{J}_{DEQ}$ is scheduled by DEQ as the OS allocator from time step t onwards, we can apply the total response time bound in Inequality (4.35) to calculate $R(\mathcal{J}_{DEQ}(\overline{t+1}))$. During the interval $\overleftarrow{t}$, job $J_{n-\sigma}$ completes work $T_1(J_{n-\sigma})$. From Lemma 4.8, we know that each job J_i with $i > n - \sigma$ has completed its work by at least $T_1(J_{n-\sigma}) - L$. Thus, such a job J_i has remaining work $T_1(J_i(\overline{t+1})) \leq T_1(J_i) - T_1(J_{n-\sigma}) + L$. The squashed work of $\mathcal{J}_{DEQ}(\overline{t+1})$ is

$$\begin{aligned} & sqw(\mathcal{J}_{DEQ}(\overline{t+1})) \\ &= \frac{1}{P} \text{sq-sum}(\langle T_1(J_i(\overline{t+1})) \mid n - \sigma + 1 \leq i \leq n \rangle) \\ &\leq \frac{1}{P} \sum_{n-\sigma+1 \leq i \leq n} (n - i + 1)T_1(J_i) - \frac{(1 + \sigma)\sigma}{2P} T_1(J_{n-\sigma}) + PL. \end{aligned} \quad (4.39)$$

According to Inequality (4.35), we obtain

$$R(\mathcal{J}_{DEQ}(\overline{t+1})) \leq 2(c + 1) \cdot sqw(\mathcal{J}_{DEQ}(\overline{t+1})) + sat(\mathcal{J}). \quad (4.40)$$

We now calculate the response time of $\mathcal{J}_{DEQ}$. Given Inequalities (4.40), (4.39), and (4.38), the response time of $\mathcal{J}_{DEQ}$ is

$$\begin{aligned} & R(\mathcal{J}_{DEQ}) \\ &= R(\mathcal{J}_{DEQ}(\overline{t+1})) + \sigma \cdot t \\ &\leq 2(c + 1)sqw(\mathcal{J}_{DEQ}(\overline{t+1})) + sat(\mathcal{J}_{DEQ}(\overline{t+1})) \\ &\quad + \sigma \left((\sigma + 1)T_1(J_{n-\sigma}) + \sum_{1 \leq i < n-\sigma} T_1(J_i) / P + L \right) \end{aligned}$$

$$\begin{aligned}
&\leq \frac{2(c+1)}{P} \sum_{n-\sigma+1 \leq i \leq n} (n-i+1)T_1(J_i) \\
&\quad + \frac{\sigma}{P} \sum_{1 \leq i \leq n-\sigma} T_1(J_i) + \text{sat}(\mathcal{J}) + (2c+3)PL. \tag{4.41}
\end{aligned}$$

(3) Calculate $R(\mathcal{J})$. Given $R(\mathcal{J}_{RR})$ in Inequality (4.36) and $R(\mathcal{J}_{DEQ})$ in Inequality (4.41), the response time of $\mathcal{J}$ is the sum of them as follows:

$$\begin{aligned}
R(\mathcal{J}) &= R(\mathcal{J}_{RR}) + R(\mathcal{J}_{DEQ}) \\
&\leq \frac{2(c+1)}{P} \sum_{1 \leq i \leq n} (n-i+1)T_1(J_i) + \text{sat}(\mathcal{J}) + L|\mathcal{J}| + (2c+3)PL \\
&= 2(c+1)\text{sqw}(\mathcal{J}) + \text{sat}(\mathcal{J}) + O(L|\mathcal{J}|).
\end{aligned}$$

We obtain the bound in Inequality (4.32), which proves the theorem. $\square$

Consider an example where a thread scheduler with instantaneous parallelism as feedback has 0 waste and $T_\infty(J_i)$ satisfied count. Theorem 4.17 tells us that I-RAD has makespan bounded by $T_1(\mathcal{J})/P + \max_{J_i \in \mathcal{J}} (T_\infty(J_i) + r(J_i))$, which is 2-competitive. This result is consistent with the result in [30], which shows that DEQ is 2-competitive for makespan. Theorem 4.18 tells us that I-RAD has total response time bounded by $2\text{sqw}(\mathcal{J}) + T_\infty(\mathcal{J})$, which is 3-competitive. It is consistent with the result in Theorem 3.6. Similarly, we can use the waste and satisfied count of A-GREEDY and A-STEAL to derive the efficiency of G-RAD and W-RAD, respectively. Theorem 4.17 and Theorem 4.18 reflect the importance of waste and satisfied count as performance measures for thread schedulers.

Chapter 5

Empirical Results

We built a discrete event simulator using DESMO-J [47] to evaluate the performance of A-STEAL, W-RAD, and G-RAD. We conducted eight sets of experiments on the simulator with synthetic workloads. The first three sets of experiments were designed to evaluate A-STEAL, the fourth set of experiments for W-RAD, and the last four sets of experiments for G-RAD and a variant. Our results are summarized below:

1. The **time experiments** investigate the performance of A-STEAL in over 2300 job runs. A linear-regression analysis of the results provides evidence that the coefficients on the number of accounted and deductible steps are considerably smaller than those in our theoretical bounds. A second linear-regression analysis indicates that A-STEAL on average completes jobs in at most twice the optimal number of time steps, which is the same bound provided by offline greedy scheduling [31, 66].
2. The **waste experiments** measure the waste A-STEAL incurs in practice, comparing the observed waste to the theoretical bounds. Our experiments indicate that the waste is almost insensitive to the parameter settings and is a tiny fraction (less than 10%) of the work for jobs with high parallelism.
3. The **time-waste experiments** compare the completion time and waste of A-STEAL with another adaptive thread scheduler called ABP [5] by running single jobs with predetermined availability profiles. ABP was introduced in [5] and named for the initials of the authors Nimar Arora, Robert Blumofe, and Greg Plaxton. ABP is an adaptive thread scheduler that does not supply parallelism feedback to the OS allocator. These experiments indicate that on large machines, when the mean availability $\bar{P}$ is considerably smaller than the number

of processors P in the machine, A-STEAL completes jobs faster than ABP while wasting fewer processor cycles than ABP. On medium-sized machines, when $\bar{P}$ is of the same order as P , ABP completes jobs slightly faster than A-STEAL, but it still wastes many more processor cycles than A-STEAL.

4. The **utilization experiments** compare the utilization of W-RAD and ABP+EQ when many jobs with varying characteristics are using the same multiprocessor resource. ABP+EQ uses ABP as the thread scheduler and EQ as the OS allocator. The experiments provide evidence that on moderately to heavily loaded large machines, W-RAD consistently yields a higher utilization than ABP+EQ for a variety of job mixes.
5. The **makespan experiments** compare the makespan produced by G-RAD against the theoretical lower bound for over 10000 runs of job sets. The results show that the makespan produced by G-RAD is no more than 1.39 times the optimal on average and it never exceeds 4.5 times.
6. The **mean response time experiments** investigate how G-RAD performs with respect to mean response time for over 8000 batched job sets. For these runs, the mean response time produced by G-RAD is no more than 2.37 times the optimal on average, and it never exceeds 5.5 times.
7. The **load experiments** investigate how the system load affects the performance of G-RAD. Under moderate and heavy load, G-RAD produces a makespan very close to optimal, and produces a mean response time approximately 2 times the optimal.
8. The **Proactive RAD experiments** compare the performance of RAD against a variant called *Proactive RAD*. The Proactive RAD always allots all processors to jobs even if the overall desire is less than the total number of processors. The experimental results suggest that Proactive RAD improves the performance of the original algorithm under light load.

The performance of A-GREEDY is not discussed explicitly in this chapter because the performance of A-GREEDY would be at least as good as A-STEAL. A-GREEDY and A-STEAL uses the same desire-estimation algorithm, and they are only different in their thread scheduling strategies. A-GREEDY uses greedy thread scheduling, which is a centralized scheme, and it assumes complete information of the ready threads at any time. In contrast, A-STEAL uses a distributed thread scheduling scheme where each processor has only local information of the ready threads. Therefore, A-GREEDY performs a more “effective” mapping of ready threads to the available processors and

should perform at least as well as A-STEAL.

The remainder of this chapter is organized as follows. Section 5.1 describes our simulation setup. Section 5.2 to Section 5.8 elaborate the results of the experiments.

5.1 Simulation Setup

We built a Java-based discrete-time simulator using DESMO-J [47]. The simulator implements four major entities — processors, jobs, thread schedulers, and OS allocators — and simulates their interactions in a two-level scheduling environment. We modeled a job as a DAG, which is scheduled for execution by the thread scheduler. When a job is submitted to the simulated multiprocessor system, an instance of the thread scheduler is created for the job. The OS allocator allots processors to the job, and the thread scheduler simulates the execution of the job using work-stealing. The simulator operates in discrete time steps: a processor can complete either a work-cycle, steal-cycle, or mug-cycle during each time step. In the simulation, we ignored the overheads that are due to the reallocation of processors.

Our benchmark applications are the *square-wave* jobs whose task graphs are typically as shown in Figure 5-1. Each job starts with an initial *serial phase* with length w_0 and alternates between a serial phase of length w_1 and a *parallel phase* (with h -way parallelism) of length w_2 . The parallelism of job's parallel phase is the *height* h of the job, and the number of iterations is denoted as *iter*. Square-wave jobs arise naturally in jobs that exhibit “data parallelism” which applies the same computational steps to a set of different data points. Many computationally intensive applications can be expressed in a data-parallel fashion [112]. The repeated serial-parallel cycle in the job reflects the often iterative nature of these computations. The average parallelism of the job is approximately $(w_1 + hw_2)/(w_1 + w_2)$. By varying the values of w_0 , w_1 , w_2 , h , and the number of iterations, we can generate jobs with different work, span, and phase length.

In the time-waste experiments and the utilization experiments, we compared the performance of A-STEAL with that of another adaptive thread scheduler, ABP [5], which does not provide parallelism feedback to the OS allocator. In these experiments, ABP is always allotted all the processors available to the job. ABP uses a nonblocking implementation of work stealing and always maintains P dequeues. When the OS allocator allots $a_q = p_q$ processors in quantum q , ABP selects a_q dequeues uniformly at random from the P dequeues, and the allotted processors start working on them.

Figure 5-1: The DAG of a square-wave job used in the simulation. This job has start-up length $w_0 = 1$, serial phase length $w_1 = 3$, parallel phase length $w_2 = 2$, parallelism $h = 7$, and the number of iterations $iter = 2$.

Arora, Blumofe, and Plaxton [5] show that ABP completes a job in expected time

$$T = O(T_1/\bar{P} + PT_\infty/\bar{P}) , \quad (5.1)$$

where $\bar{P}$ is the average number of processors allotted to the job by the OS allocator. Although Arora *et al.* provide no bounds on waste, one can prove that ABP may waste $\Omega(T_1 + PT_\infty)$ processor cycles in an adversarial setting.

We implemented four kinds of OS allocators — profile-based, equipartitioning (EQ), RAD, and Proactive RAD. A profile-based OS allocator was used in the first three sets of experiments; both EQ and RAD OS allocators were used in the utilization experiments; RAD was used in the last four sets of experiments; Proactive RAD was used in the last set of experiments. An EQ OS allocator simply allots the same number of processors to all the active jobs in the system. Since ABP provides no parallelism feedback, EQ is a suitable OS allocator for ABP’s scheduling model. RAD is described in Chapter 3. Proactive RAD is a variation of RAD and will be introduced in Section 5.9.

For the first three experiments — time, waste, and time-waste — we ran a single job with a predetermined **availability profile**: the sequence of processor availabilities p_q for all the quanta while the job is executing. For the **profile-based** OS allocator, we precomputed the availability profile and, during the simulation, the OS allocator simply used the precomputed availability for each quantum. We generated three kinds of profiles:

- **Uniform profiles:** The processor availabilities in these profiles follow the uni-

form distribution in the range from 1 to P , the maximum number of processors in the system. These profiles represent near-adversarial conditions for A-STEAL, because the availability in one quantum is unrelated to the availability in the previous quantum.

- **Smooth profiles:** In these profiles, the change of processor availabilities from one scheduling quantum to the next follows a standard normal distribution. Thus, the processor availability is unlikely to change significantly over two consecutive quanta. These profiles attempt to model situations where new arrivals of jobs are rare, and the availability changes significantly only when a new job arrives.
- **Practical profiles:** These availability profiles were generated from the workload archives [55] of various computer clusters. We computed the availability at every quantum by subtracting the number of processors that were being used at the start of the quantum from the number of processors in the machine. These profiles are meant to capture the processor availability in practical systems.

The thread schedulers A-GREEDY and A-STEAL require certain parameters as input. Some typical values of the responsiveness parameter ρ might range between 1.2 and 2.0. The responsiveness parameter was $\rho = 1.5$ for the first four sets of the experiments and was $\rho = 2.0$ for the last four sets of the experiments. Some typical values for the utilization parameter δ might be 80–95%. For all experiments except the waste experiments, the utilization parameter was set to be $\delta = 0.8$. We vary δ in the waste experiments. The quantum length L represents the time between successive reallocations of processors by the OS allocator. It is selected to amortize the overheads due to the communication between the OS allocator and the thread scheduler, as well as for the reallocation of processors. In conventional computer systems, a scheduling quantum is typically between 10 and 20 milliseconds. A steal/mug cycle in the Cilk runtime system [67], for example, takes approximately 0.5 to 5 microseconds, indicating that the quantum length L should be set to values between 10^3 and 10^5 time steps. Our theoretical bounds indicate that as long as $T_\infty \gg L \log P$, the length of L should have little effect on our results. Due to the performance limitations of our simulation environment, however, we could not afford to run very long jobs: most have a span in the order of only a few thousand time steps. Therefore, to satisfy the condition that $T_\infty \gg L \log P$, we set $L = 200$ in the first four sets of experiments and $L = 1000$ in the last four sets of experiments.

5.2 Time Experiments

The running-time bounds proved in Section 2.7, though $O(1)$ -competitive against an optimal scheduler, have weak constants. The time experiments were designed to investigate what constants occur in practice and how A-STEAL performs compared to an optimal scheduler. We performed linear-regression analysis on the results of 2331 job runs using many availability profiles of all three kinds to answer these questions.

Our first time experiment uses the bounds in Inequality (2.2) as a simple model, as in the study [21]. Assuming that equality holds and disregarding smaller terms, the model estimates performance as

$$T \approx c_1 T_1 / \tilde{P} + c_\infty T_\infty, \quad (5.2)$$

where $c_1 > 0$ is the **work overhead** factor and $c_\infty > 0$ is the **span overhead** factor. When $\delta = 0.8$, $\rho = 1.5$, and $L = 200$, the coefficients for the asymptotic bounds in Inequality (2.2) turn out to be $1.26 < c_1 < 1.27$ and $c_\infty = 480$, but a direct expected time analysis can improve the bound on span overhead to $c_\infty = 60$. Since the span overhead c_∞ is large, the bound suggests that A-STEAL may not provide linear speedup except when $T_1/T_\infty \gg 60\tilde{P}$. Moreover, on accounted time steps, A-STEAL might not provide perfect linear speedup, since the work overhead factor is $1.26 > 1$.

In practice, however, these large overheads do not materialize. First, our analysis is based on asymptotic bounds and general bounding techniques such as Markov's inequality and Chernoff bounds which are not necessarily tight. Second, our analysis assumes that the job completes the minimum number of work-cycles in each quantum — specifically, 0 in a deductible quantum and $\delta L a_q$ in an accounted quantum with allotment a_q .

Our first linear-regression analysis fits the running time of the 2331 job runs to Equation (5.2). The least-squares fit to the data to minimize relative error yields $c_1 = 0.960 \pm 0.003$ and $c_\infty = 0.812 \pm 0.009$ with 95% confidence. The R^2 correlation coefficient of the fit is 99.4%. Since $c_\infty = 0.812 \pm 0.009$, on average the jobs achieved linear speedup when $T_1/T_\infty \gg \tilde{P}$. In addition, since $c_1 = 0.960 \pm 0.003$, A-STEAL achieves almost perfect linear speedup on the accounted steps. The fact that $c_1 < 1$ stems from the fact that jobs performed some work during the deductible steps.

We performed a second set of regression tests on the same set of jobs to compare the performance of A-STEAL to an optimal scheduler. We fit the job execution time

Figure 5-2: Comparing the (true) mean availability $\bar{P}$ with the trimmed availability $\tilde{P}$ using three availability profiles. Each data point represents a job execution for which the mean availability and trimmed availability were measured. These values were normalized by dividing by the parallelism T_1/T_∞ of the job. When the parallelism satisfies $T_1/T_\infty > 5\bar{P}$, the experiments indicate that for all profiles, the trimmed availability is a good approximation of the mean availability. All these experiments used $\delta = 0.8$ and $\rho = 1.5$.

to the curve

$$T = \hat{c}_1 T_1 / \bar{P} + \hat{c}_\infty T_\infty . \quad (5.3)$$

The analysis yields $\hat{c}_1 = 0.992 \pm 0.003$ and $\hat{c}_\infty = 0.911 \pm 0.008$ with an R^2 correlation coefficient of 99.4%. Both $T_1/\bar{P}$ and T_∞ are lower bounds on the job's running time, and thus an optimal scheduler requires at least $\max\{T_1/\bar{P}, T_\infty\} \geq (T_1/\bar{P} + T_\infty)/2 \geq (\hat{c}_1 T_1/\bar{P} + \hat{c}_\infty T_\infty)/2$ time steps, since $\hat{c}_1 < 1$ and $\hat{c}_\infty < 1$. Thus, on average A-STEAL completed the jobs within no more than twice the time of an optimal scheduler.

The two job models 5.2 and 5.3 both predict performance with high accuracy, yet $\tilde{P}$ and $\bar{P}$ can diverge significantly. To resolve this paradox, we compared $\tilde{P}$ and $\bar{P}$ on the job runs. Figure 5-2 shows a graph of the results, where $\tilde{P}$ and $\bar{P}$ are each normalized by dividing with the parallelism T_1/T_∞ of the job. The diagonal line is the curve $\tilde{P} = \bar{P}$.

When a job has parallelism $T_1/T_\infty > 5\bar{P}$ (data points on the left half of Figure 5-

2), the experiment indicates that for all three kinds of availability profiles, we have $\tilde{P} \approx \bar{P}$. In this case, we have $T_1/\tilde{P} \approx T_1/\bar{P}$ and $T_1/\bar{P} \gg T_\infty$, which implies that the first terms in Equations (5.2) and (5.3) are nearly identical and dominate the running time. On the other hand, if a job has small parallelism (data points on the right half of Figure 5-2), the values of $\tilde{P}$ and $\bar{P}$ diverge and the divergence depends on the availability profile used. The values of $\tilde{P}$ are better approximation of $\bar{P}$ in practical profile than those in the smooth and uniform profile. In this region, however, the running time is dominated by the span T_∞ , and thus, the divergence of $\tilde{P}$ and $\bar{P}$ has little influence on the running time.

5.3 Waste Experiments

Our theoretical analysis shows that the waste exhibited by A-STEAL is at most $O(T_1)$. The constant hidden in the O -notation depends on the parameter settings. In our first waste experiment, we varied the value of the utilization parameter δ to study the relationship between the waste and the value of δ . For our second experiment, we investigated whether the waste incurred by a job depends on the job's parallelism.

The proof of Theorem 2.25 shows that the number of processor cycles wasted by a job is $((1 - \delta)/\delta)T_1$ in efficient quanta and approximately $(\rho/\delta)T_1$ in inefficient quanta. Substituting $\delta = 0.8$ and $\rho = 1.5$, A-STEAL could waste $0.25T_1$ processor cycles on efficient quanta and $1.875T_1$ processor cycles on inefficient quanta. Since this analysis assumes that the OS allocator is an adversary and the job completes the minimum number of work-cycles in each quantum, we do not expect these constants to materialize in practice.

We measured the waste for 300 jobs, most of which had parallelism $T_1/T_\infty > 5\bar{P}$, for $\delta = 0.5, 0.6, \dots, 1.0$. The job runs used many availability profiles drawn equally from the uniform, smooth, and practical job profiles. Figure 5-3 shows the average of waste normalized by the work T_1 of the job. For comparison we also plotted the normalized theoretical bound given by Inequality (2.4) for the total waste and the normalized bound $((1 - \delta)/\delta)T_1$ for the waste in efficient quanta. The figure shows that (although the curve is barely distinguishable from the x -axis) the observed waste is less than 10% of the work T_1 for most values of δ and is considerably less than predicted by the theoretical bounds. Moreover, the waste seems to be quite insensitive to the particular value of δ .

We also ran an experiment to determine whether job parallelism has an effect on the waste. The bound in Inequality (2.4) does not depend on the parallelism T_1/T_∞ of

5.3. Waste Experiments

93

Figure 5-3: Comparing the theoretical and practical waste (normalized by T_1) using A-STEAL for various values of the utilization parameter δ . The top line shows the total theoretical waste, the next line shows the theoretical waste on efficient quanta, and the bottom line shows the observed waste. The observed waste appears to be almost insensitive to the value of δ and is much smaller than the theoretical waste.

Figure 5-4: It shows how waste varies with parallelism. When $T_1/T_\infty > 10\bar{P}$, that is, when the job's parallelism significantly exceeds the average availability, the observed waste was only a tiny fraction of the work T_1 . For jobs with small parallelism, the waste showed a large variance but never exceeded the work T_1 in any of our runs. The utilization parameter was $\delta = 0.8$ for all job runs.

the job, but only on the work T_1 . For the 2331 job runs used in the time experiments, we measured the waste versus parallelism. Since waste is insensitive to δ , all jobs used the value $\delta = 0.8$. Figure 5-4 graphs the results. As can be seen in the figure, the higher the parallelism, the lower the waste-to-work ratio. The reason is that when the parallelism is high, the job can usually use most of the available processors without readjusting its desire. When the parallelism is low, however, the job's desire must track its parallelism closely to avoid waste. Low parallelism is where A-STEAL is most effective, as the thread scheduler pushes the theoretical waste bound to their limit.

5.4 Time-Waste Experiments

The time-waste experiments were designed to compare A-STEAL with ABP, an adaptive thread scheduler with no parallelism feedback. For our first experiment, we ran A-STEAL and ABP to execute 756 job runs on a simulated machine with $P = 512$ processors. Each head-to-head run used one of two practical availability profiles, one with $\bar{P} = 30$ and one with $\bar{P} = 60$. We measured the time and waste of A-STEAL and ABP for each run. Our second experiment was similar, but it used only $P = 128$ processors in the simulated machine over 330 job runs. Whenever the availability exceeded 128, which seldom arose, we chopped the availability to 128.

Figure 5-5 shows the ratio of ABP to A-STEAL with respect to both time and waste as a function of the mean availability $\bar{P}$, normalized by dividing it by the average parallelism T_1/T_∞ . This experiment shows that A-STEAL completed jobs about twice as fast as ABP while wasting only about 10% of the processor cycles wasted by ABP. Not surprisingly, A-STEAL wastes fewer processor cycles than ABP, since A-STEAL uses parallelism feedback to limit possible excessive allotment. Paradoxically, however, A-STEAL still completes jobs faster than ABP, even though A-STEAL's allotment in every quantum is at most that of ABP, which always gets all the available processors.

ABP's slow completion is due to how ABP manages its ready dequeues. In particular, ABP has no mechanism for increasing and decreasing the number r of ready dequeues, and it maintains $r = P$ dequeues throughout the execution. Randomized work-stealing algorithms require $\Theta(r)$ steal-cycles to reduce the span by 1 in expectation. Consequently, if r is large, each steal-cycle becomes less effective, and the job's progress along its span slows down. Thus, if the job has small or moderate parallelism (data points on the right), the span dominates the running time. If the job has

5.4. Time-Waste Experiments

95

Figure 5-5: Comparing the time and waste of A-STEAL against ABP when $P = 512$ and $\bar{P} = 30, 60$. In this experiment, where P exceeds $\bar{P}$ by a significant margin, A-STEAL completes jobs about twice as fast as ABP while wasting less than 10% of the processor cycles wasted by ABP.

large parallelism (data points on the left), however, the impact is less. In contrast, A-STEAL continues to make good progress along the span, regardless of parallelism, by reducing the number of dequeues according to its allotment.

Figure 5-6: Comparing the time and waste of A-STEAL against ABP when $P = 128$ and $\bar{P} = 30, 60$. This experiment shows that when P and $\bar{P}$ are closer in magnitude, A-STEAL runs slightly slower than ABP, but it still tends to waste fewer processor cycles than ABP.

This faster-with-fewer-processors paradox can also be understood by using the model from Equation (5.2) for A-STEAL and an analogous model based on Equation (5.1) for ABP. Let us consider three cases:

- $T_1/T_\infty < \bar{P} \ll P$ (data points on the right hand side of Figure 5-5): Whereas A-STEAL completes the job in $\Theta(T_\infty)$ time, ABP requires $\Theta(PT_\infty/\bar{P})$ time.
- $\bar{P} < T_1/T_\infty \ll P$ (data points in the middle): A-STEAL provides linear speedup since $T_1/T_\infty > \tilde{P}$, but ABP does not, since $T_1/T_\infty \ll P$.
- $P < T_1/T_\infty$ (data points on the left hand side of Figure 5-5): Both provide linear speedup in this range.

Since ABP performed relatively poorly when P is large compared to $\bar{P}$, our second experiment investigated the case when P is closer to $\bar{P}$. Figure 5-6 shows the results on 330 job runs on a simulated machine with $P = 128$. In this case, ABP performs slightly better than A-STEAL with respect to time and slightly worse with respect to waste. Since $\bar{P} \approx P$, the two models coincide, and ABP and A-STEAL perform comparably. Therefore, on small machines, where the disparity between $\bar{P}$ and P cannot be great, the advantage of parallelism feedback is diminished, and ABP may well be an effective thread-scheduling algorithm.

5.5 Utilization Experiments

The utilization experiments compared W-RAD with ABP+EQ on a large server where many jobs are running simultaneously and jobs arrive and leave dynamically. We simulated a 1000-processor machine for about 10^6 time steps, where jobs had a mean interarrival time of 1000 time steps. We compared the utilization provided by W-RAD and ABP+EQ over time.

It was unclear what distribution the parallelism and the span should follow. Although many workload models for parallel jobs have been studied [40, 49, 56, 101, 120], none appears to apply directly to multithreaded jobs. Some studies [73, 74, 97] claim that the sizes of Unix jobs follow a heavy-tailed distribution. Lacking clear evidence, we decided to try various distributions. As it turned out, our results were fairly insensitive to these distributions.

We considered 9 sets of jobs using three distributions on each of the parallelism and the span. The means of the distributions were chosen so that jobs arrive faster than they complete and the load on the machine progressively increases. Thus, we were able to measure the utilization of the machine under various loads. The three distributions we explored were the following:

- **Uniform distribution (U):** The span is picked uniformly from the range 1,000 to 99,000. The parallelism is generated uniformly in the range [1, 80].

- **Heavy-tailed distribution 1 (HT1):** We used a Zipf's-like [135] heavy-tailed distribution where the probability of generating x is proportional to $1/x$. In our experiments, the distribution for parallelism has a mean value of about 36, and the distribution for span has a mean value of 50,000.
- **Heavy-tailed distribution 2 (HT2):** In this distribution, the probability of generating x is proportional to $1/\sqrt{x}$. In our experiments, the distribution for parallelism has mean 36, and the distribution for span has a mean value of 50,000.

Of the 9 possible sets of jobs, we ran 6 experiments using parallelism and span drawn from U/U, U/HT1, HT1/U, HT1/HT1, HT2/U, and HT2/HT2. For all these experiments, the comparison between W-RAD and ABP+EQ followed a similar pattern. We broke time into intervals of 2000 time steps and measured the utilization — the fraction of processor cycles spent working — for each interval. Figure 5-7 shows the utilization as a function of time (log-scale) for the U/U experiment on the top and for HT1/HT1 at the bottom. As can be seen in both figures, ABP+EQ starts out with a higher utilization, since W-RAD initially requests just 1 processor. Before 10% of the simulation has elapsed, however, W-RAD overtakes ABP+EQ with respect to the utilization and then consistently provides a higher utilization. Although the figure does not show it, the mean completion time of jobs under ABP+EQ is nearly 50% longer than those under W-RAD for both these distributions.

5.6 Makespan Experiments

The competitive ratio of makespan in Theorem 4.2, though asymptotically strong, has a relatively large constant multiplier. The makespan experiments were designed to evaluate the constants that would occur in practice and compare G-RAD to an optimal scheduler. The experiments were conducted on more than 10,000 runs of jobs sets using many combinations of jobs and different loads.

Figure 5-8 shows how G-RAD performs compared to an optimal scheduler. The makespan of a job set $\mathcal{J}$ has two lower bounds $\max_{J_i \in \mathcal{J}}(r(J_i) + T_\infty(J_i))$ and $T_1(\mathcal{J})/P$. The makespan produced by an optimal scheduler is at least the larger of these two lower bounds. The makespan ratio in Figure 5-8 is defined as the makespan of a job set scheduled by G-RAD divided by the larger of the two lower bounds. In Figure 5-8, the X -axis represents the ranges of the makespan ratio, while the histogram shows the percentage of the job sets whose makespan ratio falls into each range. Among the over 10,000 runs, 76.19% of them use less than 1.5 times the theoretical lower

Figure 5-7: Comparing the utilization over time of W-RAD and ABP+EQ. In the figure on the top, both the span and the parallelism follow the uniform distribution, and in the figure on the bottom, they follow the HT1 distribution.

bound, 89.70% uses less than 2.0 times, and none uses more than 4.5 times. The average makespan ratio is 1.39, which suggests that in practice G-RAD has a good competitive ratio with respect to the makespan.

We now examine the relation between the theoretical bounds and experimental

Figure 5-8: Comparing the makespan of a job set produced by G-RAD with the theoretical lower bound for job sets with arbitrary job release time.

Figure 5-9: Comparing the mean response time of a job set produced by G-RAD with the theoretical lower bound for batched job sets.

results as follows. When $\rho = 2$ and $\delta = 0.8$, from Theorem 4.2, G-RAD is 13.75-competitive in the worst case. We anticipate that G-RAD's makespan ratio would be small in practical settings, however, especially when many jobs have much greater work than span and the machine is moderately or highly loaded. In this case, the

term $T_1(\mathcal{J})/P$ in Inequality (4.3) of Theorem 4.2 is much larger than the term $\max_{J_i \in \mathcal{J}} \{T_\infty(i) + r(i)\}$, which is to say, the term $T_1(\mathcal{J})/P$ generally dominates the makespan bound. The proof of Theorem 4.2 calculates the multiplicative coefficient of $T_1(\mathcal{J})/P$ as the ratio of the total allotment (total work plus total waste) versus the total work. When the OS allocator is RAD, which is not a true adversary, our simulation results indicate that the ratio of the waste versus the total work is only about 1/10 of the total work. Thus, the coefficient of $T_1(\mathcal{J})/P$ in Inequality (4.3) is about 1.1. As shown in Figure 5-8, the makespan produced by G-RAD is on average less than 2 times the lower bound.

5.7 Mean Response Time Experiments

This set of experiments was designed to evaluate the mean response time of the batch job sets scheduled by G-RAD and compare it against an optimal scheduler.

Figure 5-9 shows the distribution of the mean response time normalized with respect to the larger of the two lower bounds — the squashed work bound $sqw(\mathcal{J})/|\mathcal{J}|$ and the aggregated span bound $T_\infty(\mathcal{J})/|\mathcal{J}|$. The histogram in Figure 5-9 shows that of more than 8000 runs, 94.65% use less than 3 times the theoretical lower bound, and none uses more than 5.5 times. The average mean response time ratio is 2.37.

We relate the theoretical bounds to experimental results as follows. When $\rho = 2$ and $\delta = 0.8$, from Theorem 4.12, G-RAD is 17.5-competitive. We expect that G-RAD should perform closer to optimal in practice, however. In particular, when the job set $\mathcal{J}$ exhibits reasonably large total parallelism, we have $sqw(\mathcal{J}) \gg T_\infty(\mathcal{J})$, and thus, the term involving $sqw(\mathcal{J})$ in Theorem 4.12 dominates the total response time. More importantly, RAD is not an adversary of A-GREEDY, and as mentioned before, the waste of a job is only about 1/10 of the total work in average for over 100,000 job runs we tested. Based on this waste, the squashed area bound $sqw(\mathcal{J})$ in Inequality (4.29) of Theorem 4.12 has a coefficient of around 2.2. As shown in Figure 5-9, the mean response time produced by G-RAD is less than 3 times the lower bound.

5.8 Load Experiments

This set of experiments was designed to investigate how the load affects the performance of G-RAD. The **load** of a job set $\mathcal{J}$ on a machine with P processors, which measures how heavily jobs compete for processors on the machine, is calculated as

follows:

$$load = \frac{T_1(\mathcal{J})}{P \cdot (\max_{J_i \in \mathcal{J}} r(J_i) - \min_{J_i \in \mathcal{J}} r(J_i) + T_\infty(\mathcal{J}) / |\mathcal{J}|)} . \quad (5.4)$$

For a batched job set, the load is just the average parallelism of the set divided by the total number of processors.

Figure 5-10 shows how G-RAD performs with respect to makespan against the theoretical lower bound as the system load varies. The makespan ratio in this figure is defined as the makespan of a job set scheduled by G-RAD divided by the larger of the two lower bounds. Each data point represents the makespan ratio of a job set. The test results suggest that the makespan ratio becomes smaller as the load gets higher. Specifically, the makespan generated by G-RAD is close to the lower bound when the load is greater than 4, and the makespan ratio never exceeds 1.5 times when system load is greater than 3. When the load is less than 2, however, the makespan ratio spreads across the range from 1 to 4.

Figure 5-11 shows the performance of G-RAD for batched jobs with respect to the mean response time as the system load varies. It compares the mean response time incurred by G-RAD with the larger of the two lower bounds. Similar to makespan, the mean response time produced by G-RAD is closer to the lower bound when the system load gets larger. Specially, under heavy load, the mean response time ratio concentrates on about 2 times the optimal, while under light load, the ratio spreads across the range from 1 to slightly over 5.

The load experiments raise a question of how to improve the performance of G-RAD under light load. The OS allocator RAD makes conservative decisions when the allocating processors to jobs. When the system is lightly loaded, *e.g.*, the total demand is less than the total number of processors, RAD keeps some processors idle without allocating them to any jobs. Since a greedy thread scheduler executes a job faster with more processors allotted, an OS allocator that always allots all processors to jobs, should perform better under light load. We explore such a variation of the OS allocator RAD in the next set of the experiments.

5.9 Proactive RAD Experiments

To improve the performance under light load, we introduce Proactive RAD, the OS allocator that always allocates all processors to jobs even if the total requests are less than the total number of processors. At a quantum q , when the total requests $d(\mathcal{J}, q) = \sum_{J_i \in \mathcal{J}} d(J_i, q)$ are greater than or equal to the total number P of processors,

Figure 5-10: Comparing G-RAD with varying load against the theoretical lower bound for makespan.

Figure 5-11: Comparing G-RAD with varying load for batched jobs against the theoretical lower bound for mean response time.

Proactive RAD works exactly the same as the original RAD allocator. If $d(\mathcal{J}, q) < P$, however, Proactive RAD evenly allots the remaining $(P - d(\mathcal{J}, q))$ processors to all the jobs.

Figure 5-12 shows the makespan ratio of Proactive RAD against the original

Figure 5-12: Comparing Proactive RAD against the original RAD for makespan with varying load. The X-axis represents the load of the system. The Y-axis represents the makespan ratio between Proactive RAD and the original RAD.

Figure 5-13: Comparing Proactive RAD against the original RAD for mean response time with varying load. The X-axis represents the load of the system. The Y-axis represents the mean response time ratio between Proactive RAD and the original RAD.

RAD under various loads. Each data point in the figure represents a head-to-head comparison for a job set. The makespan ratio in the figure is defined as the makespan produced by Proactive RAD divided by that produced by the original RAD. We can see that the makespan ratio is less than 1 for most of the runs, indicating that Proactive RAD out-performs the original RAD in most of these job sets. Moreover, the difference between them becomes more significant under light load and diminishes with increasing of the system load. The reason is that Proactive RAD generally allocates more processors to jobs, especially when the load is light. The increased allotment allows the faster execution of jobs and eventually shortens the makespan of the job set. Figure 5-13 shows the mean response time ratio for batched job sets by varying load, which exhibits similar behavior to that of makespan. Figures 5-12 and 5-13 suggest that Proactive RAD improves the performance of RAD under light load.

Chapter 6

Techniques for Analyzing Two-level Adaptive Scheduling Algorithms

The preceding chapters of this thesis describe and analyze individual thread schedulers, OS allocators, and several two-level scheduling systems. In this chapter, I generalize the techniques for designing and analyzing two-level adaptive scheduling systems.

Section 6.1 discusses the desirable properties of general thread schedulers, and Section 6.2 defines four properties of general OS allocators. Section 6.3 presents two theorems that support makespan-efficient two-level adaptive schedulers. Section 6.4 introduces a partitioning approach to analyze the batched mean response time of a two-level scheduler.

6.1 Desirable Properties of Thread Schedulers

Waste and completion time are two important measures of an adaptive thread scheduler. This section suggests that “nonoverflow waste” and “satisfied count” are in a sense more fundamental and provide a better base for comparing adaptive thread schedulers.

Nonoverflow waste is the waste incurred on a job during the steps where the allotment of the job is not more than the desire of the job. Formally, for a job J_i , denote its nonoverflow time steps $S = \{t \mid a(J_i, t) \leq d(J_i, t)\}$. The nonoverflow waste of the job J_i is $\varpi(J_i) = \sum_{t \in S} w(J_i, t)$, where $w(J_i, t)$ denotes the wasted processor cycles of J_i at time step t . One reason to use nonoverflow waste instead of the total waste is that no thread scheduler can ensure any meaningful waste bound if the OS

allocator always allots it more processors than its desire. The waste incurred on the extra processors is not the responsibility of the thread scheduler. On the other hand, a reasonable OS allocator only allots more processors to a job than its request when the system is lightly loaded. In this case, waste is unavoidable and should not discredit the thread scheduler.

The satisfied count of a job J_i , which was given in Definition 4.1 as $sat(J_i) = |\{t \mid a(J_i, t) \geq d(J_i, t)\}|$, is a measure of the total number of satisfied time steps during the execution of the job. Although completion time is an important measure, a thread scheduler must take long to complete its job if the OS allocator always gives the job a small number of processors. Trim analysis provides one way to analyze the time bound of a thread scheduler under powerful adversary. Analysis of the satisfied count, in contrast, measures the performance of a thread scheduler when the OS allocator is cooperative. Intuitively, a thread scheduler can have short completion time only if it can make good progress in completing the job during the satisfied time steps. A small value of satisfied count is indeed a necessary condition to ensure that the thread scheduler performs well in terms of the completion time.

I now discuss the values of nonoverflow waste and satisfied count for good thread schedulers. Ideally, the nonoverflow waste is 0 and satisfied count is $T_\infty(J_i)$. A greedy thread scheduler using instantaneous parallelism as feedback is theoretically ideal. As mentioned in Chapter 1, however, calculating instantaneous parallelism can be costly, and matching processor allocation to it can result in large overhead. Moreover, the minimum processor reallocation interval in some systems is in the range of 10 to 20 milliseconds, which can be much longer than the variability of parallelism. Therefore, using instantaneous parallelism as feedback is often not practical. A thread scheduler must choose other desire estimation strategies with less scheduling overhead, but possibly with larger waste and satisfied count. Typically, a good thread scheduler should ensure that the satisfied count is bounded by $O(T_\infty(J_i))$ so that the job can be completed in asymptotically optimal time when given sufficient processors. In terms of the waste, it is desirable to have nonoverflow waste bounded by $O(T_1(J_i))$, which guarantees that a constant fraction of the allotted processor cycles are doing useful work.

As an example, both A-GREEDY and A-STEAL induce at most $O(T_1(J_i))$ total waste. Their total waste is equal to the nonoverflow waste, because we assume that the OS allocator never allots more processors than a job's desire. Since they do not consider the case where allotment is greater than the desire, A-GREEDY and A-STEAL

```

    PROACTIVE-A-GREEDY( $q, \delta, \rho$ )
1  if  $q = 1$                                 ▷ base case
2      then  $d_q \leftarrow 1$ 
3  elseif  $u_{q-1} < L\delta a_{q-1}$                 ▷ inefficient
4      then  $d_q \leftarrow \max\{d_{q-1}/\rho, 1\}$ 
5  elseif  $a_{q-1} \geq d_{q-1}$                   ▷ efficient-and-satisfied
6      then  $d_q \leftarrow \rho d_{q-1}$ 
7  else  $d_q \leftarrow d_{q-1}$                 ▷ efficient-but-deprived
8  Report desire  $d_q$  to the OS allocator.
9  Receive allotment  $a_q$  from the OS allocator.
10 Greedily schedule on  $a_q$  processors for  $L$  time steps.

```

Figure 6-1: Pseudocode of the Proactive A-GREEDY algorithm, which provides parallelism feedback to an OS allocator in the form of a desire for processors. Before quantum q , Proactive A-GREEDY uses the previous quantum's desire d_{q-1} , allotment a_{q-1} , and usage u_{q-1} to compute the current quantum's desire d_q based on the utilization parameter δ and the responsiveness parameter ρ .

are not designed to work with an OS allocator that gives extra processors. Moreover, the satisfied count of A-GREEDY is no more than $2T_\infty(J_i)/(1-\delta) + L \log_\rho P + L$, while the satisfied count of A-STEAL is $O(T_\infty(J_i)/(1-\delta) + L \log_\rho P)$ in expectation.

Let's consider a variation of A-GREEDY, called Proactive A-GREEDY, which occasionally receives more processors than its desire.

Example: Proactive A-Greedy

Unlike the original A-GREEDY, Proactive A-GREEDY allows extra processors to be allocated. The pseudocode of Proactive A-GREEDY is shown in Figure 6-1. The first difference of Proactive A-GREEDY from the original G-RAD in Figure 2-1 is in line 4, where the desire d_q for quantum q is the larger of 1 and d_{q-1}/ρ if its previous quantum $q-1$ is inefficient. As the second difference, Proactive A-GREEDY rewrites the definition of satisfied quantum. A quantum q is satisfied if the job's allotment is either greater than or equal to its desire, *i.e.*, $a_q \geq d_q$; otherwise, it is deprived. Specially, a satisfied quantum can be further classified as either **matching**, where the allotment is equal to the desire, or **overflowing**, where the allotment is greater than the desire. The definitions for efficient and inefficient quanta remain unchanged.

□

Let's look at the nonoverflow waste and satisfied count of Proactive A-GREEDY. By following a similar analysis of Lemma 2.8, we can determine that the satisfied count of a job J_i produced by Proactive A-GREEDY is at most $2T_\infty(J_i)/(1-\delta) + L \log_\rho P + L$ time steps. Lemma 6.1 shows that the nonoverflow waste of Proactive A-GREEDY is at most $(1 + \rho - \delta)T_1(J_i)/\delta$ processor cycles.

Lemma 6.1 *Suppose that Proactive A-GREEDY schedules a job J_i with work $T_1(J_i)$ on a machine. The Proactive A-GREEDY's nonoverflow waste is at most $(1 + \rho - \delta)T_1(J_i)/\delta$ processor cycles.*

Proof. We use an accounting argument to calculate the nonoverflow waste based on whether a quanta is inefficient or efficient.

Nonoverflowing inefficient quanta: One can show that every inefficient quantum q with desire $d(J_i, q) > 1$ corresponds to a distinct efficient-and-satisfied quantum r with desire $d(J_i, r) = d(J_i, q)/\rho$ (c.f. Lemma 2.20). Moreover, every inefficient quantum with desire $d(J_i, q) = 1$ must be overflowing. Thus, every nonoverflowing inefficient quantum q with desire $d(J_i, q) > 1$ corresponds to a distinct efficient-and-satisfied quantum $r = f(q)$ with desire $d(J_i, r) = d(J_i, q)/\rho$. Consequently, the waste on a nonoverflowing inefficient quantum q can be tied against the work on quantum r . Since quantum r is efficient-and-satisfied, its work $v(J_i, r)$ is at least $v(J_i, r) \geq L\delta a(J_i, r) \geq L\delta d(J_i, r) = L\delta d(J_i, q)/\rho$, and we have $d(J_i, q) \leq v(J_i, r)\rho/(L\delta)$. Let $w(J_i, q)$ be the waste on quantum q . Since the quantum contains at most $La(J_i, q)$ total processor cycles, we have $w(J_i, q) \leq La(J_i, q)$. Since the quantum is not overflowing, we have $w(J_i, q) \leq Ld(J_i, q) \leq (\rho/\delta)v(J_i, r)$. That is to say, the waste on the nonoverflowing inefficient quantum q is at most a ρ/δ fraction of the work done in its corresponding efficient-and-satisfied quantum r . Therefore, the total waste on all nonoverflowing inefficient quanta is at most $(\rho/\delta)T_1(J_i)$.

Nonoverflowing efficient quanta: On any efficient quantum q , the job uses at least $L\delta a(J_i, q)$ processor cycles, and at most $L(1 - \delta)a(J_i, q)$ cycles are wasted. Summing over all efficient quanta, the waste is at most $((1 - \delta)/\delta)T_1(J_i)$.

Thus, we obtain the nonoverflow waste bound, which completes the proof. $\square$

6.2 Properties of OS Allocators

This section studies certain properties of the OS allocators that affect the scheduling performance. We define four concepts to quantify the properties of OS allocators as follows,

- **Frugality** — An OS allocator is frugal if it never allots more processors than a job requests, *i.e.*, $a(J_i, t) \leq d(J_i, t)$ for all t and $J_i \in \mathcal{J}$.
- **Fairness** — An OS allocator is fair if it satisfies the following two conditions. First, a job with smaller desire does not receive a larger allotment than a job with larger desire, *i.e.*, if we have $d(J_i, t) \leq d(J_j, t)$, then $a(J_i, t) \leq a(J_j, t)$ holds. Second, all deprived jobs receive the same number of processors, *i.e.*, if we have $a(J_i, t) < d(J_i, t)$ and $a(J_j, t) < d(J_j, t)$, then $a(J_i, t) = a(J_j, t)$ holds.
- **Integrity** — An OS allocator has integrity if it does not keep any processor idle unless no job wants it. With such an OS allocator, if there exists a $J_i \in \mathcal{J}$ with $a(J_i, t) < d(J_i, t)$, then $\sum_{J_i \in \mathcal{J}} a(J_i, t) = P$ holds.
- **Prudence** — An OS allocator is prudent if it satisfies all jobs' requests before it gives any job extra processors. With a prudent OS allocator, if there exists a job J_i and time t such that $a(J_i, t) > d(J_i, t)$, we have $a(J_i, t) \geq d(J_i, t)$ for all $J_i \in \mathcal{J}$.

These properties of OS allocator are not completely independent. In particular, a frugal OS allocator must be prudent, while a prudent allocator may or may not be frugal.

As an example, equipartitioning, which allots each job an equal number of processors, is fair and has integrity, but is neither frugal nor prudent. RAD is frugal, fair, and has integrity. A-GREEDY and A-STEAL assume that their OS allocator is frugal in order to ensure their waste and completion time bound. Even though being frugal is a reasonable assumption, an OS allocator need not be frugal, which allows extra processors to be allocated to some jobs beyond those jobs' desires. One justification is that when the system is lightly loaded, *i.e.*, $\sum_{J_i \in \mathcal{J}} d(J_i, t) < P$, a frugal scheduler holds some processors unallotted and therefore wasted. In this case, it can be beneficial, as with Proactive RAD, to let the jobs use the extra processors.

Proactive RAD

Proactive RAD always allocates all processors to jobs even if the total requests are less than the total number of processors. More precisely, at a quantum q , when the total requests $d(\mathcal{J}, q) = \sum_{J_i \in \mathcal{J}} d(J_i, q)$ are greater than or equal to the total number P of processors, the Proactive RAD works exactly the same as the original RAD. If $d(\mathcal{J}, q) < P$, however, the Proactive RAD evenly allots the remaining $P - d(\mathcal{J}, q)$ processors to all the jobs.

Proactive RAD is prudent, fair, and has integrity, but it is not frugal. Unlike RAD, Proactive RAD sometimes allocate more processors than a job desires. The experimental results in Chapter 5 show that Proactive RAD outperforms the original RAD under light system load.

6.3 Makespan-Efficient Two-level Schedulers

This section studies how thread schedulers and OS allocator cooperate to produce makespan efficient two-level schedulers. Suppose that a thread scheduler completes a job quickly and only wastes a small number of processors. Intuitively, combining such a thread scheduler with a reasonable OS allocator, the resulting two-level scheduler should provide good processor utilization and perform well with respect to makespan. This section formalizes and presents sufficient conditions for two-level schedulers to generate an efficient makespan.

The next theorem bounds the makespan of a two-level scheduler that combines thread schedulers with a frugal OS allocator that has integrity. Recall that $\varpi(\mathcal{J})$ and $T_1(J_i)$ denote the total nonoverflow waste and total work of $\mathcal{J}$ respectively, and $\text{sat}(J_i)$ and $r(J_i)$ denote the satisfied count and release time of job J_i respectively. The quantum length and number of processors are denoted by L and P respectively.

Theorem 6.2 *Suppose that a two-level scheduler $X+Y$ couples thread scheduler X with OS allocator Y , where Y is frugal and has integrity. Then, the makespan of the job set $\mathcal{J}$ produced by $X+Y$ is bounded by*

$$T(\mathcal{J}) \leq \frac{\varpi(\mathcal{J}) + T_1(\mathcal{J})}{P} + \max_{J_i \in \mathcal{J}} (\text{sat}(J_i) + r(J_i)) + L. \quad (6.1)$$

Proof. The proof is a straightforward generalization of Theorem 4.17. Suppose that job J_k is the last job completed among the jobs in $\mathcal{J}$. Let $S(J_k)$ denote the set of satisfied steps for J_k , and let $D(J_k)$ denote its set of deprived steps. We have $T(\mathcal{J}) \leq r(J_k) + L + |S(J_k)| + |D(J_k)|$. The satisfied count is $\text{sat}(J_i) = |S(J_k)|$. We have that $T(\mathcal{J}) \leq |D(J_k)| + \max_{J_i \in \mathcal{J}} (\text{sat}(J_i) + r(J_i)) + L$. We only need to bound $|D(J_k)|$.

We now bound the total number of deprived steps $D(J_k)$ of job J_k . For each step $t \in D(J_k)$, since Y has integrity, Y must allot all processors to jobs. The total allotment of $\mathcal{J}$ over J_k 's deprived steps $D(J_k)$ is $a(\mathcal{J}, D(J_k)) = \sum_{t \in D(J_k)} \sum_{J_i \in \mathcal{J}} a(J_i, t) = P|D(J_k)|$. Moreover, since Y is frugal, a job does not have overflow steps, and all

waste of a job is considered as nonoverflow waste. The total allotment for any job J_i is bounded by the sum of its total work $T_1(J_i)$ and total nonoverflow waste $\varpi(J_i)$. The total number of allotted processor cycles for all jobs in $\mathcal{J}$ is at most $T_1(\mathcal{J}) + \varpi(\mathcal{J})$. Thus, we have $P |D(J_k)| \leq T_1(\mathcal{J}) + \varpi(\mathcal{J})$, and hence $|D(J_k)| \leq (T_1(\mathcal{J}) + \varpi(\mathcal{J}))/P$, which completes the proof. $\square$

Theorem 6.2 indicates that a frugal OS allocator, coupled with a reasonable thread scheduler, only needs to have integrity in order to produce a makespan-efficient two-level scheduler. More precisely, when a thread scheduler produces waste no more than $c_1 \cdot T_1(J_i)$ and satisfied count no more than $c_2 \cdot T_\infty(J_i)$ for job J_i , a frugal and integral OS allocator is $(c_1 + c_2 + 1)$ -competitive with respect to makespan. Another point to notice is that the thread scheduler need not consider the overflowed allotment if it explicitly cooperates with such a frugal OS allocator. Examples of such thread schedulers are A-GREEDY and A-STEAL, which were designed for frugal OS allocators.

Let us consider a few examples of frugal OS allocators that have integrity. RAD is certainly frugal and has integrity. Another example is an OS allocator that allots processors proportional to the desire of the jobs. Supposing that there are 12 processors and three jobs with desires of 6, 6, and 12, respectively, the three jobs would be allocated 3, 3, and 6 processors, respectively. In other words, the allotment of a job J_i at a time step t is given by $\min\left(d(J_i, t), P \cdot d(J_i, t) / \sum_{J_j \in \mathcal{J}} d(J_j, t)\right)$. One can verify that such a scheduler is frugal and has integrity.

Theorem 6.3 relaxes the requirement in Theorem 6.2 and bounds the makespan of a two-level scheduler combining the thread schedulers with a prudent OS allocator that has integrity.

Theorem 6.3 *Suppose that a two-level scheduler $X+Y$ couples thread scheduler X with OS allocator Y , where Y is prudent and has integrity. Then, the makespan of the job set $\mathcal{J}$ produced by $X+Y$ is bounded by*

$$T(\mathcal{J}) \leq \frac{\varpi(\mathcal{J}) + T_1(\mathcal{J})}{P} + \max_{J_i \in \mathcal{J}} (\text{sat}(J_i) + r(J_i)) + L. \quad (6.2)$$

Proof. As with the proof of Theorem 6.2, the key is to bound $|D(J_k)|$. Again, for each step $t \in D(J_k)$, since Y has integrity, Y must allot all processors to jobs. The total allotment of $\mathcal{J}$ over J_k 's deprived steps $D(J_k)$ is $a(\mathcal{J}, D(J_k)) = P |D(J_k)|$. Moreover, at time step $t \in D(J_k)$, since Y is prudent and J_k is deprived, none of the jobs receives extra processors. Thus, a job's waste at step t is nonoverflow waste.

The total allotment for a job J_i is bounded by the sum of its total work $T_1(J_i)$ and total nonoverflow waste $\varpi(J_i)$. The total number of allotted processor cycles for all jobs in $\mathcal{J}$ is at most $T_1(\mathcal{J}) + \varpi(\mathcal{J})$. Thus, we have $P |D(J_k)| \leq T_1(\mathcal{J}) + \varpi(\mathcal{J})$, and hence $|D(J_k)| \leq (T_1(\mathcal{J}) + \varpi(\mathcal{J}))/P$, which completes the proof. $\square$

Theorem 6.3 indicates that a prudent and integral OS allocator, coupled with a reasonable thread schedulers, can produce a makespan-efficient two-level scheduler. More precisely, when a thread scheduler produces no more than $c_1 \cdot T_1(J_i)$ waste and no more than $c_2 \cdot T_\infty(J_i)$ satisfied count for every job J_i , a prudent and integral OS allocator is $(c_1 + c_2 + 1)$ -competitive with respect to makespan. For example, Proactive RAD is not frugal, but it is prudent and integral. Therefore, Proactive RAD combined with Proactive A-GREEDY produces us a two-level scheduler with $O(1)$ -competitiveness for makespan.

6.4 Techniques for Response Time Analysis

A two-level adaptive scheduler is a composition of an OS allocator and a thread scheduler. Both the OS allocator and the thread scheduler can be rather complex, and their interaction makes the system properties even harder to analyze. We introduce a partitioning approach that separates the issues and determine their individual influences on the final result.

We first introduce two auxiliary concepts.

Definition 6.1 *The **squashed deprived work** of a job set $\mathcal{J}$ on P processors is*

$$sqdw(\mathcal{J}) = \frac{1}{P} \text{sq-sum}(\langle T_1(J_i(D(J_i))) \rangle),$$

where $T_1(J_i(D(J_i)))$ is the total work of job $J_i \in \mathcal{J}$ done during its deprived steps $D(J_i)$. The **squashed deprived allotment** of $\mathcal{J}$ is

$$sqda(\mathcal{J}) = \frac{1}{P} \text{sq-sum}(\langle a(J_i, D(J_i)) \rangle).$$

where $a(J_i, D(J_i))$ is the total allotment of job $J_i \in \mathcal{J}$ during its deprived steps $D(J_i)$.

Our approach for the analysis of mean response time consists of three steps:

1. **Analyze the OS allocator.** The key of this step is to bound the total response time in terms of the squashed deprived allotment and the total satisfied steps.

The result of this step reveals the properties of the OS allocator, which hold independently of thread schedulers. One technique is to evaluate the OS allocator for the total response time by using instantaneous parallelism feedback.

2. **Evaluate the thread scheduler.** Usually, this step involves the process of bounding the squashed deprived allotment in terms of the squashed work, and bounding the satisfied count in terms of the aggregate span.
3. **Integrate the results.** Combine the results of the previous two steps, obtaining the competitive ratio of the two-level adaptive scheduler with respect to mean response time.

I will elaborate these three steps with examples in the remainder of this section.

Step 1: Analyze the OS allocator

To analyze the OS allocator, we bound the total response time in terms of the squashed deprived allotment and the total satisfied steps. One technique is to figure out the total response time of an OS allocator using a greedy thread scheduler with instantaneous parallelism feedback. The bound with instantaneous parallelism feedback¹ usually reflects the total response time when coupling the OS allocator with other thread schedulers. The following observation allows us to express the total response time of a job set $\mathcal{J}$ in terms of the squashed deprived allotment $sqda(\mathcal{J})$ and the total satisfied count $sat(\mathcal{J})$.

Observation 6.1 *Let $\mathcal{J}$ denote an arbitrary batched job set. Suppose that OS allocator Y using instantaneous parallelism feedback satisfies the total response time bound $R(\mathcal{J}) \leq c_1 \cdot sqdw(\mathcal{J}) + c_2 \cdot T_\infty(\mathcal{J})$ for $\mathcal{J}$, where c_1 and c_2 are constants. Let $X+Y$ denote the two-level scheduler combining a thread scheduler X with the OS allocator Y . Then, the total response time produced by $X+Y$ for $\mathcal{J}$ is bounded by*

$$R(\mathcal{J}) \leq c_1 \cdot sqda(\mathcal{J}) + c_2 \cdot sat(\mathcal{J}) . \quad (6.3)$$

For example, Lemma 3.4 bounds the total response time of I-RAD in terms of the squashed deprived work and aggregate span by $R(\mathcal{J}) \leq 2sqdw(\mathcal{J}) + T_\infty(\mathcal{J})$.²

¹In this section, when we say an OS allocator Y using instantaneous parallelism feedback, we refer to that the OS allocator Y is combined with a greedy thread scheduler using instantaneous parallelism feedback.

²The bound presented in Lemma 3.4 is $R(\mathcal{J}) \leq 2sqw(\mathcal{J}) + T_\infty(\mathcal{J})$ instead of $R(\mathcal{J}) \leq 2sqdw(\mathcal{J}) + T_\infty(\mathcal{J})$. We chose not to introduce an extra concept of the squashed deprived work there in order to simplify the notations. In the proof of the lemma, we can see that all the squashed work considered

Lemma 4.4 bounds the total response time of a job set scheduled by RAD in terms of the squashed deprived allotment area and the total satisfied count by $R(\mathcal{J}) \leq 2sqda(\mathcal{J}) + sat(\mathcal{J})$.³ The relation in Lemma 4.4 holds regardless of the behavior of the thread schedulers, as long as they can send their desires to the OS allocator at the beginning of each scheduling quantum. Lemma 3.4 is a special case of Lemma 4.4, where a greedy thread scheduler with instantaneous parallelism is applied.

The main task of step 1 is to bound the total response time of an OS allocator using instantaneous parallelism feedback. Let's assume that the total response time produced by an OS allocator Y satisfies the inequality

$$R(\mathcal{J}) \leq c_1 \cdot sqdw(\mathcal{J}) + c_2 \cdot T_\infty(\mathcal{J}) , \quad (6.4)$$

where c_1 and c_2 are constants to be determined. Since $sqdw(\mathcal{J}) \leq sqw(\mathcal{J})$ holds and $sqw(\mathcal{J})$ is a lower bound on total response time, $sqdw(\mathcal{J})$ and $T_\infty(\mathcal{J})$ are lower bounds on total response time. Therefore, algorithm Y is $(c_1 + c_2)$ -competitive. The values of c_1 and c_2 can be chosen to minimize the competitive ratio. In order to establish Inequality (6.4), mathematical induction can be applied on the remaining execution time of the job set $\mathcal{J}$. Then, we only need to show that the changes in total response time denoted as Δrt , squashed deprived work denoted as $\Delta sqdw$, and aggregated span denoted as ΔT_∞ , over a time step t satisfy the inequality $c_1 \cdot \Delta sqdw + c_2 \cdot \Delta T_\infty \geq \Delta rt$. The change in total response time Δrt and aggregated span ΔT_∞ are usually not hard to calculate. The change in squashed deprived work area $\Delta sqdw$ can be bounded using properties of the squashed sum. Lemmas 3.1 and 3.2 provide examples of such properties.

Theorem 3.6 provides an example of how to apply such a technique. I will present another example here for a well-known OS allocator, equipartitioning (EQ), which gives each uncompleted job an equal number of processors. We show that EQ is $(2 + \sqrt{3})$ -competitive for batched jobs with respect to mean response time. This result is consistent with the result obtained by Edmonds *et al.* [52]. Certainly, our

---

is generated from deprived jobs. For satisfied jobs, we only consider its span reduced in each time step. Therefore, Lemma 3.4 indeed bounds the total response time by the squashed deprived work and the aggregate span by $R(\mathcal{J}) \leq 2sqdw(\mathcal{J}) + T_\infty(\mathcal{J})$.

³The bound presented in Lemma 4.4 is $R(\mathcal{J}) \leq 2sqda(\mathcal{J}) + sat(\mathcal{J})$ instead of $R(\mathcal{J}) \leq 2sqda(\mathcal{J}) + sat(\mathcal{J})$. In the proof of the lemma, we can see that all the squashed allotment considered is generated from deprived jobs. For satisfied jobs, we only consider its satisfied count. Therefore, Lemma 4.4 indeed bounds the total response time by the squashed deprived allotment and the satisfied count by $R(\mathcal{J}) \leq 2sqda(\mathcal{J}) + sat(\mathcal{J})$.

analysis is different from theirs. Moreover, they assume that jobs have multiple phases of arbitrary nondecreasing and sublinear speedup functions, while our result can be applied to arbitrary DAGs.

We now apply the general technique and prove a bound on the competitiveness of the mean response time for EQ in a batched setting. Please note that we can consider EQ as using instantaneous parallelism feedback even though it just allots each job an equal number of processors anyway. Thus, under the schedule of EQ, a step is **satisfied** for job J_i if the allotment of J_i is greater than or equal to its instantaneous parallelism, while a step is **deprived** otherwise.

Theorem 6.4 *Equipartitioning is $(2 + \sqrt{3})$ -competitive with respect to the mean response time for any batched job set $\mathcal{J}$. More precisely, the total response time is bounded by*

$$R(\mathcal{J}) \leq (1 + 2/\sqrt{3})sqdw(\mathcal{J}) + (1 + 1/\sqrt{3})T_\infty(\mathcal{J}) , \quad (6.5)$$

Proof. Assume as an inductive hypothesis that the total response time produced by equipartitioning satisfies Inequality (6.4) as

$$R(\mathcal{J}) \leq c_1 \cdot sqdw(\mathcal{J}) + c_2 \cdot T_\infty(\mathcal{J}) , \quad (6.6)$$

where c_1 and c_2 are constants. We shall prove Inequality (6.6) on the remaining execution time steps of the job set $\mathcal{J}(\vec{t})$. Then, we calculate the precise values of c_1 and c_2 . This proof uses the t-suffix notation $\vec{t}$ as defined in Section 3.3.

Basis: $t = T(\mathcal{J}) + 1$. Since we have $\mathcal{J}(\vec{t}) = \emptyset$, it follows that $R(\mathcal{J}(\vec{t})) = 0$, $sqdw(\mathcal{J}(\vec{t})) = 0$, and $T_\infty(\mathcal{J}(\vec{t})) = 0$. Thus, the claim holds trivially.

Induction: $1 \leq t \leq T(\mathcal{J})$. Suppose by induction that Inequality (6.6) holds at time step $t + 1$, i.e.,

$$R(\mathcal{J}(\overrightarrow{t+1})) \leq c_1 \cdot sqdw(\mathcal{J}(\overrightarrow{t+1})) + c_2 \cdot T_\infty(\mathcal{J}(\overrightarrow{t+1})) . \quad (6.7)$$

We will show that it still holds at time step t , i.e.,

$$R(\mathcal{J}(\vec{t})) \leq c_1 \cdot sqdw(\mathcal{J}(\vec{t})) + c_2 \cdot T_\infty(\mathcal{J}(\vec{t})) . \quad (6.8)$$

The following notations denote the changes in, respectively, the total response time, the squashed deprived work area, and the aggregate span from time t to $t + 1$:

$$\Delta rt = R(\mathcal{J}(\vec{t})) - R(\mathcal{J}(\overrightarrow{t+1})) ,$$

$$\begin{aligned}\Delta sqdw &= sqdw(\mathcal{J}(\vec{t})) - sqdw(\mathcal{J}(\overline{t+1})) , \\ \Delta T_\infty &= T_\infty(\mathcal{J}(\vec{t})) - T_\infty(\mathcal{J}(\overline{t+1})) .\end{aligned}$$

Given the induction hypothesis (Inequality (6.7)), we need only show that the following inequality holds for our claim (Inequality (6.8)) to be true.

$$c_1 \cdot \Delta sqdw + c_2 \cdot \Delta T_\infty \geq \Delta rt . \quad (6.9)$$

The changes in total response time and the aggregated span can be calculated as follows. We have $\Delta rt = |\mathcal{J}(t)|$, since each uncompleted job has its response time decreased by 1 at any time step t . Recall that $\mathcal{JS}(t)$ denotes the set of satisfied jobs at step t . Using instantaneous parallelism as feedback, each satisfied job has its span reduced by 1. Thus, we have $\Delta T_\infty \geq |\mathcal{JS}(t)|$.

The increase in squashed deprived work area $\Delta sqdw$ is more complicated. Recall that, under EQ, a step is satisfied for job J_i if the allotment of J_i is greater than or equal to its instantaneous parallelism, while a step is deprived otherwise. Therefore, satisfied jobs can still waste processors even when deprived jobs exist. Only the $|\mathcal{JD}(t)| P/n$ processors that are allocated to deprived jobs are guaranteed to be utilized. By applying the squashed sum property given in Lemma 3.2 with $S = |\mathcal{JD}(t)| P/n$ and $l = |\mathcal{JD}(t)|$, we have

$$\begin{aligned}\Delta sqdw &\geq \frac{(P |\mathcal{JD}(t)| / n) (|\mathcal{JD}(t)| + 1) / 2}{P} \\ &= \frac{|\mathcal{JD}(t)| (|\mathcal{JD}(t)| + 1)}{2n} .\end{aligned} \quad (6.10)$$

For Inequality (6.5) to hold, it suffices for c_1 and c_2 to satisfy

$$c_1 \frac{|\mathcal{JD}(t)| (|\mathcal{JD}(t)| + 1)}{2n} + c_2 |\mathcal{JS}(t)| \geq n. \quad (6.11)$$

Given $n = |\mathcal{JD}(t)| + |\mathcal{JS}(t)|$, Inequality (6.11) simplifies to

$$(c_1 - 2) |\mathcal{JD}(t)|^2 + (2c_2 - 4) |\mathcal{JD}(t)| |\mathcal{JS}(t)| + (2c_2 - 2) |\mathcal{JS}(t)|^2 \geq 0 . \quad (6.12)$$

One set of sufficient conditions that satisfy Inequality (6.12) is given by

$$c_1 - 2 \geq 0 \quad \implies \quad c_1 \geq 2$$

$$\begin{aligned}
2c_2 - 2 \geq 0 &\implies c_2 \geq 1 \\
4(c_1 - 2)(2c_2 - 2) \geq (2c_2 - 4)^2 &\implies 2c_1c_2 - 2c_1 \geq c_2^2
\end{aligned}$$

Minimizing $(c_1 + c_2)$ with the above constraints yields

$$c_1 = 1 + \frac{2}{\sqrt{3}}$$

and

$$c_2 = 1 + \frac{1}{\sqrt{3}}$$

Therefore, the competitive ratio for equipartitioning is $c_1 + c_2 = 2 + \sqrt{3}$. $\square$

Remarks — From the analysis of RAD and EQ, we can see that **fairness is an important factor to ensure good mean response time**. Ideally, the optimal mean response time can be achieved by executing short jobs first. When schedulers are nonclairvoyant, we know neither the execution time nor the speedup function of a job. Intuitively, the best we can do is to give each job an equal slice of time. When scheduling batched serial jobs on a single processor, round-robin is $(2 - 2/(|\mathcal{J}|))$ -competitive for mean response time, which is optimal for deterministic nonclairvoyant scheduling. Round-robin is fair in the sense that it always gives each job an equal amount of processing time. The importance of fairness in achieving good mean response time motivates our work to combine round robin with DEQ to obtain RAD, which always ensures a fair share of allotment under both lightly- and heavily-loaded systems. $\square$

Step 2: Evaluate thread scheduler

We now evaluate the thread scheduler in the two-level scheduling system. In particular, we would like to bound the squashed deprived allotment in terms of the squashed work, and bound the total satisfied steps in terms of the aggregate span, because squashed work and aggregate span are lower bounds for total response time.

The squashed deprived allotment of a job set is closely related to the nonoverflow waste of individual jobs. Suppose that $a(J_i, D(J_i))$, $T_1(J_i(D(J_i)))$, and $w(J_i, D(J_i))$ denote the total allotment, work and waste of job J_i at its deprived steps. For each job $J_i \in \mathcal{J}$, if its nonoverflow waste is bounded by using its total work, *i.e.*, $\varpi(J_i) \leq \alpha \cdot T_1(J_i)$, the squashed deprived allotment of $\mathcal{J}$ can be bounded by its

squashed work as follows:

$$\begin{aligned} sqda(\mathcal{J}) &= \text{sq-sum}(\langle a(J_i, D(J_i)) \rangle) / P \\ &= \text{sq-sum}(\langle T_1(J_i(D(J_i))) + w(J_i, D(J_i)) \rangle) / P \end{aligned} \quad (6.13)$$

$$\leq \text{sq-sum}(\langle T_1(J_i) + \varpi(J_i) \rangle) / P \quad (6.14)$$

$$\leq \text{sq-sum}(\langle (\alpha + 1)T_1(J_i) \rangle) / P \quad (6.15)$$

$$= (\alpha + 1) \text{sq-sum}(\langle T_1(J_i) \rangle) / P \quad (6.16)$$

$$= (\alpha + 1) sqw(\mathcal{J}) . \quad (6.17)$$

Inequality (6.14) is derived from Inequality (6.13) according to Lemma 3.1, as $T_1(J_i(D(J_i))) \leq T_1(J_i)$ and $w(J_i, D(J_i)) \leq \varpi(J_i)$. Similarly, we can get Inequality (6.15) from Inequality (6.14) since $\varpi(J_i) \leq \alpha \cdot T_1(J_i)$. Finally, based on Definition 3.2, we obtain Inequality (6.17), *i.e.*, $sqda(\mathcal{J}) \leq (\alpha + 1) sqw(\mathcal{J})$.

For example, according to Theorem 2.11, the nonoverflow waste of a job J_i scheduled by A-GREEDY is bounded by $(1 + \rho - \delta)T_1(J_i) / \delta$. From Inequality (6.17), we get that the squashed deprived allotment of job set $\mathcal{J}$ is bounded by $sqda(\mathcal{J}) \leq ((1 + \rho) / \delta) sqw(\mathcal{J})$, which is consistent with the result proved in Lemma 4.5. Similarly, according to Lemma 6.1, the nonoverflow waste of Proactive A-GREEDY is bounded by $(1 + \rho - \delta)T_1(J_i) / \delta$. Therefore, we obtain that $sqda(\mathcal{J}) \leq ((1 + \rho) / \delta) sqw(\mathcal{J})$.

The number of total satisfied steps of a job set is closely related to the satisfied count of individual jobs. For example, for each job $J_i \in \mathcal{J}$, if its satisfied count can be bounded by its span, *i.e.*, $sat(J_i) \leq \beta \cdot T_\infty(J_i)$, we can bound the total satisfied steps by the aggregate span as

$$\begin{aligned} sat(\mathcal{J}) &= \sum_{J_i \in \mathcal{J}} sat(J_i) \\ &\leq \sum_{J_i \in \mathcal{J}} (\beta \cdot T_\infty(J_i)) \\ &= \beta \cdot T_\infty(\mathcal{J}) . \end{aligned}$$

Another example is from G-RAD where the satisfied count of a job J_i is bounded by $sat(J_i) \leq 2T_\infty(J_i) / (1 - \delta) + L \log_\rho P + L$ (Lemma 4.1). Therefore, the total number of satisfied steps of $\mathcal{J}$ is bounded by

$$sat(\mathcal{J}) = \sum_{J_i \in \mathcal{J}} sat(J_i)$$

$$\begin{aligned}
&\leq \sum_{J_i \in \mathcal{J}} \left(\frac{2T_\infty(J_i)}{1-\delta} + L \log_\rho P + L \right) \\
&= \frac{2}{1-\delta} T_\infty(\mathcal{J}) + |\mathcal{J}| (L \log_\rho P + L) .
\end{aligned}$$

which is the result proved in Lemma 4.6.

Step 3: Integrate results

We can bound the mean response time of the scheduling algorithm by combining the results from Step 1 and 2. For example, suppose that an OS allocator satisfies the inequality $R(\mathcal{J}) \leq c_1 \cdot sqda(\mathcal{J}) + c_2 \cdot sat(\mathcal{J})$ from Step 1 for some constants c_1 and c_2 . When the nonoverflow waste of a thread scheduler X is not more than α times the total work and its satisfied count is not more than β times its span, the two-level algorithm $X+Y$ is $(c_1 \cdot (\alpha + 1) + c_2 \cdot \beta)$ -competitive for mean response time.

The proof of G-RAD in lightly-loaded system is a concrete example of the partitioning approach for mean response time analysis. Lemma 4.4 is the first step, which bounds the total response time by using squashed deprived allotment and total satisfied count as follows:

$$R(\mathcal{J}) \leq \left(2 - \frac{2}{|\mathcal{J}| + 1} \right) sqda(\mathcal{J}) + sat(\mathcal{J}) . \quad (6.18)$$

The second step involves Lemmas 4.5 and 4.6, where Lemma 4.5 bounds the squashed allotment by using the squashed work, *i.e.*,

$$sqda(\mathcal{J}) \leq \frac{\rho + 1}{\delta} sqw(\mathcal{J}) . \quad (6.19)$$

The Lemma 4.6 bounds the total satisfied count by using the total span, *i.e.*,

$$sat(\mathcal{J}) \leq \frac{2}{1-\delta} T_\infty(\mathcal{J}) + |\mathcal{J}| (L \log_\rho P + L) . \quad (6.20)$$

Finally, in the third step, by combining Inequalities (6.18), (6.19), and (6.20), we obtain the response time bound of G-RAD in the lightly-loaded system as

$$R(\mathcal{J}) \leq \left(2 - \frac{2}{|\mathcal{J}| + 1} \right) \frac{\rho + 1}{\delta} sqw(\mathcal{J}) + \frac{2}{1-\delta} T_\infty(\mathcal{J}) + |\mathcal{J}| L(\log_\rho P + 1) .$$

These three steps complete the analysis of a two-level adaptive scheduler with respect to the mean response time of batched jobs.

Chapter 7

Scheduling on Heterogeneous Resources

Most prior work is dedicated to the scheduling of jobs that require a single type of processing resource. Many parallel systems, however, embed special-purpose processors such as vector units, floating-point coprocessors, I/O processors, and so on. Application programs thus generally involve the interleaving of different types of tasks, where a task of specific type can only be carried out on the matching type of resource. Presently, there is no provably good online scheduling algorithm that ensures efficient use of multiple resources with heterogeneous functionalities.

Menasce and Almeida [105] defined two distinct forms of heterogeneity in high-performance computing systems. **Performance heterogeneity** exists in systems that contain general-purpose processors of different speeds. A task can execute on any of the processors, but it executes faster on some than on others. Executing parallel programs on processors with different speeds has been studied extensively [14, 35, 39, 43, 82, 83, 100, 124] in the scheduling theory. **Functional heterogeneity**, on the other hand, exists in systems that contain various types of processors, such as vector units, floating-point coprocessors, and I/O processors. With functional heterogeneity, some of the tasks may not be able to execute on some of the functional units.

Hamidzadeh *et al.* [72] propose a generic model that incorporates performance heterogeneity by assigning an infinite computation time to a task on an unmatched functional unit. They empirically study a dynamic self-adjusting scheduling algorithm for this type of heterogeneous system. Other forms of functional heterogeneity are also defined on a coarse level for machines with different types of parallel architectures such

as SIMD, MIMD, and vectors, etc. in [91,92] and more recently for cell processors [81]. To the best of my knowledge, however, no general algorithm offers provable efficiency for scheduling parallel applications on functionally heterogeneous resources.

This chapter introduces the ***K-resource scheduling model*** for functional heterogeneous scheduling environment. In this model, the processors and the tasks are categorized into K types, and a task of a given category can only be executed on a processor of the matching category. Any two tasks of a job can be executed concurrently as long as they do not violate precedence constraints.

I extend the RAD algorithm for homogeneous resources to K-RAD for K -resource scheduling. I also analyze the efficiency of K-RAD with respect to both makespan and mean response time. The main analytical results are listed as follows:

- I show that any deterministic online nonclairvoyant algorithm is at best $(K+1 - 1/P_{max})$ -competitive for makespan in K -resource system, where P_{max} denotes the maximum number of processors among the K categories of resources.
- I show that K-RAD is $(K + 1 - 1/P_{max})$ -competitive with respect to the makespan for any job set with arbitrary release times. Since this bound matches the lower bound, K-RAD is optimal for deterministic nonclairvoyant K -resource scheduling with respect to makespan.
- I show that K-RAD is $(4K + 1 - 4K/(|\mathcal{J}| + 1))$ -competitive with respect to the mean response time for any batched job set $\mathcal{J}$.

The remainder of the chapter is organized as follows. Section 7.1 formalizes the scheduling model and job model, and Section 7.2 introduces the K-RAD algorithm for functional heterogeneous environment. Section 7.3 proves the lower bound of K-RAD with respect to the makespan, and Section 7.4 follows with a competitive analysis of K-RAD for makespan. Section 7.5 presents mean response time lower bounds for batched K-DAG jobs. Section 7.6 presents the analysis of K-RAD for mean response time.

7.1 The K -Resource Scheduling Model and Algorithm

The new scheduling problem on heterogeneous resources consists of a collection of independent jobs $\mathcal{J} = \{J_1, J_2, \dots, J_{|\mathcal{J}|}\}$ to be scheduled on a collection of processors in K -resource systems. The processors and the tasks are categorized into K types, and a task can only be executed on a processor with the matching type. We refer to

Figure 7-1: An example of K -DAG job model. The job is represented by a 3-DAG with 3 different types of tasks given by circles, squares, and triangles respectively.

the processors belonging to a category α as **α -processors**, and the tasks running on α -processors as **α -tasks**. For each category $\alpha \in \{1, \dots, K\}$, we denote the number of α -processors in the system by P_α . In this section, we formalize the job model, and the scheduling model.

Job Model

As a natural extension to the conventional dag [15, 18, 19, 22, 23, 54, 80, 111], a parallel job with heterogenous tasks is represented as a K -color dag, or **K-DAG** in short. A K-DAG has up to K types of vertices, and each α -*vertex* represents a unit-time α -task where $\alpha \in \{1, \dots, K\}$. For a job $J_i \in \mathcal{J}$, we denote the set of α -vertices of the job by $V(J_i, \alpha)$. Define $V(J_i) = \bigcup_{\alpha=1, \dots, K} V(J_i, \alpha)$. The α -*work* $T_1(J_i, \alpha)$ corresponds to the total number of α -vertices in the K-DAG, *i.e.*, $T_1(J_i, \alpha) = |V(J_i, \alpha)|$. Each edge $e \in E(J_i)$ from a vertex u to a vertex v represents a dependency between the two corresponding tasks, regardless of their types. Like an ordinary DAG, the span $T_\infty(J_i)$ of a K-DAG corresponds to the number of nodes on the longest chain of the precedence dependencies.

Figure 7-1 shows an example of a K-DAG job J_i with three different types of tasks represented by three different types of vertices. In this example, the circular vertices, the square vertices, and the triangular vertices denote α_1 -tasks, α_2 -tasks, and α_3 -tasks, respectively, of the job. The work of each task is $T_1(J_i, \alpha_1) = 7$, $T_1(J_i, \alpha_2) = 4$ and $T_1(J_i, \alpha_3) = 3$. The span of the job is $T_\infty(J_i) = 7$.

Scheduling Model

The K -resource scheduling model can be defined as follows. A schedule $\chi = (\tau, \pi_1, \pi_2, \dots, \pi_K)$ of a job set $\mathcal{J}$ is defined by $K + 1$ mappings. The function $\tau : \bigcup_{J_i \in \mathcal{J}} V(J_i) \rightarrow \{1, 2, \dots, \infty\}$ maps the vertices of the jobs in the job set $\mathcal{J}$ to the set of time steps. For each resource category $\alpha \in \{1, \dots, K\}$, the function $\pi_\alpha : \bigcup_{J_i \in \mathcal{J}} V(J_i, \alpha) \rightarrow \{1, 2, \dots, P_\alpha\}$ maps the set of α -vertices of the jobs in the job set $\mathcal{J}$ to the set of α -processors on the machine. A valid schedule must preserve the precedence relationship of each job. For any two vertices $u, v \in V(J_i)$ of the job J_i , if $u \prec v$, then $\tau(u) < \tau(v)$, i.e., the task denoted by vertex u must be executed before the task denoted by vertex v . A valid schedule must also ensure that one processor can only be assigned to one job at any given time. For any two vertices $u, v \in \bigcup_{J_i \in \mathcal{J}} V(J_i, \alpha)$, both $\tau(u) = \tau(v)$ and $\pi_\alpha(u) = \pi_\alpha(v)$ hold if and only if $u = v$.

7.2 The K-RAD Algorithm

In this section, we present the K-RAD algorithm, which is an extension of the RAD algorithm in Chapter 3 which schedules jobs on homogeneous resources. To schedule jobs with heterogeneous tasks on heterogeneous processors, K-RAD assigns a variant of the RAD scheduler to each category α of processors to schedule the α -tasks of all jobs in the job set, where $\alpha = 1, \dots, K$. In this section, we discuss the variant of the RAD algorithm in detail. In fact, we can use the original RAD defined in Section 3.1, but the variant provides a better support for our mean response time analysis. I believe that the two versions of RAD should perform equally well in practice. In the remainder of this chapter, RAD refers to the variant defined in this section, unless specified otherwise.

For each job $J_i \in \mathcal{J}$, define the **α -desire** $d(J_i, \alpha, t)$ to be the total number of ready α -tasks or the **instantaneous α -parallelism** of J_i at time step t , and define the **α -allotment** $a(J_i, \alpha, t)$ to be the total number of α -processors allotted to J_i at time step t . Note that an uncompleted job J_i at any time t has total desire $\sum_{\alpha=1 \dots K} d(J_i, \alpha, t) \geq 1$, although for some $\alpha \in \{1, \dots, K\}$, its α -desire might be 0. At a given time step t , if a job J_i has nonzero α -desire, we say that J_i is **α -active**; otherwise, it is **α -inactive**. For each category α of processors, let $\mathcal{J}(\alpha, t)$ denote the set of α -active jobs at time step t , i.e., $\mathcal{J}(\alpha, t) = \{J_i \in \mathcal{J} : d(J_i, \alpha, t) > 0\}$.

Whenever $|\mathcal{J}(\alpha, t)| \leq P_\alpha$, RAD uses DEQ to partition processors among the active jobs. DEQ gives each job a fair share of allotment unless a job requests less. As a result, all jobs requesting less than fair shares of processors tend to get what

they request, while the other jobs requesting larger numbers of processors get an equal number of processors, which is the mean deprived allotment.

Once $|\mathcal{J}(\alpha, t)| > P_\alpha$, RAD starts an **RR cycle** to allot processors among the α -active jobs. In order to ensure fairness, all α -active jobs that have been scheduled once in the RR cycle are marked and RAD maintains the marked and unmarked α -active jobs in two separate queues Q and Q' . The queue Q contains the α -active jobs that have not been marked since the beginning of the cycle, and the other queue Q' contains the α -active jobs that have been marked. At any time step in the cycle, if there are more than P_α jobs in Q , RAD always schedules P_α jobs at the beginning of Q . When there are fewer than P_α jobs in Q , in order not to waste processors, RAD moves $\min(|Q'|, P_\alpha - |Q|)$ jobs from queue Q' to Q and partitions the processors among the jobs in Q using the DEQ algorithm. Such a time step also indicates the completion of the RR cycle, and all jobs are left unmarked. Figure 7-2 shows the pseudocode of the RAD algorithm. The main procedure RAD is called before every step t , and it in turn calls subprocedures DEQ or round-robin, depending on the relationship between the number of processors and the number of unmarked α -active jobs at t .

Under the schedule of RAD, at any time step t for a category α of processors, an active job J_i can be either **α -satisfied** if its α -allotment is equal to its α -desire, i.e., $a(J_i, \alpha, t) = d(J_i, \alpha, t)$, or **α -deprived** if its α -allotment is less than its α -desire, i.e., $a(J_i, \alpha, t) < d(J_i, \alpha, t)$. Moreover, we define a job J_i to be **$\forall$ -satisfied** if it is α -satisfied for all $\alpha = 1, \dots, K$, and the job is **$\exists$ -deprived** otherwise.

7.3 K-DAG Makespan Lower Bounds

This section presents two lower bounds on the makespan for scheduling any job set with arbitrary release times. We also exhibit a lower bound on the competitive ratio for any deterministic online nonclairvoyant algorithm. We show that any deterministic online nonclairvoyant algorithm for K -resource scheduling is at best $(K+1-1/P_{max})$ -competitive with respect to makespan, where $P_{max} = \max_{\alpha=1, \dots, K} P_\alpha$.

We first define a simple but important notation.

Definition 7.1 The **total α -work** of a job set $\mathcal{J}$ is

$$T_1(\mathcal{J}, \alpha) = \sum_{J_i \in \mathcal{J}} T_1(J_i, \alpha) , \quad (7.1)$$

```

DEQ ( $\alpha, t, Q, P$ )
1  if  $Q = \emptyset$    return
2   $S \leftarrow \{J_i \in Q : d(J_i, \alpha, t) \leq P/|Q|\}$ 
3  if  $S = \emptyset$ 
     $\triangleright$  Jobs requesting many processors get an equal share of processors
4    for each  $J_i \in Q$ 
5       $a(J_i, \alpha, t) \leftarrow P/|Q|$ 
6    return
7  else    $\triangleright$  Jobs requesting less processors get what they request
8    for each  $J_i \in S$ 
9       $a(J_i, \alpha, t) \leftarrow d(J_i, \alpha, t)$ 
10   DEQ( $\alpha, t, Q - S, P - \sum_{J_i \in S} d(J_i, \alpha, t)$ )

ROUND-ROBIN ( $\alpha, t, Q, P$ )
1   $S \leftarrow$  the first  $P$  jobs from  $Q$ 
2  for each  $J_i \in S$ 
3     $a(J_i, \alpha, t) \leftarrow 1$ 
4    mark  $J_i$  to indicate that  $J_i$  has been scheduled once in the RR cycle

RAD ( $\alpha, t, \mathcal{J}, P$ )
1   $Q \leftarrow \{J_i \in \mathcal{J} : J_i \text{ is unmarked and } \alpha\text{-active}\}$ 
2   $Q' \leftarrow \{J_i \in \mathcal{J} : J_i \text{ is marked and } \alpha\text{-active}\}$ 
3  if  $|Q| > P$ 
4    ROUND-ROBIN( $\alpha, t, Q, P$ )
5  else
6    Move  $\min(|Q'|, P - |Q|)$  jobs from  $Q'$  to  $Q$ 
7    DEQ( $\alpha, t, Q, P$ )
8    unmark all jobs to indicate the completion of the RR cycle

```

Figure 7-2: Pseudocode for the variant of the RAD OS allocator used in heterogeneous resource scheduling. The pseudocode includes the main procedure RAD, a subprocedure DEQ that implements the DEQ algorithm, and another subprocedure ROUND-ROBIN that implements the round-robin algorithm. Before each time step t , K-RAD calls RAD to partition the α -processors among jobs.

where $T_1(J_i, \alpha)$ is the α -work of job $J_i \in \mathcal{J}$.

We introduce two lower bounds on the makespan for a job set $\mathcal{J}$ represented by K-DAGs with arbitrary release time. Let $T^*(\mathcal{J})$ denote the makespan of $\mathcal{J}$ scheduled by an optimal clairvoyant scheduler. No scheduler on K-DAG can do better than

7.3. K-DAG Makespan Lower Bounds

127

the optimal scheduler on any single type of task in the K-DAG . Thus, based on the lower bounds [30] for jobs with a single type of task, we obtain the following two lower bounds:

$$T^*(\mathcal{J}) \geq \max_{J_i \in \mathcal{J}} (r(J_i) + T_\infty(J_i)) , \quad (7.2)$$

$$T^*(\mathcal{J}) \geq \max_{\alpha=1,\dots,K} (T_1(\mathcal{J}, \alpha) / P_\alpha) . \quad (7.3)$$

The next theorem proves a lower bound on the competitive ratio for makespan for any deterministic online nonclairvoyant algorithm. As with the analysis of many other online algorithms, the proof involves an adversarial argument. Upon knowing the strategy of any deterministic algorithm, the adversary makes the algorithm perform badly at each step so that the competitive ratio is maximized.

Theorem 7.1 *Any deterministic online nonclairvoyant algorithm for K -resource scheduling is at best $(K + 1 - 1/P_{\max})$ -competitive with respect to makespan, where $P_{\max} = \max_{\alpha=1,\dots,K} P_\alpha$.*

Proof. Without loss of generality, let us assume that $P_K = P_{\max}$. Consider a job set $\mathcal{J}$ with $n = mP_1P_K$ jobs where m is a large integer, as shown in Figure 7-3. All except one of the n jobs have only one unit of 1-task. Job J_i 's span is $T_\infty(J_i) = K + mP_K - 1$ and it is highlighted by the darkened nodes in the figure. Level 1 of J_i has one unit of 1-task, and each of the subsequent level $\alpha \in \{2, \dots, K - 1\}$ has exactly $mP_\alpha P_K$ units of α -task, all of which depend on a single task from the previous level. Level K of J_i has $mP_K(P_K - 1) + 1$ units of K -task, one of which is followed by a chain of K -task of length $mP_K - 1$.

To schedule this set of jobs, an optimal offline scheduler $\mathcal{S}$ always executes the ready tasks of the job J_i on the span first so that the tasks at the subsequent level can be executed as early as possible by the other types of processors. For $\alpha = 1, \dots, K$, all α -tasks of the job J_i are ready at time $\alpha - 1$. The K -tasks, as the last type of tasks, can be completed in mP_K time steps by executing one unit of K -task on the span at each step. Therefore, the optimal scheduler $\mathcal{S}$ produces a makespan of $T^*(\mathcal{J}) = K + mP_K - 1$.

To schedule the same set of jobs, any deterministic nonclairvoyant algorithm $\mathcal{A}$ can be prevented from performing well by the adversary in such a way that the tasks of the job J_i on the span are always executed last among the ready tasks. Such an adversarial strategy forces algorithm $\mathcal{A}$ to execute different types of tasks in a

Figure 7-3: An example that shows any deterministic online scheduler is at best $(K + 1 - K/P_{max})$ -competitive for makespan. This example presents a set of n jobs that can force any deterministic online scheduler to have a competitive ratio of $K + 1 - K/P_{max}$ with respect to makespan in an adversary setting. The total number of tasks at each level is indicated on the right of that level. While the optimal clairvoyant scheduler chooses to execute the ready tasks on the span (denoted by the dark nodes of job J_i in the figure) first, an adversary can force a deterministic online scheduler to choose the “wrong” tasks for execution. The delayed execution of the tasks on the span prevents the efficient use of the resources.

sequential manner. Therefore, in the worst case, $\mathcal{A}$ takes $T(\mathcal{J}) \geq mKP_K + mP_K - m$ time steps. Thus, the competitive ratio is given by

$$\begin{aligned} \frac{T(\mathcal{J})}{T^*(\mathcal{J})} &\geq \frac{mKP_K + mP_K - m}{K + mP_K - 1} \\ &= \frac{KP_K + P_K - 1}{\frac{K-1}{m} + P_K}. \end{aligned}$$

For $m \gg K$, the ratio $(K - 1)/m$ approaches 0. Therefore, we have

$$\begin{aligned} \frac{T(\mathcal{J})}{T^*(\mathcal{J})} &\geq \frac{KP_K + P_K - 1}{P_K} \\ &= K + 1 - \frac{1}{P_{max}}. \end{aligned}$$

□

7.4 K-RAD Makespan Analysis

This section shows that K-RAD is $(K + 1 - 1/P_{max})$ -competitive with respect to the makespan. Since this competitive ratio matches the lower bound given in Section 7.3, K-RAD is an optimal deterministic nonclairvoyant algorithm with respect to makespan.

Suppose that $T(\mathcal{J})$ is the makespan of job set $\mathcal{J}$ scheduled by K-RAD. Let $I = [1, 2, \dots, T(\mathcal{J})]$ denote the time interval in which K-RAD schedules the job set. Since $\mathcal{J}$ denotes any job set with arbitrary job release times, we may have subintervals of I in which no job is active. We refer to any subinterval $l = [t_1, \dots, t_2] \in I$ as an **idle interval** if no job is active in l , *i.e.*, all the jobs released before l are completed by $t_1 - 1$, and no new jobs are released until $t_2 + 1$. Thus, no work is done during interval l . In order to analyze the makespan of K-RAD, we first prove a lemma that bounds the makespan of any job set scheduled by K-RAD without any idle interval. Afterward, we will relax this constraint and give the main theorem.

The following lemma bounds the makespan of any job set scheduled by K-RAD without any idle interval.

Lemma 7.2 *Suppose that K-RAD schedules a job set $\mathcal{J}$ on P_α α -processors for each $\alpha = 1, \dots, K$. If there are no idle intervals during the schedule, then job set $\mathcal{J}$ completes in*

$$T(\mathcal{J}) \leq \sum_{\alpha=1, \dots, K} \frac{T_1(\mathcal{J}, \alpha)}{P_\alpha} + \left(1 - \frac{1}{P_{max}}\right) \max_{J_i \in \mathcal{J}} (T_\infty(J_i) + r(J_i)) \quad (7.4)$$

time steps.

Proof. Suppose that job J_k is the last job completed among the jobs in $\mathcal{J}$. Let $R(J_k)$ denote the set of time steps before J_k is released, and let $S(J_k)$ and $D(J_k)$ denote the sets of $\forall$ -satisfied and $\exists$ -deprived time steps for J_k , respectively. Since J_k is the last job completed, and $R(J_k)$, $S(J_k)$ and $D(J_k)$ are disjoint sets, we have $T(\mathcal{J}) = |R(J_k)| + |S(J_k)| + |D(J_k)|$. We bound these three sets separately.

(1) **To bound $|R(J_k)|$:** There are $r(J_k)$ time steps before the release of job J_k , *i.e.*,

$$|R(J_k)| = r(J_k). \quad (7.5)$$

We calculate the total amount of work done on these steps. Let $T'_1(\mathcal{J}, \alpha)$ denote the total α -work done before the release of J_k . Since there are no idle intervals, each step

in $R(J_k)$ completes at least one unit of work. Thus, we have $\sum_{\alpha=1,\dots,K} T'_1(\mathcal{J}, \alpha) \geq |R(J_k)| = r(J_k)$.

(2) To bound $|S(J_k)|$: On each $\forall$ -satisfied step t for job J_k , all of the ready tasks of J_k are executed, and so the span of J_k is reduced by 1. Therefore, the total number of $\forall$ -satisfied steps for job J_k is at most the span $T_\infty(J_k)$ of J_k , i.e.,

$$|S(J_k)| \leq T_\infty(J_k). \quad (7.6)$$

Let $T''_1(\mathcal{J}, \alpha)$ denote the total α -work done on $\forall$ -satisfied steps of J_k . Since each such step completes at least one unit of work, we have $\sum_{\alpha=1,\dots,K} T''_1(\mathcal{J}, \alpha) \geq |S(J_k)|$.

(3) To bound $|D(J_k)|$: We will first calculate the number of α -deprived steps of job J_k for each resource category α . Let $D(J_k, \alpha)$ denote the set of α -deprived steps for job J_k . According to K-RAD, on each deprived step $t \in D(J_k, \alpha)$, K-RAD must allot all α -processors to jobs. Thus, the total α -allotment on such a step is always equal to the total number of α -processors P_α . Since jobs always use allotted processors productively, there are P_α units of α -work done on such a time step. The total amount of α -work done on $D(J_k, \alpha)$ steps is $T_1(\mathcal{J}, \alpha) - T'_1(\mathcal{J}, \alpha) - T''_1(\mathcal{J}, \alpha)$, and thus we have $|D(J_k, \alpha)| \leq (T_1(\mathcal{J}, \alpha) - T'_1(\mathcal{J}, \alpha) - T''_1(\mathcal{J}, \alpha)) / P_\alpha$.

We now bound the total number of $\exists$ -deprived steps $|D(J_k)|$ for job J_k . Since $D(J_k) = \bigcup_{\alpha=1,\dots,K} D(J_k, \alpha)$, we have $|D(J_k)| \leq \sum_{\alpha=1,\dots,K} |D(J_k, \alpha)|$. Thus, we obtain

$$\begin{aligned} |D(J_k)| &\leq \sum_{\alpha=1,\dots,K} \frac{T_1(\mathcal{J}, \alpha) - T'_1(\mathcal{J}, \alpha) - T''_1(\mathcal{J}, \alpha)}{P_\alpha} \\ &\leq \sum_{\alpha=1,\dots,K} \frac{T_1(\mathcal{J}, \alpha)}{P_\alpha} - \frac{r(J_k) + |S(J_k)|}{P_{\max}}. \end{aligned} \quad (7.7)$$

We can now bound $T(\mathcal{J})$. From Equation (7.5) and Inequalities (7.6) and (7.7), we conclude that the makespan of the job set $\mathcal{J}$ is

$$\begin{aligned} T(\mathcal{J}) &= |R(J_k)| + |D(J_k)| + |S(J_k)| \\ &\leq r(J_k) + \sum_{\alpha=1,\dots,K} \frac{T_1(\mathcal{J}, \alpha)}{P_\alpha} - \frac{r(J_k) + |S(J_k)|}{P_{\max}} + |S(J_k)| \\ &\leq \sum_{\alpha=1,\dots,K} \frac{T_1(\mathcal{J}, \alpha)}{P_\alpha} + \left(1 - \frac{1}{P_{\max}}\right) (T_\infty(J_k) + r(J_k)) \end{aligned}$$

$$\leq \sum_{\alpha=1,\dots,K} \frac{T_1(\mathcal{J}, \alpha)}{P_\alpha} + \left(1 - \frac{1}{P_{\max}}\right) \max_{J_i \in \mathcal{J}} (T_\infty(J_i) + r(J_i)).$$

□

The following theorem gives the competitive ratio of K-RAD with respect to the makespan.

Theorem 7.3 K-RAD is $(K+1-1/P_{\max})$ -competitive with respect to the makespan for any set of jobs with arbitrary release times, where $P_{\max} = \max_{\alpha=1,\dots,K} P_\alpha$.

Proof. Recall that $I = [1, 2, \dots, T(\mathcal{J})]$ is the time interval in which K-RAD schedules the job set $\mathcal{J}$. We will consider two cases depending on whether I contains any idle intervals or not.

Case 1 : I does not contain idle intervals.

In this case, the makespan can be bounded by Inequality (7.4) from Lemma 7.2. Since $\sum_{\alpha=1,\dots,K} T_1(\mathcal{J}, \alpha)/P_\alpha \leq K \max_{\alpha=1,\dots,K} T_1(\mathcal{J}, \alpha)/P_\alpha$ and both $\max_{\alpha=1,\dots,K} T_1(\mathcal{J}, \alpha)/P_\alpha$ and $\max_{J_i \in \mathcal{J}} (T_\infty(J_i) + r(J_i))$ are lower bounds on makespan, we obtain

$$\begin{aligned} T(\mathcal{J}) &\leq K \max_{\alpha=1,\dots,K} \frac{T_1(\mathcal{J}, \alpha)}{P_\alpha} + \left(1 - \frac{1}{P_{\max}}\right) \max_{J_i \in \mathcal{J}} (T_\infty(J_i) + r(J_i)) \\ &\leq \left(K + 1 - \frac{1}{P_{\max}}\right) T^*(\mathcal{J}). \end{aligned} \quad (7.8)$$

Case 2 : I contains idle intervals.

If I contains idle intervals, let $l = [t_1, \dots, t_2]$ be the last such interval in I . Job set $\mathcal{J}$ can be divided into two disjoint subsets $\mathcal{J}_1$ and $\mathcal{J}_2$ such that $\mathcal{J}_1$ contains all the jobs completed before t_1 and $\mathcal{J}_2$ contains all the jobs released after t_2 . Since K-RAD completes $\mathcal{J}_1$ in $t_1 - 1$ time steps, the optimal scheduler $\mathcal{S}$ should complete $\mathcal{J}_1$ in no more than that amount of time. Therefore both K-RAD and $\mathcal{S}$ wait until time $t_2 + 1$ to start scheduling $\mathcal{J}_2$. Let $T^*(\mathcal{J}_2)$ denote the makespan of $\mathcal{J}_2$ scheduled by an optimal scheduler $\mathcal{S}$. Then, we have $T^*(\mathcal{J}) = t_2 + T^*(\mathcal{J}_2)$. From Case 1, we know that K-RAD completes $\mathcal{J}_2$ in $T(\mathcal{J}_2) \leq (K + 1 - 1/P_{\max})T^*(\mathcal{J}_2)$ time steps. Therefore, the makespan of the job set $\mathcal{J}$ scheduled by K-RAD is

$$\begin{aligned} T(\mathcal{J}) &= t_2 + T(\mathcal{J}_2) \\ &\leq t_2 + \left(K + 1 - \frac{1}{P_{\max}}\right) T^*(\mathcal{J}_2) \end{aligned}$$

$$\begin{aligned}
&\leq \left(K + 1 - \frac{1}{P_{max}} \right) (t_2 + T^*(\mathcal{J}_2)) \\
&= \left(K + 1 - \frac{1}{P_{max}} \right) T^*(\mathcal{J}).
\end{aligned} \tag{7.9}$$

Inequalities (7.8) and (7.9) indicate that under both cases, K-RAD achieves $(K + 1 - 1/P_{max})$ -competitiveness with respect to the makespan. $\square$

7.5 K-DAG Mean Response Time Lower Bounds

In this section, we identify two lower bounds on any set of batched jobs represented by K-DAGs. We start by defining a new term related to squashed work.

Definition 7.2 *The **squashed α -work area** of a job set $\mathcal{J}$ on P_α number of α -processors is*

$$sqw(\mathcal{J}, \alpha) = \frac{1}{P_\alpha} \text{sq-sum}(\langle T_1(J_i, \alpha) \rangle), \tag{7.10}$$

where $T_1(J_i, \alpha)$ is the α -work of job $J_i \in \mathcal{J}$.

Similar to homogeneous resources, we obtain the following two lower bounds for the mean response time on any set of batched jobs represented by K-DAGs:

$$\overline{R^*}(\mathcal{J}) \geq T_\infty(\mathcal{J}) / |\mathcal{J}|, \tag{7.11}$$

$$\overline{R^*}(\mathcal{J}) \geq \max_{\alpha=1, \dots, K} sqw(\mathcal{J}, \alpha) / |\mathcal{J}|. \tag{7.12}$$

Thus, the optimal total response time $R^*(\mathcal{J})$ has lower bounds of $T_\infty(\mathcal{J})$ and $\max_{\alpha=1, \dots, K} sqw(\mathcal{J}, \alpha)$.

7.6 K-RAD Mean Response Time Analysis

In this section, we analyze the mean response time of the K-RAD algorithm. Like RAD, K-RAD analysis can be divided into two cases. We show that under light workload, K-RAD has a competitive ratio of $2K + 1 - 2K/(|\mathcal{J}| + 1)$, while under heavy workload, K-RAD performs worse than optimal algorithm by at most a factor of 2.

We first define what we mean by light and heavy system workload. Recall that at any time t , $\mathcal{J}(\alpha, t)$ denotes the set of α -active jobs for $\alpha = 1, \dots, K$. We say that the system has **light load** when $|\mathcal{J}(\alpha, t)| \leq P_\alpha$ at *any* time t during the schedule for *all* $\alpha = 1, \dots, K$. In this case, K-RAD utilizes only the DEQ algorithm. On the

other hand, the system is considered to have **heavy load** when $|\mathcal{J}(\alpha, t)| > P_\alpha$ for some $\alpha = 1, \dots, K$ at some time t . In this case K-RAD utilizes both the DEQ and round-robin algorithms during the schedule. The two cases are presented in Sections 7.6.1 and 7.6.2, respectively.

7.6.1 Analysis of K-RAD under light load

The following theorem gives the competitive ratio for the mean response time produced by K-RAD scheduler under light system workload.

Theorem 7.4 *K-RAD is $(2K + 1 - 2K/(|\mathcal{J}| + 1))$ -competitive with respect to the mean response time for any batched job set $\mathcal{J}$, if at any time t , $|\mathcal{J}(\alpha, t)| \leq P_\alpha$ for each $\alpha = 1, \dots, K$, i.e., the number of jobs that require processors never exceeds the number of available processors.*

Proof. Suppose that K-RAD schedules a batched job set $\mathcal{J}$ on a machine with P_α number of α -processors for $\alpha = 1, \dots, K$. We will show that the total response time of $\mathcal{J}$ can be bounded by

$$R(\mathcal{J}) \leq \left(2 - \frac{2}{n+1}\right) \left(\sum_{\alpha=1, \dots, K} sqw(\mathcal{J}, \alpha)\right) + T_\infty(\mathcal{J}) , \quad (7.13)$$

where $n = |\mathcal{J}|$ is the total number of jobs in the set. Since $\sum_{\alpha=1, \dots, K} sqw(\mathcal{J}, \alpha) \leq K \max_{\alpha=1, \dots, K} sqw(\mathcal{J}, \alpha)$ and both $\max_{\alpha=1, \dots, K} sqw(\mathcal{J}, \alpha)$ and $T_\infty(\mathcal{J})$ are lower bounds for the total response time, Inequality (7.13) indicates that K-RAD is $(2K + 1 - 2K/(n+1))$ -competitive with respect to the total response time, or equivalently with respect to the mean response time under light workload.

As with the analysis in Theorem 3.6, we prove Inequality (7.13) by induction on the remaining execution time of the job set $\mathcal{J}(\vec{t})$.

Basis: $t = T(\mathcal{J}) + 1$. When $t = T(\mathcal{J}) + 1$, we have $\mathcal{J}(\vec{t}) = \emptyset$. It follows that $R(\mathcal{J}(\vec{t})) = 0$, $sqw(\mathcal{J}(\vec{t}), \alpha) = 0$ for $\alpha = 1, \dots, K$, and $T_\infty(\mathcal{J}(\vec{t})) = 0$. Thus, the claim holds trivially.

Induction: $1 \leq t \leq T(\mathcal{J})$. Suppose that Inequality (7.13) holds at time step $t+1$, i.e.,

$$R(\mathcal{J}(\vec{t+1})) \leq \left(2 - \frac{2}{n+1}\right) \left(\sum_{\alpha=1, \dots, K} sqw(\mathcal{J}(\vec{t+1}), \alpha)\right) + T_\infty(\mathcal{J}(\vec{t+1})) . \quad (7.14)$$

We will prove that it still holds at time step t , *i.e.*,

$$R(\mathcal{J}(\vec{t})) \leq \left(2 - \frac{2}{n+1}\right) \left(\sum_{\alpha=1, \dots, K} sqw(\mathcal{J}(\vec{t}), \alpha)\right) + T_{\infty}(\mathcal{J}(\vec{t})) . \quad (7.15)$$

The following notations denote the changes in, respectively, the total response time, the squashed α -work area, and the aggregate span from time step t and $t+1$:

$$\begin{aligned} \Delta rt &= R(\mathcal{J}(\vec{t})) - R(\mathcal{J}(\vec{t+1})) , \\ \Delta sqw(\alpha) &= sqw(\mathcal{J}(\vec{t}), \alpha) - sqw(\mathcal{J}(\vec{t+1}), \alpha) , \\ \Delta T_{\infty} &= T_{\infty}(\mathcal{J}(\vec{t})) - T_{\infty}(\mathcal{J}(\vec{t+1})) . \end{aligned}$$

Given the induction hypothesis (Inequality (7.14)), we need only prove that

$$\Delta rt \leq \left(2 - \frac{2}{n+1}\right) \left(\sum_{\alpha=1, \dots, K} \Delta sqw(\alpha)\right) + \Delta T_{\infty} \quad (7.16)$$

in order to prove our claim (Inequality (7.15)). We divide the proof of Inequality (7.16) into four steps.

(1) To bound Δrt : At any time t , the total number of uncompleted jobs is $|\mathcal{J}(\vec{t})| \leq n$. Since each uncompleted job in $\mathcal{J}(\vec{t})$ adds one time step to the total response time during step t , we have

$$\Delta rt \leq n . \quad (7.17)$$

(2) To bound ΔT_{∞} : At time step t , an uncompleted job J_i is either $\forall$ -satisfied or $\exists$ -deprived. The uncompleted jobs can be partitioned as $\mathcal{J}(\vec{t}) = \mathcal{JS}(t) \cup \mathcal{JD}(t)$, representing the set of $\forall$ -satisfied and $\exists$ -deprived jobs at time t , respectively. If $J_i \in \mathcal{JS}(t)$, the span of J_i must reduce by 1 at time step t , *i.e.*, $T_{\infty}(J_i(\vec{t})) = T_{\infty}(J_i(\vec{t+1})) + 1$. If $J_i \in \mathcal{JD}(t)$, the span of J_i never increases at any time step t , *i.e.*, $T_{\infty}(J_i(\vec{t})) \geq T_{\infty}(J_i(\vec{t+1}))$. Therefore, the aggregate span of $\mathcal{J}$ must reduce by at least $|\mathcal{JS}(t)|$ at time step t , and we have

$$\Delta T_{\infty} \geq |\mathcal{JS}(t)| . \quad (7.18)$$

(3) To bound $\Delta sqw(\alpha)$:

7.6. K-RAD Mean Response Time Analysis

135

Consider the α -work of $J_i(\vec{t})$ and $J_i(\overrightarrow{t+1})$. For a job J_i that is α -deprived at time step t , J_i has α -allotment $a(J_i, \alpha, t) = \bar{p}(\alpha, t)$, where $\bar{p}(\alpha, t)$ denotes the mean deprived allotment at time step t for α -processors. Thus, we have

$$T_1(J_i(\vec{t}), \alpha) = T_1(J_i(\overrightarrow{t+1}), \alpha) + \bar{p}(\alpha, t) . \quad (7.19)$$

For a job J_i that is α -satisfied at time step t , J_i 's α -allotment is equal to its α -desire, i.e., $a(J_i, \alpha, t) = d(J_i, \alpha, t)$. Thus, we have

$$T_1(J_i(\vec{t}), \alpha) = T_1(J_i(\overrightarrow{t+1}), \alpha) + d(J_i, \alpha, t) . \quad (7.20)$$

Let $\mathcal{JS}(\alpha, t)$ and $\mathcal{JD}(\alpha, t)$ denote the set of α -satisfied jobs and the set of α -deprived jobs at time step t respectively.

If there exist no α -deprived jobs at time t , i.e., $|\mathcal{JD}(\alpha, t)| = 0$, then for $\alpha = 1, \dots, K$, we have

$$\Delta sqw(\alpha) \geq 0. \quad (7.21)$$

If there exist α -deprived jobs at time t , i.e., $|\mathcal{JD}(\alpha, t)| > 0$, then *all* α -processors must be allotted to the jobs (otherwise, there would not be any jobs deprived of the α -processors). Recall that DEQ allots the deprived jobs no fewer processors than satisfied jobs. Thus, the squashed sum property proved in Lemma 3.2 bounds the change in the squashed α -work.

For $J_i \in \mathcal{J}$, let $a_i = T_1(J_i(\overrightarrow{t+1}), \alpha)$ and $b_i = T_1(J_i(\vec{t}), \alpha)$. By DEQ, all the α -processors are allotted, i.e., $\sum_{J_i \in \mathcal{J}} a(J_i, \alpha, t) = P_\alpha$. Therefore, according to Lemma 3.2, we have

$$\text{sq-sum}(\langle b_i \rangle) - \text{sq-sum}(\langle a_i \rangle) \geq P_\alpha (|\mathcal{JD}(\alpha, t)| + 1) / 2 . \quad (7.22)$$

From Inequality (7.22) and Definition 3.1, we get for $\alpha = 1, \dots, K$ and $|\mathcal{JD}(\alpha, t)| > 0$,

$$\begin{aligned} \Delta sqw(\alpha) &= \frac{\text{sq-sum}(\langle b_i \rangle) - \text{sq-sum}(\langle a_i \rangle)}{P_\alpha} \\ &\geq \frac{P_\alpha (|\mathcal{JD}(\alpha, t)| + 1)}{2P_\alpha} \\ &\geq \frac{|\mathcal{JD}(\alpha, t)| + 1}{2} . \end{aligned} \quad (7.23)$$

(4) **To derive Inequality (7.16):** The uncompleted jobs can be partitioned as $\mathcal{J}(\vec{t}) = \mathcal{JS}(t) \cup \mathcal{JD}(t)$, representing the disjoint set of $\forall$ -satisfied and $\exists$ -deprived jobs at time t , respectively. Therefore, we have

$$|\mathcal{JS}(t)| + |\mathcal{JD}(t)| = n. \quad (7.24)$$

Let $\mathcal{K} = \{1, 2, \dots, K\}$, let $\mathcal{K}' = \{\alpha \in \mathcal{K} : |\mathcal{JD}(\alpha, t)| > 0\}$, and let $\mathcal{K}'' = \{\alpha \in \mathcal{K} : |\mathcal{JD}(\alpha, t)| = 0\}$. From Equation (7.24), we have

$$\begin{aligned} \sum_{\alpha \in \mathcal{K}'} |\mathcal{JD}(\alpha, t)| &= \sum_{\alpha \in \mathcal{K}} |\mathcal{JD}(\alpha, t)| \\ &\geq \left| \bigcup_{\alpha \in \mathcal{K}} \mathcal{JD}(\alpha, t) \right| \\ &= n - |\mathcal{JS}(t)|. \end{aligned} \quad (7.25)$$

Hence, from Inequalities (7.18), (7.23), (7.21), and (7.25), we obtain

$$\begin{aligned} &\left(2 - \frac{2}{n+1}\right) \left(\sum_{\alpha \in \mathcal{K}} \Delta sqw(\alpha)\right) + \Delta T_\infty \\ &\geq \frac{2n}{n+1} \left(\sum_{\alpha \in \mathcal{K}'} \frac{|\mathcal{JD}(\alpha, t)| + 1}{2}\right) + |\mathcal{JS}(t)| \\ &\geq \frac{n}{n+1} (n - |\mathcal{JS}(t)| + |\mathcal{K}'|) + |\mathcal{JS}(t)| \end{aligned} \quad (7.26)$$

If $\mathcal{K}' \neq \emptyset$, which implies $|\mathcal{K}'| \geq 1$, we have

$$\begin{aligned} &\frac{n}{n+1} (n - |\mathcal{JS}(t)| + |\mathcal{K}'|) + |\mathcal{JS}(t)| \\ &\geq n - |\mathcal{JS}(t)| \frac{n}{n+1} + |\mathcal{JS}(t)| \\ &\geq n \end{aligned} \quad (7.27)$$

If $\mathcal{K}' = \emptyset$, then we have $|\mathcal{JS}(t)| = n$, and Inequality (7.27) holds trivially. Inequalities (7.17), (7.26), and (7.27) imply that Inequality (7.16) holds, and the proof is complete. $\square$

In the case where $K = 1$ for homogeneous resource scheduling, Theorem 7.4 indicates that DEQ is $(3 - 2/(|\mathcal{J}| + 1))$ -competitive with respect to the mean response time. This result tightens the performance bound for the DEQ algorithm analyzed

in [45, 46]. It is also consistent with the mean response time bound of RAD in Theorem 3.3.

7.6.2 Analysis of K-RAD under heavy load

Under heavy system workload, *i.e.*, there exists some time t and some $\alpha = 1, \dots, K$ for which $|\mathcal{J}(\alpha, t)| > P_\alpha$, K-RAD utilizes both the DEQ and round-robin algorithms. The following theorem gives the competitive ratio for the mean response time produced by K-RAD scheduler in this more general case.

Theorem 7.5 *K-RAD is $(4K + 1 - 4K/(|\mathcal{J}| + 1))$ -competitive with respect to the mean response time for any batched job set $\mathcal{J}$.*

Proof. Similar to the proof of Theorem 7.4, we show that to schedule a batch of n jobs, K-RAD achieves total response time

$$R(\mathcal{J}) \leq \left(4 - \frac{4}{n+1}\right) \left(\sum_{\alpha=1, \dots, K} sqw(\mathcal{J}, \alpha)\right) + T_\infty(\mathcal{J}). \quad (7.28)$$

Then, the main theorem follows directly. Also, to prove it by induction, we need only show that

$$\Delta rt \leq \left(4 - \frac{4}{n+1}\right) \left(\sum_{\alpha=1, \dots, K} \Delta sqw(\alpha)\right) + \Delta T_\infty \quad (7.29)$$

holds on each time step t . Since the change of the total response time Δrt and the aggregate span ΔT_∞ are still given by Inequalities (7.17) and (7.18), the rest of the proof focuses on calculating the change in the squashed α -work area $\Delta sqw(\alpha)$. By examining Inequalities (7.16), (7.29), and (7.23), it suffices to show that when scheduled by round-robin,

$$\Delta sqw(\alpha) \geq \frac{|\mathcal{JD}(\alpha, t)| + 1}{4} \quad (7.30)$$

for $\alpha = 1, \dots, K$ and $|\mathcal{JD}(\alpha, t)| > 0$. Recall from Section 7.2 that an RR cycle consists of more than one time steps. Specifically, during an RR cycle, all the jobs that are α -active are scheduled at least once. Let l denote the total number of jobs scheduled in the RR cycle, and let τ denote the total number of time steps in the RR cycle. We have

$$\tau = \lceil l/P_\alpha \rceil$$

$$\leq l/P_\alpha + 1 .$$

Since each scheduled job has reduced its α -work by at least 1, according to Lemma 3.2 and Definition 7.2, the total change of the squashed α -work area $\Delta \text{RR}(\alpha)$ during the cycle is given by

$$\Delta \text{RR}(\alpha) \geq \frac{l(l+1)}{2P_\alpha} . \quad (7.31)$$

Thus, given $P_\alpha < l$, the average change of the squashed α -work area $\Delta sqw(\alpha)$ during each time step of the cycle is

$$\begin{aligned} \Delta sqw(\alpha) &= \frac{\Delta \text{RR}(\alpha)}{\tau} \\ &\geq \frac{l(l+1)/2P_\alpha}{(l+P_\alpha)/P_\alpha} \\ &\geq \frac{l+1}{4} . \end{aligned} \quad (7.32)$$

At any time step t in the cycle, the number $|\mathcal{JD}(\alpha, t)|$ of α -deprived jobs is at most the total number l of α -active jobs in the cycle, *i.e.*, $|\mathcal{JD}(\alpha, t)| \leq l$. Inequality (7.32) then directly implies Inequality (7.30), and the proof is complete. $\square$

The results of Theorem 7.4 and Theorem 7.5 suggest that K-RAD doubles the competitive ratio under heavy workload by using round-robin. One intuitive explanation is that, when round-robin is used under heavy workload, if only a small number of jobs are active with limited desires, then almost all the processor cycles in the last time step of a RR cycle can be wasted, which causes the bound to double in the worst case.

Chapter 8

Literature Review

Job scheduling on a parallel computer means different things to different people. The surveys by various people in [57, 87, 94, 114, 121] have cited hundreds of papers. The wide variety of parallel computer architectures, parallel operating systems, and parallel programming languages means that there is no single ideal scheduling strategy. Also, it is hard to compare actual commercial schedulers that handle real workloads with theoretical scheduling models in academic studies. Thus, this literature review is not intended to cover the prior work exhaustively. I just hope to offer a glimpse of the state of the art and provide a reasonable perspective of the field of study.

The literature review is organized into four parts, with their contents outlined as follows.

- **General scheduling techniques:** I describe four commonly used techniques for scheduling and discuss their benefits and shortcomings.
- **Scheduling serial jobs for makespan and mean response time:** I summarize the results about the makespan and mean response time for nonclairvoyant scheduling of serial jobs on multiprocessors.
- **Thread scheduling for parallel jobs:** I classify and compare the thread scheduling algorithms.
- **Adaptive job scheduling:** I review and compare adaptive scheduling algorithms which can change jobs' allotments during their executions.

8.1 Common Techniques for Scheduling

Four commonly used techniques [58] for scheduling parallel jobs are global queueing, gang scheduling, variable partitioning, and dynamic partitioning. In this section, I

briefly describe these techniques and summarize their benefits and shortcomings.

Global queueing is a simple way to implement a scheduler on a multiprocessor. All ready threads of all jobs are put in a queue. Processors pick up threads at the front of the queue, execute them for a period of time, and return them to the queue. This approach is commonly used in small-scale, bus-based shared memory processor systems such as SGI multiprocessor workstations [11]. The most important benefit of a global queue is that it facilitates full utilization of processors and offers automatic load balancing. A processor is idle only when the global queue is empty. Such a central queue may become a single point of contention, however, especially when the number of processors becomes large. Hence, this approach is not scalable. Moreover, a processor is likely to execute threads from different jobs over scheduling quanta. As a result, jobs cannot stash local data, and the contents in local cache are wiped out each time jobs are rescheduled. The uncoordinated thread scheduling from such a simple time-sharing approach may reduce scheduling efficiency significantly, especially when there are intensive communications and dependencies among the threads of a job.

Gang scheduling schedules related threads of a job to run simultaneously on different processors. More precisely, it combines three features. First, threads are grouped into gangs. Second, the threads on each gang are mapped into different processors. Third, time slicing is used so that all threads in a gang can be preempted and rescheduled again. Thus, it combines coordinated context switching with both space-sharing and time-sharing strategies to support execution of parallel applications. Gang scheduling has been implemented on multiprocessors such as the CM5 from Thinking Machines [96] and the Intel Paragon [48]. It has two important merits. First, via time slicing, it usually provides fast interactive response times. Second, through coordinated context switching, all the threads in the job execute at the same time, allowing them to interact at a fine granularity. A shortcoming of gang scheduling is that the overall system performance is not always optimal. There is some interference in the cache and overhead incurred during the context switching. In addition, there may be unacceptable processor fragmentation [59].

Variable partitioning divides the processors into disjoint sets according to users' request when jobs are submitted, and it executes each job in a distinct partition. A large partition can be divided into smaller ones to allow small jobs to execute in parallel; partitions are merged when the jobs terminate. Unlike global queueing and gang scheduling, variable partitioning allocates processors to jobs by

space-sharing. Variable partitioning is popular on distributed-memory machines such as Intel hypercubes and IBM SP2. Its advantage is that a job can get a dedicated partition according to its requirement. Its disadvantage comes from the possible mismatch between the available processors and user requests, resulting in fragmentation and long queuing times. Therefore, variable partitioning is more suitable for batched processing as opposite to an interactive setting.

Dynamic partitioning, like variable partitioning, also employs space sharing. The partition size can change during the execution of jobs however to reflect the changes of jobs' parallelism. Dynamic partitioning is usually implemented in a two-level framework [57] including a kernel-level OS allocator and some application-level thread schedulers. The advantages of dynamic partitioning include:

1. Schedulers can improve system performance by shifting processors from jobs that do not require many processors to the jobs in need.
2. Compared with a thread scheduler that comes with an OS, the thread scheduler embedded within the application incurs a much smaller overhead, and it allows application specific optimizations.
3. Since the programming models using dynamic partitioning usually express parallelism by using threads instead of processes, there is no waste of CPU cycles on busy waiting for process synchronization, as thread blocking incurs little overhead.
4. It does not suffer from resource fragmentation.

Dynamic partitioning does impose restrictions, however. Firstly, it requires the applications to be written in languages that allow the change of processor allotment during their execution. Secondly, it requires close cooperation between the thread schedulers and the OS allocator. For example, a processor working on a critical section of a job cannot be preempted, since it may result in a deadlock. Thirdly, the processor reallocation in dynamic partitioning may introduce scheduling overhead.

In this thesis, I chose to use dynamic partitioning to make better use of the flexibility of malleable jobs and maximize system efficiency. Regarding the restrictions of dynamic partitioning, firstly, many parallel languages now support dynamic creation and termination of threads or processes. Examples include PVM, MPI-2, and Cilk [20, 67]. Secondly, I believe that the proper coordination between thread and OS allocators can be achieved by careful design and implementation. Thirdly, the processor reallocation overhead can be controlled by adjusting the length of the scheduling quantum to be large enough. My scheduling algorithms such as G-RAD

and W-RAD ensure good performance even when the scheduling quantum length L is large.

8.2 Scheduling Serial Jobs for Makespan and Mean Response Time

This section reviews online algorithms and lower bounds for the scheduling of serial jobs on multiprocessors to minimize makespan or mean response time.

The first proof of competitiveness of an online algorithm for a scheduling problem was given by Graham [66] in 1969. Graham studied a simple deterministic greedy algorithm, now commonly called List Scheduling, to minimize makespan. The studied model is a basic one, where there is a sequence of sequential jobs characterized by their running times. It assumes that there are precedence constraints among jobs, and all jobs arrive at time 0. The list algorithm places the jobs in a list in arbitrary order. Whenever a machine becomes idle, it executes the first ready job on the list. The list algorithm is not only online, it is also nonclairvoyant. Graham shows that this simple algorithm is $(2 - 1/P)$ -competitive with respect to the makespan.

For makespan, an algorithm that can efficiently schedule batched jobs can lead to an efficient algorithm with arbitrary job release times. Shmoys, Wein, and Williamson [124] show that there exists a 2δ -competitive algorithm that allows arbitrary release times if given a δ -competitive algorithm for batched jobs. Feldmann, Sgall, and Teng [60] show that for a certain class of batch-scheduling algorithms, the competitive ratio is increased by an additive factor of 1 rather than by a multiplicative factor of 2. This class of algorithms includes all algorithms that use a greedy approach similar to List Scheduling.

Shmoys, Wein, and Williamson [124] study the lower bounds of online nonclairvoyant scheduling of serial jobs with respect to makespan. They show that the competitive ratio is at least $2 - 1/P$ for any preemptive deterministic online algorithm, and at least $2 - 1/\sqrt{P}$ for any nonpreemptive randomized online algorithm with an oblivious adversary.

I now review some prior work on the scheduling of serial jobs on multiprocessors to minimize mean response time.

For both sequential machine and parallel machine scheduling, the Shortest Remaining Processing Time First (SRPT) algorithm offers, within constant factors, the best possible competitive ratio of any online algorithm for scheduling serial jobs to

minimize mean response time [98]. On a sequential machine, SRPT is optimal. On parallel machines, SRPT is $O(\min(\log |\mathcal{J}|/P, \log \gamma))$ -competitive, where γ is the ratio between the processing time of the longest and shortest jobs. This bound is optimal to within constant factors [98]. Later work [7, 8, 13, 36] improves the basic SRPT by reducing the number of preemption or job migrations. Note that, even though SRPT is an online algorithm, it requires clairvoyance since it needs the remaining processing time of jobs to make scheduling decision.

For nonclairvoyant scheduling of a job set $\mathcal{J}$ with serial jobs, Motwani, Phillips, and Torng [108] show that no deterministic algorithm can achieve a competitive ratio better than $\Omega(|\mathcal{J}|^{1/3})$, and no randomized algorithm can achieve a competitiveness ratio better than $\Omega(\log |\mathcal{J}|)$ for the mean response time. Becchetti and Leonardi [12] present a version of the randomized multilevel feedback algorithm (RMLF) and prove an $O(\log |\mathcal{J}| \log(|\mathcal{J}|/P))$ -competitiveness result against any oblivious adversary on a machine with P processors. Their algorithm simulates the behavior of SRPT without requiring prior information of jobs' remaining processing times.

Instead of scheduling jobs with arbitrary release times, some prior work focuses on scheduling batched job sets. Motwani, Phillips, and Torng [108] show that any deterministic algorithm is $\Omega(2 - 2/(|\mathcal{J}| + 1))$ -competitive, and any randomized algorithm is $\Omega(2 - 4/(|\mathcal{J}| + 3))$ -competitive for the scheduling of batched serial jobs on multiprocessors. Consider a round-robin scheduler, which at all times ensures that each uncompleted job receives an equal amount of processing time to within 1 quantum. Intuitively, in batched nonclairvoyant scheduling where the processing time of a job is unknown beforehand, the best we can do is to give each job an equal amount of processing time, and the shorter jobs complete earlier than the longer jobs. Indeed, round-robin is $(2 - 2/(|\mathcal{J}| + 1))$ -competitive, and a simple randomized algorithm extended from round-robin is $(2 - 4/(|\mathcal{J}| + 3))$ -competitive. Both algorithms achieve the best possible competitiveness for nonclairvoyant deterministic and randomized algorithms, respectively.

8.3 Thread Scheduling for Parallel Jobs

A thread scheduler is designed to schedule the ready tasks of a parallel job on its allotted processors. A thread scheduler is adaptive if the job's allotment can change during its execution, or nonadaptive, if the job's allotment is fixed. This section discusses both nonadaptive and adaptive thread scheduling algorithms.

Prior work on scheduling a single parallel job tends to focus on nonadaptive

scheduling [18, 23, 31, 66, 111] or adaptive scheduling without parallelism feedback [5].

The most intuitive way to map ready threads of a job to its allotted processors is to schedule them in a greedy manner. At any time, if there are at least P ready threads, a greedy scheduler maps any P of them onto the processors; otherwise, it schedules all of them. Graham’s list algorithm [66], mentioned in Section 8.2, can also be interpreted as a greedy thread scheduler for scheduling individual parallel job on a fixed number of processors. Specifically, we can treat those serial jobs with dependencies as a single parallel job that has threads with dependencies. The algorithm completes the job J in at most $T_1(J)/P + (1 - 1/P)T_\infty(J)$ time, which is $(2 - 1/P)$ -competitive. The authors of [15, 18, 111] presented several nonadaptive thread schedulers for data-parallel programs. Their schedulers use central control to execute the ready threads and schedule them greedily over the allotted processors by prioritizing threads according to the depth-first execution order to minimize the memory space. These greedy thread schedulers are centralized, where the scheduler is aware of all ready threads at any moment to make a scheduling decision.

In decentralized thread scheduling, work-stealing has been used as a heuristic since Burton and Sleep’s research [32] and Halstead’s implementation of Multilisp [71]. Many variants [61, 69, 107] have been implemented since then, and work-stealing has been analyzed in the context of load balancing [116], backtrack search [89], etc. Blumofe and Leiserson [23] prove that the work-stealing algorithm is efficient with respect to time, space, and communication for the class of “fully strict” multithreaded computations. Arora, Blumofe and Plaxton [5] extend the time bound result to arbitrary multithreaded computations. In addition, Acar, Blelloch and Blumofe [1] show that work-stealing schedulers are efficient with respect to cache misses for jobs with “nested parallelism.” Variants of work-stealing algorithms have been implemented in real systems [20, 27, 63], and empirical studies show that work-stealing schedulers are scalable and practical [26, 63].

Adaptive thread scheduling without parallelism feedback has been studied in the context of multithreading [5, 25, 26, 28]. In this work, the thread scheduler uses a randomized work-stealing strategy to schedule threads on available processors, but it does not provide the feedback about the job’s parallelism to the OS allocator. The work in [25, 28] considers networks of workstations where processors may fail or join and leave a computation while the job is running, showing that work-stealing provides a good foundation for adaptive task scheduling. Arora, Blumofe, and Plaxton [5] show that the ABP task scheduler completes a job in $O(T_1/\bar{P} + PT_\infty/\bar{P})$ expected time.

Blumofe and Papadopoulos [26] perform an empirical evaluation of ABP and show that on an 8-processor machine, ABP provides almost perfect linear speedup for jobs with reasonable parallelism. In all these experiments, the average job parallelism T_1/T_∞ is much greater than 8.

Adaptive task scheduling with parallelism feedback has been studied empirically in [119, 126, 129]. These researchers use a job's history of processor utilization to provide feedback to dynamic-equipartitioning OS allocators. Their studies use different strategies for parallelism feedback, and all have reported better system performance with parallelism feedback than without, but it is not obvious which of their strategies is best.

As discussed in Chapter 1, job completion time and waste are two important metrics for adaptive thread scheduling. Achieving both short completion time and small waste implies that a thread scheduler not only performs well in terms of scheduling the job assigned, but the system can also repurpose the unneeded processors to other needy jobs for better utilization. To the best of my knowledge, A-GREEDY and A-STEAL are the first adaptive thread schedulers designed to minimize both completion time and waste.

8.4 Adaptive Job Scheduling

Adaptive job scheduling has been studied both empirically [99, 103, 130, 134] and theoretically [9, 45, 51, 53, 68, 108]. This section reviews some of the theoretical results. For nonclairvoyant scheduling of parallel jobs, Edmonds [51] shows that any algorithm is no better than $\Omega(\sqrt{|\mathcal{J}|})$ -competitive. Therefore, for the scheduling of general parallel jobs, what a nonclairvoyant algorithm can do to minimize mean response time is limited. Thus, most of the results for mean response time focus on the batched case.

Because parallel job scheduling is closely related to the corresponding job model, I first describe four common ways to model parallel jobs.

- The most simple one is to assume that a job is **fully parallel**, which indicates that the job can always achieve perfect linear speedup with any allotment.
- Turek *et al.* [131] model speedup as a function, which specifies the rate at which work is completed given the number of processors allotted to it.
- Edmonds *et al.* [51, 53] present multi-phase parallel jobs where each phase can have different speedup function.

- Many results in the literature [15, 18, 19, 22, 23, 44, 54, 80, 111] model a job as a DAG. The DAG model explicitly represents the dependency among its threads, and precisely captures both the fine-grained and coarse-grained logic of the underlying application.

Turek *et al.* [131] consider parallel jobs with a single phase of an arbitrary nondecreasing and sublinear speedup function. Without using preemptions, they achieve a competitive ratio of 2 in the batched setting. The algorithm requires complete knowledge of the jobs' workload and speedup functions. Moreover, it incurs the excessive computation time of $O(|\mathcal{J}|(|\mathcal{J}|^2 + P))$ to calculate the allotment of jobs.

McCann, Vaswani, and Zahorjan [103] introduce the notion of dynamic equipartitioning (DEQ), which gives each job a fair allotment of processors based on the job's request, while allowing processors that cannot be used by a job to be reallocated to other jobs. Brecht, Deng, and Gu [30] consider multiphase parallel jobs, and prove that DEQ with instantaneous parallelism as feedback is 2-competitive with respect to the makespan. In this result, they do not let the scheduler know the amount of work per job, but they still assume that the scheduler knows the speedup function given by the instantaneous parallelism feedback. Under the same assumptions, Deng *et al.* [46] prove that DEQ with instantaneous parallelism is also 4-competitive for batched jobs with respect to the mean response time. One problem with using DEQ as the OS allocator is that it only applies to the case where the total number of jobs in the job set is less than or equal to the total number of processors.

Edmonds, Chinn, Brecht, and Deng [53] show that the equipartitioning achieves a competitive ratio $2 + \sqrt{3}$ for mean response time in batched settings for a class of jobs in which each phase has a nondecreasing sublinear speedup function. When the number of jobs is more than the number of processors, they assume that each job gets a fractional number of processors as allotment. With the same model, they also provide a lower bound of $e \approx 2.71$ for any nonclairvoyant scheduler.

Another line of research [10, 51, 86, 104, 113] uses **resource augmentation**, which gives nonclairvoyant scheduler more resources than the adversary in order to compensate for the nonclairvoyance. The recent popularity of resource augmentation analysis of scheduling problems emanates from a paper [86] by Kalyanasundaram and Pruhs, and the term "resource augmentation" was introduced by Phillips *et al.* [113]. In this model, they augment the online algorithm with extra resources in the form of faster processors or extra processors. An online algorithm A is an **s -speed c -competitive** algorithm in $f(A_s, I) \leq c \cdot f(OPT_1, I)$ for all input instances I . Pruhs, Sgall, and

8.4. Adaptive Job Scheduling

147

Torong [114] give a good review on this topic in their survey paper.

In summary, the prior work on nonclairvoyant adaptive scheduling has achieved various degrees of success, but it leaves large room for improvements. Whereas many prior results provide heuristics without offering any performance assurances, others make unrealistic assumptions in order to achieve analytical results. Moreover, they tend to improve efficiency without taking into account fairness and scheduling overhead. I have been motivated to study nonclairvoyant adaptive scheduling because the assumption of nonclairvoyance matches the practical requirements of parallel job scheduling, and adaptivity offers the best promise for efficiency in parallel scheduling. My scheduling algorithms G-RAD and W-RAD provide provable efficiency and fairness over a wide range of work loads with small and manageable overhead under realistic assumptions.

Chapter 9

Discussions and Conclusion

This thesis presented several provably efficient two-level adaptive scheduling algorithms with parallelism feedback, for both centralized and distributed environment, for both homogeneous and heterogeneous systems. It also presented generic techniques to design and analyze adaptive schedulers. This concluding chapter summarizes the main results of this thesis, and discusses possible future work.

9.1 Summary

In this thesis, I have explored adaptive scheduling algorithms and techniques on multiprocessors. The main results of the thesis are summarized as follows.

- I introduced two adaptive thread schedulers — A-GREEDY and A-STEAL— which provide provably good history-based feedback about the job’s parallelism without knowing the future of the job. These thread schedulers complete jobs in near-optimal time while guaranteeing low waste.
- I introduced the online nonclairvoyant OS allocator RAD. When using instantaneous parallelism as feedback, RAD achieves 3-competitiveness with mean response time for batched parallel jobs. It offers the best competitive ratio of all comparable OS allocator.
- I presented two online nonclairvoyant adaptive scheduling systems — G-RAD and W-RAD— each of which integrates an OS allocator and a thread scheduler. Both of them achieve $O(1)$ -competitiveness with respect to makespan and mean response time for nonbatched jobs and batched jobs, respectively. They are the first nonclairvoyant schedulers to ensure provable efficiency, fairness, and minimal overhead.

- The simulation results suggest that my schedulers can perform well in practice. For individual parallel jobs, A-STEAL provides almost perfect linear speedup across a variety of processor availability profiles for jobs with sufficient parallelism. For arbitrary job sets, the simulations indicate that the makespan scheduled by G-RAD is not more than 1.39 times the optimal on average, and it never exceeded 4.5 times for any simulation. For batched job sets, the mean response time scheduled by G-RAD was never more than 2.37 times the optimal on average, and it never exceeded 5.5 times.
- I presented some principles to design makespan-efficient two-level scheduling algorithms. I discussed general techniques to analyze the mean response time of adaptive schedulers.
- I introduced a novel analytical model and the K-RAD adaptive scheduler for functional heterogeneous resource systems. I proved K-RAD to be the best deterministic online nonclairvoyant algorithm with respect to makespan. In a batched setting, K-RAD is $(4K + 1 - 4K/(|\mathcal{J}| + 1))$ -competitive for mean response time, where K denotes the number of categories of the heterogeneous resources.

9.2 Future Work

I summarize some future work and open questions related to parallel job scheduling and adaptive scheduling algorithms as follows.

Stabilization. Although the desire estimation algorithm of A-GREEDY and A-STEAL ensures small waste and short completion time, the desire of a job may oscillate even when the parallelism of the job remains unchanged. The oscillation in desire may result in scheduling overhead due to redundant processor reallocation and waste due to excessive processor allocation. In A-GREEDY or A-STEAL, the range of the oscillation can be controlled by adjusting the value of the responsiveness parameter ρ . A smaller ρ gives smaller oscillation, but it also results in slower response if the job's parallelism changes.

Future work could look toward developing schedulers with a smooth transition between adaptation and stabilization. One way to approach this line of research is to treat the current desire as a continuous function with some previous states as input. By borrowing ideas from adaptive control, stabilization might be achieved without compromising much on the speed of adaptation.

Mean response time of k -batched job set. In a batched setting, nonclairvoyant algorithms can achieve 2-competitiveness for serial job scheduling [108] and 3-competitiveness for adaptive parallel job scheduling, as shown in Chapter 3. With arbitrary release times, however, a deterministic algorithm can only be $\Omega(n^{1/3})$ -competitive, and a randomized algorithm is at best $\Omega(\log n)$ -competitive [108], where n denotes the number of jobs in the job set. What can be achieved for mean response time if the jobs are released in two batches? What if jobs are released in k batches? Will the competitiveness be tied to the number of jobs n or to the number of batches k ? Our recent work indicates that k is the determining factor, not n . Thus, an interesting line of research is to design nonclairvoyant algorithms that are $O(\log k)$ -competitive with respect to mean response time for both serial and parallel job scheduling.

Scheduling on heterogeneous resources. Although Chapter 7 presents a simple algorithm for efficient scheduling on resources with functional heterogeneity, much work remains to be done. When instantaneous parallelism cannot be applied, it is important to study other desire estimation strategies such as using history-based feedback on heterogeneous resources. Moreover, adaptive scheduling on I/O devices may be addressed separately from the computational resources because of their fundamental differences. In addition to functional heterogeneity, different resources in the same category may have different speeds. A more general framework for heterogeneous resources scheduling would incorporate of functional and performance heterogeneity.

Adaptive scheduling. The most important part of an adaptive scheduling algorithm is its request-allotment protocol. My thesis has introduced one way of doing history-based feedback. It is certainly interesting to explore others. For example, instead of using only the feedback from the immediate previous quantum, information about the preceding m quanta may be able to provide a better estimation. The feedback-driven request-allotment protocol may be applied to many other application domains such as bandwidth allocation of communication channels, vehicle allocation in container ports, resource management in grid system, adaptive scheduling on multiprocessor telephones, etc.

With my current scheme, the parameters such as ρ and δ need to be tuned manually. Can these parameters be self-configured and adapted to the workload and other features of the system environment to maximize the performance of adaptive systems?

9.3 Concluding Remarks

In summary, this thesis presents algorithmic foundations for the design and analysis of adaptive scheduling algorithms. The adaptive scheduling framework and request allotment protocol can potentially improve the efficiency and utilization of multiprocessor systems. I hope that, in the near future, these scheduling algorithms can be effectively implemented and deployed in multiprocessors and multicore chips.

List of Publications

Conference Papers

1. Yuxiong He, Hongyang Sun, and Wen Jing Hsu. *Adaptive Scheduling of Parallel Jobs on Functionally Heterogeneous Resources*. Proceedings of the International Conference on Parallel Processing (ICPP), 2007.
2. Yuxiong He, Wen Jing Hsu, and Charles E. Leiserson. *Provably Efficient On-line Non-clairvoyant Scheduling*. Proceedings of the 21st IEEE International Parallel and Distributed Processing Symposium (IPDPS), 2007.
3. Kunal Agrawal, Yuxiong He, and Charles E. Leiserson. *Adaptive Work Stealing with Parallelism Feedback*. Proceedings of in the Annual ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP), 2007.
4. Kunal Agrawal, Yuxiong He, Wen Jing Hsu, and Charles E. Leiserson. *Adaptive Scheduling with Parallelism Feedback*. Proceedings of the NSF Next Generation Software Workshop in Conjunction with IPDPS, 2007.
5. Yuxiong He, Wen Jing Hsu and Charles E. Leiserson. *Provably Efficient Two-level Adaptive Scheduling*. Proceedings of the 12th Workshop on Job Scheduling Strategies for Parallel Processing (JSSPP), 2006.
6. Kunal Agrawal, Yuxiong He, and Charles E. Leiserson. *An Empirical Evaluation of Work Stealing with Parallelism Feedback*. Proceedings of International Conference on Distributed Computing Systems (ICDCS), 2006.
7. Kunal Agrawal, Yuxiong He, Wen Jing Hsu and Charles E. Leiserson. *Adaptive Task Scheduling with Parallelism Feedback*. Proceedings of Annual ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP), 2006.

8. Yuxiong He, Junqing Wang and Wen Jing Hsu. *Provably Efficient Parallel Detection of Nondeterministic Races in Parallel Programs*. Proceedings of the 16th IASTED International Conference on Parallel and Distributed Computing and Systems (PDCS), **Best Paper Award**, Nov, 2004.

Journal Papers Submitted

1. Yuxiong He, Hongyang Sun, and Wen Jing Hsu. *Nonclairvoyant Scheduling of Batched Parallel Jobs to Minimize Mean Response Time*. Submitted.
2. Yuxiong He, Wen Jing Hsu and Charles E. Leiserson. *Provably Efficient On-line Nonclairvoyant Adaptive Scheduling*. Accepted by IEEE Transactions on Parallel and Distributed Systems.
3. Kunal Agrawal, Yuxiong He, Wen Jing Hsu, and Charles E. Leiserson. *Adaptive Scheduling with Parallelism Feedback*. Accepted by ACM Transactions on Computer Systems.

Bibliography

- [1] Umut A. Acar, Guy E. Blelloch, and Robert D. Blumofe. The data locality of work stealing. In *Proceedings of the ACM Symposium on Parallel Algorithms and Architectures*, pages 1–12, New York, NY, USA, 2000.
- [2] Kunal Agrawal, Yuxiong He, Wen Jing Hsu, and Charles E. Leiserson. Adaptive scheduling with parallelism feedback. In *Proceedings of the ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, pages 100–109, New York, NY, USA, 2006.
- [3] Kunal Agrawal, Yuxiong He, and Charles E. Leiserson. An empirical evaluation of work stealing with parallelism feedback. In *Proceedings of the International Conference on Distributed Computing Systems*, pages 19 – 29, Lisboa, Portugal, 2006.
- [4] Kunal Agrawal, Yuxiong He, and Charles E. Leiserson. Work stealing with parallelism feedback. In *Proceedings of the ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, San Jose, CA, USA, 2007.
- [5] Nimar S. Arora, Robert. D. Blumofe, and C. Greg Plaxton. Thread scheduling for multiprogrammed multiprocessors. In *Proceedings of the ACM Symposium on Parallel Algorithms and Architectures*, pages 119–129, Puerto Vallarta, Mexico, 1998.
- [6] James Aspnes, Maurice Herlihy, and Nir Shavit. Counting networks. *Journal of the ACM*, 41(5):1020–1048, 1994.
- [7] Nir Avrahami and Yossi Azar. Minimizing total flow time and total completion time with immediate dispatching. In *Proceedings of the ACM Symposium on Parallel Algorithms and Architectures*, pages 11–18, San Diego, California, USA, 2003.

- [8] Baruch Awerbuch, Yossi Azar, Stefano Leonardi, and Oded Regev. Minimizing the flow time without migration. *SIAM Journal on Computing*, 31:1370–1382, 2001.
- [9] Nikhil Bansal, Kedar Dhamdhere, Jochen Konemann, and Amitabh Sinha. Non-clairvoyant scheduling for minimizing mean slowdown. *Algorithmica*, 40(4):305–318, 2004.
- [10] Nikhil Bansal and Kirk Pruhs. Server scheduling in the l_p norm: a rising tide lifts all boats. In *Proceedings of the ACM Symposium on the Theory of Computing*, pages 242–250, San Diego, CA, USA, 2003.
- [11] James M. Barton and Nawaf Bitar. A scalable multi-discipline, multiple-processor scheduling framework for IRIX. In *Job Scheduling Strategies for Parallel Processing*, Santa Barbara, CA, USA, 1995.
- [12] Luca Becchetti and Stefano Leonardi. Nonclairvoyant scheduling to minimize the total flow time on single and parallel machines. *Journal of the ACM*, 51(4):517–539, 2004.
- [13] Luca Becchetti, Stefano Leonardi, and S. Muthukrishnan. Scheduling to minimize average stretch without migration. In *Proceedings of the ACM-SIAM Symposium on Discrete Algorithms*, pages 548–557, San Francisco, CA, USA, 2000.
- [14] Michael A. Bender and Michael O. Rabin. Scheduling Cilk multithreaded computations on processors of different speeds. In *Proceedings of the ACM Symposium on Parallel Algorithms and Architectures*, pages 13–21, Bar Harbor, Maine, USA, July 2000.
- [15] Guy Blelloch, Phil Gibbons, and Yossi Matias. Provably efficient scheduling for languages with fine-grained parallelism. *Journal of the ACM*, 46(2):281–321, 1999.
- [16] Guy E. Blelloch, Siddhartha Chatterjee, Jonathan C. Hardwick, Jay Sipelstein, and Marco Zagha. Implementation of a portable nested data-parallel language. *Journal of Parallel and Distributed Computing*, 21(1):4–14, 1994.

- [17] Guy E. Blelloch, Phillip B. Gibbons, and Yossi Matias. Provably efficient scheduling for languages with fine-grained parallelism. In *Proceedings of the ACM Symposium on Parallel Algorithms and Architectures*, pages 1–12, Santa Barbara, California, 1995.
- [18] Guy E. Blelloch and John Greiner. A provable time and space efficient implementation of NESL. In *Proceedings of the ACM SIGPLAN International Conference on Functional Programming*, pages 213–225, Philadelphia, Pennsylvania, 1996.
- [19] Robert D. Blumofe. *Executing Multithreaded Programs Efficiently*. PhD thesis, Massachusetts Institute of Technology, Cambridge, MA, USA, 1995.
- [20] Robert D. Blumofe, Christopher F. Joerg, Bradley C. Kuszmaul, Charles E. Leiserson, Keith H. Randall, and Yuli Zhou. Cilk: An efficient multithreaded runtime system. In *Proceedings of the ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, pages 207–216, Santa Barbara, California, July 1995.
- [21] Robert D. Blumofe, Christopher F. Joerg, Bradley C. Kuszmaul, Charles E. Leiserson, Keith H. Randall, and Yuli Zhou. Cilk: An efficient multithreaded runtime system. *Journal of Parallel and Distributed Computing*, 37(1):55–69, 1996.
- [22] Robert D. Blumofe and Charles E. Leiserson. Space-efficient scheduling of multithreaded computations. *SIAM Journal on Computing*, 27(1):202–229, 1998.
- [23] Robert D. Blumofe and Charles E. Leiserson. Scheduling multithreaded computations by work stealing. *Journal of the ACM*, 46(5):720–748, 1999.
- [24] Robert D. Blumofe, Charles E. Leiserson, and Bin Song. Automatic processor allocation for work-stealing jobs. Technical Report, MIT, 1998.
- [25] Robert D. Blumofe and Philip A. Lisiecki. Adaptive and reliable parallel computing on networks of workstations. In *Proceedings of the USENIX Annual Technical Symposium*, pages 133–147, Anaheim, California, 1997.
- [26] Robert D. Blumofe and Dionisios Papadopoulos. The performance of work stealing in multiprogrammed environments. In *Proceedings of the International*

- Conference on Measurement and Modeling of Computer Systems*, pages 266–267, Madison, Wisconsin, USA, 1998.
- [27] Robert D. Blumofe and Dionisios Papadopoulos. Hood: A user-level threads library for multiprogrammed multiprocessors. Technical report, University of Texas at Austin, 1999.
  - [28] Robert D. Blumofe and David S. Park. Scheduling large-scale parallel computations on networks of workstations. In *ACM SIGMETRICS Conference on Measurement and Modeling of Computer Systems*, pages 96–105, San Francisco, California, 1994.
  - [29] A. Borodin and R. El-Yaniv. *On-line Computation and Competitive Analysis*. Cambridge University Press, 1998.
  - [30] T. Brecht, Xiaotie Deng, and Nian Gu. Competitive dynamic multiprocessor allocation for parallel applications. In *Parallel and Distributed Processing*, pages 448 – 455, San Antonio, TX, 1995.
  - [31] R. P. Brent. The parallel evaluation of general arithmetic expressions. *Journal of the ACM*, pages 201–206, 1974.
  - [32] F. Warren Burton and M. Ronan Sleep. Executing functional programs on a virtual tree of processors. In *Proceedings of the Conference on Functional Programming Languages and Computer Architecture*, pages 187–194, Portsmouth, New Hampshire, 1981.
  - [33] Soumen Chakrabarti, Cynthia A. Phillips, Andreas S. Schulz, David B. Shmoys, Clifford Stein, and Joel Wein. Improved scheduling algorithms for minsum criteria. In *In the Proceedings of Automata, Languages and Programming*, pages 646–657, Paderborn, Germany, 1996.
  - [34] Bradford L. Chamberlain, Sung-Eun Choi, E Christopher Lewis, Calvin Lin, Lawrence Snyder, and W. Derrick Weathersby. ZPL: A machine independent programming language for parallel computers. *IEEE Transactions on Software Engineering*, 26(3):197–211, 2000.
  - [35] Chandra Chekuri and Michael Bender. An efficient approximation algorithm for minimizing makespan on uniformly related machines. *Journal of Algorithms*, 41(2):212–224, 2001.

- [36] Chandra Chekuri, Sanjeev Khanna, and An Zhu. Algorithms for minimizing weighted flow time. In *Proceedings of the ACM Symposium on the Theory of Computing*, pages 84–93, Hersonissos, Greece, 2001.
- [37] Bo Chen and Arjen P. A. Vestjens. Scheduling on identical machines: How good is lpt in an on-line setting? *Operations Research Letters*, 21:165–169, 1998.
- [38] Jianer Chen and Antonio Miranda. A polynomial time approximation scheme for general multiprocessor job scheduling (extended abstract). In *Proceedings of the ACM Symposium on the Theory of Computing*, pages 418–427, New York, NY, USA, 1999.
- [39] Fabi A. Chudak and David B. Shmoys. Approximation algorithms for precedence-constrained scheduling problems on parallel machines that run at different speeds. In *Proceedings of the ACM-SIAM Symposium on Discrete Algorithms*, pages 581–590, Philadelphia, PA, USA, 1997.
- [40] Walfredo Cirne and Francine Berman. A model for moldable supercomputer jobs. In *Proceedings of the IEEE International Parallel and Distributed Processing Symposium*, page 59, Washington, DC, USA, 2001.
- [41] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. *Introduction to Algorithms*. The MIT Press and McGraw-Hill, second edition, 2001.
- [42] Intel Corporation. Intel thread building blocks. <http://www.intel.com/cd/software/products/asmo-na/eng/294797.htm>.
- [43] Ernest Davis and Jeffrey M. Jaffe. Algorithms for scheduling tasks on unrelated processors. *Journal of ACM*, 28(4):721–736, 1981.
- [44] X. Deng and E. Koutsoupias. Competitive implementation of parallel programs. In *Proceedings of the ACM-SIAM Symposium on Discrete Algorithms*, Austin, Texas, USA, 1993.
- [45] Xiaotie Deng and Patrick Dymond. On multiprocessor system scheduling. In *Proceedings of the ACM Symposium on Parallel Algorithms and Architectures*, pages 82–88, Padua, Italy, 1996.

- [46] Xiaotie Deng, Nian Gu, Tim Brecht, and KaiCheng Lu. Preemptive scheduling of parallel jobs on multiprocessors. In *Proceedings of the ACM-SIAM Symposium on Discrete Algorithms*, pages 159–167, Philadelphia, PA, USA, 1996.
- [47] DESMO-J: A framework for discrete-event modelling and simulation. <http://asi-www.informatik.uni-hamburg.de/desmoj/>.
- [48] Intel Super Computer Systems Division. *Paragon User's Guide*, 1994.
- [49] Allen B. Downey. A parallel workload model and its implications for processor allocation. *Cluster Computing*, 1(1):133–145, 1998.
- [50] Derek L. Eager, John Zahorjan, and Edward D. Lozowska. Speedup versus efficiency in parallel systems. *IEEE Transactions on Computers*, 38(3):408–423, 1989.
- [51] Jeff Edmonds. Scheduling in the dark. In *Proceedings of the ACM Symposium on the Theory of Computing*, pages 179–188, Atlanta, Georgia, USA, 1999.
- [52] Jeff Edmonds, Donald D. Chinn, Tim Brecht, and Xiaotie Deng. Non-clairvoyant multiprocessor scheduling of jobs with changing execution characteristics (extended abstract). In *Proceedings of the ACM Symposium on the Theory of Computing*, pages 120–129, El Paso, Texas, USA, 1997.
- [53] Jeff Edmonds, Donald D. Chinn, Timothy Brecht, and Xiaotie Deng. Non-clairvoyant multiprocessor scheduling of jobs with changing execution characteristics. *Journal of Scheduling*, 6(3):231–250, 2003.
- [54] Zhixi Fang, Peiyi Tang, Pen-Chung Yew, and Chuan-Qi Zhu. Dynamic processor self-scheduling for general parallel nested loops. *IEEE Transactions on Computers*, 39(7):919–929, 1990.
- [55] Dror Feitelson. Parallel workloads archive. <http://www.cs.huji.ac.il/labs/parallel/workload/>.
- [56] Dror G. Feitelson. Packing Schemes for Gang Scheduling. In *Proceedings of the Workshop on Job Scheduling Strategies for Parallel Processing*, pages 89–110, Honolulu, Hawaii, USA, 1996.

- [57] Dror G. Feitelson. Job scheduling in multiprogrammed parallel systems (extended version). Technical report, IBM Research Report RC 19790 (87657), 2nd Revision, 1997.
- [58] Dror G. Feitelson and Larry Rudolph. Parallel job scheduling: Issues and approaches. In *Proceedings of the Workshop on Job Scheduling Strategies for Parallel Processing*, pages 1–18, Santa Barbara, CA, USA, 1995.
- [59] Dror G. Feitelson and Larry Rudolph. Evaluation of design choices for gang scheduling using distributed hierarchical control. *Journal of Parallel and Distributed Computing*, 35(1):18–34, 1996.
- [60] Anja Feldmann, Ji Sgall, and Shang-Hua Teng. Dynamic scheduling on parallel machines. *Theoretical Computer Science*, 130(1):49–72, 1994.
- [61] Raphael Finkel and Udi Manber. DIB—A distributed implementation of backtracking. *ACM Transactions on Programming Languages and Systems*, 9(2):235–256, April 1987.
- [62] High Performance Fortran Forum. High performance Fortran language specification version 1.0. Technical report, Rice University, 1993.
- [63] Matteo Frigo, Charles E. Leiserson, and Keith H. Randall. The implementation of the Cilk-5 multithreaded language. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 212–223, Montreal, Quebec, Canada, 1998.
- [64] Joseph Gil and Yossi Matias. Fast and efficient simulations among CRCW PRAMs. *Journal on Parallel and Distributed Computing*, 23(2):135–148, 1994.
- [65] Allan Gottlieb, Boris D. Lubachevsky, and Larry Rudolph. Basic techniques for the efficient coordination of very large numbers of cooperating sequential processors. *Programming Languages and Systems*, 5(2):164–189, 1983.
- [66] R. L. Graham. Bounds on multiprocessing anomalies. *SIAM Journal on Applied Mathematics*, pages 17(2):416–429, 1969.
- [67] Supercomputing Technologies Group. *Cilk 5.3.2 Reference Manual*. MIT Laboratory for Computer Science, 2001.

- [68] Nian Gu. Competitive analysis of dynamic processor allocation strategies. Master's thesis, York University, 1995.
- [69] Michael Halbherr, Yuli Zhou, and Chris F. Joerg. MIMD-style parallel programming with continuation-passing threads. In *Proceedings of the International Workshop on Massive Parallelism: Hardware, Software, and Applications*, Capri, Italy, September 1994.
- [70] Leslie A. Hall, David B. Shmoys, and Joel Wein. Scheduling to minimize average completion time: off-line and on-line algorithms. In *Proceedings of the ACM-SIAM Symposium on Discrete Algorithms*, pages 142–151, Atlanta, Georgia, USA, 1996.
- [71] Robert H. Halstead, Jr. Implementation of Multilisp: Lisp on a multiprocessor. In *Proceedings of the ACM Symposium on Lisp and Functional Programming*, pages 9–17, Austin, Texas, August 1984.
- [72] B. Hamidzadeh, D. J. Lilja, and Y. Atif. Dynamic scheduling techniques for heterogeneous computing systems. *Concurrency: Practice and Experience*, 7(7):633–652, 1995.
- [73] Mor Harchol-Balter. The effect of heavy-tailed job size distributions on computer system design. In *Proceedings of Applications of Heavy Tailed Distributions in Economics*, Washington, DC, USA, 1999.
- [74] Mor Harchol-Balter and Allen B. Downey. Exploiting process lifetime distributions for dynamic load balancing. *ACM Transactions on Computer Systems*, 15(3):253–285, 1997.
- [75] Tim J. Harris. A survey of PRAM simulation techniques. *ACM Computing Survey*, 26(2):187–206, 1994.
- [76] Yuxiong He, Wen Jing Hsu, and Charles E. Leiserson. Provably efficient two-level adaptive scheduling. In *Proceedings of the Workshop on Job Scheduling Strategies for Parallel Processing*, Saint-Malo, France, 2006.
- [77] Yuxiong He, Wen Jing Hsu, and Charles E. Leiserson. Provably efficient online non-clairvoyant scheduling. In *Proceedings of the IEEE International Parallel and Distributed Processing Symposium*, Long Beach, CA, USA, 2007.

- [78] Yuxiong He, Hongyang Sun, and Wen-Jing Hsu. Adaptive scheduling of parallel jobs on functionally heterogeneous resources. In *Proceedings of the International Conference on Parallel Processing*, Xi An, Shan Xi, China, 2007.
- [79] K. S. Hong and J. Y. T. Leung. On-line scheduling of real-time tasks. *IEEE Transactions on Computers*, 41(10):1326–1331, 1992.
- [80] S. F. Hummel and E. Schonberg. Low-overhead scheduling of nested parallelism. *IBM Journal of Research and Development*, 35(5-6):743–765, 1991.
- [81] IBM. <http://researchweb.watson.ibm.com/cell/>.
- [82] Jeffrey M. Jaffe. An analysis of preemptive multiprocessor job scheduling. *Mathematics of Operations Research*, 5(3):415–421, 1980.
- [83] Jeffrey M. Jaffe. Efficient scheduling of tasks without full use of processor resources. *Theoretical Computer Science*, 12:1–17, 1980.
- [84] Klaus Jansen and Lorant Porkolab. Linear-time approximation schemes for scheduling malleable parallel tasks. In *Proceedings of the ACM-SIAM Symposium on Discrete Algorithms*, pages 490–498, Baltimore, Maryland, USA, 1999.
- [85] Klaus Jansen and Hu Zhang. Scheduling malleable tasks with precedence constraints. In *Proceedings of the ACM Symposium on Parallel Algorithms and Architectures*, pages 86–95, Las Vegas, Nevada, USA, 2005.
- [86] Bala Kalyanasundaram and Kirk Pruhs. Speed is as powerful as clairvoyance. *Journal of the ACM*, 47(4):617–643, 2000.
- [87] D. Karger, C. Stein, and Joel Wein. *Handbook of Algorithms and Theory of Computation*, chapter 35 - Scheduling Algorithms. CRC Press, 1997.
- [88] Anna Karlin, Mark S. Manasse, Larry Rudolph, and Daniel D. Sleator. Competitive snoopy caching. *Algorithmica*, 3:79–119, 1988.
- [89] Richard M. Karp and Yanjun Zhang. A randomized parallel branch-and-bound procedure. In *Proceedings of the ACM Symposium on the Theory of Computing*, pages 290–300, Chicago, Illinois, May 1988.

- [90] H. Kellerer, T. Tautenhahn, and G. J. Woeginger. Approximability and non-approximability results for minimizing total flow time on single machine. In *Proceedings of the ACM Symposium on the Theory of Computing*, Philadelphia, Pennsylvania, USA, 1996.
- [91] Ashfaq Khokhar, Viktor K. Prasanna, Muhammad Shaaban, and Cho-Li Wang. Heterogeneous supercomputing: Problems and issues. In *Workshop on Heterogeneous Processing*, Beverly Hills, CA, USA, 1992.
- [92] Thomas T. Kwan, Robert E. McGrath, and Daniel A. Reed. Em3: A taxonomy of heterogeneous computing systems. *Computer*, 28(12):68–70, 1995.
- [93] C. Lasser. *The Essential *Lisp Manual*. Thinking Machines Corporation, Cambridge, MA, 1986.
- [94] E. L. Lawler, J. K. Lenstra, A. H. G. Rinnooy Kan, and D. B. Shmoys. *Sequencing and Scheduling: Algorithms and Complexity*, pages 445–552. Elsevier Science Publishers, 1997.
- [95] Daan Leijen and Judd Hall. Microsoft task parallel library. <http://msdn.microsoft.com/msdnmag/issues/07/10/Futures/default.aspx>.
- [96] Charles E. Leiserson, Zahi S. Abuhamdeh, David C. Douglas, Carl R. Feynman, Mahesh N. Ganmukhi, Jeffrey V. Hill, W. Daniel Hillis, Bradley C. Kuszmaul, Margaret A. St Pierre, David S. Wells, Monica C. Wong-Chan, Shaw-Wen Yang, and Robert Zak. The network architecture of the Connection Machine CM-5. *Journal of Parallel and Distributed Computing*, 33(2):145–158, 1996.
- [97] Will Leland and Teunis J. Ott. Load-balancing heuristics and process behavior. In *Proceedings of the International Conference on Measurement and Modeling of Computer Systems*, pages 54–69, New York, NY, USA, 1986.
- [98] Stefano Leonardi and Danny Raz. Approximating total flow time on parallel machines. In *Proceedings of the ACM Symposium on the Theory of Computing*, pages 110–119, El Paso, Texas, USA, 1997.
- [99] Scott T. Leutenegger and Mary K. Vernon. The performance of multiprogrammed multiprocessor scheduling policies. In *Proceedings of the International Conference on Measurement and Modeling of Computer Systems*, pages 226–236, Boulder, Colorado, USA, 1990.

- [100] R. L. Graham, E. L. Lawler, J. K. Lenstra, and A. H. G. Rinnooy Kan. Optimization and approximation in deterministic sequencing and scheduling: A survey. *Annals of Discrete Mathematics*, 5:287–326, 1979.
- [101] Uri Lublin and Dror G. Feitelson. The workload on parallel supercomputers: Modeling the characteristics of rigid jobs. *Journal of Parallel and Distributed Computing*, 63(11):1105–1122, 2003.
- [102] Walter Ludwig and Prasoon Tiwari. Scheduling malleable and nonmalleable parallel tasks. In *Proceedings of the ACM-SIAM Symposium on Discrete Algorithms*, pages 167–176, Philadelphia, PA, USA, 1994.
- [103] Cathy McCann, Raj Vaswani, and John Zahorjan. A dynamic processor allocation policy for multiprogrammed shared-memory multiprocessors. *ACM Transactions on Computer Systems*, 11(2):146–178, 1993.
- [104] Jason McCullough and Eric Torng. Srpt optimally utilizes faster machines to minimize flow time. In *Proceedings of the ACM-SIAM Symposium on Discrete Algorithms*, pages 350–358, New Orleans, Louisiana, 2004.
- [105] D. Menasce and V. Almeida. Heterogeneous supercomputing: Is it cost-effective? In *Supercomputing*, pages 169–177, New York, NY, USA, 1990.
- [106] Sun Microsystems. Sun fortress. <http://research.sun.com/projects/plrg/>.
- [107] Eric Mohr, David A. Kranz, and Jr. Robert H. Halstead. Lazy task creation: A technique for increasing the granularity of parallel programs. In *Proceedings of the ACM Symposium on Lisp and Functional Programming*, pages 185–197, Nice, France, 1990.
- [108] Rajeev Motwani, Steven Phillips, and Eric Torng. Non-clairvoyant scheduling. In *Proceedings of the ACM-SIAM Symposium on Discrete Algorithms*, pages 422–431, Austin, Texas, USA, 1993.
- [109] Rajeev Motwani and Prabhakar Raghavan. *Randomized Algorithms*. Cambridge University Press, 1995.
- [110] Gregory Mounie, Christophe Rapine, and Dennis Trystram. Efficient approximation algorithms for scheduling malleable tasks. In *Proceedings of the ACM*

- Symposium on Parallel Algorithms and Architectures*, pages 23–32, Saint Malo, France, 1999.
- [111] Girija J. Narlikar and Guy E. Blelloch. Space-efficient scheduling of nested parallelism. *ACM Transactions on Programming Languages and Systems*, 21(1):138–173, 1999.
  - [112] Lars S. Nyland, Jan F. Prins, Allen Goldberg, and Peter H. Mills. A design methodology for data-parallel applications. *IEEE Transactions on Software Engineering*, 26(4):293–314, 2000.
  - [113] Cynthia A. Phillips, Cliff Stein, Eric Torng, and Joel Wein. Optimal time-critical scheduling via resource augmentation (extended abstract). In *Proceedings of the ACM Symposium on the Theory of Computing*, pages 140–149, El Paso, Texas, USA, 1997.
  - [114] Kirk Pruhs, Jiri Sgall, and Eric Torng. *Handbook of Scheduling: Algorithms, Models, and Performance Analysis*, chapter 15 - Online Scheduling. CRC Press Company, 2004.
  - [115] J. R. Rose and G. L. Steele Jr. C*: An extended C language for data parallel programming. In *Supercomputing*, pages 2–16, 1987.
  - [116] Larry Rudolph, Miriam Slivkin-Allalouf, and Eli Upfal. A simple load balancing scheme for task allocation in parallel machines. In *Proceedings of the ACM Symposium on Parallel Algorithms and Architectures*, pages 237–245, Hilton Head, South Carolina, 1991.
  - [117] William Scherer, Doug Lea, and Michael Scott. Scalable synchronous queues. In *Proceedings of the ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, New York City, NY, USA, 2006.
  - [118] Uwe Schwiegelshohn, Walter Ludwig, Joel L. Wolf, John Turek, and Philip S. Yu. Smart smart bounds for weighted response time scheduling. *SIAM Journal of Computing*, 28(1):237–253, 1998.
  - [119] Siddhartha Sen. Dynamic processor allocation for adaptively parallel jobs. Master’s thesis, Massachusetts Institute of technology, 2004.

- [120] Kenneth C. Sevcik. Application scheduling and processor allocation in multi-programmed parallel processing systems. *Performance Evaluation*, 19(2-3):107–140, 1994.
- [121] Jiri Sgall. On-line scheduling. In *Developments from a June 1996 seminar on Online algorithms*, pages 196–231, London, UK, 1998.
- [122] Nir Shavit and Dan Touitou. Elimination trees and the construction of pools and stacks: preliminary version. In *Proceedings of the ACM Symposium on Parallel Algorithms and Architectures*, pages 54–63, Santa Barbara, California, USA, 1995.
- [123] Nir Shavit and Asaph Zemach. Diffracting trees. *ACM Transactions of Computer Systems*, 14(4):385–428, 1996.
- [124] D. B. Shmoys, J. Wein, and D. P. Williamson. Scheduling parallel machines online. In *Proceedings of the IEEE Symposium on Foundations of Computer Science*, pages 131–140, San Juan, Puerto Rico, 1991.
- [125] Daniel D. Sleator and Robert E. Tarjan. Amortized efficiency of list update and paging rules. *Communication of ACM*, 28(2):202–208, 1985.
- [126] B. Song. Scheduling adaptively parallel jobs. Master’s thesis, Massachusetts Institute of Technology, 1998.
- [127] Mark S. Squillante. On the benefits and limitations of dynamic partitioning in parallel computer systems. In *Proceedings of the International Parallel Processing Symposium*, pages 219–238, Oakland, California, USA, 1995.
- [128] Dan Suciú and Val Tannen. Efficient compilation of high-level data parallel algorithms. In *Proceedings of the ACM Symposium on Parallel Algorithms and Architectures*, pages 57–66, Cape May, New Jersey, USA, 1994.
- [129] Kaushik Guha Timothy B. Brecht. Using parallel program characteristics in dynamic processor allocation policies. *Performance Evaluation*, 27-28:519–539, 1996.
- [130] Andrew Tucker and Anoop Gupta. Process control and scheduling issues for multiprogrammed shared-memory multiprocessors. In *Proceedings of the ACM*

- Symposium on Operating Systems Principles*, pages 159–166, New York, NY, USA, 1989.
- [131] John Turek, Walter Ludwig, Joel L. Wolf, Lisa Fleischer, Prasoon Tiwari, Jason Glasgow, Uwe Schwiegelshohn, and Philip S. Yu. Scheduling parallelizable tasks to minimize average response time. In *Proceedings of the ACM Symposium on Parallel Algorithms and Architectures*, pages 200–209, Cape May, New Jersey, USA, 1994.
- [132] John Turek, Uwe Schwiegelshohn, Joel L. Wolf, and Philip S. Yu. Scheduling parallel tasks to minimize average response time. In *Proceedings of the ACM-SIAM Symposium on Discrete Algorithms*, pages 112–121, Arlington, Virginia, USA, 1994.
- [133] Roger Wattenhofer and Peter Widmayer. The counting pyramid: an adaptive distributed counting scheme. *Journal of Parallel and Distributed Computing*, 64(4):449–460, 2004.
- [134] K. K. Yue and D. J. Lilja. Implementing a dynamic processor allocation policy for multiprogrammed parallel applications in the Solaris^{TM} operating system. *Concurrency and Computation-Practice and Experience*, 13(6):449–464, 2001.
- [135] George K. Zipf. *Human Behaviour and the Principle of Least-Effort*. Addison-Wesley, 1949.