Abstract of thesis entitled

"New Results on Online Job Scheduling and Data Stream Algorithms"

Submitted by

Lee Lap Kei

for the degree of Doctor of Philosophy
at The University of Hong Kong in April 2009

This thesis presents several new results on online job scheduling and data stream algorithms.

Job scheduling is a fundamental problem in computer science, which has been studied extensively and has applications in practical computer systems. Traditionally, the primary concern of job scheduling was the system performance. One commonly used "quality of service" measurement is the total flow time (or equivalently average response time) of jobs, which measures how long each job has to wait before it completes. The increasing computing power of processors is accompanied with dramatic increase in their energy consumption. To be more energy efficient, many modern processors now adopt the technology of dynamic speed scaling, where the processor can adjust its speed dynamically in some range. Running a job at a slower speed saves energy, yet it takes longer time and may affect the performance.

This thesis studies the tradeoff between flow time and energy and aims at finding algorithms that minimize their sum. We consider the online setting, where the information of a job is known only when it is released. To schedule a single processor, we introduce and analyze a speed scaling algorithm which is more efficient for flow plus energy and

more stable to speed change than existing speed functions. Notice that a processor can actually go to a sleep state for further energy saving, yet waking up from sleep requires extra energy. We initiate the study in a model that exploits both speed scaling and multiple sleep states. We also consider scheduling on multi-processors, given that multi-core processors are getting common. In some applications like operating systems, job size is only known when the job finishes, which is referred to as the non-clairvoyant model. We also derive a competitive algorithm in such model.

This thesis also considers monitoring statistics on a data stream, which has become a common form of data in many applications such as network monitoring and telecommunications. Since data items are time sensitive, we consider the sliding window model, which only queries the most recent items received by the stream.

We first study space-efficient algorithms. The challenge is how to represent massive data volume in a window, while allowing certain statistics to be computed with sufficient accuracy. One fundamental problem is to estimate the count of 1-bits in a bit stream, yet its memory requirement is too much for many applications. Observing that good estimate is often required only when the count is large enough, we introduce the Significant One Counting problem, which requires much smaller memory, and give a space-optimal algorithm. We also give another space-optimal algorithm for identifying frequent items.

Finally, we study communication-efficient algorithms for continuous monitoring of multiple, distributed data streams. The concern is how to minimize the communication between individual streams and a coordinator, while allowing the coordinator, at any time, to report the global statistics of all streams with sufficient accuracy. We initiate the study in the sliding window model.

(493 words)

New Results on Online Job Scheduling and Data Stream Algorithms

by

Lee Lap Kei
B.Eng. *H.K.*

A thesis submitted in partial fulfillment of the requirements for
the Degree of Doctor of Philosophy
at The University of Hong Kong

April 2009

Declaration

I declare that this thesis represents my own work, except where due acknowledgement is made, and that it has not been previously included in a thesis, dissertation or report submitted to this University or to any other institutions for a degree, diploma or other qualification.

Signed by .....

Lee Lap Kei

Acknowledgements

First of all, I would like to express my profound thanks and gratitude to my advisor, Prof. Tak-Wah Lam, for granting me the opportunity for pursuing the PhD degree under his guidance. He is a wonderful teacher as well as an excellent mentor. When I was an undergraduate student, his course Discrete Mathematics had aroused my interest in theoretical computer science which motivated me to do further research. During my postgraduate study, whenever I encounter problems in my research and even in my life, he has been willing to discuss them with me and gave me many stimulating comments and insightful suggestions. I am deeply grateful for his continuous guidance, support, encouragement and tolerance which enabled me to accomplish the work for this thesis.

I would also like to thank Prof. Francis Chin, Dr. Hing-Fung Ting, and Dr. Siu-Ming Yiu, who helped me to build up my knowledge of algorithms. My special thanks go to Dr. Hing-Fung Ting, who has spent lots of time teaching me research skills and discussing with me on research. He has also given me great support and advice, which have helped me a lot throughout my studies.

My co-authors helped me greatly in doing research. I am much indebted to H. L. Chan, W. T. Chan, X. Han, R. Hung, K. S. Mak, K. Pruhs, I. To and P. Wong for the discussions that provide me with enlightening views of the research problems which I have struggled. I have to thank my colleagues P. Y. Chan, C. W. Fok, P. Y. Fung, S. Huang, K. S. Liu, C. M. Leung, S. Y. Leung, M. H. Siu, S. L. Tam, M. K. Wu, L. Yan, B. Yang, D. Ye, W. Zhang and Y. Zhang for bringing me many joyful moments in my postgraduate life.

I would like to offer my genuine thanks to my parents and my sister for their love and patience over the years. They are always by my side to give me their support and care whenever I am confused and frustrated. I am obliged to my grandparents, especially my grandma, for their love and care and giving me the good memories. I am also extremely grateful to Cherry Nga-In Wu for her care, patience and spiritual support. Her endless love is always with me whatever happens.

Remarks. Sections 2, 3 and 4 of this thesis are joint work with Tak-Wah Lam, Isaac To and Prudence Wong, which appeared in *Proceedings of European Symposium on Algorithms (ESA)*, 2008, and *Proceedings of International Colloquium on Automata, Languages and Programming (ICALP)*, 2009, and *IEEE Transactions on Parallel and Distributed Systems*, 2008, and *Proceedings of ACM Symposium on Parallelism in Algorithms*

and Architectures (SPAA), 2008. Part of Section 5 is joint work with Ho-Leung Chan, Jeff Edmonds, Tak-Wah Lam, Alberto Marchetti-Spaccamela and Kirk Pruhs, which appeared in *Proceedings of International Symposium on Theoretical Aspects of Computer Science (STACS)*, 2009. Sections 6 is the joint work with Hing-Fung Ting, which appeared in *Proceedings of ACM-SIAM Symposium on Discrete Algorithms (SODA)*, 2006, and *Proceedings of ACM SIGMOD-SIGACT-SIGART Symposium on Principles of Database Systems (PODS)*, 2006. Section 7 is the joint work with Ho-Leung Chan, Tak-Wah Lam and Hing-Fung Ting.

Contents

1 Introduction **1**

1.1 Online Job Scheduling 1

1.1.1 Flow-energy scheduling on single processor 3

1.1.2 Flow-energy scheduling with sleep states 5

1.1.3 Non-migratory multi-processor flow-energy scheduling 7

1.1.4 Non-clairvoyant flow-energy scheduling 10

1.2 Data Stream Algorithms 12

1.2.1 Space-efficient data stream algorithms 14

1.2.2 Communication-efficient data stream algorithms 17

1.3 Organization 20

2 Flow-Energy Scheduling on Single Processor **21**

2.1 Speed Function AJC and Algorithm SRPT-AJC 22

2.2 Analysis for Infinite Speed Model 23

2.3 Analysis for Bounded Speed Model 26

3 Flow-Energy Scheduling with Sleep States **28**

3.1 Model and Notations 29
3.2 Sleep Management Algorithm IdleLonger 30
3.2.1 Sleep management algorithm for a single sleep state 31
3.2.2 Sleep management algorithm for $m \geq 2$ sleep states 34
3.3 Speed Scaling Algorithm SAJC 37
3.4 Bounded Speed Model 40

4 Non-migratory Multi-processor Flow-Energy Scheduling 48

4.1 Preliminaries 50
4.2 The Online Algorithm 53
4.3 Jobs of Power-of-2 Size 55
4.3.1 Eliminating migration in a multi-processor schedule of parallel jobs 59
4.3.2 Forward transformation of schedules: from J to J^* 62
4.3.3 Backward transformation of schedules: from J^* to J^+ and then to J 64
4.4 Jobs of Arbitrary Size 67
4.4.1 Restricted but useful optimal schedules 67
4.4.2 Constructing CRR schedules 70
4.4.3 Optimal migratory schedules 77
4.5 Lower Bound 80

5 Non-clairvoyant Flow-Energy Scheduling 81

5.1 Batched Jobs 82
5.1.1 Comparing WRR-AJW* against HDF-AJW* 83

5.1.2 Analysis of SJF-AJC* 85
5.1.3 Analysis of HDF-AJW* 86
5.2 Arbitrary Jobs 90
5.3 Sleep States 94

6 Space-efficient Data Stream Algorithms **98**
6.1 Significant One Counting 100
6.1.1 Lower bound 100
6.1.2 An one-level data structure 103
6.1.3 Improvement in memory: a multilevel data structure 109
6.1.4 Improvement in time: the optimal data structure 111
6.2 Finding Frequent Items over Sliding Window 113
6.2.1 λ-snapshot and λ-counter 114
6.2.2 Finding (θ, ε)-frequent item set 118
6.2.3 Algorithm extensions 122

7 Communication-efficient Data Stream Algorithms **130**
7.1 Basic Counting 132
7.2 Approximate Counting of All Items 137
7.2.1 A simple algorithm 138
7.2.2 The full algorithm 141
7.3 Other Extensions 144
7.3.1 Frequent items 145

7.3.2 Quantiles 145
7.3.3 Out-of-order streams 146
7.4 Lower Bounds 146

8 Conclusion **150**

8.1 Future Work 152

Chapter 1

Introduction

This thesis presents several new results on online job scheduling and data stream algorithms. Below, we will give an overview of the problems we consider, review the related work and state our contributions.

1.1 Online Job Scheduling

Job scheduling and flow time. Job scheduling is a classical and fundamental problem in computer science, which has been studied extensively and has applications in practical computer systems, e.g., operating systems, web servers and database query servers (see, e.g., [11,70,87] for a survey). The basic situation to study is as follows. We are given one or more processors. Jobs are released at different times, with different work requirement (or size). A scheduling algorithm has to select, at any time, a job to execute on each processor. Note that each job can be scheduled on at most one processor at any time. Preemption is allowed and a preempted job can be resumed at the point of preemption. One commonly used "quality of service" measurement for job scheduling is total flow time (or equivalently, average response time) [7,8,26,27,69,72]. The *flow time* (or simply *flow*) of a job is the time elapsed since the job arrives until it is completed; in other words, it measures how long the user of the computer system has to wait for the job to complete. The objective of a scheduling algorithm is to minimize the total flow time of all jobs.

Online algorithms and competitive analysis. In the *online* setting, we only know the information of jobs released so far. This is in contrast to the offline setting, where the complete job sequence is known in advance. In theoretical computer science, the performance of online algorithms is usually evaluated using *competitive analysis*, which is introduced in the seminal paper of Sleator and Tarjan [89]. Competitive analysis is a worst-case comparison between an online algorithm and the *optimal offline algorithm*. Formally speaking, given a cost function to minimize, such as total flow time, an online algorithm is said to be *c-competitive* if for any input sequence, the cost incurred is never more than c times the cost required by an optimal offline algorithm. For more details about competitive analysis, Borodin and El-Yaniv's book [19] is a good reference. For flow time scheduling, it is well-known that the online algorithm SRPT (shortest remaining processing time) is optimal, i.e., 1-competitive, for total flow time [11].

Energy concern. In the last few years, the increasing computing power of processors has caused dramatic increase in their energy consumption. This not only leads to high cooling costs but also to substantially reduced battery life in laptops and other mobile devices. Companies such as IBM, Intel and AMD have made power aware design a key priority and even scrapped the development of faster processors in favor of lower power ones. To be more energy efficient, many modern processors now adopt the technology of *dynamic speed (voltage) scaling* (see, e.g., [50, 80, 90]), where the processor can adjust its speed dynamically in some range without any overhead. For example, IBM's PowerPC 970FX [85] allows the operating system to dynamically vary the speed (with zero overhead) at various discrete points from 2.5GHz to 625MHz while the power consumption reduces from 100W to less than 10W. Running a job at a slower speed is more energy efficient, yet it takes longer time and may affect the performance. Recently, there are a lot of theory research on online job scheduling taking dynamic speed scaling and energy usage into consideration (e.g., [2, 3, 15, 16, 21, 23, 84, 92]; see [55] for a survey). The challenge basically arises from the conflicting objectives of providing good "quality of service" (e.g., total flow time) and conserving energy.

Speed scaling models. The above results are based on a speed scaling model in which a processor, when running at speed s, consumes energy at the rate of s^α, where α is typically 2 [76] or 3 (the cube-root rule [20])[1]. In the *infinite speed model* [92], a processor can run at any speed between 0 and ∞. A more realistic model, known as the

[1] In reality, this does not hold at very low speeds due to leakage power effects that do not scale with speed.

bounded speed model [23], imposes a bound T on the maximum processor speed.

Flow-energy scheduling. The objectives flow time and energy are competing. To better understand the tradeoff between them, Albers and Fujiwara [2] proposed combining the dual objectives into a single one of minimizing the sum of total flow time and energy[2], which we refer to as *flow-energy scheduling*. The intuition is that, from an economic viewpoint, flow time and energy can each be measured in money terms; thus it can be assumed that users are willing to pay one unit of energy to reduce a certain units (say, ρ units) of flow time. A large value of ρ means that energy is more of a concern; on the other hand, if $\rho = 0$, the problem reduces to the traditional flow time scheduling. In general, the objective is to optimize the total flow time plus ρ times the energy used. For positive ρ, by changing the units of either time or energy, one can further assume without loss of generality that $\rho = 1$ and thus would like to optimize total flow time plus energy.

In this thesis, we focus on online flow-energy scheduling and we study four problems, namely, flow-energy scheduling on a single processor, flow-energy scheduling with sleep states, non-migratory multi-processor flow-energy scheduling, and non-clairvoyant flow-energy scheduling. We introduce these four problems in the following sections.

1.1.1 Flow-energy scheduling on single processor

We first consider flow-energy scheduling on a single processor. Under the infinite speed model, Albers and Fujiwara [2] focused on scheduling jobs of unit size, and they gave an $8.3e((3+\sqrt{5})/2)^{\alpha}$-competitive algorithm for minimizing total flow time plus energy. Bansal, Pruhs and Stein [16] extended their work to jobs of arbitrary sizes. They gave an $O(1)$-competitive online algorithm for minimizing flow time plus energy; precisely, the competitive ratio is $\mu_\epsilon\gamma_1$, where ϵ is any positive constant, $\mu_\epsilon = \max\{(1+\frac{1}{\epsilon}), (1+\epsilon)^{\alpha}\}$ and $\gamma_1 = \max\{2, 2(\alpha-1)/(\alpha-(\alpha-1)^{1-1/(\alpha-1)})\}$. E.g., if $\alpha = 2$, the competitive ratio is 5.236. More recently, Bansal, Chan, Lam and Lee [12] adapted this result to the bounded speed model. Assuming that the online algorithm can have a higher maximum speed of $(1+\epsilon)T$ for any $\epsilon > 0$, the competitive ratio in this case increases slightly to $\mu_\epsilon\gamma_2$, where $\gamma_2 = 2\alpha/(\alpha-(\alpha-1)^{1-1/(\alpha-1)}) = (2+o(1))\alpha/\ln\alpha$. Table 1.1 shows the competitive ratios

[2]Sum of objectives are common in bi-objective optimizations, e.g., TCP acknowledgement problem [39,63] with sum of acknowledgement cost and acknowledgement delays as objective, network design problem [43] with total hardware and QoS costs, and the facility location problem [33] with facility installation and client service costs.

	$\alpha = 2$	$\alpha = 3$
Infinite speed model ($T = \infty$)	5.236 [16] **2.667** [our result]	7.940 [16] **3.252** [our result]
Bounded speed model	10.472 with max speed $1.618T$ [12] **3.6** with max speed T [our result]	11.910 with max speed $1.466T$ [12] **4** with max speed T [our result]

Table 1.1: Results on flow-energy scheduling. Note that our new results do not demand extra speed.

for some fixed α. Both results also hold for weighted flow time plus energy, where jobs may carry different weights.

Follow-up questions. The speed function of the algorithms by Bansal et al. [12, 16] depends on the *remaining work* of "active" jobs (i.e., jobs that have not been completed). There are some questions related to such a speed function.

- Work-based speed functions would demand the processor speed to change continuously which is undesirable practically. Can one design a more stable speed function that changes in a discrete manner?
- The algorithm in [12] requires extra speed. In contrast, the classic result on flow-time scheduling does not need extra speed [11]. This is perhaps due to the inefficiency of the work-based speed functions; specifically, they are sometimes slower than a critical threshold ($(\frac{n}{\alpha-1})^{1/\alpha}$ where n is the number of active jobs). When this happens, we can decrease the flow time plus energy by increasing the speed. It is natural to ask whether a speed function that never goes below this threshold can work without extra speed and gives a better competitive ratio.

Our contribution. We answer affirmatively the above questions by introducing a new speed function AJC that depends on the number of active jobs. AJC is more stable, changing speed only at job arrival or completion. Using AJC leads to improvements in both the infinite and bounded speed models, as shown below.

Speed function AJC. AJC is defined such that at any time t, the speed is $n(t)^{1/\alpha}$, where $n(t)$ is the number of active jobs at time t. We use SRPT (instead of SJF (shortest job first) in [12, 16]) to select jobs. This algorithm is more competitive for minimizing flow time plus energy and does not demand extra speed: for the infinite and bounded speed model, the competitive ratios are respectively $\beta_1 = 2/(1 - \frac{\alpha-1}{\alpha^{\alpha/(\alpha-1)}})$ and $\beta_2 =$

$2(\alpha+1)/(\alpha - \frac{\alpha-1}{(\alpha+1)^{1/(\alpha-1)}})$. Table 1.1 compares these ratios with those in [12, 16]. The improvement is more significant for large α, as β_1 and β_2 tend to $2\alpha/\ln\alpha$, while $\mu_\epsilon\gamma_1$ and $\mu_\epsilon\gamma_2$ [12, 16] tend to $2(\alpha/\ln\alpha)^2$.

Technically speaking, the analysis of existing algorithms requires indirect comparison via a notion called fractional flow. In contrast, we divide the time into "stable intervals", and directly compare the flow time of the online algorithm against an optimal offline algorithm in each interval. This makes the analysis tighter.

Remarks. The theoretical study of energy-efficient scheduling was initiated by Yao, Demers and Shenker [92]. They considered deadline scheduling in the infinite speed model, where jobs have deadlines. Their result was improved by Bansal et al. [15], and extended to the bounded speed model by Chan et al. [23] and Bansal et al. [12]. Pruhs et al. [83] also studied offline scheduling for minimizing the total flow time on a processor with a given amount of energy. The offline problem of minimizing the makespan (i.e., the maximum job completion time) subject to a fixed amount of energy has also been studied in [21, 84].

1.1.2 Flow-energy scheduling with sleep states

We extend the study of flow-energy scheduling to a model that allows both sleep management and speed scaling. The problem is to determine when to sleep and for how long, as well as which job and at what speed to run. Below, we first review the related work and then state our contributions.

Sleep management. In earlier days, energy reduction was mostly achieved by allowing a processor to enter a low-power *sleep* state, yet waking up requires extra energy. In the (embedded systems) literature, there are different energy-efficient strategies to bring a processor to sleep during a period of zero load [18]. This is an online problem, usually referred to as *dynamic power management*. The input is the length of the period, known only when the period ends. There are several interesting results with competitive analysis (e.g., [6, 56, 62]). In its simplest form, the problem assumes the processor is in either the *awake* state or the *sleep* state. The awake state always requires a static power $\sigma > 0$. To have zero energy usage, the processor must enter the sleep state, but a wake-up back to the awake state requires $\omega > 0$ energy. In general, there can be multiple intermediate

sleep states, which demand some static power but less wake-up energy.

It is natural to study job scheduling on a processor that allows both sleep states and speed scaling. More specifically, a processor in the awake state can run at any speed $s \geq 0$ and consumes energy at the rate $s^\alpha + \sigma$, where $\sigma > 0$ is static power and s^α is the dynamic power[3]. Here job scheduling requires two components: a *sleep management algorithm* to determine when to sleep or work, and a *speed scaling algorithm* to determine which job and at what speed to run. Notice that sleep management here is not the same as in dynamic power management; in particular, the length of a sleep or idle period is part of the optimization (rather than the input). Adding a sleep state actually changes the nature of speed scaling. Assuming no sleep state, running a job slower is a natural way to save energy. Now one can also save energy by sleeping more and working faster later. It is even more complicated when flow is concerned. Prolonging a sleeping period by delaying job execution can save energy, yet it also incurs extra flow. Striking a balance is not trivial. In the theory literature, the only relevant work is by Irani et al. [57]; they studied deadline scheduling on a processor with one sleep state and infinite speed scaling. They showed an $O(1)$-competitive algorithm to minimize the energy for meeting the deadlines of all jobs.

Our contributions. We initiate the study of flow-energy scheduling that exploits both speed scaling and multiple sleep states. We give a sleep management algorithm called IdleLonger, which works for a processor with one or multiple levels of sleep states. We adapt the speed scaling algorithm AJC [67] to take the static power σ into consideration. Under the infinite speed model, this adapted algorithm together with IdleLonger is shown to be $O(1)$-competitive for minimizing flow plus energy. More precisely, the ratio is $O(\frac{\alpha}{\ln \alpha})$ (recall that α is a constant).

For the bounded speed model, the problem becomes more difficult since the processor, once overslept, cannot rely on unlimited extra speed to catch up the delay. Nevertheless, we are able to enhance IdleLonger and AJC to observe the maximum processor speed. It remains $O(\frac{\alpha}{\ln \alpha})$-competitive for flow plus energy under the bounded speed model.

Sleep management algorithm IdleLonger. When the processor is sleeping, it is natural to delay waking up until sufficient jobs have arrived. The non-trivial case is when the processor is idle (i.e., awake but at zero speed), IdleLonger has to determine when

[3]Static power is dissipated due to leakage current and is independent of processor speed, and dynamic power is due to dynamic switching loss and increases with the speed.

to start working again or go to sleep. At first glance, if some new jobs arrive while the processor is idle, the processor should run the jobs immediately so as to avoid extra flow. Yet this would allow the adversary to easily keep the processor awake, and it is difficult to achieve $O(1)$-competitiveness. In an idle period, IdleLonger considers the (static) energy and flow accumulated during the period as two competing quantities. Only if the flow exceeds the energy, IdleLonger would start to work. Otherwise, IdleLonger will remain idle until the energy reaches to a certain level; then the processor goes to sleep even in the presence of jobs.

Analysis framework. Apparently, a sleep management algorithm and a speed scaling algorithm would affect each other; analyzing their relationship and their total cost could be a complicated task. Interestingly, our results stem from the fact that we can isolate the analysis of these algorithms. We divide the total cost (flow plus energy) into two parts, *working cost* (incurred while working on jobs) and *inactive cost* (incurred at other times). We upper bound the inactive cost of IdleLonger independent of the speed scaling algorithm. For the working cost, although it does depend on both algorithms, our potential analysis of the speed scaling algorithms reveals that the dependency on the sleep management algorithm is limited to a simple quantity called *inactive flow*, which is the flow part of the inactive cost. Intuitively, large inactive flow means many jobs are delayed due to prolonged sleep, and hence the processor has to work faster later to catch up, incurring a higher working cost. It is easy to minimize inactive flow at the sacrifice of the energy part of the inactive cost. IdleLonger is designed to maintain a good balance between them. In conclusion, coupling IdleLonger with AJC, we obtain competitive algorithms for flow plus energy.

1.1.3 Non-migratory multi-processor flow-energy scheduling

We extend the study of flow-energy scheduling to the setting with $m \geq 2$ processors. This extension is not only of theoretical interest, as modern processors adopt multi-core technology (dual-core and quad-core are getting common). A multi-core processor is essentially a pool of parallel processors. To make our work more meaningful, we aim at schedules that do not require job migration among processors. In practice, migrating jobs requires overheads and is avoided in many applications.

Multi-processor flow time scheduling. In the older days, when energy was not a concern, flow time scheduling on multiple processors running at fixed speed was an interesting problem by itself (e.g., [7, 8, 26, 27, 69, 82]). In the multi-processor setting, jobs remain sequential in nature and cannot be executed by more than one processor in parallel. We differentiate two types of schedules: a migratory schedule can move partially-executed jobs from one processor to another processor without any penalty, and a non-migratory schedule cannot. Different online algorithms like SRPT and IMD that are $\Theta(\log P)$-competitive have been proposed respectively under the migratory and the non-migratory model, where P is the ratio of the maximum job size to the minimum job size [7,8]. Furthermore, Chekuri et al. [26] have shown that IMD is $O(1+\frac{1}{\epsilon})$-competitive when using processors $(1+\epsilon)$ times faster. If migration is allowed, SRPT can achieve a competitive ratio one or even smaller, when using sufficiently faster processors [72, 79].

Multiprocessor scheduling for flow time and energy. The only previous work on multi-processors taking flow time and energy into consideration was by Bunde [21], which is about an offline approximation algorithm for jobs of unit size. As for online algorithms, no work has been known that takes flow time and energy into consideration.

Our contributions. To balance the energy usage of multiple processors, it is natural to consider some kind of round-robin strategy to dispatch jobs. Typical examples include IMD[4] for flow time scheduling [7] and CRR for energy-efficient deadline scheduling[5] in the infinite speed model [3]. Our main contribution is to apply CRR (classified round robin) for optimizing flow time plus energy and give a non-trivial analysis of its performance. Unlike [3], we apply CRR according to the job size rather than the job density; we classify jobs according to their sizes. Intuitively, CRR handles different classes independently; jobs of the same class are dispatched (upon their arrival) to the m processors using a round-robin strategy. Note that CRR is similar to IMD in the sense that both algorithms divide jobs into classes according to their size, but IMD is slightly more complicated in job dispatching. Recall that for minimizing flow time alone, IMD is known to be $O(\log P)$-competitive [7], and $O(1)$-competitive when using processors with extra speed [26]; yet no similar result is known for CRR.

[4]IMD divide jobs into classes according to their sizes and dispatches a job to the processor with the smallest accumulated work of jobs of the corresponding class.

[5]In [3], CRR divides jobs into classes according to their "density" (defined as the job size divided by the difference between deadline and release time), and dispatch jobs of the same class to the processors in a round-robin fashion.

Flow time	Flow time plus energy
$\Theta(\log P)$ [7, 8]	$\Theta(\log P)$ (for jobs with power-of-2 size) [our result] $\Omega(\log P)$ [our result]
$O(1)$ using $(1+\epsilon)$-speed processors [26]	$O(1)$ using $(1+\epsilon)$-speed processors [our result]

Table 1.2: The competitive ratios of non-migratory online algorithms for minimizing flow time and flow time plus energy (the constant α is absorbed by the big-Oh notation).

Jobs that are dispatched by CRR to the same processor can be scheduled by using any $O(1)$-competitive algorithm A that minimizes flow time plus energy on a single processor, such as the algorithm AJC [67] and the algorithm BCP in the very recent work of Bansal, Chan and Pruhs [13]. In analyzing the performance of the resulting algorithm, denoted as CRR-A, we focus on the bounded speed model and compare it against the optimal offline migratory algorithm using maximum speed T. Note that our analysis can also be applied to the infinite speed model, though it is of less interest. We show the following competitiveness of CRR-A for minimizing flow plus energy in the bounded speed model. Let β be the competitive ratio of algorithm A in the single-processor setting.

- For jobs restricted to power-of-2 size, CRR-A using processors with maximum speed T is $O(\log P)$-competitive for flow plus energy. More precisely, the competitive ratio is $(5.966 \log P + 2)\beta$.
- For jobs of arbitrary size, CRR-A is $O(1)$-competitive when using processors with slightly higher maximum speed. Precisely, given any $\epsilon > 0$, let $\eta_\epsilon = (1+\epsilon)^\alpha[(1+\epsilon)^{\alpha-1} + (1-1/\alpha)(2+\epsilon)/\epsilon^2]$. CRR-A is $(5\eta_\epsilon\beta)$-competitive for flow plus energy, when the maximum speed is relaxed to $(1+\epsilon)^2 T$.

Lower bound. For minimizing flow time on multi-processor running at a fixed speed, Leonardi and Raz [69] showed that any online scheduling algorithm (without extra speed) is $\Omega(\log P)$-competitive. This lower bound can be easily adapted to the problem of minimizing flow time plus energy in the bounded speed model. That is, any online algorithm using speed scaling with maximum speed T is $\Omega(\log P)$-competitive for flow plus energy. This lower bound remains valid even if jobs are all with power-of-2 size (because the original proof in [69] uses only such jobs). In other words, CRR-A when

scheduling jobs of power-of-2 size achieves the best possible performance (up to a constant factor). Table 1.2 shows a summary of the non-migratory results on scheduling for flow time with or without energy concern.

Eliminating migration offline. The analysis of CRR-A stems from an offline result to eliminate migration, which is interesting on its own. The cost of eliminating migration has been investigated in the classical setting such as deadline scheduling [24, 60] and flow time scheduling [7, 8]. Our work extends along this line to take energy consumption into consideration. Roughly speaking, to show that CRR-A is $O(\log P)$-competitive for jobs of power-of-2 size, the analysis relies on a transformation of an arbitrary job set to an "m-parallel" job set, which involves modifying the release times forward and backward. Such transformation increases the flow time plus energy of the resulting schedule by a factor of $O(\log P)$, and it cannot be improved by using extra speed. To show that CRR-A is $O(1)$-competitive using extra speed, instead of relying on such transformation, we have an observation that we can focus on some special schedules called "immediate-start" schedules, from which we can derive a much simpler algorithm to construct CRR schedules and exploit the extra speed to show that the flow time plus energy increases by only a constant factor (rather than an $O(\log P)$ factor).

Remarks. It is worth-mentioning that for other scheduling objectives (such as makespan and deadlines), the literature already contains several multi-processor results on dynamic speed scaling in the infinite speed model. In particular, Pruhs et al. [84] and Bunde [21] both studied offline algorithms for the makespan objective. Albers et al. [3] studied online algorithms for jobs with restricted deadlines.

1.1.4 Non-clairvoyant flow-energy scheduling

We initiate the study of flow-energy scheduling in the non-clairvoyant model. In some applications like operating systems, job size is only known when the job finishes. This is referred to as the *non-clairvoyant* model. This is in contrast to the clairvoyant model considered in previous sections, where the size of a job is known at release time.

Non-clairvoyant flow time scheduling. The earlier work of non-clairvoyant scheduling focused on batched jobs (i.e., all jobs have the same release time), and Motwani et al. [74] have shown that the online algorithm Round Robin that shares the processor

equally among all jobs is 2-competitive for flow time. There is a matching lower bound of 2 when the number of jobs is large. Kim and Chwa [66] further showed that a weighted version of Round Robin is also 2-competitive for weighted flow time. For jobs with arbitrary release times, the competitive ratio of deterministic non-clairvoyant algorithm is $\Omega(n^{1/3})$, and the competitive ratio of every randomized algorithm against an oblivious adversary is $\Omega(\log n)$ [74], where n is the number of jobs. The deterministic non-clairvoyant algorithm SETF (shortest elapsed time first) is $(1+\epsilon)$-speed $O(1+\frac{1}{\epsilon})$-competitive [61]. SETF shares the processor equally among all jobs that have been run the least. This result has also been generalized to weighted flow time with the same performance [14,66]. The algorithm Round Robin is $(2+\epsilon)$-speed $O(1+\frac{1}{\epsilon})$-competitive [40]. A randomized version of the Multi-Level Feedback Queue algorithm is $O(\log n)$-competitive [17,61].

Our contributions. We initiate the study of non-clairvoyant flow-energy scheduling, and give competitive online algorithm for batched jobs and arbitrary jobs, respectively.

- We first focus on batched jobs, i.e., when all jobs are released at time 0. For scheduling unweighted jobs, we use the speed function $\text{AJC}^* = (\frac{n}{\alpha-1})^{1/\alpha}$, where n is the total number of active jobs. Coupling with Round Robin, we give an algorithm that is $(2-\frac{1}{\alpha})$-competitive in the infinite speed model and 2-competitive in the bounded speed model. The latter inherits a lower bound of 2 from flow-time scheduling [74]. We can further generalize this algorithm for minimizing weighted flow time plus energy, and the corresponding ratios become $(2-\frac{1}{\alpha})^2$ and 4, respectively.

- We then consider jobs with arbitrary release times. Under the infinite speed model, we analyze the nonclairvoyant algorithm whose job selection policy is Latest Arrival Processsor Sharing (LAPS) [41] and whose speed scaling policy is to run at speed $(1+\frac{3}{\alpha})$ times the number of active jobs. LAPS shares the processor equally among the latest arriving constant fraction of the jobs. We show that this algorithm is $O(\alpha^3)$-competitive for flow plus energy in the infinite speed model. As a remark, non-clairvoyant flow-energy scheduling in the bounded speed model remains open.

- We also extend the study of non-clairvoyant flow-energy scheduling to the model that allows both sleep management and speed scaling (the model is introduced in Section 1.1.2). Consider a processor with one or multiple sleep states. We adapt LAPS to take the static power of the processor into consideration. Under the infinite speed model, this adapted algorithm together with the sleep management algorithm

IdleLonger (see Section 1.1.2) is $O(\alpha^3)$-competitive for flow plus energy.

Analysis. To analyze the algorithm for batched jobs, we first prove that the performance of our algorithm is close to that of a clairvoyant algorithm based on SJF (shortest job first) plus the speed function AJC*, denoted by SJF-AJC*. Then we show that SJF-AJC* is optimal against any offline algorithm for minimizing flow time plus energy. As for jobs with arbitrary release times, we use an amortized local competitiveness argument (see, e.g., [81]). The potential function that we use is an amalgamation of the potential function used in [41] for the fixed speed analysis of LAPS, and the potential functions used for analyzing clairvoyant speed scaling policies. When processors have sleep states, the line of reasoning is similar to that mentioned in Section 1.1.2; we can focus on the working cost (incurred while working on jobs) of the algorithm, and analyze the working cost via a potential analysis of the speed scaling algorithm LAPS.

1.2 Data Stream Algorithms

Data streams are common in many applications such as network monitoring, telecommunications and financial monitoring. For example, data packets in network monitoring and stock transactions in financial monitoring are received and processed in the form of a data stream. In the last decade, algorithms for continuous monitoring of a single massive data stream gained a lot of attention (see [1,77] for a survey), and the main challenge has been how to represent the massive data using limited space, while allowing certain statistics (e.g., item counts, quantiles) to be computed with sufficient accuracy. The space-accuracy tradeoff for representing a single stream has gradually been understood over the years (e.g., [4, 37, 51, 54]). Recently, motivated by large scale networks, the database community is enthusiastic about communication-efficient algorithms for continuous monitoring of multiple, distributed data streams (e.g., [10, 28, 29, 32, 34, 38, 49, 53, 58, 75, 78, 88]). In this thesis, we present new results on both space-efficient data stream algorithms and communication-efficient data stream algorithms.

Data stream models. A data stream is a sequence of items from a totally ordered set U. Each item is associated with a time-stamp recording its creation time. We say that a data stream is *in-order* if items arrive in non-decreasing order of their time-stamps; otherwise, it is *out-of-order*. The statistics on the data stream can be based on the *whole*

data stream [4, 37, 51, 54] or only the recent items [5, 36, 68]. The latter is motivated by the emergence of applications in which only the most recent items in a data stream are important in computing statistics and aggregates. For example, in a stream of stock market data, a software may need to track the moving average of the price of a stock over all observations made in the last hour. In network monitoring, it is useful to monitor the volume of traffic destined to a given node during the most recent window of time. Recent items can be modeled by two types of sliding windows [9, 35]. Let W be the window size, which is a positive integer. The *count-based sliding window* includes the last W items in the data stream, while the *time-based sliding window* includes items whose time-stamps are within the last W time units. The latter assumes that zero or more items can arrive at a time. Note that items in a sliding window will expire and are more difficult to handle than in the whole data stream. For example, counting the frequency of a certain item in the whole stream can be done easily by maintaining a single counter, yet the same problem requires space $\Theta(\frac{1}{\varepsilon}\log^2(\varepsilon W))$ bits even if we allow a relative error of at most ε [35, 46]. In fact, the whole data stream model can be viewed as a special case of the sliding window model with window size being infinite. Also, a count-based window is a special case of a time-based window in which exactly one item arrives at a time.

ε-Approximate queries. We will study algorithms for answering four types of classical ε-approximate queries, defined as follows. Let $0 < \varepsilon < 1$ be the user-specified error bound. For any stream σ, let $c_{j,\sigma}$ and c_σ be the count of item j and all items, respectively, in the current window. Denote $c_j = \sum_\sigma c_{j,\sigma}$ and $c = \sum_\sigma c_\sigma$.

- *Basic Counting.* Return an estimate $\hat{c}$ on the total count c such that $|\hat{c} - c| \leq \varepsilon c$. (Note that this query can be generalized to count data items of a fixed subset $X \subseteq U$; the literature often refers to the special case with $U = \{0, 1\}$ and $X = \{1\}$.)

- *Approximate Counting.* Given any item j, return an estimate $\hat{c}_j$ on the count of item j such that $|\hat{c}_j - c_j| \leq \varepsilon c$.

- *Frequent Items.* Given any user-specified threshold $0 < \phi < 1$, return a set $F \subseteq U$ which includes all items j with $c_j \geq \phi c$ and possibly some items j' with $c_{j'} \geq \phi c - \varepsilon c$.

- *Quantiles.* Given any $0 < \phi < 1$, return an item whose rank is in $[\phi c - \varepsilon c, \phi c + \varepsilon c]$, where the *rank* of an item is its position in the sorted arrangement of all items in the current window. We call any item with rank $\lceil \phi n \rceil$ a *ϕ-quantile*.

1.2.1 Space-efficient data stream algorithms

We study space-efficient data stream algorithms. To answer a statistics on a data stream, we need to handle a huge volume of items from the stream, usually in the order of gigabytes a second, and as the stream passes we have only a few nanoseconds to react to each item. Furthermore, we usually have stringent memory, e.g., the routers used in network monitoring are relatively cheap and have small main memory. Thus, any algorithm for answering the statistics on a data stream must have *extremely fast update and query time*, and use *small memory space*.

We focus on the count-based sliding window model. We introduce a new problem called *Significant One Counting*, which is more flexible in space-accuracy tradeoff than the basic counting problem. We also study finding approximate frequent items.

Basic Counting. The basic counting problem is proposed by Datar et al. [36]. Recall that the problem is to estimate, at any time, the number of 1-bits in a count-based sliding window of size W such that the relative error of the estimate is bounded by ε (see the formal definition in Section 1.2). Datar et al. [36] gave an algorithm for the problem that uses $O(\frac{1}{\varepsilon}\log^2(\varepsilon W))$ bits of memory and has $O(\log W)$ update time and $O(1)$ query time. They also proved that any algorithm for the problem must use $\Omega(\frac{1}{\varepsilon}\log^2(\varepsilon W))$ bits of memory. Later, Gibbons et al. [46] gave an improved algorithm; it uses the same $O(\frac{1}{\varepsilon}\log^2(\varepsilon W))$ bits of memory and has $O(1)$ query time, and the update time is reduced to $O(1)$. Note that this algorithm has the optimal time and space complexity.

Finding frequent items in the whole data stream. Another data stream problem that is closely related to basic counting is the problem of finding frequent items (see the formal definition in Section 1.2). Roughly speaking, for basic counting, we keep track of the count of a particular type of data item, while for frequent items, we need to keep track of the counts of many items. Finding frequent items is useful in many applications. For example, a flow in a network can be modeled as a continuous stream of items such as source/destination addresses of the TCP/UDP packets. Identifying frequent items in such stream has find important applications in network monitoring and data mining. There are many algorithms for identifying frequent items [25, 31, 42, 44, 59, 64, 71] in the whole data stream. Many of these algorithms use random sampling; they make assumptions on the distribution of the item frequencies and the quality of their results are only guaranteed probabilistically. Recently, Karp et al. [64], and independently, Demaine et al. [37],

rediscovered a deterministic algorithm of Misra and Gries [73] (the MG algorithm), which can easily be adapted to find ε-approximate frequent items in the whole data stream without making any assumption on the distribution of the item frequencies. The MG algorithm is simple and elegant; it needs $1/\varepsilon$ simple counters (each occupies a word[6]) to count the items in the stream. The update operation involves only the increment (i.e., +1) or decrement (i.e., -1) of some of these counters.

Finding frequent items over sliding window. As for identifying frequent items over a count-based sliding window, Golab et al. [47] gave some heuristics for the problem and showed empirically that they worked well. Later, Golab et al. [48] gave an algorithm for the problem when the item frequencies are multinomially-distributed. Arasu and Manku [5] gave the first deterministic algorithm for finding ε-approximate frequent items; it supports $O(\frac{1}{\varepsilon}\log\frac{1}{\varepsilon})$ query and update time and uses $O(\frac{1}{\varepsilon}\log^2(\frac{1}{\varepsilon}))$ words of space. Their algorithm divides the sliding window into a collection of possibly overlapping sub-windows with different sizes. As the sliding window shifts, it applies the MG algorithm to each of these sub-windows to find the frequent items in these sub-windows. These sub-windows are organized cleverly into levels so that whenever there is a query on the frequent items, we can traverse these web of sub-windows efficiently to identify the frequent items.

Our contributions. Recall that the basic counting problem requires $\Theta(\frac{1}{\varepsilon}\log^2(\varepsilon W))$ bits of memory. Unfortunately, this memory requirement is still too much to be practical in many applications. We break the $\Omega(\frac{1}{\varepsilon}\log^2(\varepsilon W))$ memory requirement barrier, not by giving a better algorithm (which is impossible), but by defining a new problem. The lower bound of $\Omega(\frac{1}{\varepsilon}\log^2(\varepsilon W))$ bits [36] for basic counting comes from the fact that any correct algorithm should cover all cases; in particular, it needs to guarantee the relative error bound even when the window has only a few 1-bits. However, this is not a requirement for many applications, e.g., telecommunication and financial monitoring. For such applications, if the actual number of 1-bits in the window is small, it is enough to know that it is small and an estimate with a bounded relative error is not required. E.g., in telecommunications, it is usually the case that users with usage above some threshold are charged by usage while the rest are charged by some fixed rate. These applications are called *threshold accounting* in [42], and for them we introduce the following problem:

Significant One Counting. For a stream of bits, let c be the number of

[6]By convention, the memory usage for finding frequent items is stated in terms of word (i.e., $\log N$ bits for a stream of N items).

> 1-bits in the current sliding window (of size W). Given a *threshold* $0 < \theta < 1$, and a *relative error bound* $0 < \varepsilon < 1$, the problem is to return an estimate $\hat{c}$ of the number c of 1-bits such that if $c \geq \theta W$, we have $|\hat{c} - c| \leq \varepsilon c$.

Optimal algorithm for significant one counting. We assume that $\varepsilon \geq \frac{1}{\theta W}$; otherwise the error bound will force us to find the exact number of 1-bits in the stream. Note that when $\theta = 1/W$, our problem becomes the basic counting problem. We first prove a lower bound on the memory of any algorithm for the significant one counting problem. We show that any algorithm for the problem must have memory at least $\Omega(\frac{1}{\varepsilon}\log^2(\frac{1}{\theta}) + \log(\varepsilon\theta W))$ bits. Then, we give an algorithm that has constant update and query time, and uses memory matching the lower bound, i.e., the algorithm has the optimal time and space complexity. This algorithm returns an estimate $\hat{c}$ of the number c of 1-bits in the sliding window such that if $c \geq \theta W$, then $c \leq \hat{c} \leq c + \varepsilon c$, and if $c < \theta W$, then $c \leq \hat{c} \leq c + \varepsilon\theta W$. Note that the algorithm becomes an optimal algorithm for basic counting when we set θ to $1/W$. On the other hand, for any fixed θ, its memory usage is only $O(\frac{1}{\varepsilon} + \log(\varepsilon W))$ bits, which is much smaller than $\Theta(\frac{1}{\varepsilon}\log^2(\varepsilon W))$ bits.

Application to finding frequent items. Our algorithm for significant one counting is simple and easy to implement, and can be used directly to the problem of finding frequent items over a count-based sliding window of size W. Recall that U is the universe of data items. Given the error bound ε and the threshold $\phi \geq \varepsilon$ for the frequent items problem, we can use $|U|$ instances of the algorithm for significant one counting; the total memory usage is $O(|U|(\frac{1}{\varepsilon}\log^2(\frac{1}{\phi}) + \log(\varepsilon\phi W)))$ bits. When the size of the universe U is small, this result is better than the algorithm of Arasu and Manku [5], whose memory usage is $O(\frac{1}{\varepsilon}\log^2(\frac{1}{\varepsilon}))$ words, or equivalently, $O(\frac{1}{\varepsilon}\log^2(\frac{1}{\varepsilon})\log W)$ bits.

Space-optimal algorithm for finding frequent items. We also give another deterministic algorithm for the ε-approximate frequent items problem over sliding window. This algorithm supports $O(\frac{1}{\varepsilon})$ update and query time, and uses $O(\frac{1}{\varepsilon})$ words of space. This substantially improves Arasu and Manku's result. Note that our memory usage is essentially optimal; the number of frequent items can be $\Omega(\frac{1}{\varepsilon})$ in the worst case and thus any algorithm must use $\Omega(\frac{1}{\varepsilon})$ words of memory. More importantly, this algorithm is much simpler than Arasu and Manku's result; our algorithm has about twenty lines of codes. It uses only $O(\frac{1}{\varepsilon})$ simple variables and $O(\frac{1}{\varepsilon})$ queues, and the total length of these queues is $O(\frac{1}{\varepsilon})$. To update these data structures when the window slides, we need only to increment/decrement some of the variables for most cases, and we seldom need to insert

or delete entries in the queues.

Variable-size count-based sliding window. By adapting a technique in [5], we extend our algorithm to identify frequent items in a sliding window whose size can be changed by the user. As pointed out in [5], fixed- and variable-size window capture the essential features of many common types of windows, e.g., time-based sliding window.

1.2.2 Communication-efficient data stream algorithms

Finally, we study communication-efficient algorithms for continuous monitoring of multiple, distributed data streams. The problems studied are best illustrated by the following puzzle. John and Mary work in different laboratories and communicate by telephone only. In a forever-running experiment, John records which devices have an exceptional signal in every 10 seconds. To adjust her devices, Mary at any time needs to keep track of the number of exceptional signals generated by each device of John in the last one hour. John can call Mary every 10 seconds to report the exceptional signals, yet this requires too many calls in an hour and the total message size per hour is linear to the total number N of exceptional signals in an hour. Mary's devices actually allow some small error. Can the number of calls and message size be reduced to $o(N)$, or even poly-log N if a small error (say, 0.1%) is allowed? In general, is there an efficient way to trade accuracy for less communication? It is important to note that the input is given online and Mary needs to know the answers continuously; this makes the problem different from those studied in the classical communication complexity model of Yao [91], in which all inputs are given in advance and the two parties need to compute an answer only once.

Continuous monitoring of multiple, distributed data streams. The formal problem is as follows. We have $k \geq 1$ remote sites each monitoring a data stream, and there is a root (or coordinator) responsible for computing some global statistics. A remote site needs to maintain certain statistics itself, and has to communicate with the root often enough so that the root can compute, at any time, the statistics of the union of all data streams within a certain error. The objective is to minimize the communication. We will study the four classical ε-approximate queries, namely, basic counting, approximate counting, frequent items and quantiles. The communication aspects of data streams introduce several challenging theoretical questions such as what is the optimal communication-accuracy tradeoff for maintaining a particular statistic, and whether two-

way communication is inherently more efficient than one-way communication.

Previous works. Gibbons and Tirthapura [45, 46] are among the first to study the distributed data streams; they considered the case where the root is required to compute the statistics only at fixed times (rather than at any time); see also [71]. Recently, the database literature has a flurry of results on continuous monitoring of distributed data streams [10, 28, 29, 32, 34, 38, 49, 53, 58, 75, 78, 88]. The algorithms studied can be classified into two types: *one-way* algorithms only allow messages sent from each remote site to the root, and *two-way* algorithms allow bi-directional communication between the root and each site. One-way algorithms are often very simple as a remote site has little information and all it can do is to update the root when its local statistics deviate significantly from those previously sent. On the other hand, most two-way algorithms are complicated and often involve non-trivial heuristics. It is commonly believed in the database community that two-way algorithms are more efficient; however, for most existing two-way algorithms, their worst-case communication costs are still waiting for rigorous mathematical analysis, and existing works often rely on experimental results.

The literature contains several results on the mathematical analysis of the worst-case performance of one-way algorithms. They are all for the easier whole data stream setting. Keralapura et al. [65] studied the thresholded-count problem, which leads to an algorithm for basic counting with communication cost $O(\frac{1}{\varepsilon}\log n)$ words per stream, where n is the number of items in that stream[7]. Cormode et al. [29] gave a one-way algorithm for quantiles with communication cost $O(\frac{1}{\varepsilon^2}\log n)$ words per stream. They also showed how to handle frequent items via a reduction to quantiles, the communication cost remains the same. More recently, Yi and Zhang [93] improved the communication cost of frequent items to $O(\frac{1}{\varepsilon}\log n)$ words, and that of quantiles to $O(\frac{1}{\varepsilon}\log^2(\frac{1}{\varepsilon})\log n)$ words.

There have been attempts to devise heuristics to extend the above whole-data-stream algorithms to sliding windows, yet not much has been known about their worst-case performance. For example, Cormode et al. [29] have extended their algorithms for quantiles and frequent items to sliding window. They believed that the communication cost would only have a mild increase, but they did not give supporting analysis. The analysis of sliding-window algorithms is more difficult because the expiry of items destroys some monotonic property that is important to the analysis for whole data stream.

[7] If based on $k > 1$ remote sites, the total communication cost can be stated as $O(\frac{k}{\varepsilon}\log\frac{N}{k})$, where N is the total number of items received over all streams.

<table>
<tr><th></th><th>Basic Counting
(bits)</th><th>Approixmate Counting/
Frequent items (words)</th><th>Quantiles
(words)</th></tr>
<tr><td rowspan="2">Whole data stream</td><td rowspan="2">$O(\frac{1}{\varepsilon}\log n)$ words [65]
$\Theta(\frac{1}{\varepsilon}\log(\varepsilon n))$ bits</td><td>$O(\frac{1}{\varepsilon}\log n)$ [93]
$O(\frac{1}{\varepsilon}\log n)$</td><td>$O(\frac{1}{\varepsilon}\log^2(\frac{1}{\varepsilon})\log n)$ [93]
$O(\frac{1}{\varepsilon^2}\log n)$</td></tr>
<tr><td colspan="2">$\Omega(\frac{1}{\varepsilon}\log(\varepsilon n))$</td></tr>
<tr><td rowspan="2">Sliding window</td><td rowspan="2">$\Theta(\frac{1}{\varepsilon}\log(\varepsilon B))$</td><td>$O(\frac{1}{\varepsilon}\log B)$</td><td>$O(\frac{1}{\varepsilon^2}\log B)$</td></tr>
<tr><td colspan="2">$\Omega(\frac{1}{\varepsilon}\log(\varepsilon B))$</td></tr>
<tr><td rowspan="2">Sliding window &
Out-of-order streams</td><td>$O((\frac{W}{W-\tau})\frac{1}{\varepsilon}\log(\varepsilon B))$</td><td>$O((\frac{W}{W-\tau})\frac{1}{\varepsilon}\log B)$</td><td>$O((\frac{W}{W-\tau})\frac{1}{\varepsilon^2}\log B)$</td></tr>
<tr><td>$\Omega(\max\{\frac{W}{W-\tau}, \frac{1}{\varepsilon}\log(\varepsilon B)\})$</td><td colspan="2">$\Omega(\max\{\frac{W}{W-\tau}, \frac{1}{\varepsilon}\log(\varepsilon B)\})$</td></tr>
</table>

Table 1.3: Communication cost for each remote site.

Our contributions. We give the first mathematical analysis of the communication cost in the sliding window model. We derive lower bounds on the worst-case communication cost of any two-way algorithm (and hence any one-way algorithm) for answering the four types of ε-approximate queries. More interestingly, we analyze some common-sense algorithms that use one-way communication only and prove that their communication costs match or almost match the corresponding lower bounds. These results demonstrate optimal or near optimal communication-accuracy tradeoffs for supporting these queries over the sliding window. Furthermore, our work reveals that two-way algorithms could not be much better than one-way algorithms (at least in the worst case).

Below we state the lower and upper bounds precisely. Consider a sliding window of W time steps and let B be the maximum number of items arriving at each stream within a window. We prove that for basic counting, each remote site needs to communicate $\Omega(\frac{1}{\varepsilon}\log(\varepsilon B))$ bits with the root in every consecutive interval of $2W$ time units in the worst case, and $\Omega(\frac{1}{\varepsilon}\log(\varepsilon B))$ words for the other three queries. For upper bounds, our analysis shows that basic counting requires $O(\frac{1}{\varepsilon}\log(\varepsilon B))$ bits within any window, and approximate counting $O(\frac{1}{\varepsilon}\log B)$ words. Note that the estimates given by approximate counting is sufficient to find frequent items, and thus the latter problem has the same communication cost. For quantiles, it takes $O(\frac{1}{\varepsilon^2}\log B)$ words. See the second row (sliding window) of Table 1.3 for a summary. Note that our algorithms do not need to know the value of B in advance, it is only needed in the analysis.

As mentioned before, sliding-window algorithms can be applied to handle the special case of whole data streams. The first row of Table 1.3 includes our results as well as

previous results on whole data streams (where n denotes the number of items in an entire stream). Except quantiles, our results are better or at least match the previous ones.

Our algorithms can be readily applied to out-of-order streams. For an out-of-order stream, we say that the stream has *tardiness* τ if any item with time-stamp t must arrive within τ time units from t, i.e., at any time in $[t, t+\tau]$. Without loss of generality, we assume that $\tau \in \{0, 1, 2, \ldots, W-1\}$ (if an item time-stamped at t arrives after $t+W-1$, it has already expired and can be ignored). Note that in-order data streams have tardiness 0. The previous lower bounds for in-order streams are all valid in the out-of-order setting. In addition, we obtain a lower bound related to τ, namely, $\Omega(\frac{W}{W-\tau})$ bits for basic counting and $\Omega(\frac{W}{W-\tau})$ words for the other three problems. Regarding upper bounds, our algorithms when applied to out-of-order streams with tardiness τ will just increase the communication cost by a factor of $\frac{W}{W-\tau}$. The results are summarized in the last row of Table 1.3.

1.3 Organization

Chapters 2 to 5 give our results on online flow-energy scheduling. Chapter 2 presents improved algorithms for a single processor. Chapter 3 introduces the model that allows both sleep management and speed scaling, and gives the first competitive algorithm in such model. Chapter 4 presents the first non-migratory competitive algorithm on multiple processors and analyzes it for processors with or without extra maximum speed. Chapter 5 initiates the study of non-clairvoyant flow-energy scheduling and gives competitive algorithms for batched jobs and arbitrary job sets, respectively.

Space-efficient and communication-efficient data stream algorithms are discussed in Chapters 6 and 7, respectively. In Chapter 6, we propose the new problem Significant One Counting, and give optimal algorithm for the problem. We also present a space-optimal algorithm for finding frequent items over a sliding window. In Chapter 7, we study continuous monitoring of multiple and distributed data streams over a time-based sliding window, and give algorithms with optimal or nearly optimal communication-accuracy tradeoff for basic counting, approximate counting, frequent items and quantiles.

In Chapter 8, we give the conclusion and future work for both online flow-energy scheduling and data stream algorithms.

Chapter 2

Flow-Energy Scheduling on Single Processor

In this chapter, we consider flow-energy scheduling on single processor and give improved algorithms for both the infinite and bounded speed models. The formal problem is defined as follows. We consider a job set J to be scheduled on a processor whose speed can be varied between 0 and T, where T is the speed bound. In the infinite speed model, we have $T = \infty$. When running at speed s, the processor processes s units of work and consumes s^α units of energy in each unit of time, where $\alpha \geq 2$. Preemption is allowed; a preempted job can resume at the point of preemption. We use $r(j)$ and $p(j)$ to denote the release time and size of a job j in J, respectively. Consider a particular time t in a schedule of J. For any job j in J, we let $q(j)$ denote its remaining work at t, and say it is *active* if $r(j) \leq t$ and $q(j) > 0$. The *flow time* $F(j)$ of job j is the time elapsed since j arrives until j is completed. The *total flow time* is given by $F = \sum_{j \in J} F(j)$. Note that it is equivalent to $F = \int_0^\infty n(t)\,\mathrm{d}t$, where $n(t)$ denotes the number of active jobs at time t, and the energy consumption is $E = \int_0^\infty s(t)^\alpha\,\mathrm{d}t$, where $s(t)$ is the speed of the processor at time t. Our aim is to minimize the total flow time plus energy, $G = F + E$. Each of E, F and G is the integration over all time from 0 to ∞; we call the integration over a period of time to be the amount of their respective value *incurred* during that period.

In Section 2.1, we introduce the new speed function AJC and the algorithm SRPT-AJC. AJC is a more stable speed function than existing speed functions [12,16]; existing speed functions change continuously, which is undesirable practically, while AJC depends on the

count of active jobs and therefore changes only at job arrival or completion. We also give a brief overview of our approach to analyzing SRPT-AJC. Section 2.2 gives the analysis of SRPT-AJC in the infinite speed model, showing that the competitive ratio of SRPT-AJC for flow plus energy is $\beta_1 = 2/(1 - \frac{\alpha-1}{\alpha^{\alpha/(\alpha-1)}})$. Section 2.3 adapts the analysis in Section 2.2 to the bounded speed model, showing that the competitive ratio of SRPT-AJC for flow plus energy is $\beta_2 = 2(\alpha+1)/(\alpha - \frac{\alpha-1}{(\alpha+1)^{1/(\alpha-1)}})$. These results improve the best competitive ratios in the infinite speed model [16] and the bounded speed model [12], respectively. The improvement is more significant for large α, since β_1 and β_2 tend to $2\alpha/\ln\alpha$, while the competitive ratios of both the algorithms in [12, 16] tend to $2(\alpha/\ln\alpha)^2$. Technically speaking, the analysis of existing algorithms requires indirect comparison via a notion called fractional flow. In contrast, we divide the time into "stable intervals", and directly compare the flow time of the online algorithm against an optimal offline algorithm in each interval. This makes the analysis tighter.

2.1 Speed Function AJC and Algorithm SRPT-AJC

AJC and SRPT-AJC. The speed function AJC (*active job count*) is defined as $\min\{T, n(t)^{1/\alpha}\}$, where $n(t)$ is the number of active jobs at time t. The algorithm SRPT-AJC uses AJC with the SRPT (shortest remaining processing time) policy: at any time, select the job with the smallest remaining work to run; tie is broken arbitrarily.

We first explain that SRPT gives the best job selection as follows.

Lemma 2.1. *Consider a job sequence J. Suppose a schedule S for J uses a speed function f. Then, among all schedules of J using the speed function f, the one selecting jobs in accordance with SRPT incurs the least total flow time.*

Proof. We modify S in multiple steps to a SRPT schedule. In each step as shown below, the new schedule S' has total flow time reduced. Then the lemma follows.

Let t be the first time when S does not follow SRPT, running j_i instead of the shortest remaining work job j_j. S' differs from S during the time intervals since t when S runs either job: j_j is run to completion before j_i, using the same speed function. j_j thus completes in S' earlier than j_i does in S. So the sum of their completion times, and thus the total flow time, is less in S' than in S. □

Overview of analysis. To analyze SRPT-AJC, we compare it against an optimal offline schedule OPT. By Lemma 2.1, OPT uses the SRPT policy as well. Consider any time t. Let $G_{\mathrm{a}}(t)$ and $G_{\mathrm{o}}(t)$ be the flow time plus energy incurred up to t by SRPT-AJC and OPT, respectively. We drop the parameter t when it refers to the current time. Our analysis exploits amortization and potential functions (e.g., [16, 23]): if there is a potential function $\Phi(t)$ and a value β such that the followings hold, then SRPT-AJC is β-competitive.

- *Boundary condition:* Φ is initially 0 and finally non-negative.
- *Arrival condition:* When a job is released, Φ does not increase.
- *Running condition:* At any other time, the rate of change of G_{a} plus that of Φ is no more than β times the rate of change of G_{o}, i.e., $\frac{\mathrm{d}G_{\mathrm{a}}}{\mathrm{d}t} + \frac{\mathrm{d}\Phi}{\mathrm{d}t} \leq \beta \frac{\mathrm{d}G_{\mathrm{o}}}{\mathrm{d}t}$.

2.2 Analysis for Infinite Speed Model

In this section, we consider the infinite speed model (i.e., $T = \infty$) and show the following theorem.

Theorem 2.2. SRPT-AJC *is β_1-competitive for flow time plus energy in the infinite speed model, where* $\beta_1 = 2/(1 - \frac{\alpha-1}{\alpha^{\alpha/(\alpha-1)}})$.

We first give a potential function Φ, whose design is motivated by the work in [12,16]. We will choose β to be $2/(1 - \frac{\alpha-1}{\alpha^{\alpha/(\alpha-1)}})$. Then we show that the boundary, arrival and running conditions hold, and thus can conclude the competitiveness of SRPT-AJC.

Potential function $\Phi(t)$. Consider any time t. For any $q \geq 0$, let $n_{\mathrm{a}}(q)$ denote the current number of active jobs with remaining work at least q in SRPT-AJC, and similarly $n_{\mathrm{o}}(q)$ for OPT. So $n_{\mathrm{a}}(0)$ and $n_{\mathrm{o}}(0)$ are the total number of active jobs in the schedules, which we abbreviate as n_{a} and n_{o}, respectively. It is useful to consider $n_{\mathrm{a}}(q)$ and $n_{\mathrm{o}}(q)$ as functions of q which change when a job arrives or runs for a while (see Figure 2.1). We define the potential function as

$$\Phi(t) = \eta \int_0^\infty \phi(q)\,\mathrm{d}q \ \text{, where } \phi(q) = \left(\sum_{i=1}^{n_{\mathrm{a}}(q)} i^{1-1/\alpha}\right) - n_{\mathrm{a}}(q)^{1-1/\alpha} n_{\mathrm{o}}(q) \text{ and } \eta = \frac{2}{1 - \frac{\alpha-1}{\alpha^{\alpha/(\alpha-1)}}}$$

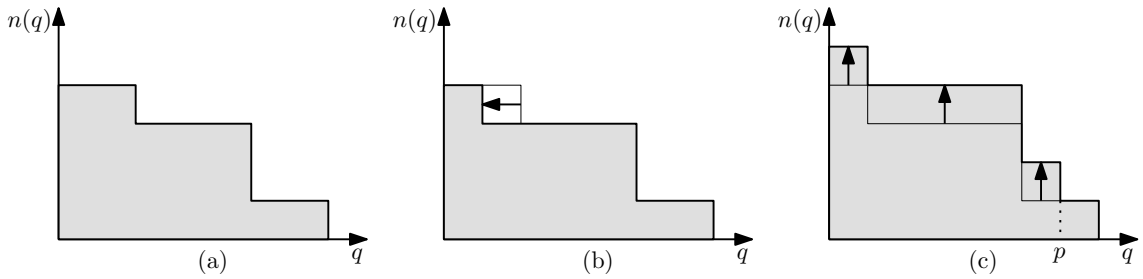

Figure 2.1: (a) At any time, $n_{\mathrm{a}}(q)$ or $n_{\mathrm{o}}(q)$ (denoted by $n(q)$ above) is a step function containing unit height stripes, the area under $n(q)$ is the total remaining work. (b) If we run a job selected with SRPT at speed s for a period of time Δ, the top stripe shrinks by $s\Delta$. (c) When a job of size p is released, $n(q)$ increases by 1 for all $q \le p$.

At $t = 0$ or ∞, no job remains, so $\Phi = 0$. The integration of the first term of $\phi(q)$ is proportional to the total flow time plus energy of SRPT-AJC after t if no more job arrives (which is $2\int_0^\infty \sum_{i=1}^{n_{\mathrm{a}}(q)} i^{1-1/\alpha}\,\mathrm{d}q$), paying the cost once OPT completes all jobs. The second term of $\phi(q)$ makes the arrival condition hold.

Lemma 2.3. *When a job arrives, the change of* Φ *is non-positive.*

Proof. Suppose a job j arrives. For $q > p(j)$, $n_{\mathrm{a}}(q)$ and $n_{\mathrm{o}}(q)$, and hence $\phi(q)$, are unchanged. For $q \le p(j)$, both $n_{\mathrm{a}}(q)$ and $n_{\mathrm{o}}(q)$ increase by 1 (see Figure 2.1(c)). Thus the first term of $\phi(q)$ increases by $(n_{\mathrm{a}}(q)+1)^{1-1/\alpha}$. The increase of the term $n_{\mathrm{a}}(q)^{1-1/\alpha} n_{\mathrm{o}}(q)$ can be interpreted in two-steps: (i) increase to $(n_{\mathrm{a}}(q)+1)^{1-1/\alpha} n_{\mathrm{o}}(q)$, and (ii) increase to $(n_{\mathrm{a}}(q)+1)^{1-1/\alpha}(n_{\mathrm{o}}(q)+1)$. The increase in step (ii), i.e., $(n_{\mathrm{a}}(q)+1)^{1-1/\alpha}$, covers the increase in the first term of $\phi(q)$, so $\phi(q)$ does not increase. □

Running condition. The rest of this section proves the following lemma.

Lemma 2.4. *When no job arrives,* $\frac{\mathrm{d}G_{\mathrm{a}}}{\mathrm{d}t} + \frac{\mathrm{d}\Phi}{\mathrm{d}t} \le \beta_1 \frac{\mathrm{d}G_{\mathrm{o}}}{\mathrm{d}t}$ *where* $\beta_1 = 2/(1 - \frac{\alpha-1}{\alpha^{\alpha/(\alpha-1)}})$.

Consider any time t. Let s_{a} and s_{o} be the speed of SRPT-AJC and OPT, respectively, and q_{a} and q_{o} be the remaining work of the job they run. We divide the time line into *stable intervals* by breaking it when the following events occur.

- A job arrives, or is completed by either SRPT-AJC or OPT.
- The speed s_{a} or the job chosen by SRPT-AJC changes.

- The speed s_{o} or the job chosen by OPT changes.
- Either $n_{\mathrm{o}}(q_{\mathrm{a}})$ or $n_{\mathrm{a}}(q_{\mathrm{o}})$ changes.

Φ does not increase on job arrival (Lemma 2.3), and is unchanged on other events above ($\phi(q)$ changes for only single q). Therefore, we focus on its change in a stable interval.

We first bound $\frac{\mathrm{d}\Phi}{\mathrm{d}t}$. Consider a stable interval of length $\mathrm{d}t$. We analyze $\frac{\mathrm{d}\Phi}{\mathrm{d}t}$ as if $\phi(q)$ is changed in two steps: (i) $n_{\mathrm{a}}(q)$ decreases due to the execution of SRPT-AJC. (ii) $n_{\mathrm{o}}(q)$ decreases due to the execution of OPT. We denote these changes by $\frac{\mathrm{d}\Phi_1}{\mathrm{d}t}$ and $\frac{\mathrm{d}\Phi_2}{\mathrm{d}t}$, respectively. Then $\frac{\mathrm{d}\Phi}{\mathrm{d}t} = \frac{\mathrm{d}\Phi_1}{\mathrm{d}t} + \frac{\mathrm{d}\Phi_2}{\mathrm{d}t}$.

Lemma 2.5. *Consider a stable interval of length* $\mathrm{d}t$. (i) $\frac{\mathrm{d}\Phi_1}{\mathrm{d}t} \le \eta(n_{\mathrm{o}} - n_{\mathrm{a}})$; (ii) $\frac{\mathrm{d}\Phi_2}{\mathrm{d}t} \le \eta n_{\mathrm{a}}^{1-1/\alpha} s_{\mathrm{o}}$.

Proof. (i) Consider SRPT-AJC. $n_{\mathrm{a}}(q)$ decreases from n_{a} to $n_{\mathrm{a}}-1$ for all $q \in [q_{\mathrm{a}} - s_{\mathrm{a}}\mathrm{d}t, q_{\mathrm{a}}]$ (see Figure 2.1(b)). For any $q \in [q_{\mathrm{a}} - s_{\mathrm{a}}\mathrm{d}t, q_{\mathrm{a}}]$, $\phi(q)$ is changed by:

$$\begin{aligned}&\left(\sum_{i=1}^{n_{\mathrm{a}}-1} i^{1-1/\alpha}\right) - (n_{\mathrm{a}}-1)^{1-1/\alpha} n_{\mathrm{o}}(q) - \left(\sum_{i=1}^{n_{\mathrm{a}}} i^{1-1/\alpha}\right) + n_{\mathrm{a}}^{1-1/\alpha} n_{\mathrm{o}}(q) \\ &\quad \le -n_{\mathrm{a}}^{1-1/\alpha} + n_{\mathrm{a}}^{-1/\alpha} n_{\mathrm{o}}(q) \le -n_{\mathrm{a}}^{1-1/\alpha} + n_{\mathrm{a}}^{-1/\alpha} n_{\mathrm{o}} \ ,\end{aligned}$$

where the first inequality is due to $n^{1-1/\alpha} - (n-1)^{1-1/\alpha} \le n^{-1/\alpha}$ for any $n \ge 1$, and the second holds since $n_{\mathrm{o}}(q) \le n_{\mathrm{o}}$. Recall that $s_{\mathrm{a}} = n_{\mathrm{a}}^{1/\alpha}$. Integrating over all q,

$$\frac{\mathrm{d}\Phi_1}{\mathrm{d}t} \le \eta(-n_{\mathrm{a}}^{1-1/\alpha} + n_{\mathrm{a}}^{-1/\alpha} n_{\mathrm{o}})(n_{\mathrm{a}}^{1/\alpha}) = \eta(n_{\mathrm{o}} - n_{\mathrm{a}}) \ .$$

(ii) Consider OPT. Similarly to (i), $\phi(q)$ is changed only for $q \in [q_{\mathrm{o}} - s_{\mathrm{o}}\mathrm{d}t, q_{\mathrm{o}}]$. Note that $n_{\mathrm{a}}(q) \le n_{\mathrm{a}}$ for all q. Then for any $q \in [q_{\mathrm{o}} - s_{\mathrm{o}}\mathrm{d}t, q_{\mathrm{o}}]$, $\phi(q)$ is changed by

$$-n_{\mathrm{a}}(q)^{1-1/\alpha}(n_{\mathrm{o}}(q)-1) + n_{\mathrm{a}}(q)^{1-1/\alpha} n_{\mathrm{o}}(q) = n_{\mathrm{a}}(q)^{1-1/\alpha} \le n_{\mathrm{a}}^{1-1/\alpha} \ .$$

Integrating over all q, we have $\frac{\mathrm{d}\Phi_2}{\mathrm{d}t} \le \eta n_{\mathrm{a}}^{1-1/\alpha} s_{\mathrm{o}}$. □

Proof of Lemma 2.4. By Lemma 2.5 (ii), we have $\frac{\mathrm{d}\Phi_2}{\mathrm{d}t} \le \eta n_{\mathrm{a}}^{1-1/\alpha} s_{\mathrm{o}}$. We introduce a constant $\mu > 0$ and apply the Young's Inequality [52], which is formally stated in Lemma 2.6,

by setting $f(x) = x^{\alpha-1}$, $f^{-1}(x) = x^{1/(\alpha-1)}$, $g = s_{\mathrm{o}}/\mu$, $h = n_{\mathrm{a}}^{1-1/\alpha}\mu$. Then, we have

$$\frac{\mathrm{d}\Phi_2}{\mathrm{d}t} \le \eta(n_{\mathrm{a}}^{1-1/\alpha}\mu)\left(\frac{s_{\mathrm{o}}}{\mu}\right) \le \eta\Big(\int_0^{\frac{s_{\mathrm{o}}}{\mu}} x^{\alpha-1}\,\mathrm{d}x + \int_0^{n_{\mathrm{a}}^{1-1/\alpha}\mu} x^{1/(\alpha-1)}\,\mathrm{d}x\Big) \le \frac{\eta s_{\mathrm{o}}^{\alpha}}{\alpha\mu^{\alpha}} + \eta(1-\frac{1}{\alpha})\mu^{\alpha/(\alpha-1)}n_{\mathrm{a}}\ .$$

Since $s_{\mathrm{a}} = n_{\mathrm{a}}^{1/\alpha}$, $\frac{\mathrm{d}G_{\mathrm{a}}}{\mathrm{d}t} = n_{\mathrm{a}} + s_{\mathrm{a}}^{\alpha} = 2n_{\mathrm{a}}$. Also, $\frac{\mathrm{d}G_{\mathrm{o}}}{\mathrm{d}t} = n_{\mathrm{o}} + s_{\mathrm{o}}^{\alpha}$. Since $\frac{\mathrm{d}\Phi}{\mathrm{d}t} = \frac{\mathrm{d}\Phi_1}{\mathrm{d}t} + \frac{\mathrm{d}\Phi_2}{\mathrm{d}t}$, by Lemma 2.5 (i), we have

$$\frac{\mathrm{d}G_{\mathrm{a}}}{\mathrm{d}t} + \frac{\mathrm{d}\Phi}{\mathrm{d}t} \le 2n_{\mathrm{a}} + \eta\Big(n_{\mathrm{o}} - n_{\mathrm{a}} + \big(1-\frac{1}{\alpha}\big)\mu^{\alpha/(\alpha-1)}n_{\mathrm{a}} + \frac{s_{\mathrm{o}}^{\alpha}}{\alpha\mu^{\alpha}}\Big)\ . \tag{2.1}$$

Lemma 2.4 follows with $\mu = \alpha^{-1/\alpha}$ and the fact that $\beta_1 = \eta = 2/(1 - \frac{\alpha-1}{\alpha^{\alpha/(\alpha-1)}})$. □

Below is the formal statement of Young's Inequality, which is used in the proof of Lemma 2.4.

Lemma 2.6 (Young's Inequality [52])**.** *Let f be any real-value, continuous and strictly increasing function f such that $f(0) = 0$. Then, for all $g, h \ge 0$, $\int_0^g f(x)\,\mathrm{d}x + \int_0^h f^{-1}(x)\,\mathrm{d}x \ge gh$, where f^{-1} is the inverse function of f.*

2.3 Analysis for Bounded Speed Model

In this section, we adapt the analysis in Section 2.2 to the bounded speed model and show the following theorem.

Theorem 2.7. SRPT-AJC *is β_2-competitive for flow time plus energy in the bounded speed model, where $\beta_2 = 2(\alpha+1)/(\alpha - \frac{\alpha-1}{(\alpha+1)^{1/(\alpha-1)}})$.*

We use the same potential function as in Section 2.2 while setting $\eta = 2/(1 - \frac{1-1/\alpha}{(\alpha+1)^{1/(\alpha-1)}})$. Most of the analysis in Section 2.2 still holds, except Lemma 2.5 (i). As the speed used might be T instead of $n^{1/\alpha}$ if n is large, the decrease $\frac{\mathrm{d}\Phi_1}{\mathrm{d}t}$ is not sufficient to guarantee the original competitive ratio. Nevertheless, we observe that at any time t, the difference $(n_{\mathrm{a}}(t) - n_{\mathrm{o}}(t))$ can be upper bounded by T^{α} (Lemma 2.8). Then the competitiveness of SRPT-AJC is only slightly worse as shown in Theorem 2.7.

Lemma 2.8. *At any time t, the number of active jobs in* SRPT-AJC *and* OPT *satisfy $n_{\mathrm{a}}(t) - n_{\mathrm{o}}(t) \le T^{\alpha}$.*

Proof. Consider a time t. The non-trivial case is $n_{\mathrm{a}}(t) > T^{\alpha}$. Let t_0 be the last time before t that $n_{\mathrm{a}}(t_0) \leq T^{\alpha}$. Consider the interval $[t_0, t]$, where SRPT-AJC runs at full speed. Suppose k jobs arrive, and OPT completes x of them, so $n_{\mathrm{o}} \geq k - x$. Since running SRPT at the full speed T maximizes the number of jobs completed at any time [86], SRPT-AJC also completes at least x jobs. So n_{a} increases by at most $k - x \leq n_{\mathrm{o}}$ after time t_0, and the lemma follows. □

Proof of Theorem 2.7. The boundary and arrival conditions still hold. It remains to establish the running condition. Its analysis is split into two cases.

Case 1: $n_{\mathrm{a}} \leq T^{\alpha}$. The arguments in Section 2.2 still hold, leading to Inequality (2.1). Setting $\mu = (\alpha+1)^{-1/\alpha}$ and recalling that $\eta = 2/(1 - \frac{1-1/\alpha}{(\alpha+1)^{1/(\alpha-1)}})$, (2.1) implies $\frac{\mathrm{d}G_{\mathrm{a}}}{\mathrm{d}t} + \frac{\mathrm{d}\Phi}{\mathrm{d}t} \leq \eta n_{\mathrm{o}} + (1+1/\alpha)\eta s_{\mathrm{o}}^{\alpha} \leq \beta_2 \frac{\mathrm{d}G_{\mathrm{o}}}{\mathrm{d}t}$, where the last inequality is due to $\beta_2 = (1+1/\alpha)\eta$.

Case 2: $n_{\mathrm{a}} > T^{\alpha}$. The arguments in Section 2.2 hold, except Lemma 2.5 (i). We still have $\frac{\mathrm{d}G_{\mathrm{o}}}{\mathrm{d}t} = n_{\mathrm{o}} + s_{\mathrm{o}}^{\alpha}$ and $\frac{\mathrm{d}G_{\mathrm{a}}}{\mathrm{d}t} = n_{\mathrm{a}} + T^{\alpha} < 2n_{\mathrm{a}}$. For $\frac{\mathrm{d}\Phi_1}{\mathrm{d}t}$,

$$\frac{\mathrm{d}\Phi_1}{\mathrm{d}t} \leq \eta(-n_{\mathrm{a}}^{1-1/\alpha} + n_{\mathrm{a}}^{-1/\alpha} n_{\mathrm{o}})T = \eta(n_{\mathrm{o}} - n_{\mathrm{a}})\Big(\frac{T^{\alpha}}{n_{\mathrm{a}}}\Big)^{1/\alpha} \ .$$

If $n_{\mathrm{o}} \geq n_{\mathrm{a}}$, $\frac{\mathrm{d}\Phi_1}{\mathrm{d}t} \leq \eta(n_{\mathrm{o}} - n_{\mathrm{a}})$ as before (since $T^{\alpha}/n_{\mathrm{a}} \leq 1$), leading to (2.1), so the arguments in Case 1 apply. If $n_{\mathrm{o}} < n_{\mathrm{a}}$, Lemma 2.8 implies

$$\frac{\mathrm{d}\Phi_1}{\mathrm{d}t} \leq -\eta(n_{\mathrm{a}} - n_{\mathrm{o}})\Big(\frac{T^{\alpha}}{n_{\mathrm{a}}}\Big)^{1/\alpha} \leq \frac{-\eta(n_{\mathrm{a}} - n_{\mathrm{o}})^{1+1/\alpha}}{n_{\mathrm{a}}^{1/\alpha}} \ .$$

Note that for the convex function $f(x) = x^{1+1/\alpha}$, we have $f(n_{\mathrm{a}} - n_{\mathrm{o}}) \geq f(n_{\mathrm{a}}) - f'(n_{\mathrm{a}})\, n_{\mathrm{o}}$, which is equivalent to $(n_{\mathrm{a}} - n_{\mathrm{o}})^{1+1/\alpha} \geq n_{\mathrm{a}}^{1+1/\alpha} - (1+1/\alpha) n_{\mathrm{a}}^{1/\alpha} n_{\mathrm{o}}$. We thus have

$$\frac{\mathrm{d}\Phi_1}{\mathrm{d}t} \leq \frac{-\eta(n_{\mathrm{a}}^{1+1/\alpha} - (1+1/\alpha) n_{\mathrm{a}}^{1/\alpha} n_{\mathrm{o}})}{n_{\mathrm{a}}^{1/\alpha}} = \eta\big((1+\frac{1}{\alpha})n_{\mathrm{o}} - n_{\mathrm{a}}\big) \ , \text{ and}$$
$$\frac{\mathrm{d}G_{\mathrm{a}}}{\mathrm{d}t} + \frac{\mathrm{d}\Phi}{\mathrm{d}t} \leq 2n_{\mathrm{a}} + \eta\Big((1+\frac{1}{\alpha})n_{\mathrm{o}} - n_{\mathrm{a}} + \big(1 - \frac{1}{\alpha}\big)\mu^{\alpha/(\alpha-1)} n_{\mathrm{a}} + \frac{s_{\mathrm{o}}^{\alpha}}{\alpha\mu^{\alpha}}\Big) \ .$$

Setting $\mu = (\alpha+1)^{-1/\alpha}$ and recalling that $\eta = 2/(1 - \frac{1-1/\alpha}{(\alpha+1)^{1/(\alpha-1)}})$ and $\beta_2 = (1+1/\alpha)\eta$, this implies

$$\frac{\mathrm{d}G_{\mathrm{a}}}{\mathrm{d}t} + \frac{\mathrm{d}\Phi}{\mathrm{d}t} \leq (1+\frac{1}{\alpha})\eta(n_{\mathrm{o}} + s_{\mathrm{o}}^{\alpha}) = \beta_2 \frac{\mathrm{d}G_{\mathrm{o}}}{\mathrm{d}t} \ .$$ □

Chapter 3

Flow-Energy Scheduling with Sleep States

In this chapter, we initiate the study of flow-energy scheduling that exploits both speed scaling and sleep states. A processor has two states, *awake* and *sleep*. The awake state always requires a static power $\sigma > 0$. More specifically, a processor in the awake state consumes a dynamic power that depends on its speed, as well as the static power σ (see Section 3.1 for the formal definition of the model). To have zero energy usage, the processor must enter the sleep state, but a wake-up back to the awake state requires $\omega > 0$ energy. We consider processor that can have multiple intermediate sleep states, which demand some static power but less wake-up energy. For such processor, job scheduling requires two components: a *sleep management algorithm* to determine when to sleep or work, and a *speed scaling algorithm* to determine which job and at what speed to run. Adding a sleep state actually changes the nature of speed scaling. Assuming no sleep state, running a job slower is a natural way to save energy. Now one can also save energy by sleeping more and working faster later. It is even more complicated when flow is concerned. Prolonging a sleeping period by delaying job execution can save energy, yet it also incurs extra flow. Striking a balance is not trivial.

Section 3.1 defines the model formally. Sections 3.2 and 3.3 focus on the infinite speed model. In Section 3.2, we give a sleep management algorithm called IdleLonger, which works for a processor with one or multiple levels of sleep states. In Section 3.3, we give the speed scaling algorithm SAJC by adapting the algorithm SRPT-AJC in Chapter 2. We

also show that SAJC together with IdleLonger is $O(1)$-competitive for the infinite speed model. Finally, Section 3.4 extends the results to the bounded speed model, showing that SAJC together with IdleLonger is still $O(1)$-competitive for the bounded speed model.

Analysis framework. Apparently, a sleep management algorithm and a speed scaling algorithm would affect each other; analyzing their relationship and their total cost could be a complicated task. Interestingly, our results stem from the fact that we can isolate the analysis of these algorithms. We divide the total cost (flow plus energy) into two parts, *working cost* (incurred while working on jobs) and *inactive cost* (incurred at other times). We upper bound the inactive cost of IdleLonger independent of the speed scaling algorithm. For the working cost, although it does depend on both algorithms, our potential analysis of the speed scaling algorithms reveals that the dependency on the sleep management algorithm is limited to a simple quantity called *inactive flow*, which is the flow part of the inactive cost. Intuitively, large inactive flow means many jobs are delayed due to prolonged sleep, and hence the processor has to work faster later to catch up, incurring a higher working cost. It is easy to minimize inactive flow at the sacrifice of the energy part of the inactive cost. IdleLonger is designed to maintain a good balance between them. In conclusion, coupling IdleLonger with SAJC, we obtain competitive algorithms for flow plus energy.

3.1 Model and Notations

Speed and power. We first consider the setting with one sleep state. At any time, a processor is in either the *awake* state or the *sleep* state. In the former, the processor can run at any speed $s \geq 0$ and demands power in the form $s^\alpha + \sigma$, where $\alpha > 1$ and $\sigma > 0$ are constants. We call s^α the *dynamic power* and σ the *static power*. In the sleep state, the speed is zero and the power is zero. State transition requires energy; without loss of generality, we assume a transition from the sleep state to the awake state requires an amount ω of energy, and the reverse takes zero energy. To simplify our work, we assume state transition takes no time.

Next we consider the setting with $m > 1$ levels of sleep. A processor is in either the *awake* state or the *sleep-i* state, where $1 \leq i \leq m$. The awake state is the same as before, demanding static power σ and dynamic power s^α. For convenience, we let $\sigma_0 = \sigma$. The

sleep-m state is the only "real" sleep state, which has static power $\sigma_m = 0$; other sleep-i states have decreasing positive static power σ_i such that $\sigma_0 > \sigma_1 > \sigma_2 > \cdots > \sigma_{m-1} > \sigma_m = 0$. We denote the wake-up energy from the sleep-i state to the awake state as ω_i. Note that $\omega_m > \omega_{m-1} > \cdots > \omega_1 > 0$.

It is useful to differentiate two types of awake state: with zero speed and with positive speed. The former is called *idle* state and the latter is called *working* state.

Flow and energy. Given a set of jobs to schedule, we denote the release time and size of a job j as $r(j)$ and $p(j)$, respectively. Consider any schedule of jobs. The flow $F(j)$ of a job j is the time elapsed since it arrives and until it is completed. The total flow is $F = \sum_j F(j)$, or equivalently, $F = \int_0^\infty n(t)\,\mathrm{d}t$, where $n(t)$ is the number of unfinished jobs at time t. Based on this view, we divide F into two parts: F_{w} is the flow incurred during time intervals of working state, and F_{i} for idle or sleep state. The energy usage is also divided into three parts: W denotes the energy due to wake-up transitions, E_{i} is the idling energy (static power consumption in the idle or intermediate sleep state), and E_{w} is the working energy (static and dynamic power consumption in the working state). Our objective is to minimize the *total cost* $G = F_{\mathrm{w}} + F_{\mathrm{i}} + E_{\mathrm{i}} + E_{\mathrm{w}} + W$. We call $F_{\mathrm{w}} + E_{\mathrm{w}}$ the *working cost*, and $F_{\mathrm{i}} + E_{\mathrm{i}} + W$ the *inactive cost*.

3.2 Sleep Management Algorithm IdleLonger

This section presents a sleep management algorithm called IdleLonger that determines when the processor should sleep, idle, and work (with speed > 0). IdleLonger can be coupled with any *speed scaling algorithm*, which specifies which job and at what speed to run when the processor is working. As a warm-up, we first consider the case with a single sleep state. Afterwards, we consider the general case of multiple sleep states.

In this section, we derive an upper bound of the inactive cost of IdleLonger independent of the choice of the speed scaling algorithm. In Section 3.3, we will present the speed scaling algorithm SAJC and analyze its working cost when it is coupled with IdleLonger. In conclusion, putting IdleLonger and SAJC together, we can show that both the inactive cost and working cost are $O(1)$ times of the total cost of the optimal offline algorithm OPT. Note that since IdleLonger and the analysis of its inactive cost is generic for any speed scaling algorithm, when we consider non-clairvoyant flow-energy scheduling with

sleep states in Chapter 5, we only need to give a non-clairvoyant speed scaling algorithm and analyze its working cost when it is coupled with IdleLonger.

3.2.1 Sleep management algorithm for a single sleep state

When the processor is in the working state and sleep state, it is relatively simple to determine the next transition. In the former, the processor keeps on working as long as there is an unfinished job; otherwise switch to the idle state. In the sleep state, we avoid waking up immediately after a new job arrives as this requires energy. It is natural to wait until the new jobs have accumulated enough flow, say, at least the wake-up energy ω, then we let the processor to switch to working state direct. Below we refer the flow accumulated due to new jobs over a period of idle or sleep state as the *inactive flow* of that period.

When the processor is in idle state, it is non-trivial when to switch to the sleep or working state. Intuitively, the processor should not stay in idle state too long, because it consumes energy (at the rate of σ) but does not get any work done. Yet to avoid frequent wake-up in future, the processor should not sleep immediately. Instead the processor should wait for possible job arrival and sleep only after the idling energy (i.e., σ times the length of idling interval) reaches the wake-up energy ω. When a new job arrives in the idle state, a naive idea is to let the processor switch to the working state to process the job immediately; this avoids accumulating inactive flow. Yet this turns out to be a bad strategy as it becomes too difficult to sleep; e.g., the adversary can use some tiny jobs sporadically, then the processor would never accumulate enough idling energy to sleep.

It is perhaps counter-intuitive that IdleLonger always prefers to idle a bit longer, and it can switch to the sleep state even in the presence of unfinished jobs. The idea is to consider the inactive flow and idling energy at the same time. Note that when an idling period gets longer, both the inactive flow and idling energy increase, but at different rates. We imagine that these two quantities are competing with each other.

> The processor switches from the idle state to the working state once the inactive flow catches up with the idling energy. If the idling energy has exceeded ω before the inactive flow catches up with the idling energy, the processor switches to the sleep state.

Below is a summary of the above discussion. For simplicity, IdleLonger is written in a way that it is being executed continuously. In practice, we can rewrite it such that the execution is driven by discrete events like job arrival, job completion and wake-up.

Algorithm 1 IdleLonger(A): A is any speed scaling algorithm

At any time t, let $n(t)$ be the number of unfinished jobs at t.
In working state: If $n(t) > 0$, keep working on jobs according to the algorithm A; else (i.e., $n(t) = 0$), switch to idle state.
In idle state: Let $t' \le t$ be the last time in working state ($t' = 0$ if undefined). If the inactive flow over $[t', t]$ equals $(t - t')\sigma$, then switch to working state;
Else if $(t - t')\sigma = \omega$, switch to sleep state.
In sleep state: Let $t' \le t$ be the last time in working state ($t' = 0$ if undefined). If the inactive flow over $[t', t]$ equals ω, switch to working state.

Below we upper bound the inactive cost of IdleLonger (the working cost will be dealt with in Section 3.3). It is useful to define three types of time intervals. An $\mathrm{I_w}$-interval is a maximal interval in idling state with a transition to the working state at the end, and similarly an $\mathrm{I_s}$-interval for that with a transition to the sleep state at the end. Furthermore, an $\mathrm{IS_w}$-interval is a maximal interval comprising an $\mathrm{I_s}$-interval, then a sleeping interval, and finally a wake-up transition. As the processor starts in the sleep state, we allow the first $\mathrm{IS_w}$-interval containing no $\mathrm{I_s}$-interval.

Consider a schedule of IdleLonger(A). Recall that the inactive cost is composed of W (wake-up energy), F_{i} (inactive flow), and E_{i} (idling energy). We further divide E_{i} into two types: $\boldsymbol{E_{\mathrm{iw}}}$ is the idling energy incurred in all $\mathrm{I_w}$-intervals, and $\boldsymbol{E_{\mathrm{is}}}$ for all $\mathrm{I_s}$-intervals.

By the definition of IdleLonger, we have the following property.

Property 3.1. (i) $F_{\mathrm{i}} \le W + E_{\mathrm{iw}}$, *and* (ii) $E_{\mathrm{is}} = W$.

Therefore, the inactive cost of IdleLonger, defined as $W + F_{\mathrm{i}} + E_{\mathrm{iw}} + E_{\mathrm{is}}$, is at most $3W + 2E_{\mathrm{iw}}$. The non-trivial part is to upper bound W and E_{iw}. Our main result is stated below. For the optimal offline algorithm OPT, we divide its total cost G^* into two parts: W^* is the total wake-up energy, and $C^* = G^* - W^*$ (i.e., the total flow plus the working and idling energy).

Theorem 3.2. $W + E_{\mathrm{iw}} \le C^* + 2W^*$.

Corollary 3.3. *The inactive cost of* IdleLonger *is at most* $3C^* + 6W^*$.

The rest of this section is devoted to proving Theorem 3.2. Note that W is the wake-up energy consumed at the end of all IS_w-intervals, and E_iw is the idling energy of all I_w-intervals. All these intervals are disjoint. Below we show a charging scheme such that, for each IS_w-interval, we charge OPT a cost at least ω, and for each I_w-interval, we charge OPT at least the idling energy of this interval. Thus, the total charge to OPT is at least $W + E_\text{iw}$. On the other hand, we argue that the total charge is at most $C^* + 2W^*$. Therefore, $W + E_\text{iw} \leq C^* + 2W^*$.

The charging scheme for an IS_w-interval $[t_1, t_2]$ is as follows. The target is at least ω.

Case 1. If OPT switches from or to the sleep state in $[t_1, t_2]$, we charge OPT the cost ω of the first wake-up in $[t_1, t_2]$ (if it exists) or of the last wake-up before t_1.

Case 2. If OPT is awake throughout $[t_1, t_2]$, we charge OPT the static energy $(t_2 - t_1)\sigma$. Note that in an IS_w-interval, IdleLonger has an idle-sleep transition, and hence $(t_2 - t_1)\sigma > \omega$.

Case 3. If OPT is sleeping throughout $[t_1, t_2]$, we charge OPT the inactive flow (i.e., the flow incurred by new jobs) over $[t_1, t_2]$. In this case, OPT and IdleLonger have the same amount of inactive flow during $[t_1, t_2]$, which equals ω (because IdleLonger wakes up at t_2).

For an I_w-interval, we use the above charging scheme again. The definition of I_w-interval allows the scheme to guarantee a charge of $(t_2 - t_1)\sigma$ instead of ω. Specifically, as an I_w-interval ends with an idle-working transition, the inactive flow accumulated in $[t_1, t_2]$ is $(t_2 - t_1)\sigma$, and the latter cannot exceed ω. Therefore, the charge of Case 1, which equals ω, is at least $(t_2 - t_1)\sigma$. Case 2 charges exactly $(t_2 - t_1)\sigma$. For Case 3, we charge OPT the inactive flow during $[t_1, t_2]$. Note that OPT and IdleLonger accumulate the same inactive flow, which is $(t_2 - t_1)\sigma$.

Summing over all I_w- and IS_w-intervals, we have charged OPT at least $W + E_\text{iw}$. On the other hand, since all these intervals are disjoint, in Cases 2 and 3, the charge comes from non-overlapping flow and energy of C^*. In Case 1, each OPT's wake-up from the sleep state is charged for ω at most twice, thus the total charge is at most $2W^*$. In conclusion, $W + E_\text{iw} \leq C^* + 2W^*$.

3.2.2 Sleep management algorithm for $m \geq 2$ sleep states

We extend the previous sleep management algorithm to allow intermediate sleep states, which demand less idling (static) energy than the idling state, and also less wake-up energy than the final sleep state (i.e., sleep-m state). We treat the sleep-m state as the only sleep state in the single-level setting, and adapt the transition rules of the idling state for the intermediate sleep states. The key idea is again to compare inactive flow against idling energy continuously. To ease our discussion, we treat the idle state as the sleep-0 state with wake-up energy $\omega_0 = 0$.

Algorithm 2 IdleLonger(A): A is any speed scaling algorithm

At any time t, let $n(t)$ be the number of unfinished jobs at t.
In working state: If $n(t) > 0$, keep working on the jobs according to the algorithm A; else if $n(t) = 0$, switch to idle state.
In sleep-j state, where $0 \leq j \leq m-1$: Let $t' \leq t$ be the last time in the working state, and let t'' be the last time in $[t', t]$ switching from sleep-$(j-1)$ state to sleep-j state. If the inactive flow over $[t', t]$ equals $(t - t'')\sigma_j + \omega_j$, then wake up to the working state;
Else if $(t - t'')\sigma_j = (\omega_{j+1} - \omega_j)$, switch to sleep-$(j+1)$ state.
In sleep-m state: Let $t' \leq t$ be the last time in the working state. If the inactive flow over $[t', t]$ equals ω_m, then wake up to the working state.

When we analyze the multi-level algorithm, the definition of W (total wake-up cost) and F_{i} (total inactive flow) remain the same, but E_{is} and E_{iw} have to be generalized. Below we refer a maximal interval during which the processor is in a particular sleep-j state, where $0 \leq j \leq m$, as a *sleep interval* or more specifically, a sleep-j interval. Note that all sleep intervals, except sleep-m intervals, demand idling (static) energy. We denote $\boldsymbol{E_{\mathrm{iw}}}$ as the idling energy for all sleep intervals that end with a wake-up transition, and $\boldsymbol{E_{\mathrm{is}}}$ the idling energy of all sleep intervals ending with a (deeper) sleep transition.

IdleLonger imposes a rigid structure of sleep intervals. Define $\ell_j = (\omega_{j+1} - \omega_j)/\sigma_j$. A sleep-$j$ interval can appear only after a sequence of lower level sleep intervals, which starts with a sleep-0 interval of length ℓ_0, followed by a sleep-1 interval of length ℓ_1, ..., and finally a sleep-$(j-1)$ interval of length ℓ_{j-1}. Consider a maximum sequence of such sleep intervals that ends with a transition to the working state. We call the entire time interval enclosed by this sequence an $\mathrm{IS_w}[j]$-interval for some $0 \leq j \leq m$ if the deepest (also the last) sleep subinterval is of level j. It is useful to observe the following lemma

about an $\mathrm{IS_w}[j]$-interval.

Lemma 3.4. *Consider any* $\mathrm{IS_w}[j]$*-interval* $[t_1, t_2]$*, where* $0 \le j \le m$*. Assume that the last sleep-*j *(sub)interval is of length* ℓ*. Then,* $\omega_j + \ell\sigma_j \le \omega_k + (t_2 - t_1)\sigma_k$ *for any* $0 \le k \le m$*.*

Proof. We consider two cases depending on whether $k > j$.

If $k > j$, then $j \le k - 1 \le m - 1$. Since $j \le m - 1$, by the definition of IdleLonger, $\ell \le \ell_j$. Then $\omega_j + \ell\sigma_j \le \omega_j + \ell_j\sigma_j = \omega_{j+1} \le \omega_k$ and thus $\omega_j + \ell\sigma_j \le \omega_k + (t_2 - t_1)\sigma_k$.

Otherwise, $k \le j$ and we count the energy usage of the $\mathrm{IS_w}[j]$-interval in two different ways; one count is exactly $\omega_j + \ell\sigma_j$ and the other is at most $\omega_k + (t_2 - t_1)\sigma_k$, which implies $\omega_j + \ell\sigma_j \le \omega_k + (t_2 - t_1)\sigma_k$. Note that for $0 \le i \le j-1$, the energy usage in a sleep-i interval is $\ell_i\sigma_i = \omega_{i+1} - \omega_i$. Thus, the energy usage in the $\mathrm{IS_w}[j]$-interval is $\sum_{0 \le i \le j-1} \ell_i\sigma_i + \ell\sigma_j = (\omega_j - \omega_0) + \ell\sigma_j = \omega_j + \ell\sigma_j$, where the last equality is due to $\omega_0 = 0$. On the other hand, the energy usage in the first k sleep intervals is $\sum_{0 \le i \le k-1} \ell_i\sigma_i = \omega_k - \omega_0 = \omega_k$, while the other energy usage is $\sum_{k \le i \le j-1} \ell_i\sigma_i + \ell\sigma_j \le (\sum_{k \le i \le j-1} \ell_i + \ell) \cdot \sigma_k \le (t_2 - t_1)\sigma_k$. In conclusion, $\omega_j + \ell\sigma_j \le \omega_k + (t_2 - t_1)\sigma_k$. □

The rigid sleeping structure of IdleLonger also allows us to maintain Property 3.1 as before. That is, (i) $F_{\mathrm{i}} \le W + E_{\mathrm{iw}}$, and (ii) $E_{\mathrm{is}} = W$. The proof is given below.

Proof of Property 3.1. (i) Note that F_{i} is equal to the inactive flow incurred in all $\mathrm{IS_w}[j]$-intervals, where $0 \le j \le m$. Consider any $\mathrm{IS_w}[j]$-interval. Assume that the last sleep-j (sub)interval is of length ℓ. By definition of IdleLonger, the inactive flow is at most the idling energy $\ell\sigma_j$ of the last sleep subinterval plus the energy ω_k of the wake-up at the end. Summing over all $\mathrm{IS_w}[j]$-intervals, we have $F_{\mathrm{i}} \le E_{\mathrm{iw}} + W$.

(ii) Consider all $\mathrm{IS_w}[j]$-intervals. The first $\mathrm{IS_w}[j]$-interval is simply a sleep-m interval, which does not incur any idling energy, while in any other $\mathrm{IS_w}[j]$-interval, the total idling energy of all sleep subintervals except the last subinterval is $\sum_{0 \le i \le j-1} \ell_i\sigma_i = \omega_j - \omega_0 = \omega_j$. Note that E_{is} is the sum of such idling energy ω_j of all $\mathrm{IS_w}[j]$-intervals (except the first $\mathrm{IS_w}[j]$-interval) plus the idling energy incurred in the sleep intervals which occur after the last wake up. By the rigid sleeping structure of IdleLonger, the latter term equals $\sum_{0 \le i \le m-1} \ell_i\sigma_i = \omega_m - \omega_0 = \omega_m$. On the other hand, W is the sum of the wake-up energy ω_m of the first $\mathrm{IS_w}[j]$-interval and the wake-up energy ω_j of the other $\mathrm{IS_w}[j]$-intervals. In conclusion, we have $E_{\mathrm{is}} = W$. □

Due to Property 3.1, the inactive cost, which is equal to $F_{\mathrm{i}} + E_{\mathrm{iw}} + E_{\mathrm{is}} + W$, is still at most $3W + 2E_{\mathrm{iw}}$, i.e., Corollary 3.3 still holds. In the rest of this section, we prove that W and E_{iw} have the same upper bound as before.

Theorem 3.5. *In the setting of $m \geq 2$ sleep states, $W + E_{\mathrm{iw}} \leq C^* + 2W^*$.*

To account for W and E_{iw}, it suffices to look at all $\mathrm{IS_w}[j]$-intervals, where $0 \leq j \leq m$. For each $\mathrm{IS_w}[j]$-interval, we show how to charge OPT a cost $\omega_j + \ell\sigma_j$, where ℓ is length of the deepest sleep subinterval (it is useful to recall that $\omega_0 = 0$ and $\sigma_m = 0$). Then we argue that the total cost charged is at least $W + E_{\mathrm{iw}}$ and at most $C^* + 2W^*$.

Without loss of generality, we can assume that in a maximal interval $[r_1, r_2]$ that OPT is not working, if OPT has ever slept (in sleep-1 or deeper sleep state), then $[r_1, r_2]$ contains only one sleep transition, which occurs at r_1, and the processor remains in the same sleep state until r_2 and then wakes up to work.

Charging scheme. Consider any $\mathrm{IS_w}[j]$-interval $[t_1, t_2]$, where $0 \leq j \leq m$. Let ℓ be the length of the sleep-j (sub)interval in this $\mathrm{IS_w}[j]$-interval.

Case 1. If OPT has ever switched from or to the sleep-1 or deeper sleep state in $[t_1, t_2]$, let $k \geq 1$ be the deepest sleep level involved in the entire interval. Note that OPT uses static energy at least $(t_2 - t_1)\sigma_k$ during $[t_1, t_2]$. We charge OPT the sum of $(t_2 - t_1)\sigma_k$ and ω_k (in view of a wake-up from sleep-k state inside $[t_1, t_2]$ or after t_2; if there is no wake-up after t_2, then we charge OPT the first wake-up). By Lemma 3.4, this charge is at least $\omega_j + \ell\sigma_j$.

Case 2. If OPT is working or idle throughout $[t_1, t_2]$, we charge OPT the static energy $(t_2 - t_1)\sigma_0$, which, by Lemma 3.4, is at least $\omega_j + \ell\sigma_j$.

Case 3. If OPT is sleeping (at any level except zero) throughout $[t_1, t_2]$, we charge OPT the inactive flow over $[t_1, t_2]$. Note that OPT has the same amount of inactive flow as IdleLonger. By the definition of a wake-up transition in IdleLonger, the inactive flow equals $\omega_j + \ell\sigma_j$.

Since $\mathrm{IS_w}[j]$-intervals are all disjoint, the flow and idling (static) energy charged to OPT by Cases 1, 2 and 3 come from different parts of C^*. For Case 1, each of OPT's wake-up from a sleep state is charged at most twice. Thus, $W + E_{\mathrm{iw}} \leq C^* + 2W^*$, completing the proof of Theorem 3.5.

3.3 Speed Scaling Algorithm SAJC

When there is no sleep state and the power function is in the form of s^α, Chapter 2 showed that the speed scaling algorithm SRPT-AJC is $O(1)$-competitive for flow plus energy. SRPT-AJC always runs the job with the smallest remaining work (SRPT) at the speed $n(t)^{1/\alpha}$, where $n(t)$ is the number of unfinished jobs at time t. In the sleep setting, when the processor is working, it requires at least the static power σ; if σ is large, running at the speed $n(t)^{1/\alpha}$ would be too slow to be cost effective as the dynamic power could be way smaller than σ. Indeed SRPT-AJC has unbounded competitive ratio no matter what sleep management algorithm is used. This section shows how to analyze a simple adaptation of SRPT-AJC to the sleep setting, and upper bound its working cost in terms of OPT's total cost.

> **Algorithm SAJC.** At any time t, SAJC runs the job with the shortest remaining work at the speed $(n(t) + \sigma)^{1/\alpha}$.

The following analysis of SAJC is valid no matter what sleep management algorithm Slp is being used together. Ideally we want to upper bound the working cost of SAJC solely in terms of the total cost of OPT, yet this is not possible as the working cost also depends on Slp. Below we give an analysis of SAJC in which the dependency on Slp is bounded by the inactive flow incurred by Slp. More specifically, let G_{w} and F_{i} be the working cost and the inactive flow of Slp(SAJC), respectively. Again, we use C^* to denote the total cost of OPT minus the wake-up energy; the latter is denoted by W^*. We will give a potential analysis to prove that $G_{\mathrm{w}} = O(C^* + F_{\mathrm{i}})$.

Let us look at a simple case. If Slp always switches to working state whenever there are unfinished jobs, then $F_{\mathrm{i}} = 0$. In this case we can easily adapt the analysis in Chapter 2 to bound G_{w} in terms of C^* only. However, the inactive cost of Slp may be unbounded in this case. On the other hand, consider a sleep management algorithm that prefers to wait for more jobs before waking up to work (e.g., IdleLonger). Then SAJC would start at a higher speed and G_{w} can be much larger than C^*. Roughly speaking, the excess is due to the fact that the online algorithm is sleeping while OPT is working. Note that the cost to catch up the work lagged behind increases at a rate depending on $n(t)$. This motivates us to bound the excess in terms of F_{i}. The main result of this section is stated in the

lemma below, which, together with the results on inactive cost of IdleLonger, imply that IdleLonger(SAJC) is $(2\beta+2)$-competitive for flow plus energy, where $\beta = 2/(1-\frac{\alpha-1}{\alpha^{\alpha/(\alpha-1)}})$.

Theorem 3.6. *With respect to* Slp(SAJC), $G_{\mathrm{w}} \leq \beta C^* + (\beta-2)F_{\mathrm{i}}$.

Corollary 3.7. *In the setting of single sleep state or multiple sleep states, the total cost of* IdleLonger(SAJC) *is at most* $(2\beta+2)$ *times of the total cost of* OPT.

Proof of Corollary 3.7. Consider the coupling of IdleLonger and SAJC. As proven in Section 3.2, $F_{\mathrm{i}} \leq C^* + 2W^*$, and the inactive cost is at most $3C^* + 6W^*$. The total cost of IdleLonger(SAJC), comprised of the inactive cost and the working cost, is at most $(2\beta+1)C^* + (2\beta+2)W^*$, which is at most $(2\beta+2)$ times of OPT's total cost. □

To prove Theorem 3.6, we adapt the potential function $\Phi(t)$ in Chapter 2 to relate the change rate of G_{w}, C^* and F_{i}. Consider any time t. Let $G_{\mathrm{w}}(t)$, $F_{\mathrm{i}}(t)$ and $C^*(t)$ denote the corresponding value of G_{w}, F_{i} and C^* incurred up to t. We drop the parameter t when it is clear that t is the current time. The potential function $\Phi(t)$ attempts to capture the difference of the remaining work of SAJC and OPT. The formal definition is as follows. At time t, for any $q \geq 0$, let $n_{\mathrm{a}}(q)$ be the number of unfinished jobs of SAJC with remaining work at least q, and similarly for $n_{\mathrm{o}}(q)$ and OPT. Then

$$\Phi(t) = \beta \int_0^\infty \phi(q)dq \ , \text{ where } \phi(q) = \left(\sum_{j=1}^{n_{\mathrm{a}}(q)} (j+\sigma)^{1-1/\alpha}\right) - (n_{\mathrm{a}}(q)+\sigma)^{1-1/\alpha} n_{\mathrm{o}}(q) \ .$$

It can be verified that $\Phi(t) = 0$ initially and is non-negative when Slp(SAJC) and OPT finish all jobs. Furthermore, $\Phi(t)$ does not increase or even does not change at any discrete-event time including when a job arrives, and when SAJC or OPT finish a job or change the speed. To prove Theorem 3.6, it suffices to show that at any other time, the rate of change of G_{w} and Φ can be upper bounded in terms of that of C^* and F_{i}.

Lemma 3.8. *At any time when no discrete events occur,* $\frac{\mathrm{d}G_{\mathrm{w}}}{\mathrm{d}t} + \frac{\mathrm{d}\Phi}{\mathrm{d}t} \leq \beta \frac{\mathrm{d}C^*}{\mathrm{d}t} + (\beta-2)\frac{\mathrm{d}F_{\mathrm{i}}}{\mathrm{d}t}$.

The rest of this section is devoted to proving Lemma 3.8. At any time t, if Slp(SAJC) is in the working state, we let s_{a} be the current speed of SAJC and let q_{a} be the remaining work of the current job of SAJC. And s_{o} and q_{o} are defined similarly for OPT.

When SAJC is working, $\frac{\mathrm{d}G_{\mathrm{w}}}{\mathrm{d}t} = s_{\mathrm{a}}^{\alpha} + \sigma + n_{\mathrm{a}} = 2(n_{\mathrm{a}} + \sigma)$ and $\frac{\mathrm{d}F_{\mathrm{i}}}{\mathrm{d}t} = 0$; otherwise, $\frac{\mathrm{d}G_{\mathrm{w}}}{\mathrm{d}t} = 0$ and $\frac{\mathrm{d}F_{\mathrm{i}}}{\mathrm{d}t} = n_{\mathrm{a}}$. When OPT is working, $\frac{\mathrm{d}C^*}{\mathrm{d}t} = s_{\mathrm{o}}^{\alpha} + \sigma + n_{\mathrm{o}}$; otherwise, $\frac{\mathrm{d}C^*}{\mathrm{d}t} \geq n_{\mathrm{o}}$. To upper bound $\frac{\mathrm{d}\Phi}{\mathrm{d}t}$, we follow the framework in Chapter 2 to divide the analysis into two parts: (i) the execution of SAJC affects $\phi(q)$ for $q \in (q_{\mathrm{a}} - s_{\mathrm{a}}\,\mathrm{d}t, q_{\mathrm{a}}]$, and (ii) the execution of OPT affects $\phi(q)$ for $q \in (q_{\mathrm{o}} - s_{\mathrm{o}}\,\mathrm{d}t, q_{\mathrm{o}}]$. We denote these changes as $\mathrm{d}\Phi_1$ and $\mathrm{d}\Phi_2$, respectively, and $\mathrm{d}\Phi = \mathrm{d}\Phi_1 + \mathrm{d}\Phi_2$. We upper bound $\frac{\mathrm{d}\Phi_1}{\mathrm{d}t}$ and $\frac{\mathrm{d}\Phi_2}{\mathrm{d}t}$ as follows. Note that in the infinite speed model, $T = \infty$ and thus we always have $s_{\mathrm{a}} < T$.

Claim 3.9. (i) *Consider* $\frac{\mathrm{d}\Phi_1}{\mathrm{d}t}$. *If* $s_{\mathrm{a}} > 0$, $\frac{\mathrm{d}\Phi_1}{\mathrm{d}t} \leq -\beta(n_{\mathrm{a}} + \sigma - n_{\mathrm{o}})$; *otherwise,* $\frac{\mathrm{d}\Phi_1}{\mathrm{d}t} \leq 0$.
(ii) $\frac{\mathrm{d}\Phi_2}{\mathrm{d}t} \leq \beta(n_{\mathrm{a}} + \sigma)^{1-1/\alpha} s_{\mathrm{o}}$.

Proof. (i) We first analyze $\frac{\mathrm{d}\Phi_1}{\mathrm{d}t}$. Note that $n_{\mathrm{a}}(q_{\mathrm{a}}) = n_{\mathrm{a}}$. For any $q \in (q_{\mathrm{a}} - s_{\mathrm{a}}\mathrm{d}t, q_{\mathrm{a}}]$, $\phi(q)$ is changed by $-(n_{\mathrm{a}} + \sigma)^{1-1/\alpha} + ((n_{\mathrm{a}} + \sigma)^{1-1/\alpha} - (n_{\mathrm{a}} + \sigma - 1)^{1-1/\alpha}) n_{\mathrm{o}}(q_{\mathrm{a}})$. Note that $x^{1-1/\alpha} - (x-1)^{1-1/\alpha} \leq x^{-1/\alpha}$ for all $x \geq 1$. By setting $x = n_{\mathrm{a}} + \sigma$ and the fact that $n_{\mathrm{o}}(q_{\mathrm{a}}) \leq n_{\mathrm{o}}$, $\phi(q)$ changes by at most $-(n_{\mathrm{a}} + \sigma)^{1-1/\alpha} + (n_{\mathrm{a}} + \sigma)^{-1/\alpha} n_{\mathrm{o}}$. Integrating over all $q \in (q_{\mathrm{a}} - s_{\mathrm{a}}\mathrm{d}t, q_{\mathrm{a}}]$,

$$\frac{\mathrm{d}\Phi_1}{\mathrm{d}t} \leq \beta(-(n_{\mathrm{a}} + \sigma)^{1-1/\alpha} + (n_{\mathrm{a}} + \sigma)^{-1/\alpha} n_{\mathrm{o}}) s_{\mathrm{a}} \leq -\beta(n_{\mathrm{a}} + \sigma - n_{\mathrm{o}})(n_{\mathrm{a}} + \sigma)^{-1/\alpha} s_{\mathrm{a}} \ . \quad (3.1)$$

If $s_{\mathrm{a}} = 0$, then Inequality (3.1) implies $\frac{\mathrm{d}\Phi_1}{\mathrm{d}t} \leq 0$. If $s_{\mathrm{a}} > 0$, then $s_{\mathrm{a}} = (n_{\mathrm{a}} + \sigma)^{1/\alpha}$ and hence Inequality (3.1) implies $\frac{\mathrm{d}\Phi_1}{\mathrm{d}t} \leq -\beta(n_{\mathrm{a}} + \sigma - n_{\mathrm{o}})$.

(ii) We now analyze $\frac{\mathrm{d}\Phi_2}{\mathrm{d}t}$. Note that $n_{\mathrm{a}}(q_{\mathrm{o}}) \leq n_{\mathrm{a}}$. For any $q \in (q_{\mathrm{o}} - s_{\mathrm{o}}\mathrm{d}t, q_{\mathrm{o}}]$, $\phi(q)$ is changed by $(n_{\mathrm{a}}(q) + \sigma)^{1-1/\alpha} \leq (n_{\mathrm{a}} + \sigma)^{1-1/\alpha}$. Integrating over all $q \in (q_{\mathrm{o}} - s_{\mathrm{o}}\mathrm{d}t, q_{\mathrm{o}}]$, we have $\frac{\mathrm{d}\Phi_2}{\mathrm{d}t} \leq \beta(n_{\mathrm{a}} + \sigma)^{1-1/\alpha} s_{\mathrm{o}}$. □

Using the Young's Inequality (see Lemma 2.6 in Chapter 2), we can further introduce any constant $\mu > 0$ into the bound of $\frac{\mathrm{d}\Phi_2}{\mathrm{d}t}$ in Claim 3.9(ii) as follows. (Later we will set $\mu = \alpha^{-1/\alpha}$.) By setting $f(x) = x^{\alpha-1}$, $f^{-1}(x) = x^{1/(\alpha-1)}$, $g = s_{\mathrm{o}}/\mu$, $h = (n_{\mathrm{a}} + \sigma)^{1-1/\alpha}\mu$,

$$\begin{aligned}\frac{\mathrm{d}\Phi_2}{\mathrm{d}t} \leq \beta((n_{\mathrm{a}} + \sigma)^{1-1/\alpha}\mu)\left(\frac{s_{\mathrm{o}}}{\mu}\right) &\leq \beta\left(\int_0^{\frac{s_{\mathrm{o}}}{\mu}} x^{\alpha-1}\,\mathrm{d}x + \int_0^{n_{\mathrm{a}}^{1-1/\alpha}\mu} x^{1/(\alpha-1)}\,\mathrm{d}x\right) \\ &\leq \frac{\beta s_{\mathrm{o}}^{\alpha}}{\alpha\mu^{\alpha}} + \beta(1 - \frac{1}{\alpha})(n_{\mathrm{a}} + \sigma)\mu^{\alpha/(\alpha-1)} \ . \quad (3.2)\end{aligned}$$

We are now ready to prove the inequality of Lemma 3.8. We give a case analysis depending on whether Slp(SAJC) or OPT are working. The interesting case is Case iii, when OPT is working but Slp(SAJC) is not. In Case iii, the increase of potential is partially offset by the increase of inactive flow.

Case i. $s_{\mathbf{a}} > 0$**,** $s_{\mathbf{o}} > 0$**:** In this case, $\frac{\mathrm{d}G_{\mathrm{w}}}{\mathrm{d}t} = 2(n_{\mathrm{a}}+\sigma)$, $\frac{\mathrm{d}C^*}{\mathrm{d}t} = s_{\mathrm{o}}^{\alpha}+\sigma+n_{\mathrm{o}}$, and $\frac{\mathrm{d}F_{\mathrm{i}}}{\mathrm{d}t} = 0$. By Claim 3.9(i) and Inequality (3.2), we have

$$\frac{\mathrm{d}G_{\mathrm{w}}}{\mathrm{d}t} + \frac{\mathrm{d}\Phi}{\mathrm{d}t} \leq 2(n_{\mathrm{a}}+\sigma) - \beta(n_{\mathrm{a}}+\sigma) + \beta n_{\mathrm{o}} + \beta(1-\frac{1}{\alpha})(n_{\mathrm{a}}+\sigma)\mu^{\alpha/(\alpha-1)} + \frac{\beta s_{\mathrm{o}}^{\alpha}}{\alpha\mu^{\alpha}} \ .$$

Recalling that $\beta = 2/(1-\frac{\alpha-1}{\alpha^{\alpha/(\alpha-1)}})$ and setting $\mu = \alpha^{-1/\alpha}$, we have $\frac{\mathrm{d}G_{\mathrm{w}}}{\mathrm{d}t} + \frac{\mathrm{d}\Phi}{\mathrm{d}t} \leq \beta(s_{\mathrm{o}}^{\alpha} + \sigma + n_{\mathrm{o}}) = \beta\frac{\mathrm{d}C^*}{\mathrm{d}t}$.

Case ii. $s_{\mathbf{a}} > 0$**,** $s_{\mathbf{o}} = 0$**:** In this case, $\frac{\mathrm{d}G_{\mathrm{w}}}{\mathrm{d}t} = 2(n_{\mathrm{a}}+\sigma)$, $\frac{\mathrm{d}C^*}{\mathrm{d}t} \geq n_{\mathrm{o}}$, $\frac{\mathrm{d}F_{\mathrm{i}}}{\mathrm{d}t} = 0$, and $\frac{\mathrm{d}\Phi_2}{\mathrm{d}t} = 0$ (Claim 3.9(ii)). Using Claim 3.9(i), we have

$$\frac{\mathrm{d}G_{\mathrm{w}}}{\mathrm{d}t} + \frac{\mathrm{d}\Phi}{\mathrm{d}t} \leq 2(n_{\mathrm{a}}+\sigma) - \beta(n_{\mathrm{a}}+\sigma) + \beta n_{\mathrm{o}} \leq \beta n_{\mathrm{o}} \leq \beta\frac{\mathrm{d}C^*}{\mathrm{d}t} \ .$$

Case iii. $s_{\mathbf{a}} = 0$**,** $s_{\mathbf{o}} > 0$**:** In this case, $\frac{\mathrm{d}G_{\mathrm{w}}}{\mathrm{d}t} = 0$, $\frac{\mathrm{d}C^*}{\mathrm{d}t} = s_{\mathrm{o}}^{\alpha}+\sigma+n_{\mathrm{o}}$, $\frac{\mathrm{d}F_{\mathrm{i}}}{\mathrm{d}t} = n_{\mathrm{a}}$, and $\frac{\mathrm{d}\Phi_1}{\mathrm{d}t} = 0$ (Claim 3.9(i)). Using Inequality (3.2) and setting $\mu = \alpha^{-1/\alpha}$, we have

$$\frac{\mathrm{d}G_{\mathrm{w}}}{\mathrm{d}t}+\frac{\mathrm{d}\Phi}{\mathrm{d}t} \leq \beta(1-\frac{1}{\alpha})(n_{\mathrm{a}}+\sigma)\mu^{\alpha/(\alpha-1)}+\frac{\beta s_{\mathrm{o}}^{\alpha}}{\alpha\mu^{\alpha}} \leq \beta(s_{\mathrm{o}}^{\alpha}+\sigma)+(\beta-2)n_{\mathrm{a}} \leq \beta\frac{\mathrm{d}C^*}{\mathrm{d}t}+(\beta-2)\frac{\mathrm{d}F_{\mathrm{i}}}{\mathrm{d}t} \ .$$

Case iv. $s_{\mathbf{a}} = s_{\mathbf{o}} = 0$**:** In this case, $\frac{\mathrm{d}G_{\mathrm{w}}}{\mathrm{d}t} = 0$, $\frac{\mathrm{d}C^*}{\mathrm{d}t} \geq n_{\mathrm{o}}$, $\frac{\mathrm{d}F_{\mathrm{i}}}{\mathrm{d}t} = n_{\mathrm{a}}$, and $\frac{\mathrm{d}\Phi}{\mathrm{d}t} = 0$ (Claim 3.9). Therefore, $\frac{\mathrm{d}G_{\mathrm{w}}}{\mathrm{d}t} + \frac{\mathrm{d}\Phi}{\mathrm{d}t} = 0 \leq \beta\frac{\mathrm{d}C^*}{\mathrm{d}t}$.

3.4 Bounded Speed Model

This section extends the sleep management algorithm IdleLonger and the speed scaling algorithm SAJC to the bounded speed model. We consider the setting where the processor speed is upper bounded by a constant $T > 0$, and there are $m \geq 1$ levels of sleep states. We show that the total cost (comprising inactive and working cost) of IdleLonger(SAJC) is $O(1)$ times of the optimal offline algorithm OPT.

Adaptation. In the bounded speed model, IdleLonger (see Section 3.2) still works and the inactive cost is $O(1)$ times of OPT's total cost. However, IdleLonger often allows a long sleep, then a speed scaling algorithm, without the capability to speed up arbitrarily, cannot always catch up the progress of OPT and may have unbounded working cost. Thus, we adapt IdleLonger to wake up earlier, especially when too many new jobs have arrived. To this end, we add one more wake-up condition to IdleLonger. Recall that $\sigma(=\sigma_0)$ is the static power in the working state.

> In the sleep-j state, where $0 \le j \le m$, if the number of unfinished jobs exceeds σ, the processor wakes up to the working state.

Recall that SAJC runs at the speed $(n(t)+\sigma)^{1/\alpha}$, where $n(t)$ is the number of unfinished jobs at time t. To adapt SAJC to the bounded speed model, we simply cap the speed at T. That is, at any time t, the processor runs at the speed $\min\{(n(t)+\sigma)^{1/\alpha}, T\}$.

Inactive cost of IdleLonger. The rigid structure of sleep intervals remains the same as before, and the inactive cost is still at most $3W + 2E_{\text{iw}}$, where W is the wake-up energy and E_{iw} is the idling energy incurred in those idling or intermediate sleep intervals that end with a wake-up transition (see Section 3.2 for details). However, due to the additional wake-up rule, IdleLonger has a slightly worse bound on W plus E_{iw}. Our main result is stated in Lemma 3.10. Again, W^* denotes the wake-up energy of OPT, and C^* is the total cost of OPT minus W^*.

Theorem 3.10. (i) $W + E_{\text{iw}} \le C^* + 3W^*$. (ii) *The inactive cost of* IdleLonger *is at most* $3C^* + 9W^*$.

To prove Theorem 3.10(i), we extend the charging scheme in Section 3.2.2 to show that for each $\text{IS}_{\text{w}}[j]$-interval, OPT can be charged with a cost at least $\omega_j + \ell\sigma_j$, where ℓ is the length of the deepest sleep subinterval (recall that $\omega_0 = 0$, $\sigma_0 = \sigma$ and $\sigma_m = 0$). The three cases of the old charging scheme remain the same, except that Case 3 is restricted to $\text{IS}_{\text{w}}[j]$-intervals where IdleLonger wakes up at the end due to excessive inactive flow. We supplement Case 3 with a new scheme (Case 4) to handle $\text{IS}_{\text{w}}[j]$-intervals with wake-ups due to more than σ unfinished jobs.

Charging scheme – Case 4. If OPT is sleeping (at any level except zero) throughout an $\text{IS}_{\text{w}}[j]$-interval $[t_1, t_2]$, and IdleLonger has accumulated more than σ unfinished jobs

at t_2, we consider two scenarios to charge OPT, depending on $n_o(t_1)$, the number of unfinished jobs in OPT at t_1.
(a) Suppose $n_o(t_1) \geq \sigma_0$. We charge OPT the inactive flow of these jobs over $[t_1, t_2]$, which is at least $(t_2 - t_1)\sigma_0$. By Lemma 3.4, this charge is at least $\omega_j + \ell\sigma_j$.
(b) Suppose $n_o(t_1) < \sigma_0$. Note that OPT stays in a sleep-k state, for some $k \geq 1$, in the entire interval and uses static energy $(t_2 - t_1)\sigma_k$ during $[t_1, t_2]$. We charge OPT the sum of $(t_2 - t_1)\sigma_k$ and ω_k (in view of OPT's first wake-up after t_2, which must exist because new jobs have arrived within $[t_1, t_2]$). By Lemma 3.4, this charge is at least $\omega_j + \ell\sigma_j$.

In conclusion, we are able to charge OPT, for each $\mathrm{IS}_w[j]$-interval, a cost at least $\omega_j + \ell\sigma_j$. Therefore, the sum of the charges to all $\mathrm{IS}_w[j]$-intervals is at least $W + E_{iw}$. On the other hand, recall that Case 1 has a total charge at most $2W^*$. Case 2, 3 and 4(a) have a total charge at most C^*. Below, we argue that OPT is charged by Case 4(b) with a cost at most W^* (Lemma 3.11). Then we have $W + E_{iw} \leq C^* + 3W^*$. And Lemma 3.10(ii) follows directly.

Lemma 3.11. *With respect to the above charging scheme,* OPT *is charged by Case 4(b) with a cost of at most W.*

Proof. To prove that the total charge due to Case 4(b) is at most W^*, it suffices to show that each wake-up of OPT is charged at most once by an $\mathrm{IS}_w[j]$-interval in Case 4(b). Suppose, for the sake of contradiction, there are two $\mathrm{IS}_w[j]$-intervals $[t_1, t_2]$ and $[r_1, r_2]$ (with possibly different j), where $t_2 < r_1$, both charging to the same wake-up transition in Case 4(b). This implies OPT is sleeping during $[t_1, r_2]$. At time t_2, IdleLonger, as well as OPT, have at least $\sigma = \sigma_0$ unfinished jobs. As OPT is sleeping during $[t_1, r_2]$, the number of unfinished jobs of OPT at time r_1 is also at least σ_0 and Case 4(a) should have been applied for $[r_1, r_2]$, which is a contradiction. □

Working cost of SAJC. It remains to analyze the working cost of IdleLonger(SAJC). Our main result is stated in Theorem 3.12 below, which, together with Theorem 3.10(ii), implies that IdleLonger(SAJC) is $\max\{\beta(2 + 1/\alpha) + \zeta + 1, 3\beta + 3\} = O(\frac{\alpha}{\ln \alpha})$-competitive for flow plus energy, where $\zeta = \max\{4, \beta(1 - 1/\alpha)\}$.

Theorem 3.12. *With respect to* IdleLonger(SAJC), $G_w \leq (\beta(1+1/\alpha)+\zeta)C^* + (\beta-2)F_i$.

Corollary 3.13. *In the bounded speed model with single or multiple sleep states, the total cost of* IdleLonger(SAJC) *is at most* $\max\{\beta(2 + 1/\alpha) + \zeta + 1, 3\beta + 3\}$ *times of the total*

cost of OPT.

Proof of Corollary 3.13. By Theorem 3.10(ii), the inactive cost is at most $3C^* + 9W^*$. On the other hand, by Theorem 3.12, the working cost of IdleLonger(SAJC) is $G_{\mathrm{w}} \leq (\beta(1+1/\alpha)+\zeta)C^* + (\beta-2)F_{\mathrm{i}}$. By Property 3.1 in Section 3.2 and Theorem 3.10(i), $F_{\mathrm{i}} \leq W + E_{\mathrm{iw}} \leq C^* + 3W^*$. It follows that the total cost of IdleLonger(SAJC), comprised of the inactive cost and the working cost, is at most $(\beta(2+1/\alpha)+\zeta+1)C^* + (3\beta+3)W^*$. Therefore, the total cost of IdleLonger(SAJC) is at most $\max\{\beta(2+1/\alpha)+\zeta+1, 3\beta+3\}$ times OPT's total cost. □

To prove Theorem 3.12, we use the same potential function $\Phi(t)$ as in Section 3.3, except that we set $\beta = 2/(1 - \frac{1-1/\alpha}{(\alpha+1)^{1/(\alpha-1)}})$. Below, we repeat the definition of $\Phi(t)$. At time t, for any $q \geq 0$, let $n_{\mathrm{a}}(q)$ be the number of unfinished jobs of SAJC with remaining work at least q, and similarly for $n_{\mathrm{o}}(q)$ and OPT. Then

$$\Phi(t) = \beta \int_0^\infty \phi(q)dq \ , \text{ where } \phi(q) = \left(\sum_{j=1}^{n_{\mathrm{a}}(q)} (j+\sigma)^{1-1/\alpha}\right) - (n_{\mathrm{a}}(q)+\sigma)^{1-1/\alpha} n_{\mathrm{o}}(q) \ .$$

As shown in Section 3.3, $\Phi(t) = 0$ initially and is non-negative when IdleLonger(SAJC) and OPT finish all jobs. Furthermore, $\Phi(t)$ does not increase or even does not change at any discrete-event time including when a job arrives, and when IdleLonger(SAJC) or OPT finish a job or change the speed.

Similarly as in Section 3.3, it suffices to show that at any time when no discrete events occur, $\frac{\mathrm{d}G_{\mathrm{w}}(t)}{\mathrm{d}t} + \frac{\mathrm{d}\Phi(t)}{\mathrm{d}t} = O(\frac{\mathrm{d}C^*(t)}{\mathrm{d}t} + \frac{\mathrm{d}F_{\mathrm{i}}(t)}{\mathrm{d}t})$. We omit the parameter t from the notations as it is clear that t is the current time. At time t, if IdleLonger(SAJC) is working, let s_{a} be the current speed of SAJC, and let q_{a} be the remaining work of the current job of SAJC. And s_{o} and q_{o} are define similarly for OPT. Let n_{a} and n_{o} be the number of unfinished jobs at time t in IdleLonger(SAJC) and OPT, respectively. In the infinite speed model, n_{a} may be arbitrary larger than n_{o}. Below, we bound $n_{\mathrm{a}} - n_{\mathrm{o}}$, which shows how the extension of IdleLonger and SAJC helps IdleLonger(SAJC) to keep up with OPT.

Lemma 3.14. *With respect to* IdleLonger(SAJC)*, at any time* t*,* $n_{\mathrm{a}} - n_{\mathrm{o}} \leq \max\{T^\alpha, \sigma\}$.

Proof. The proof idea is the same as that of Lemma 2.8 in Chapter 2. The non-trivial case is $n_{\mathrm{a}} > \max\{T^\alpha, \sigma\}$ at time t. Let $t_0 < t$ be the last time when $n_{\mathrm{a}} \leq \max\{T^\alpha, \sigma\}$.

By the extension of IdleLonger and SAJC, IdleLonger(SAJC) is working at maximum speed T using SRPT during $[t_0, t]$. Since SRPT maximizes the number of jobs completed by any time [86], the number of jobs completed after t_0 by IdleLonger(SAJC) is at least the number of jobs that are released and completed in $[t_0, t]$ by OPT. It follows that $n_{\mathrm{a}} - n_{\mathrm{o}} \leq \max\{T^{\alpha}, \sigma\}$. □

Intuitively, SAJC sometimes works at speed T, which is slower than $(n_{\mathrm{a}} + \sigma)^{1/\alpha}$, and this leads to a longer working period and hence an excess of flow time and static energy. At such time, we can use Lemma 3.14 to bound the rate of increase of excess flow time in terms of $\frac{\mathrm{d}C^*}{\mathrm{d}t}$, yet it is difficult to bound the rate of increase of excess static energy and show $\frac{\mathrm{d}G_{\mathrm{w}}}{\mathrm{d}t} + \frac{\mathrm{d}\Phi}{\mathrm{d}t} = O(\frac{\mathrm{d}C^*}{\mathrm{d}t} + \frac{\mathrm{d}F_{\mathrm{i}}}{\mathrm{d}t})$.

We extend the potential analysis with yet another new notion E_{ws}, the *total working static energy*, which is the static power σ times the total length of working intervals. When IdleLonger(SAJC) is working, $\frac{\mathrm{d}E_{\mathrm{ws}}}{\mathrm{d}t} = \sigma$. It allows us to show the following lemma.

Lemma 3.15. *At any time when no discrete events occur,* $\frac{\mathrm{d}G_{\mathrm{w}}}{\mathrm{d}t} + \frac{\mathrm{d}\Phi}{\mathrm{d}t} \leq \beta(1 + \frac{1}{\alpha})\frac{\mathrm{d}C^*}{\mathrm{d}t} + (\beta - 2)\frac{\mathrm{d}F_{\mathrm{i}}}{\mathrm{d}t} + \zeta\frac{\mathrm{d}E_{\mathrm{ws}}}{\mathrm{d}t}$.

To prove Lemma 3.15, we follow the framework in Section 3.3. When IdleLonger(SAJC) is working, $\frac{\mathrm{d}G_{\mathrm{w}}}{\mathrm{d}t} = s_{\mathrm{a}}^{\alpha} + \sigma + n_{\mathrm{a}}$, $\frac{\mathrm{d}F_{\mathrm{i}}}{\mathrm{d}t} = 0$ and $\frac{\mathrm{d}E_{\mathrm{ws}}}{\mathrm{d}t} = \sigma$; otherwise, $\frac{\mathrm{d}G_{\mathrm{w}}}{\mathrm{d}t} = 0$, $\frac{\mathrm{d}F_{\mathrm{i}}}{\mathrm{d}t} = n_{\mathrm{a}}$ and $\frac{\mathrm{d}E_{\mathrm{ws}}}{\mathrm{d}t} = 0$. When OPT is working, $\frac{\mathrm{d}C^*}{\mathrm{d}t} = s_{\mathrm{o}}^{\alpha} + \sigma + n_{\mathrm{o}}$; otherwise, $\frac{\mathrm{d}C^*}{\mathrm{d}t} \geq n_{\mathrm{o}}$. To upper bound $\frac{\mathrm{d}\Phi}{\mathrm{d}t}$, we divide the analysis of $\frac{\mathrm{d}\Phi}{\mathrm{d}t}$ into two parts: (i) the execution of SAJC affects $\phi(q)$ for $q \in (q_{\mathrm{a}} - s_{\mathrm{a}}\mathrm{d}t, q_{\mathrm{a}}]$, and (ii) the execution of OPT affects $\phi(q)$ for $q \in (q_{\mathrm{o}} - s_{\mathrm{o}}\mathrm{d}t, q_{\mathrm{o}}]$. We denote these changes as $\mathrm{d}\Phi_1$ and $\mathrm{d}\Phi_2$, respectively, and $\mathrm{d}\Phi = \mathrm{d}\Phi_1 + \mathrm{d}\Phi_2$.

Claim 3.16. (i) *Consider* $\frac{\mathrm{d}\Phi_1}{\mathrm{d}t}$. *If* $0 < s_{\mathrm{a}} < T$, *or* $s_{\mathrm{a}} = T$ *and* $n_{\mathrm{a}} - n_{\mathrm{o}} \leq -\sigma$, *then* $\frac{\mathrm{d}\Phi_1}{\mathrm{d}t} \leq -\beta(n_{\mathrm{a}} + \sigma - n_{\mathrm{o}})$; *else, if* $s_{\mathrm{a}} = T$ *and* $n_{\mathrm{a}} - n_{\mathrm{o}} > \sigma$, *then* $\frac{\mathrm{d}\Phi_1}{\mathrm{d}t} \leq -\beta(n_{\mathrm{a}} - (1 + 1/\alpha)n_{\mathrm{o}})$; *otherwise,* $\frac{\mathrm{d}\Phi_1}{\mathrm{d}t} \leq 0$. (ii) $\frac{\mathrm{d}\Phi_2}{\mathrm{d}t} \leq \beta(n_{\mathrm{a}} + \sigma)^{1 - 1/\alpha} s_{\mathrm{o}}$.

Proof. (i) We analyze $\frac{\mathrm{d}\Phi_1}{\mathrm{d}t}$. Using the same arguments in the proof of Claim 3.9(i), we have the following Inequality (3.1):

$$\frac{\mathrm{d}\Phi_1}{\mathrm{d}t} \leq -\beta(n_{\mathrm{a}} + \sigma - n_{\mathrm{o}})(n_{\mathrm{a}} + \sigma)^{-1/\alpha} s_{\mathrm{a}} \ .$$

If $s_{\mathrm{a}} = 0$, then Inequality (3.1) implies $\frac{\mathrm{d}\Phi_1}{\mathrm{d}t} \leq 0$. If $0 < s_{\mathrm{a}} < T$, then $s_{\mathrm{a}} = (n_{\mathrm{a}} + \sigma)^{1/\alpha}$ and hence Inequality (3.1) implies $\frac{\mathrm{d}\Phi_1}{\mathrm{d}t} \leq -\beta(n_{\mathrm{a}} + \sigma - n_{\mathrm{o}})$.

Now consider $s_a = T$. Note that $T \le (n_a+\sigma)^{1/\alpha}$. If $n_a - n_o \le -\sigma$, then Inequality (3.1) also implies $\frac{d\Phi_1}{dt} \le -\beta(n_a+\sigma-n_o)$. Otherwise, if $-\sigma < n_a - n_o \le \sigma$, then Inequality (3.1) simply implies $\frac{d\Phi_1}{dt} \le 0$. It remains to consider the case that $n_a - n_o > \sigma$. By Lemma 3.14, we have $\sigma < n_a - n_o \le T^\alpha$, i.e., $(n_a - n_o)^{1/\alpha} \le T$. Note that for any non-negative x and y, if $x \le y$, then $1+\sigma/x \ge 1+\sigma/y \ge (1+\sigma/y)^{1/\alpha}$, and thus $(x+\sigma)/(y+\sigma)^{1/\alpha} \ge x/y^{1/\alpha}$. By setting $x = n_a - n_o$ and $y = n_a$, $\frac{d\Phi_1}{dt} \le -\beta(n_a - n_o)n_a^{-1/\alpha}T$. Since $T \ge (n_a - n_o)^{1/\alpha}$, we have

$$\frac{d\Phi_1}{dt} \le \frac{-\beta(n_a - n_o)^{1+1/\alpha}}{n_a^{1/\alpha}} \le \frac{-\beta(n_a^{1+1/\alpha} - (1+1/\alpha)n_a^{1/\alpha}n_o)}{n_a^{1/\alpha}} = -\beta(n_a - (1+1/\alpha)n_o) \ .$$

(ii) To upper bound $\frac{d\Phi_2}{dt}$, we use the same proof of Claim 3.9(ii). □

We are now ready to prove the inequality of Lemma 3.15, i.e., $\frac{dG_w}{dt} + \frac{d\Phi}{dt} \le \beta(1+\frac{1}{\alpha})\frac{dC^*}{dt} + (\beta-2)\frac{dF_i}{dt} + \zeta\frac{dE_{ws}}{dt}$. Below is a case analysis depending on whether IdleLonger(SAJC) and OPT is working.

Case a: $s_a < T$. By Claim 3.16, the bound on $\frac{d\Phi}{dt}$ is the same as in the Lemma 3.8 in Section 3.3, except that the constant β is different. Furthermore, since $s_a = (n_a + \sigma)^{1/\alpha}$, the bounds on $\frac{dG_w}{dt}$, $\frac{dC^*}{dt}$ and $\frac{dF_i}{dt}$ all remain the same. Therefore, we use the same arguments in Lemma 3.8, except that we made a minor change on the constant μ to set $\mu = (\alpha+1)^{-1/\alpha}$. Then for all the four cases considered in the proof of Lemma 3.8, we can show that $\frac{dG_w}{dt} + \frac{d\Phi}{dt} \le \beta\frac{dC^*}{dt} + (\beta-2)\frac{dF_i}{dt} \le \beta(1+1/\alpha)\frac{dC^*}{dt} + (\beta-2)\frac{dF_i}{dt}$.

Case b: $s_a = T$ and $n_a - n_o \le -\sigma$. By Claim 3.16, the bound on $\frac{d\Phi}{dt}$ is the same as Case a. Again, the bounds on $\frac{dC^*}{dt}$ and $\frac{dF_i}{dt}$ remain the same. Since $s_a = T \ge (n_a - \sigma)^{1/\alpha}$, we have $\frac{dG_w}{dt} = s_a^\alpha + \sigma + n_a \le 2(n_a+\sigma)$. We can show $\frac{dG_w}{dt} + \frac{d\Phi}{dt} \le \beta(1+1/\alpha)\frac{dC^*}{dt} + (\beta-2)\frac{dF_i}{dt}$ in the same way as Case a.

Case c: $s_a = T$ and $n_a - n_o > -\sigma$. Since $s_a = T$, $\frac{dG_w}{dt} \le 2(n_a+\sigma)$. Now we show $\frac{dG_w}{dt} + \frac{d\Phi}{dt} \le \beta(1+\frac{1}{\alpha})\frac{dC^*}{dt} + (\beta-2)\frac{dF_i}{dt} + \zeta\frac{dE_{ws}}{dt}$ depending on whether OPT is working.

Case c.1: $s_o > 0$. In this case, $\frac{dG_w}{dt} \le 2(n_a+\sigma)$, $\frac{dC^*}{dt} = n_o + s_o^\alpha + \sigma$, $\frac{dF_i}{dt} = 0$, and $\frac{dE_{ws}}{dt} = \sigma$.

If $n_a - n_o \le \sigma$, by Claim 3.16, $\frac{d\Phi_1}{dt} \le 0$ and $\frac{d\Phi_2}{dt} \le \beta(n_a+\sigma)^{1-1/\alpha}s_o$. Similarly as in the proof of Lemma 3.8, using the Young's Inequality, we can further introduce any constant

$\mu > 0$ into the above bound of $\frac{\mathrm{d}\Phi_2}{\mathrm{d}t}$ (see Inequality 3.2 in Section 3.3) and thus we have

$$\begin{aligned}\frac{\mathrm{d}G_{\mathrm{w}}}{\mathrm{d}t} + \frac{\mathrm{d}\Phi}{\mathrm{d}t} &\leq 2(n_{\mathrm{a}} + \sigma) + \beta(1 - 1/\alpha)\mu^{\alpha/(\alpha-1)}(n_{\mathrm{o}} + 2\sigma) + \frac{\beta}{\alpha\mu^{\alpha}}s_{\mathrm{o}}^{\alpha} \\ &\leq (2 + \beta(1 - 1/\alpha)\mu^{\alpha/(\alpha-1)})(n_{\mathrm{o}} + 2\sigma) + \frac{\beta}{\alpha\mu^{\alpha}}s_{\mathrm{o}}^{\alpha} \ .\end{aligned}$$

By setting $\mu = (\alpha + 1)^{-1/\alpha}$ and recalling that $\beta = 2/(1 - \frac{1-1/\alpha}{(\alpha+1)^{1/(\alpha-1)}})$, we have $\beta = 2 + \beta(1 - 1/\alpha)\mu^{\alpha/(\alpha-1)}$. Therefore, $\frac{\mathrm{d}G_{\mathrm{w}}}{\mathrm{d}t} + \frac{\mathrm{d}\Phi}{\mathrm{d}t} \leq \beta(n_{\mathrm{o}} + 2\sigma) + \beta(1 + 1/\alpha)s_{\mathrm{o}}^{\alpha} \leq \beta(1 + 1/\alpha)(s_{\mathrm{o}}^{\alpha} + n_{\mathrm{o}} + \sigma) + (2\beta - \beta(1 - 1/\alpha))\sigma = \beta(1 + 1/\alpha)\frac{\mathrm{d}C^*}{\mathrm{d}t} + \beta(1 - 1/\alpha)\frac{\mathrm{d}E_{\mathrm{ws}}}{\mathrm{d}t} \leq \beta(1 + 1/\alpha)\frac{\mathrm{d}C^*}{\mathrm{d}t} + \zeta\frac{\mathrm{d}E_{\mathrm{ws}}}{\mathrm{d}t}$.

If $n_{\mathrm{a}} - n_{\mathrm{o}} > \sigma$, similarly, Claim 3.16 implies

$$\begin{aligned}\frac{\mathrm{d}G_{\mathrm{w}}}{\mathrm{d}t} + \frac{\mathrm{d}\Phi}{\mathrm{d}t} \leq& 2(n_{\mathrm{a}} + \sigma) - \beta n_{\mathrm{a}} + \beta(1 + 1/\alpha)n_{\mathrm{o}} + \beta(1 - 1/\alpha)\mu^{\alpha/(\alpha-1)}(n_{\mathrm{a}} + \sigma) + \frac{\beta}{\alpha\mu^{\alpha}}s_{\mathrm{o}}^{\alpha} \\ =& \Big(2 - \beta(1 - (1 - 1/\alpha)\mu^{\alpha/(\alpha-1)})\Big)n_{\mathrm{a}} \\ &+ \Big(2 + \beta(1 - 1/\alpha)\mu^{\alpha/(\alpha-1)}\Big)\sigma + \beta(1 + 1/\alpha)n_{\mathrm{o}} + \frac{\beta}{\alpha\mu^{\alpha}}s_{\mathrm{o}}^{\alpha} \ .\end{aligned}$$

Again, by setting $\mu = (\alpha + 1)^{-1/\alpha}$, the above inequality implies $\frac{\mathrm{d}G_{\mathrm{w}}}{\mathrm{d}t} + \frac{\mathrm{d}\Phi}{\mathrm{d}t} \leq \beta\sigma + \beta(1 + 1/\alpha)(n_{\mathrm{o}} + s_{\mathrm{o}}^{\alpha}) \leq \beta(1 + 1/\alpha)(n_{\mathrm{o}} + s_{\mathrm{o}}^{\alpha} + \sigma) = \beta(1 + 1/\alpha)\frac{\mathrm{d}C^*}{\mathrm{d}t}$.

Case c.2: $s_{\mathbf{o}} = 0$**.** In this case, $\frac{\mathrm{d}G_{\mathrm{w}}}{\mathrm{d}t} \leq 2(n_{\mathrm{a}} + \sigma)$, $\frac{\mathrm{d}C^*}{\mathrm{d}t} \geq n_{\mathrm{o}}$, $\frac{\mathrm{d}F_{\mathrm{i}}}{\mathrm{d}t} = 0$, and $\frac{\mathrm{d}E_{\mathrm{ws}}}{\mathrm{d}t} = \sigma$.

If $n_{\mathrm{a}} - n_{\mathrm{o}} \leq \sigma$, by Claim 3.16,

$$\frac{\mathrm{d}G_{\mathrm{w}}}{\mathrm{d}t} + \frac{\mathrm{d}\Phi}{\mathrm{d}t} \leq 2(n_{\mathrm{a}} + \sigma) \leq 2(n_{\mathrm{o}} + 2\sigma) = 2n_{\mathrm{o}} + 4\sigma \leq \beta(1 + 1/\alpha)\frac{\mathrm{d}C^*}{\mathrm{d}t} + \zeta\frac{\mathrm{d}E_{\mathrm{ws}}}{\mathrm{d}t} \ ,$$

where the last inequality follows from the fact that $\beta \geq 2$ and $\zeta \geq 4$.

If $n_{\mathrm{a}} - n_{\mathrm{o}} > \sigma$, by Claim 3.16,

$$\begin{aligned}\frac{\mathrm{d}G_{\mathrm{w}}(t)}{\mathrm{d}t} + \frac{\mathrm{d}\Phi(t)}{\mathrm{d}t} &\leq 2(n_{\mathrm{a}} + \sigma) - \beta n_{\mathrm{a}} + \beta(1 + 1/\alpha)n_{\mathrm{o}} = (2 - \beta)n_{\mathrm{a}} + 2\sigma + \beta(1 + 1/\alpha)n_{\mathrm{o}} \\ &\leq \beta(1 + 1/\alpha)\frac{\mathrm{d}C^*}{\mathrm{d}t} + 2\frac{\mathrm{d}E_{\mathrm{ws}}}{\mathrm{d}t} \ ,\end{aligned}$$

where the last inequality follows from the fact that $\beta \geq 2$ and $\zeta \geq 2$.

In conclusion, for any case, $\frac{\mathrm{d}G_{\mathrm{w}}}{\mathrm{d}t} + \frac{\mathrm{d}\Phi}{\mathrm{d}t} \leq \beta(1+1/\alpha)\frac{\mathrm{d}C^*}{\mathrm{d}t} + (\beta-2)\frac{\mathrm{d}F_{\mathrm{i}}}{\mathrm{d}t} + \zeta\frac{\mathrm{d}E_{\mathrm{ws}}}{\mathrm{d}t}$, completing the proof of Lemma 3.15. Lemma 3.15 implies $G_{\mathrm{w}} \leq \beta(1+1/\alpha)C^* + (\beta-2)F_{\mathrm{i}} + \zeta E_{\mathrm{ws}}$. We further observe that $E_{\mathrm{ws}} \leq C^*$. Then Theorem 3.12 follows.

Lemma 3.17. *With respect to* IdleLonger(SAJC)*,* $E_{\mathrm{ws}} \leq C^*$.

Proof. Let x be the total size of all jobs. First, consider $T \geq \sigma^{1/\alpha}$. When IdleLonger(SAJC) is working, its speed is at least $\sigma^{1/\alpha}$ and thus $E_{\mathrm{ws}} \leq \sigma \cdot (x/\sigma^{1/\alpha}) = x\sigma^{1-1/\alpha}$. Note that running a job at the critical speed $s_{\mathrm{crit}} = (\sigma/(\alpha-1))^{1/\alpha}$ minimizes the energy usage of the job. Therefore, the energy usage of any schedule and hence C^* is at least $(x/s_{\mathrm{crit}}) \cdot (s_{\mathrm{crit}}^{\alpha} + \sigma) \geq (\alpha/(\alpha-1)^{1-1/\alpha}) \cdot (x\sigma^{1-1/\alpha}) \geq (\alpha/(\alpha-1)^{1-1/\alpha})E_{\mathrm{ws}}$. For any $\alpha > 1$, $(\alpha/(\alpha-1)^{1-1/\alpha}) \geq 1$, and hence $C^* \geq E_{\mathrm{ws}}$.

Now consider the case that $T < \sigma^{1/\alpha}$. When IdleLonger(SAJC) is working, its speed is always T and thus $E_{\mathrm{ws}} \leq \sigma \cdot (x/T)$. If $s_{\mathrm{crit}} \leq T$, then the energy usage of any schedule and hence C^* is at least $(x/s_{\mathrm{crit}}) \cdot (s_{\mathrm{crit}}^{\alpha} + \sigma) \geq \sigma \cdot (x/s_{\mathrm{crit}}) \geq \sigma \cdot (x/T) = E_{\mathrm{ws}}$. Otherwise, if $s_{\mathrm{crit}} > T$, then the optimal schedule would always run a job at speed T. It is because when running a job below s_{crit}, the slower the speed, the more energy as well as flow time are incurred. Thus, $C^* \geq (x/T) \cdot (T^{\alpha} + \sigma) \geq \sigma \cdot (x/T) = E_{\mathrm{ws}}$. □

Chapter 4

Non-migratory Multi-processor Flow-Energy Scheduling

This chapter extends the study of flow-energy scheduling to the setting with $m \geq 2$ processors. This extension is not only of theoretical interest, as modern processors adopt multi-core technology (dual-core and quad-core are getting common). A multi-core processor is essentially a pool of parallel processors. The formal problem is defined as follows. Given a job set J, we want to schedule J on a pool of $m \geq 2$ processors. Jobs are sequential in nature and cannot be executed by more than one processor in parallel. All processors are identical and a job can be executed in any processor. A processor can run at any speed between 0 and T; when running at speed s, it processes s units of work and consumes s^α units of energy in each unit of time, where $\alpha \geq 2$. Preemption is allowed and a preempted job can be resumed at the point of preemption. We differentiate two types of schedules: a migratory schedule can move partially-executed jobs from one processor to another processor without any penalty, and a non-migratory schedule dispatches each job to one of the m processors and runs the job entirely in that processor. In practice, migrating jobs requires overheads and is avoided in many applications. To make our work more meaningful, we aim at schedules that do not require job migration among processors. The objective is to minimize the total flow time of all jobs plus the total energy usage of all processors.

In Section 4.1, we give some definitions and notations necessary for discussion; in particular, the notions of fractional weight and fractional flow are introduced. We also

introduce the definitions of critical speed and global critical speed. In Section 4.2, we introduce the job dispatching policy CRR and present the online algorithm CRR-A. We show the following two competitive ratios of CRR-A for flow plus energy. First, for jobs of power-of-2 size (i.e., every job has size 2^k for some k), CRR-A is $O(\log P)$-competitive, where P is the ratio of the maximum job size to the minimum job size. Second, for jobs of arbitrary size, CRR-A is $O(1)$-competitive when using processors with slightly higher maximum speed. These results are based on an offline result to eliminate migration (Sections 4.3 and 4.4). Section 4.3 considers jobs of power-of-2 size, showing that given any migratory schedule, we can transform it to a CRR schedule (which is nonmigratory) such that the flow time is increased by an $O(\log P)$ factor. Section 4.4 considers jobs of arbitrary size, showing how to construct a CRR schedule from an optimal schedule with an increase of $O(1)$ factor in flow plus energy and a mild increase in maximum speed. In Section 4.5, we show that any online scheduling algorithm (without extra maximum speed) is $\Omega(\log P)$-competitive. This lower bound holds even if jobs are all power-of-2 size and thus implies that the competitiveness of CRR-A is optimal (up to a constant factor) for jobs of power-of-2 size.

Remarks for fixed-speed scheduling. The analysis of CRR also reveals its performance in the context of traditional flow-time scheduling, where processors are of fixed-speed and the concern is on flow time only. In this case, CRR (plus SRPT for individual processor) would give a non-migratory online algorithm which, when compared with the optimal migratory algorithm, can have a competitive ratio of one or even any constant arbitrarily smaller than one, when using sufficiently fast processors. The competitive ratio can be $(56.72/s)$ when using s-speed processors for $s \geq 56.72$.[1] Note that if migration is allowed, the efficiency can be much better as McCullough and Torng [72] have showed that the migratory algorithm SRPT is $\frac{1}{s}$-competitive when using s-speed processors, where $s \geq 2 - \frac{1}{m}$.

[1]Precisely, we can show that for any $\epsilon > 0$ such that $\frac{5}{\epsilon^2}(2+\epsilon) \geq 1$, CRR using processors of speed $(1+\epsilon)^2$ is $\frac{5}{\epsilon^2}(2+\epsilon)$-competitive. Combining with the result of McCullough and Torng [72] that SRPT is $\frac{1}{s}$-competitive for single processor when using processor of speed s, this implies CRR using processors of speed $s(1+\epsilon)^2$ is $\frac{5}{s\epsilon^2}(2+\epsilon)$-competitive. By changing variables, we can show that with $\sigma = 56.72$, CRR using processors of speed $s \geq \sigma$ is $\frac{5\sigma(\sqrt{\sigma}+1)}{s(\sqrt{\sigma}-1)^2}$-competitive, and $\frac{5\sigma(\sqrt{\sigma}+1)}{(\sqrt{\sigma}-1)^2} \approx 56.72$.

4.1 Preliminaries

Jobs and fractional weight. We use $r(j)$ and $p(j)$ to denote respectively the release time and size of a job j. For a set J of jobs, we let $p(J) = \sum_{j \in J} p(j)$ be the total size of J, and let $P(J)$ (or simply P when it is clear what J is referring to) denote the ratio of the largest job size to the smallest job size. We define the *fractional weight* of a job j at a particular time to be $q/p(j)$, where q is the remaining work of j at the time of concern. Note that the fractional weight of j decreases from 1 to 0 in the course of executing j.

Schedules and fractional flow. Throughout this chapter, migratory and non-migratory schedules are usually represented by the symbols $\mathcal{S}$ and $\mathcal{N}$, respectively. With respect to a schedule $\mathcal{S}$ of a job set J, we use *max-speed*$_J(\mathcal{S})$, $E_J(\mathcal{S}), F_J(\mathcal{S})$, and $\hat{F}_J(\mathcal{S})$ to denote the maximum speed, energy usage, total flow time, and total fractional flow time of $\mathcal{S}$, respectively. Note that the total flow is $F_J(\mathcal{S}) = \int_0^\infty n(t)\,\mathrm{d}t$, where $n(t)$ is the number of unfinished jobs at time t. On the other hand, the total *fractional flow* $\hat{F}_J(\mathcal{S})$ is defined as $\hat{F}_J(\mathcal{S}) = \int_0^\infty \hat{w}(t)\,\mathrm{d}t$, where $\hat{w}(t)$ is the total *fractional weight* of unfinished jobs at time t. Obviously, $\hat{F}_J(\mathcal{S}) \leq F_J(\mathcal{S})$. Note that processor speed can vary dynamically and the time to execute a job j is not necessarily equal to $p(j)$. We use $X(j)$ to denote the *execution time* of job j (i.e., flow time minus waiting time), and define $X_J(\mathcal{S}) = \sum_{j \in J} X(j)$ to be the total execution time of $\mathcal{S}$. Our objective is to minimize total flow time plus energy. Yet it is also helpful to analyze the fractional flow time plus energy. It is convenient to define $G_J(\mathcal{S}) = F_J(\mathcal{S}) + E_J(\mathcal{S})$ and $\hat{G}_J(\mathcal{S}) = \hat{F}_J(\mathcal{S}) + E_J(\mathcal{S})$. When the context is clear, we will omit the subscript J from the above notations.

The following lemma shows a lower bound on $G(\mathcal{S})$ which depends on $p(J)$, irrelevant of the number of processors.

Lemma 4.1. *For any m-processor schedule $\mathcal{S}$ for a job set J, $G(\mathcal{S}) \geq \frac{\alpha}{(\alpha-1)^{1-1/\alpha}} p(J)$.*

Proof. Suppose that a job j in $\mathcal{S}$ has flow time t. The energy usage for j is minimized if j is run at constant speed $p(j)/t$ throughout, and it is at least $(p(j)/t)^\alpha t = p(j)^\alpha / t^{\alpha-1}$. Since $t + p(j)^\alpha / t^{\alpha-1}$ is minimized when $t = (\alpha-1)^{1/\alpha} p(j)$, we have $t + p(j)^\alpha / t^{\alpha-1} \geq \frac{\alpha}{(\alpha-1)^{1-1/\alpha}} p(j)$. Summing over all jobs, we obtain the desired lower bound. □

Global critical speed and flow time in multi-processor schedules. To optimize flow time plus energy, it is useful to define the *global critical speed* to be $1/(\alpha-1)^{1/\alpha}$.

Throughout this chapter, we assume that at any time the optimal schedules never run a job at speed less than the global critical speed $1/(\alpha-1)^{1/\alpha}$, and the maximum speed T is at least the global critical speed. The assumption stems from an observation (Lemma 4.3) that if necessary, a multi-processor schedule can be transformed without increasing the flow time plus energy so that it never runs a job j at speed less than the global critical speed. It also implies $X(j) \leq (\alpha-1)^{1/\alpha}p(j)$, which is at most $1.322p(j)$ for any $\alpha \geq 2$.

Lemma 4.3 makes use of a result by Albers and Fujiwara [2] that when scheduling a single job j on a processor for minimizing total flow time plus energy, j should be executed at the global critical speed, i.e., $1/(\alpha-1)^{1/\alpha}$.

Lemma 4.2. [2] *At any time after a job j has been run on a processor for a while, suppose that we want to further execute j for another $x > 0$ units of work and minimize the flow time plus energy incurred to this period. The optimal strategy is to let the processor always run at the global critical speed.*

Lemma 4.3. *Given any m-processor schedule $\mathcal{S}$ for a job set J, we can construct an m-processor schedule $\mathcal{S}'$ for J such that $\mathcal{S}'$ never runs a job at speed less than the global critical speed and $G(\mathcal{S}') \leq G(\mathcal{S})$. Moreover, $\mathcal{S}'$ needs migration if and only if $\mathcal{S}$ does; and max-speed$(\mathcal{S}') \leq \max\{$max-speed$(\mathcal{S}), 1/(\alpha-1)^{1/\alpha}\}$.*

Proof. Assume that there is a time interval I in $\mathcal{S}$ during which a processor i is running a job j below the global critical speed. If $\mathcal{S}$ needs migration, we transform $\mathcal{S}$ to a migratory schedule $\mathcal{S}_1$ of J such that job j is always scheduled in processor i. This can be done by swapping the schedules of processor i and other processors for different time intervals. If $\mathcal{S}$ does not need migration, job j is entirely scheduled in processor i and $\mathcal{S}_1$ is simply $\mathcal{S}$. In both cases, $G(\mathcal{S}_1) = G(\mathcal{S})$.

We can then improve $G(\mathcal{S}_1)$ by modifying the schedule of processor i in $\mathcal{S}_1$ as follows. Let x be the amount of work of j processed during I on processor i. First, we schedule this amount of work of j at the global critical speed. Note that the time required is shortened. Then we move the remaining schedule of j backward to fill up the time shortened. By Lemma 4.2, the flow time plus energy for j is preserved. Other jobs in J are left intact. To obtain the schedule $\mathcal{S}'$, we repeat this process to eliminate all such intervals I. □

Assumption on maximum speed T. We assume that the maximum speed T is at least the global critical speed. Otherwise, any multi-processor schedule including the

optimal one would always run a job at the maximum speed. It is because when running a job below the global critical speed, the slower the speed, the more total flow time plus energy is incurred. In other words, the problem is reduced to minimizing flow time alone.

Critical speed and fractional flow time in single-processor schedules. When analyzing non-migratory schedules, we sometimes need to focus on individual processors and analyze the fractional flow time for each processor. In this case we need to consider fractional weight and make a different assumption of the minimum speed and transformation. At any time, we define the *critical speed* of a job j to be $(q/(\alpha-1)p(j))^{1/\alpha}$, where $q \leq p(j)$ denotes the remaining work of job j at the time of concern. Note that the critical speed of j changes over time. We give a non-trivial observation that when scheduling a single job j on a processor for minimizing the fractional flow time plus energy, the processor should always keep up with its critical speed (Lemma 4.4). Then we can show that a single-processor schedule can be transformed without increasing the fractional flow time plus energy so that it never runs a job at speed less than its critical speed (Lemma 4.5).

Lemma 4.4. *At any time after a job j has been run on a processor for a while, suppose that we want to further execute j for another $x > 0$ units of work and minimize the fractional flow time plus energy incurred to this period. The optimal strategy is to let the processor always run at the critical speed.*

Proof. Let $p = p(j)$, and let $q \leq p$ be the remaining work of j. We first consider the case to further process an infinitesimal amount of work (i.e., $x \to 0$). In this case we can assume that the speed s is constant. The time required is $t = x/s$, and the fractional flow time plus energy incurred, denoted by $\Delta\hat{G}$, is $s^\alpha t + (\frac{q-x/2}{p})t$. To find the optimal speed (or equivalently, optimal time) to minimize $\Delta\hat{G}$, we set $\frac{d\Delta\hat{G}}{dt} = 0$. That is,

$$(1-\alpha)\left(\frac{x}{t}\right)^\alpha + \frac{q-x/2}{p} = 0 \ ,$$

or equivalently,

$$\frac{x}{t} = \left(\frac{q-x/2}{(\alpha-1)p}\right)^{1/\alpha} \ .$$

Since $x \to 0$, we have $s = x/t = (q/(\alpha-1)p)^{1/\alpha}$.

Next, consider the case that x is arbitrarily large. Consider a schedule in which the

processor is not running at the critical speed after processing $x' < x$ units of work. Let Δq be an infinitesimal amount. From the discussion above we can change the speed to the critical speed to obtain the optimal fractional flow time plus energy for processing the next Δq units of work. Thus we can eventually obtain a schedule that always runs at the critical speed. □

Lemma 4.5. *Given any single-processor schedule $\mathcal{N}$ for a job set J, we can construct another schedule $\mathcal{N}'$ for J such that $\mathcal{N}'$ never runs a job at speed less than its critical speed and $\hat{G}(\mathcal{N}') \leq \hat{G}(\mathcal{N})$. Moreover, max-speed$(\mathcal{N}') \leq \max\{$max-speed$(\mathcal{N}), 1/(\alpha-1)^{1/\alpha}\}$.*

Proof. The transformation uses Lemma 4.4 and the same arguments in the proof of Lemma 4.3. □

4.2 The Online Algorithm

This section presents the formal definition of the online algorithm CRR-A, which makes use of any $O(1)$-competitive online algorithm A for minimizing flow plus energy on a single processor, and produces a non-migratory schedule for $m \geq 2$ processors. As mentioned earlier, the analysis of CRR-A stems from an offline result to eliminate migration. We will state two theorems about this offline result (to be proved in later sections), one for jobs of power-of-2 size and another for jobs of arbitrary size. Then we analyze the performance of CRR-A based on this offline result.

CRR(λ) dispatching. Consider any $\lambda > 0$. We define a CRR(λ) (or simply CRR) schedule based on the following notion of *classes*. A job is said to be in class k if its size is in the range $((1+\lambda)^{k-1}, (1+\lambda)^k]$. In a CRR($\lambda$) schedule, jobs of the same class are dispatched upon their arrival to the m processors using a round-robin strategy, and different classes are handled independently. Jobs once dispatched to a processor will be processed there until they finish. Thus a CRR(λ) schedule is non-migratory in nature.

The intuition of using a CRR schedule comes from the offline result on eliminating migration (Theorem 4.6 and Theorem 4.7). Theorem 4.6 states that for jobs of power-of-2 size, there is a CRR schedule such that the total flow time plus energy is $O(\log P)$ times that of the optimal migratory offline schedule and the maximum speed remains the same. Furthermore, Theorem 4.7 states that for jobs of arbitrary size, there is a CRR schedule

such that the total flow time plus energy is $O(1)$ times that of the optimal migratory offline schedule and the maximum allowable speed is only slightly higher. Theorems 4.6 and 4.7 will be proved in Sections 4.3 and 4.4, respectively. Note that in Theorem 4.7, we will also compare CRR-A with an optimal offline non-migratory schedule; in such case, the competitive ratio can be improved by a factor of 2.5. We define the constant η_ϵ as $(1+\epsilon)^\alpha[(1+\epsilon)^{\alpha-1} + (1-1/\alpha)(2+\epsilon)/\epsilon^2]$.

Theorem 4.6. *Given a job set J, where jobs are of power-of-2 size, let $\mathcal{S}$ be a migratory schedule of J. Then there is a* CRR(1) *schedule $\mathcal{N}$ for J such that $G(\mathcal{N}) \leq (5.966 \log P + 2)G(\mathcal{S})$, and max-speed$(\mathcal{N}) \leq$ max-speed$(\mathcal{S})$.*

Theorem 4.7. *Given a job set J, let $\mathcal{O}_1$ and $\mathcal{O}_2$ be respectively an optimal non-migratory schedule and an optimal migratory schedule for J using maximum speed T. Then,*

i. for any $\epsilon > 0$, there is a CRR(ϵ) schedule $\mathcal{S}_1$ for J such that $G(\mathcal{S}_1) \leq 2\eta_\epsilon G(\mathcal{O}_1)$, and max-speed$(\mathcal{S}_1) \leq (1+\epsilon)^2 \times$ max-speed$(\mathcal{O}_1)$; and

ii. for any $\epsilon > 0$, there is a CRR(ϵ) schedule $\mathcal{S}_2$ for J such that $G(\mathcal{S}_2) \leq 5\eta_\epsilon G(\mathcal{O}_2)$, and max-speed$(\mathcal{S}_2) \leq (1+\epsilon)^2 \times$ max-speed$(\mathcal{O}_2)$.

The above theorems naturally suggest an online algorithm that first dispatches jobs using the policy CRR, and then schedules jobs in each processor independently and in a way that is competitive in the single-processor setting. For the latter, we make use of any $O(1)$-competitive algorithm A, e.g., SRPT-AJC in Chapter 2, and BCP in [13].

> **Algorithm CRR-A.** Jobs are dispatched to the m processors with the CRR(λ) policy. Jobs in each processor are scheduled independently using algorithm A.

Analysis of CRR-A. We analyze the competitiveness of CRR-A in the bounded speed model. Note that our analysis can also be applied to the infinite speed model, though it is of less interest. Suppose the algorithm A is β-competitive for flow plus energy in the single-processor setting (in the bounded speed model). With Theorems 4.6 and 4.7, we can easily derive the performance of CRR-A against the optimal migratory or non-migratory algorithm. Recall that λ is the parameter for classifying jobs in CRR(λ).

Corollary 4.8. *In the bounded speed model, the performance of* CRR-A *for minimizing flow time plus energy on $m \geq 2$ processors is as follows.*

i. For jobs of power-of-2 size, against a migratory optimal schedule, CRR-A *(with $\lambda = 1$) is $(5.966 \log P + 2)\beta$-competitive using processors with maximum speed T.*

ii. For jobs of arbitrary size, for any $\epsilon > 0$, against a non-migratory optimal schedule, CRR-A *(with $\lambda = \epsilon$) is $2\eta_\epsilon\beta$-competitive when using processors with maximum speed relaxed to $(1+\epsilon)^2 T$.*

iii. For jobs of arbitrary size, for any $\epsilon > 0$, against a migratory optimal schedule, CRR-A *(with $\lambda = \epsilon$) is $5\eta_\epsilon\beta$-competitive when using processors with maximum speed relaxed to $(1+\epsilon)^2 T$.*

Proof. To show (i), consider a job set J_2, where jobs are of power-of-2 size. Let $\mathcal{O}_2$ be the optimal migratory schedule for J_2. By Theorem 4.6, there exists a CRR(1) schedule $\mathcal{N}_2$ for J_2 such that $G(\mathcal{N}_2) \leq (5.966 \log P + 2)G(\mathcal{O}_2)$, and *max-speed*$(\mathcal{N}_2) =$ *max-speed*$(\mathcal{O}_2)$. Let $\mathcal{S}_2$ be the CRR(1) schedule produced by CRR-A (with $\lambda = 1$) for J_2. Applying to individual processors the fact that algorithm A is β-competitive for flow plus energy, we conclude that $G(\mathcal{S}_2) \leq \beta G(\mathcal{N}_2) \leq (5.966 \log P + 2)\beta G(\mathcal{O}_2)$, and *max-speed*$(\mathcal{S}_2) \leq$ *max-speed*$(\mathcal{O}_2)$.

To show (ii) and (iii), we consider an arbitrary job set J. Let $\mathcal{O}$ be the optimal non-migratory schedule for J. Consider any $\epsilon > 0$. By Theorem 4.7 (i), there exists a CRR(ϵ) schedule $\mathcal{S}$ for J such that $G(\mathcal{S}) \leq 2\eta_\epsilon G(\mathcal{O})$ and *max-speed*$(\mathcal{S}) \leq (1+\epsilon)^2 \times$ *max-speed*$(\mathcal{O})$. Let $\mathcal{S}'$ be the $CRR(\epsilon)$ schedule produced by CRR-A (with $\lambda = \epsilon$) for J. Applying to individual processors the fact that algorithm A is β-competitive for flow plus energy, we conclude that $G(\mathcal{S}') \leq \beta G(\mathcal{S}) \leq 2\eta_\epsilon\beta G(\mathcal{O})$, and *max-speed*$(\mathcal{S}') \leq (1+\epsilon)^2 \times$ *max-speed*$(\mathcal{O})$. Thus, (ii) follows. The analysis of the migratory case, i.e., (iii), is the same. □

4.3 Jobs of Power-of-2 Size

This section is devoted to proving Theorem 4.6 in Section 4.2. We focus on jobs of power-of-2 size. Now we give an overview of an offline method for transforming a given migratory schedule to a non-migratory schedule. Roughly speaking, the method involves

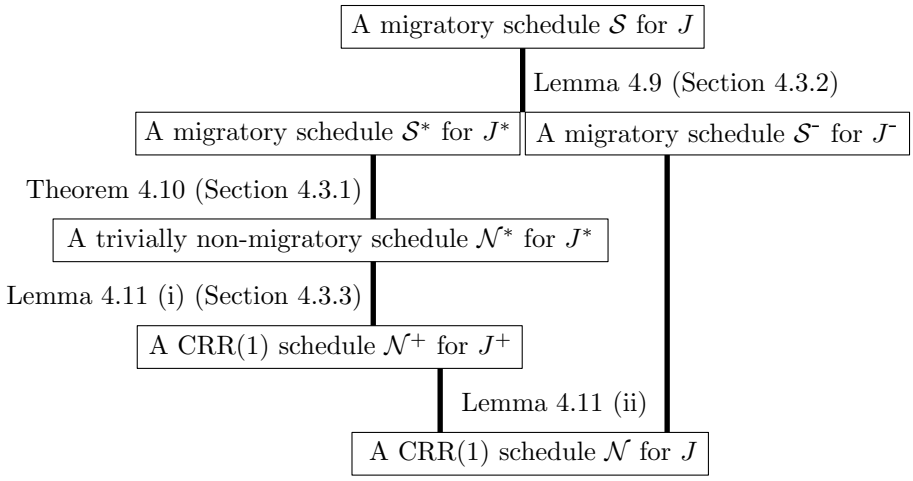

Figure 4.1: Transforming a migratory schedule to a non-migratory schedule for jobs J of power-of-2 size

eliminating migration in two types of schedules: schedule for a set of special jobs called parallel jobs, and schedule for any set of jobs of power-of-2 size. This section states the lemmas and theorems related to these two steps (to be proved respectively in Section 4.3.1 and in Sections 4.3.2 and 4.3.3). More importantly, we explain how to apply these results to obtain the main theorem of transforming a migratory schedule to a non-migratory schedule. To ease discussion, Figure 4.1 is given as an overview of how various lemmas and theorems are applied.

We first define and recall some definitions to be used in this section. We consider a set J of jobs of power-of-2 size and denote the maximum-minimum ratio of job sizes by P. Note that J has at most $\log P$ distinct job sizes. A job set is said to be *m-parallel* if the jobs can be partitioned into batches, each with m jobs of identical release time and size. For an m-parallel job set, a *trivially non-migratory* schedule is defined as any schedule in which all processors have identical schedules and at any time, execute respectively the m different jobs in a batch.

The key ideas and steps involved in the transformation are as follows.

1. It is easy to convert J into an m-parallel job set J^* (see the procedure **Make_Parallel** below). And a migratory schedule for J would naturally define a migratory schedule for J^*.

2. More interestingly, the migratory schedule for J^* can be transformed to a trivially non-migratory schedule for J^*.

3. Finally, any trivially non-migratory schedule for J^* can be transformed to a CRR(1) schedule for J.

Furthermore, all transformations incur only a moderate increase in the flow time plus energy.

The procedure **Make_Parallel** is defined as follows. It transforms J to two job sets, each still has at most $\log P$ distinct job sizes.

J^+: Jobs of the same size in J are grouped into batches of m jobs, in the order of release time. The last batch may not be full. J^+ is the set of all jobs that are in a "full" batch. Let $J^- = J - J^+$.

J^*: For each batch in J^+, we pick a job with the earliest release time as the *leader*, and change the release time of every other job to that of the leader. We use $r(j)$ and $r^*(j)$ to denote the original and the new release time of a job j. The resulting job set is denoted by J^*, which is m-parallel.

The following lemmas and theorem define a sequence of transformations from a migratory schedule $\mathcal{S}$ of J to different intermediate schedules (for J^+, J^- and J^*) and eventually to a non-migratory CRR(1) schedule of J. See Figure 4.1 for a summary of these transformations. Each transformation consists of a few steps only; yet the analysis of the increase of flow time and energy is often quite involved. The details and proofs will be given in Sections 4.3.1, 4.3.2 and 4.3.3. In the rest of this section, we need to deal with different migratory and non-migratory schedules of the job sets J, J^+, J^- and J^*; it is noteworthy that their migratory schedules are always denoted by $\mathcal{S}, \mathcal{S}^+, \mathcal{S}^-$, and $\mathcal{S}^*$, respectively, and their non-migratory schedules are denoted by $\mathcal{N}, \mathcal{N}^+, \mathcal{N}^-$, and $\mathcal{N}^*$, respectively.

Lemma 4.9. *Given a migratory schedule $\mathcal{S}$ for J, we can construct two migratory schedules $\mathcal{S}^*$ for J^* and $\mathcal{S}^-$ for J^- in such a way that $G(\mathcal{S}^*) + G(\mathcal{S}^-) \leq G(\mathcal{S}) + 1.322(\log P + 1) \cdot p(J^+)$. Both $\mathcal{S}^*$ and $\mathcal{S}^-$ use at most the maximum speed of $\mathcal{S}$.*

The next transformation is the most non-trivial, it converts a migratory schedule for J^* to a trivially non-migratory schedule.

Theorem 4.10. *Given a migratory m-processor schedule $\mathcal{S}^*$ for J^*, we can construct a trivially non-migratory schedule $\mathcal{N}^*$ for J^* such that $G(\mathcal{N}^*) \leq G(\mathcal{S}^*) + 2\log P \cdot p(J^*)$. Furthermore, max-speed$(\mathcal{N}^*) \leq$ max-speed$(\mathcal{S}^*)$.*

Recall that jobs in J^* may have their release time moved backward and thus we need another transformation of $\mathcal{N}^*$ to obtain a valid schedule for J^+ and then J.

Lemma 4.11. (i) *Given a trivially non-migratory schedule $\mathcal{N}^*$ of J^*, we can construct a* CRR(1) *schedule $\mathcal{N}^+$ for J^+ such that $G(\mathcal{N}^+) \leq G(\mathcal{N}^*) + 1.322\log P \cdot p(J^+)$. Moreover, max-speed$(\mathcal{N}^+) \leq$ max-speed$(\mathcal{N}^*)$.* (ii) *Together with a migratory schedule $\mathcal{S}^-$ for J^-, we can construct a* CRR(1) *schedule $\mathcal{N}$ for J, such that $G(\mathcal{N}) \leq G(\mathcal{N}^+) + G(\mathcal{S}^-) + 1.322\log P \cdot p(J)$. Moreover, max-speed$(\mathcal{N})$ is at most* $\max\{$*max-speed$(\mathcal{N}^+)$, max-speed$(\mathcal{S}^-)\}$.

With the above lemmas and theorem, we can prove the main result on eliminating migration, i.e., Theorem 4.6 in Section 4.2.

Proof of Theorem 4.6. Given a migratory schedule $\mathcal{S}$ of J, we can apply Lemma 4.9, Theorem 4.10, and Lemma 4.11 to obtain a CRR(1) schedule $\mathcal{N}$ for J such that

$$\begin{aligned}
G(\mathcal{N}) &\leq G(\mathcal{N}^+) + G(\mathcal{S}^-) + 1.322\log P \cdot p(J) \\
&\leq G(\mathcal{N}^*) + 1.322\log P \cdot p(J^+) + G(\mathcal{S}^-) \\
&\quad + 1.322\log P \cdot p(J) \\
&\leq G(\mathcal{S}^*) + 2\log P \cdot p(J^*) \\
&\quad + 1.322\log P \cdot p(J^+) + G(\mathcal{S}^-) \\
&\quad + 1.322\log P \cdot p(J) \\
&\leq G(\mathcal{S}) + 2\log P \cdot p(J^*) \\
&\quad + (2.644\lceil\log P\rceil + 1)p(J^+) \\
&\quad + 1.322\log P \cdot p(J).
\end{aligned}$$

Note that $p(J^+) = p(J^*) \leq p(J)$. Furthermore, Lemma 4.1 implies $p(J) \leq G(\mathcal{S})$. Thus, $G(\mathcal{N}) \leq (5.966\log P + 2)G(\mathcal{S})$ and *max-speed*$(\mathcal{N}) \leq$ *max-speed*$(\mathcal{S})$. Thus, Theorem 4.6 holds. □

4.3.1 Eliminating migration in a multi-processor schedule of parallel jobs

In this section we prove Theorem 4.10 (of Section 4.3) that transforms a migratory schedule $\mathcal{S}^*$ of an m-parallel job set J^* to a trivially non-migratory schedule $\mathcal{N}^*$ for J^* with a moderate increase in flow time plus energy. Let K denote the maximum number of distinct job sizes in J^*, i.e., $K = \log P$.

The transformation consists of two steps. Each step preserves the total *fractional* flow time plus energy. The first step makes use of an "averaging" technique to determine the speed and to distribute the workload among the processors, this results in a non-migratory and indeed trivially non-migratory schedule $\mathcal{N}_1^*$. The second step attempts to locally "tidy up" the schedule of each individual processor in $\mathcal{N}_1^*$ so that the total flow time of the resulting schedule $\mathcal{N}^*$ does not exceed its total fractional flow time too much (precisely, by at most $2Kp(J^*)$). Then Theorem 4.10 follows. Details are as follows.

Step 1: Speed Averaging. The speed function of the new schedule $\mathcal{N}_1^*$ is determined as follows. At any time, every processor in $\mathcal{N}_1^*$ runs at the average speed of all processors of $\mathcal{S}^*$. That is, if the processors in $\mathcal{S}^*$ are at speed $s_1, s_2, \ldots, s_m$, respectively, then every processor in $\mathcal{N}_1^*$ runs at speed $\sum_{i=1}^m s_i/m$. Note that the "total" speed of $\mathcal{S}^*$ and $\mathcal{N}_1^*$ are the same. Since the energy function s^α is convex, the rate of energy consumption of $\mathcal{N}_1^*$ (i.e., $m\,(\sum_{i=1}^m s_i/m)^\alpha$) is at most that of $\mathcal{S}^*$ (i.e., $\sum_{i=1}^m s_i^\alpha$).

Work Averaging. Next we describe how $\mathcal{N}_1^*$ selects jobs for execution. In $\mathcal{S}^*$, jobs in a batch may have different progress. To ease our discussion, we divide $\mathcal{S}^*$ into consecutive time intervals at the moment when a batch of jobs is released or has just been completed. Let $\mathcal{I}$ be such a time interval. Within $\mathcal{I}$, suppose $\mathcal{S}^*$ has worked on some jobs of a batch B for a total of u units (note that the work done on each job of B may vary), then $\mathcal{N}_1^*$ would schedule each processor to work on u/m units of a different job of B in parallel. Note that the total work done on B is still u (though the progress of individual jobs might differ from $\mathcal{S}^*$). $\mathcal{S}^*$ might have worked on several batches within $\mathcal{I}$; at any time within $\mathcal{I}$, $\mathcal{N}_1^*$ uses the Smallest Job Size First (SJF) strategy to select the next batch for execution.

Analysis. At any time in an interval $\mathcal{I}$, the SJF strategy ensures that $\mathcal{N}_1^*$ gives priority to jobs that would give the biggest decrease of fractional weight; thus, $\mathcal{N}_1^*$ has a total fractional weight no more than that of $\mathcal{S}^*$. At the end of $\mathcal{I}$, $\mathcal{N}_1^*$ and $\mathcal{S}^*$ have performed

the same amount of work for each batch, and they have the same total fractional weight. Applying the same argument to every time interval of $\mathcal{S}^*$, we conclude that at any time, the total fractional weight in $\mathcal{N}_1^*$ is no more than that of $\mathcal{S}^*$. Thus, the total fractional flow time of $\mathcal{N}_1^*$ is also no more than that of $\mathcal{S}^*$.

Step 2 (Tidying). We further transform $\mathcal{N}_1^*$ to $\mathcal{N}^*$ to reduce the total flow time. Recall that $\mathcal{N}_1^*$ has an identical schedule for all processors. The changes made in Step 2 are local to each processor, and all processors undergo the same changes.

(a) **Critical speed.** We ensure that at any time, $\mathcal{N}_1^*$ executes a job j at speed at least its critical speed (recall that this critical speed property can be enforced by invoking the transformation stated in Lemma 4.5 to each processor).

(b) **Minimizing partially processed jobs.** Next we want to ensure that in each processor, there is at most one partially processed job of each size. To obtain such a schedule, we consider all jobs of a particular size each time, and shuffle these jobs using "earliest release time first" strategy. The speed used at any time is not changed, so after shuffling, a job may be executed faster or slower.

(c) **Eliminating unnecessary idle time.** Finally, we "compact" the schedule of each processor so that it is never idle when there are unfinished jobs. To do so, we consider all the jobs executed in a processor in the order of release time. For each job j, we move its schedule as close as possible to its release time, filling out all the idle time. Note that we do not change the speed and the total time to execute j.

Analysis. Denote the schedule produced by Step 2 as $\mathcal{N}^*$. By Lemma 4.5, Step 2(a) does not increase the total fractional flow time plus energy. Steps 2(b) and (c) do not change the energy usage. Furthermore, in Step 2(b), shuffling jobs of the same size among themselves does not alter the total fractional weight of these jobs. Thus the total fractional flow time is preserved. Step 2(c) could only decrease the total fractional flow time. Thus $\mathcal{N}^*$ does not increase the total fractional flow time plus energy.

Next, we consider the total flow time. We first upper bound the flow time of $\mathcal{N}^*$ in terms of its fractional flow time and total execution time (Lemma 4.12), and the latter can be further shown to be at most twice of the total size of all jobs (Lemma 4.13).

Lemma 4.12. *Consider the schedule $\mathcal{N}^*$, $F(\mathcal{N}^*) \leq \hat{F}(\mathcal{N}^*) + K \cdot X(\mathcal{N}^*)$, where K is the number of different job sizes in J^*. (Recall that $X(\mathcal{N})$ is the total execution time of $\mathcal{N}$.)*

Proof. As $\mathcal{N}^*$ is non-migratory, we can analyze $F(\mathcal{N}^*)$ and $\hat{F}(\mathcal{N}^*)$ by summing up the total flow time and fractional flow time of individual processors. Consider any processor i, let $F_i(\mathcal{N}^*)$ and $\hat{F}_i(\mathcal{N}^*)$ be the corresponding total flow time and total fractional flow time, respectively. At any time, the number of unfinished jobs in a processor can exceed the total fractional weight of unfinished jobs by at most the number of partially processed jobs, which is at most K. Furthermore, whenever a processor is idle, there is no unfinished jobs to charge to $F_i(\mathcal{N}^*)$ and $\hat{F}_i(\mathcal{N}^*)$. Thus, $F_i(\mathcal{N}^*) - \hat{F}_i(\mathcal{N}^*) \leq K \sum_{j \in J_i^*} X(j)$, where J_i^* is the subset of jobs executed in processor i. Summing over all processors, we have $F(\mathcal{N}^*) - \hat{F}(\mathcal{N}^*) \leq K \sum_{j \in J^*} X(j)$. □

Lemma 4.13. *Consider the schedule $\mathcal{N}^*$. For any job j, $X(j) \leq (\alpha/(\alpha-1)^{1-1/\alpha})p(j) \leq 2p(j)$; and $X(\mathcal{N}^*) \leq 2p(J^*)$.*

Proof. Consider a job j running at its critical speed starting at time 0 until it completes at some time t_c. At a time t, let q be the remaining work of j. Then the speed at t is $-\frac{dq}{dt} = (\frac{q}{(\alpha-1)p(j)})^{1/\alpha}$. This implies

$$\int_0^{t_c} dt = \int_{p(j)}^{0} -((\alpha-1)\,p(j))^{1/\alpha} q^{-1/\alpha}\, dq \ ,$$

or equivalently, $t_c = \frac{\alpha\, p(j)}{(\alpha-1)^{1-1/\alpha}}$. Note that Step 2(a) produces a schedule that runs a job at speed no less than its critical speed. Then for any job j,

$$X(j) \leq (\alpha/(\alpha-1)^{1-1/\alpha})p(j) \leq 2p(j),$$

where the last inequality is due to the fact that $\alpha/(\alpha-1)^{1-1/\alpha}$ is maximized when $\alpha = 2$.

Therefore, summing over all jobs j, the total execution time of the schedule produced by Step 2(a) is $\sum_{j \in J^*} X(j) \leq \sum_{j \in J^*} 2p(j) = 2p(J^*)$. Steps 2(b) and 2(c) do not change the total execution time, so we conclude that $X(\mathcal{N}^*) \leq 2p(J^*)$. □

Lemmas 4.12 and 4.13 imply that $F(\mathcal{N}^*) \leq \hat{F}(\mathcal{N}^*) + 2Kp(J^*)$. Hence $G(\mathcal{N}^*) \leq \hat{G}(\mathcal{N}^*) + 2Kp(J^*)$. Since $\mathcal{N}^*$ preserves the total fractional flow time plus energy of $\mathcal{S}^*$ and the fractional flow time is always upper bounded by flow time, we have $\hat{G}(\mathcal{N}^*) \leq \hat{G}(\mathcal{S}^*) \leq G(\mathcal{S}^*)$, and $G(\mathcal{N}^*) \leq G(\mathcal{S}^*) + 2Kp(J^*)$. Theorem 4.10 is proved.

4.3.2 Forward transformation of schedules: from J to J^*

In this section we show how to make use of the previous result on parallel jobs to derive a way to eliminate migration in a schedule for an arbitrary job set J of power-of-2 size. Recall that we have defined the job sets J^+, J^- and J^* from a given job set J. We will present two transformations: a forward transformation of a migratory schedule for J to a migratory schedule for J^* (this section) and a backward transformation of a trivially non-migratory schedule for J^* to a CRR(1) schedule for J (Section 4.3.3).

Now we show how to transform a migratory schedule $\mathcal{S}$ for J to a migratory schedule $\mathcal{S}^*$ for J^*. Recall that a job j in J^* has an earlier deadline ($r^*(j)$), and the desired lemma is as follows.

Lemma 4.9 *Given a migratory schedule $\mathcal{S}$ for J, we can construct two migratory schedules $\mathcal{S}^*$ for J^* and $\mathcal{S}^-$ for J^- in such a way that $G(\mathcal{S}^*) + G(\mathcal{S}^-) \leq G(\mathcal{S}) + 1.322(\log P + 1) \cdot p(J^+)$. Both $\mathcal{S}^*$ and $\mathcal{S}^-$ use at most the maximum speed of $\mathcal{S}$.*

To construct $\mathcal{S}^*$ from $\mathcal{S}$, we advance the schedule of each job j in view of its new release time $r^*(j)$. Below we assume that $\mathcal{S}$ always schedules every processor to run at speed at least the global critical speed (see Lemma 4.3). Recall that $J = J^+ \cup J^-$. Restricting $\mathcal{S}$ with respect to J^+ and J^- gives two schedules $\mathcal{S}^+$ and $\mathcal{S}^-$, respectively. Note that $E(\mathcal{S}^+) + E(\mathcal{S}^-) = E(\mathcal{S})$, and $F(\mathcal{S}^+) + F(\mathcal{S}^-) = F(\mathcal{S})$. $\mathcal{S}^+$ is also a valid schedule for J^*, though it might induce a larger total flow time (because jobs in J^* have earlier release time). To limit the increase in flow time, we modify $\mathcal{S}^+$ into a better schedule $\mathcal{S}^*$ for J^* as follows. Note that we will not change the energy used, i.e., $E(\mathcal{S}^*) = E(\mathcal{S}^+)$.

Advance the schedule. Tag the jobs in each group of J^+ with unique integers from 0 to $m-1$, 0 being the leader. The schedule of all leaders are not modified. We modify $\mathcal{S}^+$ in rounds, one for a particular job size (say, from the smallest to the largest). In each round, we consider groups of jobs in the order of the release time. Consider a job j of a particular group, which is tagged with $i > 0$. Let j' be its leader. To reduce the flow time of j, we use the following trick to advance the schedule of j using a single processor, in particular, processor labeled i. Let t_1 be the total amount of time over all processors that $\mathcal{S}^+$ executes j, and suppose that between the (original) release times of j' and j, i.e., $[r^*(j), r(j)]$ (recall that $r^*(j) = r(j')$), the current schedule of processor i has a total

of $t_2 \geq 0$ units of idling time. If $t_1 < t_2$, we move the schedule of j to processor i only, occupying the earliest idle time starting from $r^*(j)$. The speed function for processing j remains unchanged. If $t_1 \geq t_2$, the schedule of j is left unmodified.

Analysis. The above modification guarantees that in the new schedule, the job j, whose release time has been set to $r^*(j)$, has only a moderate increase in waiting time, precisely, at most the sum of execution time of processor i during $[r^*(j), r(j)]$ and $X(j)$. This is proved in Lemma 4.14, which also shows that from J^+ to J^*, the total increase of waiting time (as well as flow time) over all jobs is at most $(\log P + 1)X(\mathcal{S}^+)$. Since every processor run at speed at least the global critical speed (Lemma 4.3), $X(\mathcal{S}^+) \leq (\alpha - 1)^{1/\alpha} p(J^+) \leq 1.322 p(J^+)$, and Lemma 4.9 follows.

Lemma 4.14. *The increase in flow time caused by the transformation in Lemma 4.9, when transforming a migratory schedule $\mathcal{S}^+$ for J^+ to a schedule $\mathcal{S}^*$ for J^*, is at most $(\log P + 1) \cdot X(\mathcal{S}^+)$.*

Proof. Since the same speed is being used for all jobs, $X(\mathcal{S}^*) = X(\mathcal{S}^+)$. It remains to analyze the waiting time. This is done by bounding the change in waiting time of each modified job during each round. Let $\mathcal{S}_a$ and $\mathcal{S}_b$ be the schedules before and after a round in the transformation. Obviously there is no change in the waiting time for leaders. Consider a job j tagged with $i > 0$, with leader j'. We claim that the increase in waiting time is at most the sum of the execution time of j in $\mathcal{S}_a$ and the busy time of processor i in schedule $\mathcal{S}_a$ during $[r^*(j), r(j)]$. This is clear in the case where the scheduling of j is modified: indeed the waiting time of j in $\mathcal{S}_b$ is no more than the latter term. In case where the scheduling of j is unmodified, the increase in waiting time is $r(j) - r^*(j)$, which equals to the amount of idle time plus busy time of processor i in schedule $\mathcal{S}_a$ during $[r^*(j), r(j)]$. Since the scheduling of j is not modified, the amount of idle time is at most the execution time of j in $\mathcal{S}_a$. The claim thus holds.

Summing over all the jobs concerned for the modification from $\mathcal{S}_a$ to $\mathcal{S}_b$ for the round of size 2^k jobs, the increase in waiting time is at most the sum of $X(\mathcal{S}_a) = X(\mathcal{S}^+)$ and $\sum_{j \text{ is of size } 2^k} X(j)$. Summing over all the $\log P$ rounds, the waiting time is increased by at most $\log P \cdot X(\mathcal{S}^+) + \sum_k \sum_{j \text{ is of size } 2^k} X(j) = (\log P + 1) \cdot X(\mathcal{S}^+)$. □

4.3.3 Backward transformation of schedules: from J^* to J^+ and then to J

Recall that J^* is m-parallel. Suppose that we have used Theorem 4.10 to obtain a trivially non-migratory schedule $\mathcal{N}^*$ for J^*. Below we show how to transform $\mathcal{N}^*$ to a CRR(1) schedule $\mathcal{N}^+$ for J^+ (Lemma 4.11 (i)). Then it is relatively easy to obtain a CRR(1) schedule for J (Lemma 4.11 (ii)). Lemma 4.11 of Section 4.3 is reiterated as follows.

Lemma 4.11 (i) *Given a trivially non-migratory schedule $\mathcal{N}^*$ of J^*, we can construct a* CRR(1) *schedule $\mathcal{N}^+$ for J^+ such that $G(\mathcal{N}^+) \leq G(\mathcal{N}^*) + 1.322 \log P \cdot p(J^+)$. Moreover, max-speed($\mathcal{N}^+$) $\leq$ max-speed($\mathcal{N}^*$).* (ii) *Together with a migratory schedule $\mathcal{S}^-$ for J^-, we can construct a* CRR(1) *schedule $\mathcal{N}$ for J, such that $G(\mathcal{N}) \leq G(\mathcal{N}^+) + G(\mathcal{S}^-) + 1.322 \log P \cdot p(J)$. Moreover, max-speed($\mathcal{N}$) is at most* $\max\{$*max-speed*($\mathcal{N}^+$), *max-speed*($\mathcal{S}^-$)$\}$.

From J^* to J^+. Note that $\mathcal{N}^*$ may not be a valid schedule for J^+, since a job j in J^* has a release time $r^*(j)$ earlier than the release time $r(j)$ in J^+. We transform $\mathcal{N}^*$ to move the execution period of jobs so that the schedule becomes valid. The changes made are local to each processor, and all processors undergo the same changes. The following discussion focuses on the schedule of a processor x in $\mathcal{N}^*$. Without loss of generality, we assume that jobs of the same size are processed in the order of release time in J^+. Furthermore, by Lemma 4.3, the processor runs at speed at least the global critical speed.

Transformation. We focus on the schedule of a particular processor x in $\mathcal{N}^*$. The transformation runs in multiple rounds. Initially, R is the schedule of the processor x in $\mathcal{N}^*$. In each round, R is modified with respect to the jobs of a particular size. Recall that $r(j)$ and $r^*(j)$ denote the release time of a job j in J^+ and J^*, respectively. Let $j_1, j_2, \ldots, j_n$ be the jobs of size p running in processor x, where $r^*(j_1) \leq r^*(j_2) \leq \cdots \leq r^*(j_n)$. We observe that $r^*(j_i) \leq r(j_i) \leq r^*(j_{i+1})$; the latter inequality is due to the fact that $r^*(j_{i+1})$ is actually the release time of the leader of j_{i+1} in J^+, which is at least $r(j_i)$. In other words, the period where R schedules j_{i+1} is a feasible period to schedule j_i in J^+. The transformation first removes the scheduling of $j_1, \ldots, j_n$ from R. Then, for each j_i from $i = 1$ to $n-1$, we schedule j_i to the first idle intervals of the schedule after $r(j_i)$, using the same speeds of j_{i+1} in R. Finally, we schedule j_n to the first idle intervals of

the schedule after $r(j_n)$ using the same speeds of j_1 in R. The transformation is repeated for each job size. The final schedule obtained for all processors is $\mathcal{N}^+$.

As the transformation is local to each processor, $\mathcal{N}^+$ follows the way $\mathcal{N}^*$ dispatches jobs. Consider jobs in J^+ in the order of release time. Because $\mathcal{N}^*$ is trivially non-migratory, $\mathcal{N}^+$ dispatches every m jobs of the same size (which are power-of-2) to different processors. Thus $\mathcal{N}^+$ is a CRR(1) schedule.

Analysis. In the transformation above, the maximum speed and energy of $\mathcal{N}^+$ remains the same as that of $\mathcal{N}^*$. Yet the analysis of the flow time is quite tricky. The increase in flow time is the increase in execution time plus waiting time of all jobs. Consider a particular processor and a particular job size p for which a schedule R is transformed to R_1. *Execution time:* The speed used by R_1 for the jobs is the same as that used by R, for a permutation of the jobs. Thus there is no change in total execution time for all jobs involved. *Waiting time:* According to the way we transform the execution period and speed of j_i, one can argue (by induction) that the start time (resp. completion time) of j_i in R_1 is no later than the start time (resp. completion time) of j_{i+1} in R (see Lemma 4.15 below). This property enables us to show that from $\mathcal{N}^*$ to $\mathcal{N}^+$, the increase of total flow time is at most $\log P \cdot X(\mathcal{N}^+)$ (see Lemma 4.15), which is at most $1.322 \log P \cdot p(J^+)$ (by Lemma 4.3). Then Lemma 4.11 (i) follows.

Lemma 4.15. *The transformation in Lemma 4.11 transforms a trivially non-migratory schedule $\mathcal{N}^*$ for J^* to a non-migratory* CRR(1) *schedule $\mathcal{N}^+$ for J^+ such that $F(\mathcal{N}^+) \leq F(\mathcal{N}^*) + \log P \cdot X(\mathcal{N}^+)$.*

Proof. As mentioned before, it suffices to bound the waiting time among all jobs. Due to the tidying step in the course of constructing the trivially non-migratory schedule $\mathcal{N}^*$ of J^*, we can assume that the jobs of the same size are processed in the order of release time in J^+. Consider a round of the transformation that converts from schedule R to R_1 for a particular processor and a particular job size p. Note that the waiting time of a job which is not of size p does not change, so we focus on jobs of size p. Recall that the transformation is done by first removing all jobs $j_1, j_2, \ldots, j_n$ of size p from R. We use R_0 to denote this partial schedule. Let $c^*(j)$ and $c(j)$ denote the completion time of job j in R and R_1, respectively. We first observe the following property about the start time and completion time of jobs j_i in R and R_1.

Proposition: For $i = 1, \ldots, n-1$, j_i starts running (respectively, is com-

pleted) in R_1 at or before j_{i+1} starts running (respectively, is completed) in R.

The proposition can be proved by induction on i. Job j_i starts running in R_1 at either (1) the first idle time in R_0 at or after $c(j_{i-1})$, or (2) the first idle time in R_0 at or after $r(j_i)$, whichever larger. By induction, the time (1) is no later than the first idle time in R_0 at or after $c^*(j_i)$. By our assumption that jobs of the same size are scheduled in the order of release time, this is the first time that R can process job j_{i+1}. The time (2) is no later than the first idle time in R_0 at or after $r^*(j_{i+1})$ (since $r(j_i) \leq r^*(j_{i+1})$), which is obviously no later than the start time of j_{i+1}. So the start time of j_i in R_1 is no later than the start time of j_{i+1} in R. Now $c(j_i) \leq c^*(j_{i+1})$ follows immediately, since the speed function that R_1 uses for j_i is copied from the speed function that R uses for j_{i+1}.

To bound the increase in waiting time among $j_1, j_2, \ldots, j_n$, we characterize such increase, i.e., times which contribute to the waiting time of j_i in R_1 but not in R. They can be times when one of the followings happen.

- R_0 works on a job of size other than p when j_i is already completed in R but waiting in R_1. In this case the time must be during $[c^*(j_i), c(j_i)]$.
- R_0 is idle when j_{i-1} has already completed in R but is running in R_1. In this case the time must be during $[c^*(j_{i-1}), c(j_{i-1})]$.

In both cases, these times contributed to the waiting time are times when R_1 is running; furthermore, by the Proposition, such times do not overlap for different j_i. The increase in waiting time over all jobs is thus at most $X(R_1)$. Summing over all processors and all $\log P$ rounds, the lemma follows. □

From J^+ and J^- to J. Let us turn to the modification of the schedule to accommodate the jobs in J^- and to handle the reduction in job size when scheduling J.

To construct $\mathcal{N}$ in Lemma 4.11 (ii), we simply insert each job $j \in J^-$ into the given schedule $\mathcal{N}^+$. By definition, j has a release time later than jobs in J^+ of the same size, and would be dispatched by CRR(1) after all these jobs. Suppose CRR(1) dispatches j to processor i. Then we schedule j to processor i during the first idle periods after j's release time, using the speed function of j in $\mathcal{S}^-$. The total waiting time of j is at most

the total execution time of processor i. After inserting all jobs in J^-, we obtain a CRR(1) schedule $\mathcal{N}$ for J, with energy usage and maximum speed preserved. Since at most $\log P$ jobs in J^- are dispatched to each processor, the total increase in flow time is at most $\log P$ times the total execution time of all processors in $\mathcal{N}$, which is at most $1.322 \log P \cdot p(J)$, by Lemma 4.3.

4.4 Jobs of Arbitrary Size

This section is devoted to proving Theorem 4.7 in Section 4.2. In essence, for any $\epsilon > 0$, we want to construct a CRR(ϵ) schedule from an optimal (non-migratory or migratory) schedule with a mild increase in flow time plus energy and in maximum speed. Section 4.4.1 restricts the possible optimal schedules so as to ease the construction. Section 4.4.2 presents an algorithm to construct a CRR schedule from an optimal schedule, and analyzes the performance of the CRR schedule constructed. For optimal migratory schedule, the analysis relies on a property of optimal migratory schedule, which is shown in Section 4.4.3.

4.4.1 Restricted but useful optimal schedules

In this section, we introduce two notions to restrict the possible optimal (non-migratory or migratory) schedules so as to ease the construction.

- A job set J is said to be *power-of-*$(1+\epsilon)$ if every job in J has size $(1+\epsilon)^k$ for some k.
- For any job set J and schedule $\mathcal{S}$, we say that $\mathcal{S}$ is *immediate-start* if every job starts at exactly its release time in J.

In general, an optimal schedule may not be immediate-start. The rest of this section shows that it suffices to focus on job sets that are power-of-$(1+\epsilon)$ and admit optimal schedules that are also immediate-start (such schedules will be referred to as *immediate-start, optimal schedules*). See Corollary 4.18 below for a technical summary. The job size restriction is relatively easy to observe as we can exploit a slightly higher maximum speed (Lemma 4.16). The immediate-start property is non-trivial and perhaps counter-intuitive (Lemma 4.17).

Technically speaking, the results below (Lemmas 4.16, 4.17 and Corollary 4.18) hold in both the migratory and the non-migratory setting. To simplify the presentation of this section, we will not mention whether schedules are migratory or non-migratory. One should read the lemmas and proofs by assuming all schedules are either migratory or non-migratory.

Lemma 4.16. *Given a job set J, we can construct a power-of-$(1+\epsilon)$ job set J' such that*

i. any schedule $\mathcal{S}_1$ for J defines a schedule $\mathcal{S}'_1$ for J' such that $G(\mathcal{S}'_1) \leq (1+\epsilon)^\alpha G(\mathcal{S}_1)$ and max-speed$(\mathcal{S}'_1) \leq (1+\epsilon) \times$ max-speed$(\mathcal{S}_1)$; and

ii. any schedule $\mathcal{S}'_2$ for J' defines a schedule $\mathcal{S}_2$ for J with $G(\mathcal{S}_2) \leq G(\mathcal{S}'_2)$ and max-speed$(\mathcal{S}_2) =$ max-speed$(\mathcal{S}'_2)$.

Proof. J' can be constructed from J by rounding up the size of each job in J to the nearest power of $(1+\epsilon)$. (i) $\mathcal{S}_1$ naturally defines a schedule $\mathcal{S}'_1$ for J' as follows. Whenever $\mathcal{S}_1$ runs a job j at speed s, $\mathcal{S}'_1$ runs the corresponding job in $\mathcal{S}'$ at speed $s' = s \times (1+\epsilon)^{\lceil \log_{1+\epsilon} p(j) \rceil}/p(j)$. Note that $s' \leq (1+\epsilon)s$, $E(\mathcal{S}'_1) \leq (1+\epsilon)^\alpha E(\mathcal{S}_1)$, and $F(\mathcal{S}'_1) = F(\mathcal{S}_1)$. (ii) is obvious as we can apply any schedule $\mathcal{S}'_2$ for J' to schedule J with extra idle time. □

The next lemma explains why we can focus on optimal schedules that are immediate-start. Unless otherwise stated, an optimal schedule $\mathcal{O}$ below means a schedule that has smallest flow time plus energy among all schedules with maximum speed not exceeding max-speed$(\mathcal{O})$. To ease the discussion, we add a subscript J to the notations F, E, and G to denote that the job set under concern is J.

Lemma 4.17. *Given a power-of-$(1+\epsilon)$ job set J_1 and an optimal schedule $\mathcal{O}_1$ for J_1, we can construct a power-of-$(1+\epsilon)$ job set J_2 and an immediate-start, optimal schedule $\mathcal{O}_2$ for J_2 with max-speed$(\mathcal{O}_2) \leq$ max-speed$(\mathcal{O}_1)$. Furthermore, any CRR(ϵ) schedule $\mathcal{S}_2$ for J_2 defines a CRR(ϵ) schedule $\mathcal{S}_1$ for J_1, and if $G_{J_2}(\mathcal{S}_2) \leq \gamma G_{J_2}(\mathcal{O}_2)$ for some $\gamma \geq 1$, then $G_{J_1}(\mathcal{S}_1) \leq \gamma G_{J_1}(\mathcal{O}_1)$.*

Proof. We first construct $\mathcal{O}_2$ from J_1 and $\mathcal{O}_1$. The idea is to repeatedly pick two jobs of the same size and swap their schedules in $\mathcal{O}_1$. More specifically, each time we consider all jobs in J_1 of a particular size, and swap their schedules so that their release times and start times in $\mathcal{O}_2$ are in the same order (note that the speed at any time stays unchanged).

That is, for all i, the job with the i-th smallest release time will take up the schedule of the job with the i-th smallest start time; note that the i-th smallest start time can never be earlier than the i-th smallest release time. Thus, $\mathcal{O}_2$ is also a valid schedule for J_1.

Next, we modify J_1 to J_2, by replacing the release time of each job j with its start time in $\mathcal{O}_2$. Note that the release time of j can only be delayed (and never gets advanced). Any schedule for J_2 (including $\mathcal{O}_2$) is also a valid schedule for J_1.

By construction, $\mathcal{O}_2$ is an immediate-start schedule for J_2. Next, we analyze the relationship between $\mathcal{O}_1$ and $\mathcal{O}_2$.

$\mathcal{O}_1$ and $\mathcal{O}_2$ incur the same flow time plus energy for J_1. Since $\mathcal{O}_1$ and $\mathcal{O}_2$ use the same speed at any time, $E_{J_1}(\mathcal{O}_1) = E_{J_1}(\mathcal{O}_2)$. Furthermore, at any time, $\mathcal{O}_1$ completes a job if and only if $\mathcal{O}_2$ completes a (possibly different) job, and thus $\mathcal{O}_1$ and $\mathcal{O}_2$ always have the same number of unfinished jobs. This means that $F_{J_1}(\mathcal{O}_1) = F_{J_1}(\mathcal{O}_2)$ and $G_{J_1}(\mathcal{O}_1) = G_{J_1}(\mathcal{O}_2)$.

$\mathcal{O}_2$ is optimal for J_2 (in terms of flow time plus energy). Suppose on the contrary that there is a schedule $\mathcal{O}'$ for J_2 with $G_{J_2}(\mathcal{O}') < G_{J_2}(\mathcal{O}_2)$. Any schedule for J_2, including $\mathcal{O}'$ and $\mathcal{O}_2$, is also a valid schedule for J_1. Note that $E_{J_1}(\mathcal{O}') = E_{J_2}(\mathcal{O}')$, and $F_{J_1}(\mathcal{O}') = F_{J_2}(\mathcal{O}') + d$, where d is the total delay of release times of all jobs in J_2 (when comparing with J_1). Therefore, $G_{J_1}(\mathcal{O}') = G_{J_2}(\mathcal{O}') + d$, and similarly for $\mathcal{O}_2$. Thus, if $G_{J_2}(\mathcal{O}') < G_{J_2}(\mathcal{O}_2)$, then

$$\begin{aligned} G_{J_1}(\mathcal{O}') &= G_{J_2}(\mathcal{O}') + d \\ &< G_{J_2}(\mathcal{O}_2) + d = G_{J_1}(\mathcal{O}_2) = G_{J_1}(\mathcal{O}_1) \ . \end{aligned}$$

This contradicts the optimality of $\mathcal{O}_1$ for J_1.

CRR preserves performance. Consider any CRR(ϵ) schedule $\mathcal{S}_2$ for J_2 satisfying $G_{J_2}(\mathcal{S}_2) \leq \gamma G_{J_2}(\mathcal{O}_2)$, for some $\gamma \geq 1$. By definition, jobs of the same class are also of same size and have the same order of release times in J_1 and J_2. Therefore, $\mathcal{S}_2$ is also an CRR(ϵ) schedule for J_1. For total flow time plus energy,

$$\begin{aligned} G_{J_1}(\mathcal{S}_2) &= G_{J_2}(\mathcal{S}_2) + d \leq \gamma G_{J_2}(\mathcal{O}_2) + d \\ &\leq \gamma(G_{J_2}(\mathcal{O}_2) + d) = \gamma G_{J_1}(\mathcal{O}_2) = \gamma G_{J_1}(\mathcal{O}_1) \ . \end{aligned}$$

Thus the lemma follows. □

In summary, the two lemmas above allow us to focus on power-of-$(1+\epsilon)$ job sets that admit immediate-start, optimal schedules.

Corollary 4.18. *Let J be a job set and let $\mathcal{O}$ be an optimal schedule for J. For any $\epsilon > 0$, there exists a power-of-$(1+\epsilon)$ job set J' and a schedule $\mathcal{O}'$ for J' that is immediate-start and optimal among all schedules with maximum speed $(1+\epsilon) \times$ max-speed$(\mathcal{O})$. Furthermore, any CRR(ϵ) schedule $\mathcal{S}'$ for J' defines a CRR(ϵ) schedule $\mathcal{S}$ for J, and if $G_{J'}(\mathcal{S}') \leq \gamma G_{J'}(\mathcal{O}')$ for some $\gamma \geq 1$, then $G_J(\mathcal{S}) \leq \gamma(1+\epsilon)^\alpha G_J(\mathcal{O})$.*

Proof. By Lemma 4.16 (i), we construct from J and $\mathcal{O}$ a power-of-$(1+\epsilon)$ job set J_1 and a schedule $\mathcal{S}_1$ for J_1 with $G_{J_1}(\mathcal{S}_1) \leq (1+\epsilon)^\alpha G_J(\mathcal{O})$ and *max-speed*$(\mathcal{S}_1) \leq (1+\epsilon) \times$ *max-speed*$(\mathcal{O})$. Let $\mathcal{O}_1$ be the optimal schedule for $\mathcal{S}_1$ with maximum speed $(1+\epsilon) \times$ *max-speed*$(\mathcal{O})$. Then $G_{J_1}(\mathcal{O}_1) \leq G_{J_1}(\mathcal{S}_1) \leq (1+\epsilon)^\alpha G_J(\mathcal{O})$. Next we apply Lemma 4.17 to J_1 and $\mathcal{O}_1$, and we obtain J' and an immediate start, optimal schedule $\mathcal{O}'$ with maximum speed at most *max-speed*$(\mathcal{O}_1)$, which is at most $(1+\epsilon)$*max-speed*$(\mathcal{O})$. Furthermore, by Lemma 4.17 and Lemma 4.16 (ii), $\mathcal{S}'$ defines a CRR(ϵ) schedule $\mathcal{S}$ for J such that if $G_{J'}(\mathcal{S}') \leq \gamma G_{J'}(\mathcal{O}')$ for some $\gamma \geq 1$, then $G_J(\mathcal{S}) \leq \gamma G_{J_1}(\mathcal{O}_1) \leq \gamma(1+\epsilon)^\alpha G_J(\mathcal{O})$. □

In the rest of this section, we further exploit the fact that an optimal schedule runs a job at the same speed throughout its lifespan. This is due to the convexity of the power function s^α. Recall that, without loss of generality, at any time an optimal schedule never runs a job at speed less than the global critical speed, defined as $1/(\alpha-1)^{1/\alpha}$, and the maximum speed T is at least the global critical speed (see Section 4.1).

4.4.2 Constructing CRR schedules

This section presents an algorithm, called MakeCRR, to construct a CRR schedule from an optimal non-migratory or migratory schedule $\mathcal{S}^*$ for a job set J. By Corollary 4.18, we focus on the case where J consists of power-of-$(1+\epsilon)$ jobs only and $\mathcal{S}^*$ is also immediate-start. Note that in a CRR(ϵ) schedule for J, jobs in each class are of identical size, and the round robin policy is effectively applied independently to every subset of jobs of the same size. The non-trivial part is how to ensure a moderate increase of flow time and energy.

Before we detail the algorithm, it is useful to observe the nature of the CRR schedule $\mathcal{S}$ to be constructed. The ordering of job execution in $\mathcal{S}$ could be very different from $\mathcal{S}^*$. Roughly speaking, $\mathcal{S}$ only makes reference to the speed used by $\mathcal{S}^*$. Recall that in $\mathcal{S}^*$, a job is run at the same speed throughout its lifespan. For any job, $\mathcal{S}$ determines its speed as the average of a certain subset of $(b+1)m$ jobs in $\mathcal{S}^*$, where $b = 1$ or 4 depending on whether $\mathcal{S}^*$ is non-migratory or migratory. The constant b arises from an upper bound of the number of jobs of the same size that have started but not yet finished at any time. We assume that the processors are numbered from 0 to $m-1$.

Algorithm MakeCRR**.** The algorithm has a parameter $\lambda > 0$ to control the extra speed (we will eventually set $\lambda = \epsilon$ to derive the desired result).

The construction runs in multiple rounds, from the smallest job size to the largest. Let $\mathcal{S}_0$ denote the intermediate schedule, which is initially empty and eventually becomes $\mathcal{S}$. We modify $\mathcal{S}_0$ in each round to include more jobs. In the round for size p, suppose that J contains n jobs $\{j_1, j_2, \ldots, j_n\}$ of size p, arranged in increasing order of release times. It is convenient to define $j_{n+1} = j_1$, $j_{n+2} = j_2$, etc. For $i = 1$ to n, let x_i be the average speed in $\mathcal{S}^*$ of the fastest m jobs among the following $(b+1)m$ jobs: $j_i, j_{i+1}, \ldots, j_{i+(b+1)m-1}$. We modify $\mathcal{S}_0$ by adding a schedule for j_i in processor $(i \bmod m)$: it can start as early as at its release time, runs at constant speed $(1+\lambda)x_i$, and occupies the earliest possible times, while avoiding times already committed to earlier jobs for processor $(i \bmod m)$.

Performance of the constructed schedule $\mathcal{S}$. To study the performance of $\mathcal{S}$ constructed by Algorithm MakeCRR, we need to be very specific about the properties of the job set J and the optimal schedule $\mathcal{S}^*$. By Corollary 4.18, we assume that J consists of power-of-$(1+\epsilon)$ jobs only and $\mathcal{S}^*$ is immediate-start. Furthermore, if $\mathcal{S}^*$ is non-migratory, it is useful to observe the following property.

Property 4.19. *Consider any optimal non-migratory schedule for J on $m \geq 2$ processors. At any time, for each job size, there are at most m jobs which have started but not yet finished. We call this property* m-proceeding.

Property 4.19 holds because in any optimal non-migratory schedule (no matter whether it is immediate-start or not), jobs of the same size dispatched to a processor must work in a First-Come-First-Serve manner. Otherwise we can shuffle the execution order to First-Come-First-Serve and reduce the total flow time, and the schedule is not optimal.

Note that the m-proceeding property may not hold for an optimal migratory schedule. Nevertheless, we observe a weaker property. In Section 4.4.3, we will prove that there exists an optimal migratory schedule for J that is $4m$-proceeding and immediate-start.

Now we are ready to state the performance of the schedule $\mathcal{S}^*$ constructed by Algorithm MakeCRR. Roughly speaking, if $\mathcal{S}^*$ is immediate-start and m-proceeding (or $4m$-proceeding), then $\mathcal{S}$ is a CRR schedule with comparable performance. Details are as follows. It is useful to define $\mu_\epsilon = (1+\epsilon)^{\alpha-1} + (1-1/\alpha)(2+\epsilon)/\epsilon^2$ for any $\epsilon > 0$. Note that by the definition of η_ϵ in Section 4.2, we have $\eta_\epsilon = (1+\epsilon)^\alpha \mu_\epsilon$.

Lemma 4.20. *Consider any $\epsilon > 0$. Given a power-of-$(1+\epsilon)$ job set J with an optimal (migratory or non-migratory) schedule $\mathcal{S}^*$ that is immediate-start and bm-proceeding for some $b \geq 1$, Algorithm* MakeCRR *(with $\lambda = \epsilon$) constructs a CRR(ϵ) schedule $\mathcal{S}$ for J such that $G(\mathcal{S}) \leq (b+1)\mu_\epsilon G(\mathcal{S}^*)$, and max-speed$(\mathcal{S}) \leq (1+\epsilon) \times$ max-speed$(\mathcal{S}^*)$.* [2]

The rest of this section is devoted to proving Lemma 4.20. We first analyze the energy usage. The analysis of flow time is based on an upper bound on the execution time $\mathcal{S}$ spends on jobs of certain classes within a period of time. We will state and prove this upper bound. Finally, with this upper bound, we can analyze the flow time.

Before going into the details of Lemma 4.20, we show how Lemma 4.20 immediately leads to a CRR schedule with the flow time and energy as stated in Theorem 4.7 (which was first mentioned in Section 4.2).

Theorem 4.7. *Given a job set J, let $\mathcal{O}_1$ and $\mathcal{O}_2$ be respectively an optimal non-migratory schedule and an optimal migratory schedule for J using maximum speed T. Then,*

i. for any $\epsilon > 0$, there is a CRR(ϵ) schedule $\mathcal{S}_1$ for J such that $G(\mathcal{S}_1) \leq 2\eta_\epsilon G(\mathcal{O}_1)$, and max-speed$(\mathcal{S}_1) \leq (1+\epsilon)^2 \times$ max-speed$(\mathcal{O}_1)$; and

ii. for any $\epsilon > 0$, there is a CRR(ϵ) schedule $\mathcal{S}_2$ for J such that $G(\mathcal{S}_2) \leq 5\eta_\epsilon G(\mathcal{O}_2)$, and max-speed$(\mathcal{S}_2) \leq (1+\epsilon)^2 \times$ max-speed$(\mathcal{O}_2)$.

Proof. We prove the non-migratory case only. The migratory case can be proven in the same way.

[2] In general, if Algorithm MakeCRR uses an arbitrary λ, then we have $G(\mathcal{S}) \leq (b+1)((1+\lambda)^{\alpha-1} + (1-1/\alpha)(2+\epsilon)/\lambda\epsilon)G(\mathcal{S}^*)$, and max-speed$(\mathcal{S}) \leq (1+\lambda) \times$ max-speed$(\mathcal{S}^*)$.

First of all, we apply Corollary 4.18 on J and $\mathcal{O}_1$, and we obtain a power-of-$(1+\epsilon)$ job set J' and an immediate-start, optimal non-migratory schedule $\mathcal{O}'$ for J' with maximum speed $(1+\epsilon)\times$ *max-speed*$(\mathcal{O}_1)$. Recall that every optimal non-migratory schedule including $\mathcal{O}'$ is m-proceeding.

Next, we apply Algorithm MakeCRR to J' and $\mathcal{O}'$ and construct a CRR(ϵ) schedule $\mathcal{S}'$ for J'. By Lemma 4.20, $G_{J'}(\mathcal{S}') \leq 2\mu_\epsilon G_{J'}(\mathcal{O}')$, and *max-speed*$(\mathcal{S}') \leq (1+\epsilon) \times$ *max-speed*$(\mathcal{O}')$. By Corollary 4.18, $\mathcal{S}'$ also defines a CRR(ϵ) schedule $\mathcal{S}_1$ for J such that $G_J(\mathcal{S}_1) \leq 2(1+\epsilon)^\alpha \mu_\epsilon G_J(\mathcal{O}_1)$. Note that *max-speed*$(\mathcal{S}_1) \leq (1+\epsilon) \times$ *max-speed*$(\mathcal{O}') \leq (1+\epsilon)^2 \times$ *max-speed*$(\mathcal{O}_1)$. □

Speed and energy

We now start to prove Lemma 4.20 in which the job set J in concern consists of power-of-$(1+\epsilon)$ jobs, $\mathcal{S}^*$ is an optimal schedule that is immediate-start and bm-proceeding, and $\mathcal{S}$ is the schedule constructed by Algorithm MakeCRR. We first note that in $\mathcal{S}$, the speed of a job is $(1+\epsilon)$ times the average speed of m jobs in $\mathcal{S}^*$, so *max-speed*$(\mathcal{S}) \leq (1+\epsilon) \times$ *max-speed*$(\mathcal{S}^*)$. Next, we consider the energy.

Lemma 4.21. *The energy used by $\mathcal{S}$ produced by Algorithm* MakeCRR *is at most* $(b+1)(1+\epsilon)^{\alpha-1} G(\mathcal{S}^*)$.

Proof. We first note that the energy incurred by running a job of size p at constant speed s is $s^\alpha p/s = s^{\alpha-1}p$, which is a convex function of the speed. Consider m jobs of the same size being run at different constant speeds, and let x be their average speed. Energy is a function of speed to the power of $\alpha - 1 \geq 1$, which is convex. Running a job of the same size at speed x incurs energy at most $1/m$ times the total energy for running these m jobs. If we further increase the speed to $(1+\epsilon)x$, the power increases by a factor of $(1+\epsilon)^\alpha$, and the running time decreases by a factor of $(1+\epsilon)$. Thus, the energy usage increases by a factor of $(1+\epsilon)^{\alpha-1}$. In $\mathcal{S}$, running a job at $(1+\epsilon)$ times the average speed of m jobs in $\mathcal{S}^*$ requires no more energy than $(1+\epsilon)^{\alpha-1}/m$ times the sum of the energy usage of those m jobs in $\mathcal{S}^*$.

To bound $E(\mathcal{S})$, we use a simple charging scheme: for a job j in $\mathcal{S}$, we charge to every one of the m jobs j' chosen for determining the speed of j in Algorithm MakeCRR; the amount to be charged is $1/m$ times of the energy usage of j' in $\mathcal{S}^*$. By Algorithm

MakeCRR, each job can be charged by at most $(b+1)m$ jobs. Thus,

$$\begin{aligned} E(\mathcal{S}) &\leq \frac{(1+\epsilon)^{\alpha-1}}{m}(b+1)mE(\mathcal{S}^*) \\ &\leq (b+1)(1+\epsilon)^{\alpha-1}E(\mathcal{S}^*) \\ &\leq (b+1)(1+\epsilon)^{\alpha-1}G(\mathcal{S}^*) \ . \end{aligned}$$ □

Upper bound on job execution time of $\mathcal{S}$

To analyze the flow time of a job in $\mathcal{S}$, we attempt to upper bound the execution time of other jobs dispatched to the same processor during its lifespan. The lemmas below look technical, yet the key observation is quite simple—For any processor z, if we consider all jobs that $\mathcal{S}$ dispatches to z during an interval I, excluding the last $(b+1)$ jobs of each class (size), their total execution time is at most $\ell/(1+\epsilon)$, where ℓ is the length of I.

Consider any job $h_0 \in J$. Let $h_1, h_2, \ldots, h_n$ be all the jobs in J such that $r(h_0) \leq r(h_1) \leq \cdots \leq r(h_n)$ and they have the same size as h_0. Suppose that $n \geq im$ for some $i \geq b+1$. We focus on two sets of jobs: $\{h_0, h_1, \ldots, h_{im-1}\}$ and $\{h_0, h_m, h_{2m}, \ldots, h_{(i-b-1)m}\}$. The latter contains jobs dispatched to the same processor as h_0. Lemma 4.22 below gives an upper bound on the execution time of $\mathcal{S}$ for $\{h_0, h_m, h_{2m}, \ldots, h_{(i-b-1)m}\}$ with respect to $\mathcal{S}^*$. This lemma stems from the fact that $\mathcal{S}^*$ is immediate-start.

Lemma 4.22. *For any job h_0 and $i \geq b+1$, suppose h_{im} exists. Let t be the execution time of $\mathcal{S}^*$ for the jobs $h_0, h_1, \ldots, h_{im-1}$ during the interval $[r(h_0), r(h_{im})]$. Then in the entire schedule of $\mathcal{S}$, the total execution time of the jobs $h_0, h_m, \ldots, h_{(i-b-1)m}$ is at most $t/m(1+\epsilon)$.*

Proof. Since $\mathcal{S}^*$ is immediate-start, jobs $h_0, \ldots, h_{im-1}$ each starts within the interval $[r(h_0), r(h_{im})]$. As $\mathcal{S}^*$ is bm-proceeding, at time $r(h_{im})$, at most bm jobs among these im jobs have not yet finished, or equivalently, $\mathcal{S}^*$ has completed at least $(i-b)m$ jobs. Let Δ denote a set of any $(i-b)m$ such completed jobs. Based on release times, we partition Δ accordingly into $i-b$ subsets Δ_0, Δ_1, …, Δ_{i-b-1}, each of size exactly m. Δ_0 contains the m jobs with smallest release times in Δ, Δ_1 contains jobs with the next m smallest release times in Δ, etc.

Since Δ misses out only bm jobs in $\{h_0, h_1, \ldots, h_{im-1}\}$, each Δ_u, for $u \in \{0, \ldots, i-$

$b-1\}$, is a subset of the $(b+1)m$ jobs $\{h_{um}, h_{um+1}, \ldots, h_{um+(b+1)m-1}\}$. Because the speed used by $\mathcal{S}$ for h_{um} is $(1+\epsilon)$ times the average speed of the m fastest jobs in $h_{um}, h_{um+1}, \ldots, h_{um+(b+1)m-1}$ used by $\mathcal{S}^*$, which is faster than $(1+\epsilon)$ times the average speed of Δ_u in $\mathcal{S}^*$, it follows that the execution time of h_{um} in $\mathcal{S}$ is at most $1/m(1+\epsilon)$ times the total execution time of Δ_u in $\mathcal{S}^*$. Summing over all $u \in \{0, \ldots, i-b-1\}$, the execution time of $\mathcal{S}$ for $h_0, h_m, \ldots, h_{(i-b-1)m}$ is no more than $1/m(1+\epsilon)$ times the total execution time of Δ in $\mathcal{S}^*$. In $\mathcal{S}^*$, Δ is only executed during $[r(h_0), r(h_{im})]$, and the lemma follows. □

Below is the main result to upper bound the job execution time of $\mathcal{S}$ (to be used for analyzing the flow time of $\mathcal{S}$).

Lemma 4.23. *Consider any k and any time interval I of length ℓ. For jobs of size at most $(1+\epsilon)^k$ that are released during I, the total execution time of any processor in $\mathcal{S}$ for these jobs is at most $\ell/(1+\epsilon) + (b+1)(1+\epsilon)^{k+1} \cdot (\alpha-1)^{1/\alpha}/\epsilon(1+\epsilon)$.*

Proof. Consider a particular $k' \leq k$. Let y be the execution time over all processors that $\mathcal{S}^*$ uses for jobs of size $(1+\epsilon)^{k'}$ during the interval I. Consider a particular processor z in $\mathcal{S}$; suppose that $\mathcal{S}$ dispatches i jobs of size $(1+\epsilon)^{k'}$ to processor z during I, and denote these i jobs as $J' = \{h'_0, h'_m, \ldots, h'_{(i-1)m}\}$, arranged in the order of their release times. We claim that the execution time of processor z in $\mathcal{S}$ for these i jobs is at most $y/m(1+\epsilon)$ plus the execution time of $\mathcal{S}$ for the last $b+1$ jobs of J'. This is obvious if J' contains $b+1$ or fewer jobs. It remains to consider the case when J' has $i \geq b+2$ jobs. By Lemma 4.22, if t is the execution time of $\mathcal{S}^*$ for $h'_0, h'_1, \ldots, h'_{(i-1)m-1}$ during $[r(h'_0), r(h'_{(i-1)m})]$, then $\mathcal{S}$ uses no more than $t/m(1+\epsilon)$ time to execute $h'_0, h'_m, \ldots, h'_{(i-b-2)m}$. The claim then follows by noticing that $t \leq y$, and we only have $b+1$ jobs $h'_{(i-b-1)m} \cdots h'_{(i-1)m}$ not being counted.

Now we sum over all $k' \leq k$ the upper bound of these flow times, i.e., $y/m(1+\epsilon)$ plus the execution time of $\mathcal{S}$ for the last $b+1$ jobs in J'. The sum of the first part is $\sum y/m(1+\epsilon)$. Note that $\sum y$ is the execution time of $\mathcal{S}^*$ during I, so $\sum y \leq m|I| = m\ell$, and the sum of the first part is

$$\frac{\sum y}{m(1+\epsilon)} \leq \frac{\ell}{1+\epsilon} \ .$$

The sum of the second part is over at most $b+1$ jobs for each k'. Recall that the speed

used by $\mathcal{S}^*$ is at least the global critical speed $1/(\alpha-1)^{1/\alpha}$, and the speed used by $\mathcal{S}$ is $(1+\epsilon)$ times the average of some job speeds in $\mathcal{S}^*$. Thus the speed used by $\mathcal{S}$ for any job is at least $(1+\epsilon)/(\alpha-1)^{1/\alpha}$, and the execution time of each job of size $(1+\epsilon)^{k'}$ is at most $(1+\epsilon)^{k'}(\alpha-1)^{1/\alpha}/(1+\epsilon)$. Summing over all k' the execution time for these jobs, we have

$$\sum_{k'=0}^{k} \frac{(b+1)(1+\epsilon)^{k'}(\alpha-1)^{1/\alpha}}{1+\epsilon} < \frac{(b+1)(1+\epsilon)^{k+1}(\alpha-1)^{1/\alpha}}{\epsilon(1+\epsilon)} \ .$$

The lemma follows by summing the two parts. □

Flow time

Now we show that the flow time of each job in $\mathcal{S}$ is $O(1/\epsilon^2)$ times of its job size (Lemma 4.24), which implies that the total flow time is $O(1/\epsilon^2)G(\mathcal{S}^*)$ (Corollary 4.25). Together with Lemma 4.21, Lemma 4.20 can be proved. We first bound the flow time of a job of a particular job size in $\mathcal{S}$, making use of Lemma 4.23.

Lemma 4.24. *In $\mathcal{S}$, the flow time of a job of size $(1+\epsilon)^k$ is at most $(b+1)(2+\epsilon)(1+\epsilon)^k(\alpha-1)^{1/\alpha}/\epsilon^2$.*

Proof. Consider a job j of size $(1+\epsilon)^k$ that is scheduled on some processor z in $\mathcal{S}$. Let $r = r(j)$, and f be the flow time of j in $\mathcal{S}$, i.e., j completes at time $r+f$. To determine f, we focus on the scheduling of processor z in the intermediate schedule $\mathcal{S}_0$ immediate after Algorithm MakeCRR has scheduled j. Note that f is due to jobs that have been executed in $\mathcal{S}$ during $[r, r+f]$. They can be partitioned into two subsets: J_1 for jobs released at or before r, and J_2 for jobs released during $(r, r+f]$. Let F_1 and F_2 be the contribution on f by J_1 and J_2, respectively, i.e., $f = F_1 + F_2$.

We first consider J_1. Let t be the last time $< r$ such that processor z is idle right before t in $\mathcal{S}_0$. Thus all jobs executed by processor z at or after t, and hence all jobs in J_1, must be released at or after t. By Lemma 4.23, the execution time of processor z for jobs in J_1 is no more than $(r-t)/(1+\epsilon) + [(b+1)(1+\epsilon)^{k+1}(\alpha-1)^{1/\alpha}/\epsilon(1+\epsilon)]$. Since processor z is busy throughout $[t, r)$, the amount of execution time for jobs in J_1

remaining at r is at most

$$\frac{r-t}{1+\epsilon} + \frac{(b+1)(1+\epsilon)^{k+1}(\alpha-1)^{1/\alpha}}{\epsilon(1+\epsilon)} - (r-t)$$
$$\leq \frac{(b+1)(1+\epsilon)^{k+1}(\alpha-1)^{1/\alpha}}{\epsilon(1+\epsilon)} \ .$$

This implies $F_1 \leq (b+1)(1+\epsilon)^{k+1}(\alpha-1)^{1/\alpha}/\epsilon(1+\epsilon)$.

Next we consider J_2. Since Algorithm MakeCRR schedules jobs from the smallest to the largest size, jobs in J_2 are of size at most $(1+\epsilon)^{k-1}$. We apply Lemma 4.23 to the interval $[r, r+f]$ for jobs of size at most $(1+\epsilon)^{k-1}$. The execution time of processor z for jobs in J_2, i.e., F_2, is no more than

$$\frac{f}{1+\epsilon} + \frac{(b+1)(1+\epsilon)^{k}(\alpha-1)^{1/\alpha}}{\epsilon(1+\epsilon)} \ .$$

Then we have

$$f = F_1 + F_2 \leq \frac{(b+1)(2+\epsilon)(1+\epsilon)^{k}(\alpha-1)^{1/\alpha}}{\epsilon(1+\epsilon)} + \frac{f}{1+\epsilon} \ ,$$

immediately implying $f \leq (b+1)(2+\epsilon)(1+\epsilon)^{k}(\alpha-1)^{1/\alpha}/\epsilon^2$. □

Summing over all jobs and recalling that $G(\mathcal{S}^*) \geq \frac{\alpha}{(\alpha-1)^{1-1/\alpha}} p(J)$ (see Lemma 4.1), we have the following corollary.

Corollary 4.25. *The total flow time incurred by $\mathcal{S}$ produced by Algorithm* MakeCRR *is at most* $((b+1)(1-1/\alpha)(2+\epsilon)/\epsilon^2)G(\mathcal{S}^*)$.

By Lemma 4.21 and Corollary 4.25, we have $G(\mathcal{S}) = E(\mathcal{S}) + F(\mathcal{S}) \leq (b+1)[(1+\epsilon)^{\alpha-1} + (1-1/\alpha)(2+\epsilon)/\epsilon^2]G(\mathcal{S}^*) = (b+1)\mu_\epsilon G(\mathcal{S}^*)$. We have also noted that *max-speed*$(\mathcal{S}) \leq (1+\epsilon) \times$ *max-speed*$(\mathcal{S}^*)$. Hence, Lemma 4.20 follows.

4.4.3 Optimal migratory schedules

Algorithm MakeCRR and Lemma 4.20 can be applied to an optimal migratory schedule as long as it is immediate-start and bm-proceeding for some integer $b \geq 1$. Note that

the m-proceeding property or even the $4m$-proceeding property does not hold for every optimal migratory schedule. Nevertheless, Lemma 4.26 below shows that at least one optimal schedule satisfies the $4m$-proceeding property (i.e., at any time, there are at most $4m$ jobs of the same size started but not yet completed). Once we know the existence of such schedule, we can apply the construction in Lemma 4.17 to further modify the job set and the schedule to obtain an optimal migratory schedule that is $4m$-proceeding and immediate-start (since the only manipulation done by Lemma 4.17 is to swap the schedule of pairs of jobs). The rest of the arguments then follow, leading to Theorem 4.7 (ii).

Lemma 4.26. *For any job set J, there exists an optimal migratory schedule $\mathcal{S}^*$ that is $4m$-proceeding.*

The rest of this section is devoted to proving the above lemma. Recall that m-proceeding property holds for every optimal non-migratory schedule. For optimal migratory schedules, we find that some of them, which we call *lazy-start* optimal migratory schedules, satisfy the $4m$-proceeding property. The definition is as follows: Given a schedule $\mathcal{S}$, we define its "start time sequence" to be the sequence of start time of each job, sorted in the order of time. Among all optimal migratory schedules (which may or may not be immediate-start), a lazy-start optimal schedule is the one with lexicographically maximum start time sequence. Such a schedule has the following property.

Lemma 4.27. *In a lazy-start optimal schedule, suppose a job j_1 starts at time t, while another job j_2 of the same size that has already started before t but has not finished at t is not running at t. Then after t, j_1 runs whenever j_2 runs.*

Proof. Suppose the contrary, and let t' be the first time after t that j_2 runs but j_1 does not. Let p_0 be the amount of work processed for j_1 during $[t, t']$ when j_2 is not running. We divide the analysis into three cases, each arriving at a contradiction.

Case 1: j_1 is not yet completed by t'. We can exchange some work of j_2 done starting from t' with those of j_1 done starting from t, without changing processor speed at any time. The start time of j_1 is thus delayed without changing the start times of other jobs or increasing the energy or flow time, so the original schedule is not lazy-start.

Case 2: j_1 is completed by t', and the amount of work processed for j_2 after t' is at most p_0. We can exchange all work of j_2 after t' with some work of j_1

starting from t. The completion times of j_1 and j_2 are exchanged, but the total flow time and energy is preserved. The start time of j_1 is delayed without changing the start times of other jobs, so the original schedule is not lazy-start.

Case 3: j_1 is completed by t', and the amount of work processed for j_2 after t' is more than p_0. Since j_1 and j_2 are of the same size, these conditions imply that there must be some work processed for j_1 when both j_1 and j_2 are running. Furthermore, j_2 must be running slower than j_1 during this period, otherwise the total amount of work processed for j_2 would be larger than the size of j_1, so the two jobs cannot be of the same size. Since jobs run at constant speed in optimal schedules, the speed of j_1 is higher than the speed of j_2.

Note that j_1 lags behind j_2 at t but is ahead of j_2 at t'. So there must be a time $t_0 \in (t', t)$ such that j_1 and j_2 has been processed for the same amount of work. Exchange the scheduling of j_1 and j_2 after t_0 gives a schedule with the completion time of j_1 and j_2 exchanged, while the energy consumption and flow time remain the same. But now j_1 and j_2 are not running at constant speed, so the schedule is not optimal. □

Now we are ready to prove Lemma 4.26 by showing that a lazy-start optimal migratory schedule is $4m$-proceeding.

Proof of Lemma 4.26. Given a job set J, let $\mathcal{S}$ be a lazy-start optimal migratory schedule. Suppose, for the sake of contradiction, that at some time in $\mathcal{S}$, there are $4m$ jobs started but not yet finished. Consider these $4m$ jobs. At the time when the $(m+i)$-th job j starts, at least i jobs which started earlier must be idle. For each such idling job j', Lemma 4.27 dictates that after $r(j)$, whenever j' runs, j must also be running. In this case, we say that j' *implies* j. Since there are $4m$ jobs, there are $1 + 2 + \cdots + 3m = \frac{3}{2}m(3m+1)$ such relations. Thus some job j_0 implies at least $\frac{3}{2}m(3m+1)/4m > m$ other jobs. After all these other jobs are released, they must all run whenever j_0 runs. This contradicts that there are only m processors. The lemma follows. □

4.5 Lower Bound

We show a lower bound for minimizing total flow time plus energy when there is a maximum speed. The bound holds even if jobs are restricted to power-of-2 sizes, implying that CRR-A is optimal in this setting. We start with a lower bound of $\Omega(\log P)$ given by Leonardi and Raz [69] on the competitive ratio of any migratory online algorithm for flow time scheduling on fixed-speed processors. Their proof is based on power-of-2 sized jobs. This proof can be adapted easily to show the following theorem.

Theorem 4.28. *Any online (migratory) algorithm for minimizing flow time plus energy on multiple processors with a maximum speed has a competitive ratio of* $\Omega(\log P)$. *This holds even if jobs are restricted to have power-of-2 sizes.*

Proof. Consider the case that $T = 1$, which is greater than the global critical speed when $\alpha > 2$. We consider a particular online algorithm. Let F_a and G_a be the total flow time and total flow time plus energy of the online algorithm, respectively. By the lower bound result in [69], there is some constant c that for any online algorithm (including the one which we are considering), we can find some input J consisting of jobs of power-of-2 size, such that $F_a \geq (c \log P)F_s$, where F_s is the minimum total flow time among all schedules of J. Obviously, the minimum total flow time F_s is achieved by running all processors at the maximum speed 1. The energy consumption of such a schedule is thus the time the processor is running, which is at most F_s. We thus have a schedule of J with total flow time plus energy at most $2F_s$, which implies that the optimal algorithm have total flow time plus energy $G_o \leq 2F_s$. Together with the bound of F_a, we have $G_a \geq F_a \geq (c \log P)F_s \geq (c/2)(\log P)G_o$, which completes the proof. □

Chapter 5

Non-clairvoyant Flow-Energy Scheduling

In this chapter, we initiate the study of flow-energy scheduling in the non-clairvoyant model. In some applications like operating systems, job size is only known when the job finishes. This is referred to as the *non-clairvoyant* model. This is in contrast to the clairvoyant model considered in previous chapters, where the size of a job is known at its release time.

We consider a job set J to be scheduled on a processor. For each job $j \in J$, its release time and size are denoted as $r(j)$ and $p(j)$, respectively. In the nonclairvoyant model, when a job j arrives, $p(j)$ is not given and it is known only when j is completed. Preemption is allowed and has no cost; a preempted job can resume at the point of preemption. The processor can vary its speed dynamically to any value between 0 and the maximum speed T. Most results in this chapter are on the infinite speed model, where $T = \infty$. When running at speed s, the processor processes s units of work per unit time and consumes s^α units of energy per unit time, where $\alpha > 1$ is some fixed constant.

Consider a certain schedule A of J. We say a job j is *active* at time t if j is released by time t but not yet completed by time t. For any job j in J, the flow time of j is denoted by $F_A(j)$, and the total flow time of A is $F_A = \sum_{j \in I} F_A(j) = \int_0^\infty n_A(t)dt$, where $n_A(t)$ is the number of active jobs at time t in A. Let $s_A(t)$ be the speed of the processor at time t in A. Then the total energy usage of A is $E_A = \int_0^\infty (s(t))^\alpha dt$. The objective is to minimize

the sum of total flow time and energy usage, which is denoted by $G_A = F_A + E_A$.

In Section 5.1, we first focus on *batched jobs*, i.e., all jobs are released at time 0. We give an algorithm that is $(2 - \frac{1}{\alpha})$-competitive in the infinite speed model and 2-competitive in the bounded speed model. The latter inherits a lower bound of 2 from flow-time scheduling [74]. We will further generalize this algorithm for jobs with different weights and the objective becomes minimizing weighted flow time plus energy. The competitive ratios become $(2 - \frac{1}{\alpha})^2$ and 4 in the infinite and bounded speed models, respectively. Sections 5.2 and 5.3 consider jobs with arbitrary release times on a processor without and with sleep states, respectively. Both sections focus on the infinite speed model; the problem in the bounded speed model remains open. In Section 5.2, we analyze a nonclairvoyant algorithm LAPS and show that it is $O(\alpha^3)$-competitive for flow plus energy in the infinite speed model. In Section 5.3, we extend the study to a model that allows both sleep management and speed scaling (see Chapter 3). We adapt LAPS and show that the adapted algorithm together with the sleep management algorithm IdleLonger (which is introduced in Chapter 3) remains $O(\alpha^3)$-competitive for flow plus energy.

5.1 Batched Jobs

This section considers non-clairvoyant scheduling of batched jobs J, where all jobs $j \in J$ has the same release time $r(j) = 0$. We first define the speed function AJC* and the algorithm RR-AJC*, as follows. Recall that T is the maximum speed of the processor.

AJC* and RR-AJC*. The speed function AJC* is defined as $\min\{T, (\frac{n(t)}{\alpha-1})^{1/\alpha}\}$, where $n(t)$ is the number of active jobs at t. To cope with unknown job sizes, RR-AJC* uses AJC* with round robin (RR): split the processor equally among all active jobs.

To analyze the non-clairvoyant algorithm RR-AJC*, we consider a clairvoyant algorithm SJF-AJC* (shortest job first plus AJC*). In Section 5.1.1, we show an interesting relation that the flow time plus energy incurred by RR-AJC* is close to that of SJF-AJC*. In Section 5.1.2, we complete the analysis by showing SJF-AJC* is optimal.

In some applications, jobs may carry weights to reflect their importance. We use $w(j)$ to denote the weight of a job j, and define its *density* $\rho(j)$ to be $w(j)/p(j)$. The *weighted flow time* of a job is simply its flow time multiplied by its weight. The above result can

be generalized for weighted flow time plus energy as follows.

AJW* and WRR-AJW*. The speed function AJW* (active job weight) is defined as $\min\{T, (\frac{w(t)}{\alpha-1})^{1/\alpha}\}$, where $w(t)$ is the total weight of active jobs at t. The algorithm WRR-AJW* uses AJW* with weighted round robin (WRR): split the processor among all active jobs in the ratio of their weights. We define the *normalized work* of a job as its work divided by its weight. Every active job has the same normalized *processed* work at any time.

To analyze WRR-AJW*, we consider a clairvoyant algorithm HDF-AJW* (highest density first plus AJW*). The relation between WRR-AJW* and HDF-AJW* is the same as before (Section 5.1.1). However, HDF-AJW* is not optimal for weighted flow plus energy. Yet we show that HDF-AJW* is $O(1)$-competitive in Section 5.1.3.

We use the weighted setting as a common platform, where RR-AJC* (resp. SJF-AJC*) is WRR-AJW* (resp. HDF-AJW*) with job weights all equal to one. Without loss of generality, we assume the input $J = \{j_1, j_2, \ldots, j_n\}$ is in increasing job density order, i.e., $\rho(j_1) \leq \rho(j_2) \leq \cdots \leq \rho(j_n)$ (ties are broken by job IDs), Note that this is equivalent to that the jobs are in decreasing normalized work order. To simplify notations, for any job j_i, we also use r_i, p_i, w_i and ρ_i to represent $r(j_i)$, $p(j_i)$, $w(j_i)$ and $\rho(j_i)$.

5.1.1 Comparing WRR-AJW* against HDF-AJW*

As jobs are batched, WRR implies that jobs complete in increasing order of normalized work, from j_n to j_1. Thus, if j_i is the smallest normalized work job at time t, then the total weight of active jobs at t is $w(t) = \sum_{k=1}^{i} w_k$. We can thus compare WRR-AJW* against HDF-AJW* easily.

Lemma 5.1. *For scheduling batched jobs, the weighted flow time plus energy of* WRR-AJW* *is (i) at most* $(2 - 1/\alpha)$ *times of* HDF-AJW* *in the infinite speed model, and (ii) at most* 2 *times of* HDF-AJW* *in the bounded speed model.*

To prove Lemma 5.1, we focus on the contributions to weighted flow time and energy during the time when a particular job j_i is the job with the smallest normalized work in WRR-AJW*. Due to the WRR policy, each of the remaining jobs j_j with $1 \leq j \leq i$ is run for the same amount of normalized work. We thus evaluate the weighted flow time

and energy incurred by WRR-AJW* during this period, denoted as $F_{\mathrm{w}}(j_i)$ and $E_{\mathrm{w}}(j_i)$. They are compared against the weighted flow time and energy incurred by HDF-AJW* to process that same amount of normalized work for each of these jobs, denoted as $F_{\mathrm{h}}(j_i)$ and $E_{\mathrm{h}}(j_i)$. We show that both $F_{\mathrm{w}}(j_i)/F_{\mathrm{h}}(j_i)$ and $E_{\mathrm{w}}(j_i)/E_{\mathrm{h}}(j_i)$ are no greater than $(2-1/\alpha)$ for $T=\infty$, and 2 for general T. Summing over all j_i leads to Lemma 5.1.

Lemma 5.2. *If $T=\infty$, $F_{\mathrm{w}}(j_i) \le (2-1/\alpha)F_{\mathrm{h}}(j_i)$ and $E_{\mathrm{w}}(j_i) \le (2-1/\alpha)E_{\mathrm{h}}(j_i)$.*

Proof. By the definition of the speed function AJW*, when $T=\infty$, we have $F_{\mathrm{w}}(j_i) = (\alpha-1)E_{\mathrm{w}}(j_i)$ and $F_{\mathrm{h}}(j_i) = (\alpha-1)E_{\mathrm{h}}(j_i)$. Thus it suffices to show $F_{\mathrm{w}}(j_i) \le (2-1/\alpha)F_{\mathrm{h}}(j_i)$.

Let us first consider WRR-AJW*. Recall that we are considering the time period when the job j_i is the job with the smallest normalized work in WRR-AJW*. Let Δh be the normalized work processed in this period for each job $j_1,\ldots,j_i$. We have seen that $w(t) = \sum_{k=1}^{i} w_k$ and the processor speed in this period is $s = (\frac{w(t)}{\alpha-1})^{1/\alpha}$. The amount of work processed for j_j $(1 \le j \le i)$ is $w_j\Delta h$, so the total weighted flow time incurred $F_{\mathrm{w}}(j_i) = \sum_{j=1}^{i} w_j\Delta h \sum_{k=1}^{i} w_k/s = (\alpha-1)^{1/\alpha}(\sum_{j=1}^{i} w_j)^{2-1/\alpha}\Delta h$.

In contrast, HDF-AJW* runs only the highest density (i.e., least normalized work) job. At time t when processing a job j_j, $w(t) = \sum_{k=1}^{j} w_k$ and the speed is $s = (\frac{w(t)}{\alpha-1})^{1/\alpha}$. The weighted flow time incurred for processing an amount of work $w_j\Delta h$ for j_j is thus $w_j\Delta h(\sum_{k=1}^{j} w_k)/s = (\alpha-1)^{1/\alpha}\Delta h(\sum_{k=1}^{j} w_k)^{1-1/\alpha}w_j$. We thus have

$$F_{\mathrm{h}}(j_i) = \sum_{j=1}^{i}\left((\alpha-1)^{1/\alpha}\Delta h\Big(\sum_{k=1}^{j} w_k\Big)^{1-1/\alpha} w_j\right) = (\alpha-1)^{1/\alpha}\Delta h\sum_{j=1}^{i}\Big(\sum_{k=1}^{j} w_k\Big)^{1-1/\alpha} w_j \ .$$

To approximate $\sum_{j=1}^{i}\big(\sum_{k=1}^{j} w_k\big)^{1-1/\alpha}w_j$, we use the staircase-like function

$$f(x) = (\textstyle\sum_{k=1}^{j} w_k)^{1-1/\alpha} \qquad \text{if } x \in [\sum_{k=1}^{j-1} w_k, \sum_{k=1}^{j} w_k) \text{ where } 1 \le j \le i \quad .$$

Note that $\sum_{j=1}^{i}(\sum_{k=1}^{j} w_k)^{1-1/\alpha}w_j$ is exactly $\int_0^{\sum_{j=1}^{i} w_j} f(x)\,\mathrm{d}x$. On the other hand, $f(x) \ge x^{1-1/\alpha}$ for all $x \in [0, \sum_{j=1}^{i} w_j)$. We thus have

$$\sum_{j=1}^{i}\Big(\sum_{k=1}^{j} w_k\Big)^{1-1/\alpha} w_j \ge \int_0^{\sum_{j=1}^{i} w_j} x^{1-1/\alpha}\,\mathrm{d}x = \frac{(\sum_{j=1}^{i} w_j)^{2-1/\alpha}}{2-1/\alpha} \ ,$$

and $F_{\mathrm{h}}(j_i) \ge (\alpha-1)^{1/\alpha}\Delta h\big(\sum_{j=1}^{i} w_j\big)^{2-1/\alpha}/(2-\frac{1}{\alpha}) = F_{\mathrm{w}}(j_i)/(2-\frac{1}{\alpha})$. □

Proof of Lemma 5.1. We obtain (i) ($T = \infty$) by summing the relations about F_{h} and F_{w} and those about E_{h} and E_{w} in Lemma 5.2 over all j_i.

For (ii) ($T < \infty$), suppose WRR-AJW* processes Δh normalized work for each of $j_1, \ldots, j_i$. We first focus on $E_{\mathrm{w}}(j_i)$ and $E_{\mathrm{h}}(j_i)$. If $(\sum_{j=1}^{i} w_j/(\alpha-1))^{1/\alpha} \leq T$, Lemma 5.2 applies, so $E_{\mathrm{w}}(j_i) \leq (2 - 1/\alpha)E_{\mathrm{h}}(j_i)$. Otherwise, we try to find a $k \in \{1, \ldots, i\}$ such that $\sum_{j=1}^{k} w_j$ is exactly $(\alpha - 1)T^\alpha$ (so that the speed bound is just not exceeded). If no such k exists, for the sake of analysis we split some job j_u into two jobs j_{u1} and j_{u2} with the same density and total weight, so that $w_{u1} + \sum_{j=1}^{u-1} w_j = (\alpha - 1)T^\alpha$. We set $k = u1$. This job splitting does not affect the speed function, and thus the energy consumption, of either WRR-AJW* and HDF-AJW* (speed used for j_{u1}, j_{u2} and j_u are all T). We now notice that both WRR-AJW* and HDF-AJW* run $j_{k+1}, \ldots, j_i$ at speed T, consuming the same energy E_0. The other jobs are run as if $T = \infty$, so Lemma 5.2 leads to $E_{\mathrm{w}}(j_i) - E_0 \leq (2 - 1/\alpha)(E_{\mathrm{h}}(j_i) - E_0)$. This implies $E_{\mathrm{w}}(j_i) \leq (2 - 1/\alpha)E_{\mathrm{h}}(j_i)$.

We now compare $F_{\mathrm{w}}(j_i)$ and $F_{\mathrm{h}}(j_i)$. Again the interesting case is when $(\sum_{j=1}^{i} w_j/(\alpha - 1))^{1/\alpha} > T$, otherwise $F_{\mathrm{w}}(j_i) \leq (2-1/\alpha)F_{\mathrm{h}}(j_i)$ by Lemma 5.2. For WRR-AJW*, the processor uses speed T, so the time needed is $\sum_{j=1}^{i} w_j \Delta h/T$, and $F_{\mathrm{w}}(j_i) = (\sum_{j=1}^{i} w_j)^2 \Delta h/T$. For HDF-AJW*, the time needed to run j_j for $w_j \Delta h$ units of work is at least $w_j \Delta h/T$ (since the speed used cannot be faster than T), incurring weighted flow time of at least $\sum_{k=1}^{j} w_k w_j \Delta h/T$. We thus have $F_{\mathrm{h}}(j_i) \geq \sum_{j=1}^{i} \sum_{k=1}^{j} w_k w_j \Delta h/T > (\sum_{j=1}^{i} w_j)^2 \Delta h/2T = F_{\mathrm{w}}(j_i)/2$.

Summing these relations over all j_i gives the desired ratio in Lemma 5.1. □

5.1.2 Analysis of SJF-AJC*

We show that the speed function AJC* minimizes the flow time plus energy for scheduling batched jobs using SJF (Lemma 5.3), implying the optimality of SJF-AJC* for flow time plus energy. Combining with Lemma 5.1, we obtain the competitive ratio of RR-AJC* (Theorem 5.4).

Lemma 5.3. *Consider a set of batched jobs J. Among all schedules of J using SJF for job selection, the schedule that incurs the minimum flow time plus energy sets the speed at any time t as* $\min\{T, (\frac{n(t)}{\alpha-1})^{1/\alpha}\}$*, where $n(t)$ is the number of active jobs at t.*

Proof. Consider a particular job j_i in the optimal schedule. We only need to consider cases where its speed is constant, otherwise we can average the speed to reduce energy usage without affecting flow time. Note that $n(t)$ is unchanged when j_i is run. Suppose j_i runs at speed s. Then its contribution to flow time plus energy is $n(t)p_i/s + s^{\alpha-1}p_i$, which is minimized when $s = (\frac{n(t)}{\alpha-1})^{1/\alpha}$. □

Theorem 5.4. *For scheduling batched jobs to minimize flow time plus energy, the algorithm* RR-AJC* *is (i)* $(2 - 1/\alpha)$*-competitive in the infinite speed model, and (ii)* 2*-competitive in the bounded speed model.*

Proof. Note that for scheduling batched jobs, the policy SRPT is equivalent to SJF. Furthermore, Lemma 2.1 in Chapter 2 states that if there is a schedule for a job set J uses a speed function f, then among all schedules of J using f, the one selecting jobs in accordance with SRPT incurs the least total flow time. Therefore, every schedule can be modified to the SJF-AJC* schedule by applying Lemma 2.1 and then Lemma 5.3 without increasing the total flow time plus energy. This implies SJF-AJC* is optimal for total flow time plus energy. The theorem thus follows naturally from Lemma 5.1. □

5.1.3 Analysis of HDF-AJW*

When jobs are weighted, the clairvoyant algorithm HDF-AJW* is not optimal for weighted flow time plus energy. Instead we analyze HDF-AJW* via a variant concerning fractional flow. Consider a schedule of J. At any time t, the *fractional weight* $\widehat{w}_i$ of j_i is defined to be $w_i(q_i/p_i)$, where q_i is the remaining work of j_i at t. We define $\widehat{w}(t) = \sum_{j_i \text{ is active at } t} \widehat{w}_i$. The *fractional flow* is defined as $\widehat{F} = \int_0^\infty \widehat{w}(t)\,\mathrm{d}t$. We now define our variant of HDF-AJW*.

Algorithm HDF-FW. It differs from HDF-AJW* only in the speed function, which uses the speed function FW defined as $(\frac{\widehat{w}(t)}{\alpha-1})^{1/\alpha}$ at any time t.

We follow the framework in Section 5.1.2 to show the optimality of HDF-FW for fractional flow plus energy, as follows. Firstly, HDF minimizes fractional flow for a fixed speed function (Lemma 5.5). Secondly, FW minimizes the fractional flow plus energy for scheduling batched jobs using HDF (Lemma 5.6). Therefore, every schedule can be modified to the HDF-FW schedule by applying Lemma 5.5 and then Lemma 5.6 without increasing the fractional flow plus energy. That is, HDF-FW is optimal (Corollary 5.7).

Lemma 5.5. *Consider a set of weighted batched jobs J. Suppose there is a schedule for J using a speed function f. Then, among all schedules of J using the speed function f, the schedule that selects jobs using HDF incurs the minimum fractional flow.*

Proof. Given any schedule S_0 of J that does not follow HDF policy, we first show that S_0 can be modified in multiple steps to the HDF schedule. Then we show that in each step, its fractional flow decreases. Thus the lemma follows.

In each step, we modify a schedule S to a new schedule S'. We find the first maximal time interval $[t_1, t_2]$ where S schedules two jobs j_i and then j_j, and $\rho_i < \rho_j$. We modify S to S' by reversing the order of processing for the work of jobs j_i and j_j during the interval $[t_1, t_2]$ such that j_j is run before j_i. Repeating the above steps, we obtain a schedule that uses HDF.

Now consider the fractional flow of S and S'. To ease the discussion, we let $\widehat{F}(S)$ be fractional flow of schedule S, and let $\widehat{w}(S)$ be a function over time representing the total fractional weight of S. Recall that $\widehat{F}(S) = \int_0^\infty \widehat{w}(S)\,\mathrm{d}t$. We split $\widehat{w}(S)$ into two parts: $\widehat{w}_{\mathrm{a}}(S)$ consists of the fractional flow of j_1 and j_2, and $\widehat{w}_{\mathrm{b}}(S)$ for all other jobs. Then $\widehat{F}(S) = \int_0^\infty \widehat{w}_{\mathrm{a}}(S) + \widehat{w}_{\mathrm{b}}(S)\,\mathrm{d}t$. Similarly, we split $\widehat{w}(S')$ into $\widehat{w}_{\mathrm{a}}(S')$ and $\widehat{w}_{\mathrm{b}}(S')$. Note that at all time, $\widehat{w}_{\mathrm{b}}(S) = \widehat{w}_{\mathrm{b}}(S')$. Also, at any time outside (t_1, t_2), $\widehat{w}_{\mathrm{a}}(S) = \widehat{w}_{\mathrm{a}}(S')$. Thus it suffices to compare $\widehat{w}_{\mathrm{a}}(S)$ and $\widehat{w}_{\mathrm{a}}(S')$ at any time during (t_1, t_2), showing that the former is always larger than the latter, meaning $\widehat{F}(S) > \widehat{F}(S')$.

Consider either schedule S or S'. Suppose x units of work in j_i and y units of work in j_j remains at a time t during (t_1, t_2). Then the corresponding $\widehat{w}_{\mathrm{a}}$ at t is $\rho_i x + \rho_j y = \rho_i(x+y) + (\rho_j - \rho_i)y$. Since $x + y$ is always the same for schedules S and S', whichever schedule having the smaller y will have the smaller $\widehat{w}_{\mathrm{a}}$. Note that S decreases x before y while S' decreases y before x. Therefore, y is always smaller in S' than in S during (t_1, t_2), implying $\widehat{w}_{\mathrm{a}}(S) > \widehat{w}_{\mathrm{a}}(S')$. □

Lemma 5.6. *Consider a set of weighted batched jobs J. Among all schedules of J using HDF for job selection, the schedule that incurs the minimum fractional flow time plus energy sets the speed at any time t as $\min\{T, (\frac{\widehat{w}(t)}{\alpha-1})^{1/\alpha}\}$, where $\widehat{w}(t)$ is the total fractional weight at t.*

Proof. Since all jobs are released at time 0, HDF schedules jobs J contiguously from j_n to j_1. Consider a particular time t that the optimal schedule is working on some job j_i.

Let $W_i = \sum_{j=1}^{i-1} w_j$ denote the total weight of all other active jobs. Then $\widehat{w}(t) = W_i + \rho_i q_i$ (recall that q_i is the remaining work of j_i at t). We consider the case to further process j_i for an infinitesimal amount of work x. In this case, we can assume that the speed s is constant, and the time required is $\Delta t = x/s$. Then the contribution to fractional flow plus energy during this period is $s^\alpha \Delta t + ((W_i + \rho_i q_i) + (W_i + \rho_i(q_i - x)))\Delta t/2 = s^\alpha \Delta t + (W_i + \rho_i q_i))\Delta t$ (since $x \to 0$), i.e., $(s^{\alpha-1} + s^{-1}\widehat{w}(t))x$. This is minimized when $s = \left(\frac{\widehat{w}(t)}{\alpha-1}\right)^{1/\alpha}$. The lemma follows. □

Corollary 5.7. *Consider a set of weighted batched jobs. The algorithm* HDF-FW *is optimal for minimizing the fractional flow plus energy of the schedule.*

The following lemma further relates the weighted flow time plus energy of HDF-AJW* and the fractional flow plus energy of HDF-FW.

Lemma 5.8. *Consider a set of batched jobs. Let $\widehat{G}_{\text{hf}}$ be the fractional flow plus energy with* HDF-FW*. Then the weighted flow time plus energy with* HDF-AJW* *is (i) $G_{\text{h}} \leq (2 - 1/\alpha)\widehat{G}_{\text{hf}}$ in the infinite speed model, and (ii) $G_{\text{h}} \leq 2\widehat{G}_{\text{hf}}$ in the bounded speed model.*

Now we prove Lemma 5.8. To ease the discussion, we add a subscript "h" to the notations E, F, $\widehat{F}$, G and $\widehat{G}$ when HDF-AJW* is under concern, and add "hf" when HDF-FW is under concern. We add a parameter j_i if we focus on the contribution to these measures when a job j_i is running, For example, $F_{\text{h}}(j_i)$ denotes the contribution to the weighted flow F in the HDF-AJW* schedule when j_i is running. We aim to show that the required competitiveness is achieved for each job, so summing up all these contributions leads to the desired competitive ratio.

Lemma 5.9. *In the infinite speed model, consider the* HDF-AJW* *and* HDF-FW *schedules of a set of batched jobs J. For each $j_k \in J$, $F_{\text{h}}(j_k) \leq (2 - 1/\alpha)\widehat{F}_{\text{hf}}(j_k)$ and $E_{\text{h}}(j_k) \leq (2 - 1/\alpha)E_{\text{hf}}(j_k)$.*

Proof. Due to the way HDF-AJW* and HDF-FW choose their speeds, $F_{\text{h}}(j_k) = (\alpha - 1)E_{\text{h}}(j_k)$ and $\widehat{F}_{\text{hf}}(j_k) = (\alpha - 1)E_{\text{hf}}(j_k)$ for each job $j_k \in J$. We focus on the flow time only, and prove that $F_{\text{h}}(j_k) \leq (2 - 1/\alpha)\widehat{F}_{\text{hf}}(j_k)$. Then $E_{\text{h}}(j_k) \leq (2 - 1/\alpha)E_{\text{hf}}(j_k)$ follows immediately.

Note that both HDF-AJW* and HDF-FW use HDF policy. Jobs are both scheduled in decreasing order of job density, i.e., from j_n to j_1. Thus we can let $W_k = \sum_{j=1}^{k-1} w_j$ be the total weight of the remaining jobs after running j_k in both schedules.

We first consider the contribution of j_k for HDF-FW. Consider a short period of time where the fractional weight of j_k changes by $\mathrm{d}\widehat{w}_k$. If the time is short enough, the running speed can be considered constant, which is $s = \left(\frac{\widehat{w}_k + W_k}{\alpha - 1}\right)^{1/\alpha}$. The length of time is $(-\mathrm{d}\widehat{w}_k)/(\rho_k s)$. The contribution to $\widehat{F}_{\mathrm{hf}}$ during this period is thus

$$-(\widehat{w}_k + W_k)\,\mathrm{d}\widehat{w}_k \left(\frac{\widehat{w}_k + W_k}{\alpha - 1}\right)^{-1/\alpha} \Big/ \rho_k = \frac{-(\alpha-1)^{1/\alpha}(\widehat{w}_k + W_k)^{1-1/\alpha}\,\mathrm{d}\widehat{w}_k}{\rho_k} \ .$$

During the execution of J, $\widehat{w}_k$ changes from w_k to 0, so the total contribution to $\widehat{F}_{\mathrm{hf}}$ is

$$\widehat{F}_{\mathrm{hf}}(j_k) = \int_{w_k}^{0} \frac{-(\alpha-1)^{1/\alpha}(\widehat{w}_k + W_k)^{1-1/\alpha}}{\rho_k}\,\mathrm{d}\widehat{w}_k = \frac{(\alpha-1)^{1/\alpha}((\widehat{w}_k + W_k)^{2-1/\alpha} - W_k^{2-1/\alpha})}{(2-1/\alpha)\rho_k} \ .$$

Now consider HDF-AJW*, which run the job j_k at speed $s' = \left(\frac{w_k + W_k}{\alpha - 1}\right)^{1/\alpha}$, for a period of length $w_k/(\rho_k s')$. So its contribution to F_{h} is

$$\begin{aligned} F_{\mathrm{h}}(j_k) &= (w_k + W_k)\, w_k \left(\frac{w_k + W_k}{\alpha - 1}\right)^{-1/\alpha} \Big/ \rho_k \\ &= (\alpha-1)^{1/\alpha}((w_k + W_k)^{2-1/\alpha} - (w_k + W_k)^{1-1/\alpha} W_k)/\rho_k \le (2 - 1/\alpha)\widehat{F}_{\mathrm{hf}}(j_k). \end{aligned} \qquad \square$$

Proof of Lemma 5.8. For (i) ($T = \infty$), by Lemma 5.9, it is obvious that $G_{\mathrm{h}}(j_k) \le (2 - 1/\alpha)\widehat{G}_{\mathrm{hf}}(j_k)$. Summing over all jobs leads to the claimed competitive ratio.

For (ii) ($T < \infty$), we first focus on the energy consumption of a job j_k. Note that for the part of j_k that HDF-FW runs at the speed bound T (if exist), same energy E_0 is consumed by HDF-AJW* and HDF-FW. The energy consumption of the other part (running in either HDF-AJW* or HDF-FW) is exactly the same as if the job is run when $T = \infty$. Thus Lemma 5.9 holds for this part, leading to $E_{\mathrm{h}}(j_k) - E_0 \le (2 - 1/\alpha)(E_{\mathrm{hf}}(j_k) - E_0)$. This implies $E_{\mathrm{h}}(j_k) \le (2 - 1/\alpha)E_{\mathrm{hf}}(j_k)$.

We now move to the comparison between $F_{\mathrm{h}}(j_k)$ and $\widehat{F}_{\mathrm{hf}}(j_k)$. Note that for running j_k, HDF-FW always use a speed no faster than HDF-AJW*, so the fractional flow $\widehat{F}_{\mathrm{h}}(j_k)$ incurred by HDF-AJW* when running the job j_k is at most $\widehat{F}_{\mathrm{hf}}(j_k)$. Suppose HDF-AJW* runs the job for t unit of time during which the total fractional weight of the system changes from $w_k + W_k$ to W_k. Then $F_{\mathrm{h}}(j_k) = (w_k + W_k)t$. Since HDF-AJW* runs job j_k at a fixed speed, $\widehat{F}_{\mathrm{h}}(j_k) = ((w_k + W_k) + W_k)t/2 = (w_k/2 + W_k)t$. We thus have $F_{\mathrm{h}}(j_k) \le 2\widehat{F}_{\mathrm{h}}(j_k) \le 2\widehat{F}_{\mathrm{hf}}(j_k)$.

Summing these relations over all jobs leads to the desired competitive ratio. □

We note that for any schedule, the fractional flow is at most the weighted flow time. Thus, by Corollary 5.7 and Lemma 5.8, HDF-AJW* is $(2-1/\alpha)$-competitive for weighted flow time plus energy. Together with Lemma 5.1, we obtain the competitive ratio of WRR-AJW*.

Theorem 5.10. *Consider a set of weighted batched jobs. The algorithm* WRR-AJW* *is (i)* $(2-1/\alpha)^2$*-competitive in the infinite speed model, and (ii)* 4*-competitive in the bounded speed model.*

5.2 Arbitrary Jobs

In this section, we consider jobs with arbitrary release times. Under the infinite speed model, we give an online non-clairvoyant algorithm that is $O(\alpha^3)$-competitive for total flow time plus energy. Our algorithm is defined as follows.

> **Algorithm LAPS**(δ, β)**.** Let $0 < \delta, \beta \leq 1$ be any real. At any time t, the processor speed is $(1+\delta)(n(t))^{1/\alpha}$, where $n(t)$ is the number of active jobs at time t. The processor processes the $\lceil \beta n(t) \rceil$ active jobs with the latest release times (ties are broken by job ids) by splitting the processing speed equally among these jobs.

Our main result is the following.

Theorem 5.11. *When* $\delta = \frac{3}{\alpha}$ *and* $\beta = \frac{1}{2\alpha}$, LAPS$(\delta, \beta)$ *is c-competitive for total flow time plus energy, where* $c = 4\alpha^3(1 + (1 + \frac{3}{\alpha})^\alpha) = O(\alpha^3)$.

The rest of this section is devoted to proving Theorem 5.11. For any time t, let $G_a(t)$ and $G_o(t)$ be the total flow time plus energy incurred up to time t by LAPS(δ, β) and the optimal algorithm OPT, respectively. To show that LAPS(δ, β) is c-competitive, it suffices to give a potential function $\Phi(t)$ such that the following four conditions hold.

- *Boundary condition:* $\Phi = 0$ before any job is released and $\Phi \geq 0$ after all jobs are completed.

- *Job arrival:* When a job is released, Φ does not increase.
- *Job completion:* When a job is completed by LAPS(δ, β) or OPT, Φ does not increase.
- *Running condition:* At any other time, the rate of change of G_a plus that of Φ is no more than c times the rate of change of G_o. That is, $\frac{\mathrm{d}G_a(t)}{\mathrm{d}t} + \frac{\mathrm{d}\Phi(t)}{\mathrm{d}t} \leq c \cdot \frac{\mathrm{d}G_o(t)}{\mathrm{d}t}$ during any period of time without job arrival or completion.

Let $n_a(t)$ and $s_a(t)$ be the number of active jobs and the speed in LAPS(δ, β) at time t, respectively. Define $n_o(t)$ and $s_o(t)$ similarly for that of OPT. Then

$$\frac{\mathrm{d}G_a(t)}{\mathrm{d}t} = \frac{\mathrm{d}F_{LAPS}(t)}{\mathrm{d}t} + \frac{\mathrm{d}E_{LAPS}(t)}{\mathrm{d}t} = n_a(t) + (s_a(t))^\alpha$$

and, similarly, $\frac{\mathrm{d}G_o(t)}{\mathrm{d}t} = n_o(t) + (s_o(t))^\alpha$. We define our potential function as follows.

Potential function $\Phi(t)$. Consider any time t. For any job j, let $q_a(j,t)$ and $q_o(j,t)$ be the remaining work of j at time t in LAPS(δ, β) and OPT, respectively. Let $\{j_1, \ldots, j_{n_a(t)}\}$ be the set of active jobs in LAPS(δ, β), ordered by their release time such that $r(j_1) \leq r(j_2) \leq \cdots \leq r(j_{n_a(t)})$. Then,

$$\Phi(t) = \gamma \sum_{i=1}^{n_a(t)} \left(i^{1-1/\alpha} \cdot \max\{0, q_a(j_i,t) - q_o(j_i,t)\} \right)$$

where $\gamma = \alpha(1 + (1 + \frac{3}{\alpha})^\alpha)$. We call $i^{1-1/\alpha}$ the *coefficient* of j_i.

We first check the boundary, job arrival and job completion conditions. Before any job is released or after all jobs are completed, there is no active job in both LAPS(δ, β) and OPT, so $\Phi = 0$ and the boundary condition holds. When a new job j arrives at time t, $q_a(j,t) - q_o(j,t) = 0$ and the coefficients of all other jobs remain the same, so Φ does not change. If LAPS(δ, β) completes a job j, the term for j in Φ is removed. The coefficient of any other job either stays the same or decreases, so Φ does not increase. If OPT completes a job, Φ does not change.

It remains to check the running condition. In the following, we focus on a certain time t within a period of time without job arrival or completion. We omit the parameter

t from the notations as t refers only to this certain time. For example, we denote $n_a(t)$ and $q_a(j,t)$ as n_a and $q_a(j)$, respectively. For any job j, if LAPS(δ,β) has processed less than OPT on j at time t, i.e., $q_a(j) - q_o(j) > 0$, then we say that j is a *lagging job* at time t. We start by evaluating $\frac{\mathrm{d}\Phi}{\mathrm{d}t}$.

Lemma 5.12. *Assume* $\delta = \frac{3}{\alpha}$ *and* $\beta = \frac{1}{2\alpha}$. *At time* t, *if* LAPS(δ,β) *is processing less than* $(1-\frac{1}{2\alpha})\lceil \beta n_a \rceil$ *lagging jobs, then* $\frac{\mathrm{d}\Phi}{\mathrm{d}t} \le \frac{\gamma}{\alpha}s_o^\alpha + \gamma(1-\frac{1}{\alpha})n_a$. *Else if* LAPS$(\delta,\beta)$ *is processing at least* $(1-\frac{1}{2\alpha})\lceil \beta n_a \rceil$ *lagging jobs, then* $\frac{\mathrm{d}\Phi}{\mathrm{d}t} \le \frac{\gamma}{\alpha}s_o^\alpha - \frac{\gamma}{\alpha}n_a$.

Proof. We consider $\frac{\mathrm{d}\Phi}{\mathrm{d}t}$ as the combined effect due to the processing of LAPS(δ,β) and OPT. Note that for any job j, $q_a(j)$ is decreasing at a rate of either 0 or $-s_a/\lceil \beta n_a \rceil$. Thus the rate of change of Φ due to LAPS(δ,β) is non-positive. Similarly, the rate of change of Φ due to OPT is non-negative.

We first bound the rate of change of Φ due to OPT. The worst case is that OPT is processing the job with the largest coefficient, i.e., $n_a^{1-1/\alpha}$. Thus the rate of change of Φ due to OPT is at most $\gamma n_a^{1-1/\alpha}(-\frac{\mathrm{d}q_o(j_{n_a})}{\mathrm{d}t}) = \gamma n_a^{1-1/\alpha}s_o$. We apply Young's Inequality [52], which is formally stated in Lemma 2.6 in Chapter 2, by setting $f(x) = x^{\alpha-1}$, $f^{-1}(x) = x^{1/(\alpha-1)}$, $g = s_o$ and $h = n_a^{1-1/\alpha}$. Then, we have

$$s_o n_a^{1-1/\alpha} \le \int_0^{s_o} x^{\alpha-1}\,\mathrm{d}x + \int_0^{n_a^{1-1/\alpha}} x^{1/(\alpha-1)}\,\mathrm{d}x = \frac{1}{\alpha}s_o^\alpha + (1-\frac{1}{\alpha})n_a$$

If LAPS(δ,β) is processing less than $(1-\frac{1}{2\alpha})\lceil \beta n_a \rceil$ lagging jobs, we just ignore the effect due to LAPS(δ,β) and take the bound that $\frac{\mathrm{d}\Phi}{\mathrm{d}t} \le \frac{\gamma}{\alpha}s_o^\alpha + \gamma(1-\frac{1}{\alpha})n_a$.

If LAPS(δ,β) is processing at least $(1-\frac{1}{2\alpha})\lceil \beta n_a \rceil$ lagging jobs, let j_i be one of these lagging jobs. We notice that j_i is among the $\lceil \beta n_a \rceil$ active jobs with the latest release times. Thus, the coefficient of j_i is at least $(n_a - \lceil \beta n_a \rceil + 1)^{1-1/\alpha}$. Also, j_i is being processed at a speed of $s_a/\lceil \beta n_a \rceil$, so $q_a(j_i,t)$ is decreasing at this rate. LAPS(δ,β) is processing at least $(1-\frac{1}{2\alpha})\lceil \beta n_a \rceil$ such lagging jobs, so the rate of change of Φ due to

LAPS(δ, β) is more negative than

$$\begin{aligned}
& \gamma\left((1-\frac{1}{2\alpha})\lceil\beta n_a\rceil\right)(n_a - \lceil\beta n_a\rceil + 1)^{1-1/\alpha}\left(\frac{-s_a}{\lceil\beta n_a\rceil}\right) \\
\leq\ & -\gamma(1-\frac{1}{2\alpha})(n_a-\beta n_a)^{1-1/\alpha}(s_a) && (\text{since } -\lceil\beta n_a\rceil + 1 \geq -\beta n_a) \\
\leq\ & -\gamma(1-\frac{1}{2\alpha})(1-\beta)(1+\delta)n_a && (\text{since } s_a = (1+\delta)n_a^{1/\alpha})
\end{aligned}$$

When $\beta = \frac{1}{2\alpha}$ and $\delta = \frac{3}{\alpha}$, simple calculation shows that $(1-\frac{1}{2\alpha})(1-\beta)(1+\delta) \geq 1$ and hence the last term above is at most $-\gamma n_a$. It follows that $\frac{\mathrm{d}\Phi}{\mathrm{d}t} \leq \frac{\gamma}{\alpha}s_o^\alpha + \gamma(1-\frac{1}{\alpha})n_a - \gamma n_a = \frac{\gamma}{\alpha}s_o^\alpha - \frac{\gamma}{\alpha}n_a$. □

We are now ready to show the following lemma about the running condition.

Lemma 5.13. *Assume $\delta = \frac{3}{\alpha}$ and $\beta = \frac{1}{2\alpha}$. At time t, $\frac{\mathrm{d}G_a}{\mathrm{d}t} + \frac{\mathrm{d}\Phi}{\mathrm{d}t} \leq c \cdot \frac{\mathrm{d}G_o}{\mathrm{d}t}$, where $c = 4\alpha^3(1+(1+\frac{3}{\alpha})^\alpha)$.*

Proof. We consider two cases depending on the number of lagging jobs that LAPS(δ, β) is processing at time t. If LAPS(δ, β) is processing at least $(1-\frac{1}{\alpha})\lceil\beta n_a\rceil$ lagging jobs, then

$$\begin{aligned}
\tfrac{\mathrm{d}G_a}{\mathrm{d}t} + \tfrac{\mathrm{d}\Phi}{\mathrm{d}t} &= n_a + s_a^\alpha + \tfrac{\mathrm{d}\Phi}{\mathrm{d}t} \\
&\leq n_a + (1+\delta)^\alpha n_a + \frac{\gamma}{\alpha}s_o^\alpha - \frac{\gamma}{\alpha}n_a && (\text{by Lemma 5.12}) \\
&= (1+(1+\delta)^\alpha - \frac{\gamma}{\alpha})n_a + \frac{\gamma}{\alpha}s_o^\alpha
\end{aligned}$$

Since $\delta = \frac{3}{\alpha}$ and $\gamma = \alpha(1+(1+\frac{3}{\alpha})^\alpha)$, the coefficient of n_a becomes zero and $\frac{\mathrm{d}G_a}{\mathrm{d}t} + \frac{\mathrm{d}\Phi}{\mathrm{d}t} \leq \frac{\gamma}{\alpha}s_o$. Note that $\frac{\gamma}{\alpha} = (1+(1+\frac{3}{\alpha})^\alpha) \leq c$ and $\frac{\mathrm{d}G_o}{\mathrm{d}t} = n_o + s_o^\alpha$, so we have $\frac{\mathrm{d}G_a}{\mathrm{d}t} + \frac{\mathrm{d}\Phi}{\mathrm{d}t} \leq c \cdot \frac{\mathrm{d}G_o}{\mathrm{d}t}$.

If LAPS(δ, β) is processing less than $(1-\frac{1}{2\alpha})\lceil\beta n_a\rceil$ lagging jobs, the number of jobs remaining in OPT is $n_o \geq \lceil\beta n_a\rceil - (1-\frac{1}{2\alpha})\lceil\beta n_a\rceil = \frac{1}{2\alpha}\lceil\beta n_a\rceil \geq \frac{1}{2\alpha}\beta n_a = \frac{1}{4\alpha^2}n_a$. Therefore,

$$\begin{aligned}
\tfrac{\mathrm{d}G_a}{\mathrm{d}t} + \tfrac{\mathrm{d}\Phi}{\mathrm{d}t} &= n_a + s_a^\alpha + \tfrac{\mathrm{d}\Phi}{\mathrm{d}t} \\
&\leq n_a + (1+\delta)^\alpha n_a + \frac{\gamma}{\alpha}s_o^\alpha + \gamma(1-\frac{1}{\alpha})n_a && (\text{by Lemma 5.12}) \\
&= (1+(1+\delta)^\alpha + \gamma(1-\frac{1}{\alpha}))n_a + \frac{\gamma}{\alpha}s_o^\alpha \\
&\leq 4\alpha^2(1+(1+\delta)^\alpha + \gamma(1-\frac{1}{\alpha}))n_o + \frac{\gamma}{\alpha}s_o^\alpha
\end{aligned}$$

Since $\delta = \frac{3}{\alpha}$ and $\gamma = \alpha(1+(1+\frac{3}{\alpha})^{\alpha})$, the coefficient of n_o becomes $4\alpha^3(1+(1+\frac{3}{\alpha})^{\alpha}) = c$. The coefficient of s_o^{α} is $(1+(1+\frac{3}{\alpha})^{\alpha}) \leq c$. Since $\frac{\mathrm{d}G_o(t)}{\mathrm{d}t} = n_o + s_o^{\alpha}$, we obtain $\frac{\mathrm{d}G_a(t)}{\mathrm{d}t} + \frac{\mathrm{d}\Phi}{\mathrm{d}t} \leq c \cdot \frac{\mathrm{d}G_o(t)}{\mathrm{d}t}$. Note that this case is the bottleneck leading to the current competitive ratio. □

Combining Lemma 5.16 with the discussion on the boundary, job arrival and job completion conditions, Theorem 5.11 follows.

5.3 Sleep States

In this section, we consider non-clairvoyant scheduling in the model that allows both sleep management and speed scaling, which is introduced in Section 3.1 of Chapter 3. We consider a processor with the awake state and $m \geq 1$ levels of sleep states as in Section 3.1. Recall that in the awake state, the processor can run at any speed s, demanding the static power σ and the dynamic power s^{α}.

Now, job scheduling requires two components: a sleep management algorithm to determine when to sleep or work, and a speed scaling algorithm to determine which job and at what speed to run. The sleep management algorithm IdleLonger introduced in Chapter 3 can work for non-clairvoyant scheduling as its decision does not depend on the job size. When there is no sleep state and the power function is in the form of s^{α}, Section 5.2 has given a non-clairvoyant speed scaling algorithm LAPS that is $O(\alpha^3)$-competitive for flow plus energy. We adapt LAPS to the sleep setting in a similar way as AJC was adapted in Section 3.3; we call the resulting algorithm SLS (Sleep-aware Latest arrival processor Sharing). We also show that IdleLonger(SLS) (i.e., IdleLonger coupled with SLS) is $O(\alpha^3)$-competitive for total flow time plus energy in the infinite speed model.

> **Algorithm SLS.** Consider any time t. Let $n(t)$ be the number of active jobs at time t. Then the processor speed is $(1+\frac{3}{\alpha})(n(t)+\sigma)^{1/\alpha}$, and the processor processes the $\lceil(\frac{1}{2\alpha})n(t)\rceil$ active jobs with the latest release times (ties are broken by job ids) by splitting the processing speed equally among these jobs.

To analyze the performance of IdleLonger(SLS), we divide the total cost G (i.e., flow plus energy) into the *working cost* (incurred while working on jobs) and the *inactive cost*

(incurred at other times) as in Section 3.1. For the optimal offline algorithm OPT, we divide its total cost G^* into two parts: W^* is the total wake-up energy, and $C^* = G^* - W^*$ (i.e., the total flow plus the working and idling energy). In Section 3.2, we have shown that the inactive cost of IdleLonger is at most $3C^* + 6W^*$ (Corollary 3.3). It remains to upper bound the working cost of IdleLonger(SLS) in terms of OPT's total cost.

Below, we give an analysis of the working cost of SLS when it is coupled with any non-clairvoyant sleep management algorithm Slp (including IdleLonger). Let G_{a} and F_{i} denote the working cost and inactive flow of Slp(SLS), respectively. Similar to Section 3.3, we modify the potential analysis of LAPS in Section 5.2 to take F_{i} into consideration. The result is stated in Theorem 5.14 below, which, together with the results on inactive cost of IdleLonger, implies that IdleLonger(SLS) is $((4\alpha^3+\alpha)\gamma+3)$-competitive for flow plus energy, where $\gamma = (1+(1+\frac{3}{\alpha})^{\alpha}) \le 1+e^3$.

Theorem 5.14. *With respect to* Slp(SLS)*,* $G_{\mathrm{a}} \le (4\alpha^3+1)\gamma C^* + (\alpha-1)\gamma F_{\mathrm{i}}$.

Corollary 5.15. *In the setting of single sleep state or multiple sleep states, the total cost of* IdleLonger(SLS) *is at most* $((4\alpha^3+\alpha)\gamma+3)$ *times of the total cost of OPT.*

Proof of Corollary 5.15. Consider the coupling of IdleLonger and SLS. As proven in Section 3.2 of Chapter 3, $F_{\mathrm{i}} \le C^* + 2W^*$ and the inactive cost is at most $3C^* + 6W^*$. By Theorem 5.14, the working cost is at most $(4\alpha^3+1)\gamma C^* + (\alpha-1)\gamma F_{\mathrm{i}}$. It follows that the total cost of IdleLonger(SLS), comprised of the inactive cost and the working cost, is at most $((4\alpha^3+\alpha)\gamma+3)C^* + (2(\alpha-1)\gamma+6)W^*$. Note that $\gamma = (1+(1+\frac{3}{\alpha})^{\alpha}) \ge 2$ and hence $((4\alpha^3+\alpha)\gamma+3) - (2(\alpha-1)\gamma+6) = (4\alpha^3-\alpha)\gamma + (2\gamma-3) \ge 0$. Therefore, the total cost of IdleLonger(SLS) is at most $((4\alpha^3+\alpha)\gamma+3)$ times OPT's total cost. □

The rest of this section is devoted to Theorem 5.14. We adapt the potential function $\Phi(t)$ in Section 5.2 to include σ. At any time t, let $n_{\mathrm{a}}(t)$ and $q_{\mathrm{a}}(j,t)$ denote respectively the number of active jobs and the remaining work of job j in Slp(SLS). And $n_{\mathrm{o}}(t)$ and $q_{\mathrm{o}}(J,t)$ are defined similarly for OPT. We denote the set of active jobs in Slp(SLS) by $\{j_1, \ldots, j_{n_{\mathrm{a}}(t)}\}$, listed in non-decreasing order of release times. Define

$$\Phi(t) = \alpha\gamma \textstyle\sum_{i=1}^{n_{\mathrm{a}}(t)} \left((i+\sigma)^{1-1/\alpha} \cdot \max\{0, q_{\mathrm{a}}(j_i,t) - q_{\mathrm{o}}(j_i,t)\} \right)$$

We call the term $(i+\sigma)^{1-1/\alpha}$ the *coefficient* of j_i. Intuitively, $\Phi(t)$ measures the amount of work that Slp(SLS) is lagging behind OPT.

Ideally, Theorem 5.14 can be proven by showing that $\frac{\mathrm{d}G_\mathrm{a}}{\mathrm{d}t} + \frac{\mathrm{d}\Phi}{\mathrm{d}t} = O(\frac{\mathrm{d}C^*}{\mathrm{d}t} + \frac{\mathrm{d}F_\mathrm{i}}{\mathrm{d}t})$ at all times when no discrete events occur (i.e., no job arrival, completion or speed change). Yet this is not always correct. Note that Φ is always at least zero. When Slp(SLS) is working and OPT is sleeping, $\frac{\mathrm{d}G_\mathrm{a}}{\mathrm{d}t} \geq \sigma$, and $\frac{\mathrm{d}C^*}{\mathrm{d}t}$ and $\frac{\mathrm{d}F_\mathrm{i}}{\mathrm{d}t}$ may be much smaller than σ. If Φ reaches the minimum value zero, then $\frac{\mathrm{d}\Phi}{\mathrm{d}t} \geq 0$ and we cannot bound $\frac{\mathrm{d}G_\mathrm{a}}{\mathrm{d}t} + \frac{\mathrm{d}\Phi}{\mathrm{d}t}$ by $O(\frac{\mathrm{d}C^*}{\mathrm{d}t} + \frac{\mathrm{d}F_\mathrm{i}}{\mathrm{d}t})$.

We extend the potential with yet another new notion E_ws, the *total working static energy*, which is the static power σ times the total length of working intervals. When Slp(SLS) is working, $\frac{\mathrm{d}E_\mathrm{ws}}{\mathrm{d}t} = \sigma$. This allows us to show the following lemma.

Lemma 5.16. *At any time when no discrete events occur,* $\frac{\mathrm{d}G_\mathrm{w}}{\mathrm{d}t} + \frac{\mathrm{d}\Phi}{\mathrm{d}t} \leq 4\alpha^3\gamma\frac{\mathrm{d}C^*}{\mathrm{d}t} + (\alpha - 1)\gamma\frac{\mathrm{d}F_\mathrm{i}}{\mathrm{d}t} + \gamma\frac{\mathrm{d}E_\mathrm{ws}}{\mathrm{d}t}$.

We now prove Lemma 5.16. At any time t, if Slp(SLS) is working, let s_a be the current speed of SLS. We define s_o similarly for OPT. Furthermore, for any job j, if $q_\mathrm{a}(j,t) - q_\mathrm{o}(j,t) > 0$, we say that j is a *lagging job* at time t. Below we drop t from the notations since t refers the current time. We first analyze $\frac{\mathrm{d}\Phi}{\mathrm{d}t}$ as if Φ is modified in two steps: (i) the execution of Slp(SLS), and (ii) the execution of OPT. We denote these changes as $\frac{\mathrm{d}\Phi_1}{\mathrm{d}t}$ and $\frac{\mathrm{d}\Phi_2}{\mathrm{d}t}$, respectively, i.e., $\frac{\mathrm{d}\Phi}{\mathrm{d}t} = \frac{\mathrm{d}\Phi_1}{\mathrm{d}t} + \frac{\mathrm{d}\Phi_2}{\mathrm{d}t}$. Using the same arguments in Lemma 5.12, we can prove the following claim.

Claim 5.17. (i) *Consider* $s_\mathrm{a} > 0$. *If* Slp(SLS) *is processing less than* $(1 - \frac{1}{2\alpha})\left\lceil(\frac{1}{2\alpha})n_\mathrm{a}\right\rceil$ *lagging jobs, then* $\frac{\mathrm{d}\Phi_1}{\mathrm{d}t} \leq 0$. *Else if* Slp(SLS) *is processing at least* $(1-\frac{1}{2\alpha})\left\lceil(\frac{1}{2\alpha})n_\mathrm{a}\right\rceil$ *lagging jobs, then* $\frac{\mathrm{d}\Phi_1}{\mathrm{d}t} \leq -\alpha\gamma(n_\mathrm{a}+\sigma)$; (ii) *Consider* $s_\mathrm{o} > 0$. *Then* $\frac{\mathrm{d}\Phi_2}{\mathrm{d}t} \leq \gamma s_\mathrm{o}^\alpha + (\alpha-1)\gamma(n_\mathrm{a}+\sigma)$.

We are now ready to show Lemma 5.16 by a case analysis of whether Slp(SLS) and OPT is working. When $s_\mathrm{a} > 0$, $\frac{\mathrm{d}G_\mathrm{a}}{\mathrm{d}t} = n_\mathrm{a} + s_\mathrm{a}^\alpha + \sigma = n_\mathrm{a} + (1+\frac{3}{\alpha})^\alpha(n_\mathrm{a}+\sigma) + \sigma = (1+(1+\frac{3}{\alpha})^\alpha)(n_\mathrm{a}+\sigma) = \gamma(n_\mathrm{a}+\sigma)$. Also, if Slp(SLS) is processing less than $(1-\frac{1}{2\alpha})\left\lceil(\frac{1}{2\alpha})n_\mathrm{a}\right\rceil$ lagging jobs, the number of jobs remaining in OPT is $n_\mathrm{o} \geq \left\lceil(\frac{1}{2\alpha})n_\mathrm{a}\right\rceil - (1-\frac{1}{2\alpha})\left\lceil(\frac{1}{2\alpha})n_\mathrm{a}\right\rceil = \frac{1}{2\alpha}\left\lceil(\frac{1}{2\alpha})n_\mathrm{a}\right\rceil \geq \frac{1}{2\alpha}((\frac{1}{2\alpha})n_\mathrm{a}) = \frac{1}{4\alpha^2}(n_\mathrm{a})$, i.e., $n_\mathrm{a} \leq 4\alpha^2 n_\mathrm{o}$.

Case 1: $s_\mathbf{a} > 0, s_\mathbf{o} > 0$**.** In this case, $\frac{\mathrm{d}G_\mathrm{a}}{\mathrm{d}t} = \gamma(n_\mathrm{a}+\sigma)$, $\frac{\mathrm{d}C^*}{\mathrm{d}t} = s_\mathrm{o}^\alpha + \sigma + n_\mathrm{o}$, $\frac{\mathrm{d}F_\mathrm{i}}{\mathrm{d}t} = 0$, and $\frac{\mathrm{d}E_\mathrm{ws}}{\mathrm{d}t} = \sigma$. If Slp(SLS) is processing less than $(1 - \frac{1}{2\alpha})\left\lceil(\frac{1}{2\alpha})n_\mathrm{a}\right\rceil$ lagging jobs at time t, then $n_\mathrm{a} \leq 4\alpha^2 n_\mathrm{o}$. By Claim 5.17 (i) and (ii), we have

$$\begin{aligned}\frac{\mathrm{d}G_\mathrm{a}}{\mathrm{d}t} + \frac{\mathrm{d}\Phi}{\mathrm{d}t} &\leq \gamma(n_\mathrm{a}+\sigma) + \gamma s_\mathrm{o}^\alpha + (\alpha-1)\gamma(n_\mathrm{a}+\sigma) \\ &\leq \alpha\gamma(n_\mathrm{a}+\sigma) + \gamma s_\mathrm{o}^\alpha \leq 4\alpha^3\gamma(n_\mathrm{o} + s_\mathrm{o}^\alpha + \sigma) \leq 4\alpha^3\gamma\frac{\mathrm{d}C^*}{\mathrm{d}t}\ .\end{aligned}$$

If Slp(SLS) is processing at least $(1-\frac{1}{2\alpha})\left\lceil(\frac{1}{2\alpha})n_{\mathrm{a}}\right\rceil$ lagging jobs at time t, then by Claim 5.17 (i) and (ii), we have

$$\tfrac{\mathrm{d}G_{\mathrm{a}}}{\mathrm{d}t}+\tfrac{\mathrm{d}\Phi}{\mathrm{d}t}\leq\gamma(n_{\mathrm{a}}+\sigma)-\alpha\gamma(n_{\mathrm{a}}+\sigma)+\gamma s_{\mathrm{o}}^{\alpha}+(\alpha-1)\gamma(n_{\mathrm{a}}+\sigma)\leq\gamma s_{\mathrm{o}}^{\alpha}\leq\gamma\tfrac{\mathrm{d}C^{*}}{\mathrm{d}t}\ .$$

Case 2: $s_{\mathbf{a}}>0, s_{\mathbf{o}}=0$**.** In this case, $\frac{\mathrm{d}G_{\mathrm{a}}}{\mathrm{d}t}=\gamma(n_{\mathrm{a}}+\sigma)$, $\frac{\mathrm{d}C^{*}}{\mathrm{d}t}=n_{\mathrm{o}}$, $\frac{\mathrm{d}F_{\mathrm{i}}}{\mathrm{d}t}=0$, and $\frac{\mathrm{d}E_{\mathrm{ws}}}{\mathrm{d}t}=\sigma$. Since $s_{\mathrm{o}}=0$, it is clear that $\frac{\mathrm{d}\Phi_{2}}{\mathrm{d}t}=0$. If Slp(SLS) is processing less than $(1-\frac{1}{2\alpha})\left\lceil(\frac{1}{2\alpha})n_{\mathrm{a}}\right\rceil$ lagging jobs at time t, then $n_{\mathrm{a}}\leq 4\alpha^{2}n_{\mathrm{o}}$. By Claim 5.17 (i), we have

$$\tfrac{\mathrm{d}G_{\mathrm{a}}}{\mathrm{d}t}+\tfrac{\mathrm{d}\Phi}{\mathrm{d}t}\leq\gamma(n_{\mathrm{a}}+\sigma)\leq 4\alpha^{2}\gamma n_{\mathrm{o}}+\gamma\sigma\leq 4\alpha^{3}\gamma\tfrac{\mathrm{d}C^{*}}{\mathrm{d}t}+\gamma\tfrac{\mathrm{d}E_{\mathrm{ws}}}{\mathrm{d}t}\ .$$

If Slp(SLS) is processing at least $(1-\frac{1}{2\alpha})\left\lceil(\frac{1}{2\alpha})n_{\mathrm{a}}\right\rceil$ lagging jobs at time t, then by Claim 5.17 (i), we have

$$\tfrac{\mathrm{d}G_{\mathrm{a}}}{\mathrm{d}t}+\tfrac{\mathrm{d}\Phi}{\mathrm{d}t}\leq\gamma(n_{\mathrm{a}}+\sigma)-\alpha\gamma(n_{\mathrm{a}}+\sigma)\leq 0\leq 4\alpha^{3}\gamma\tfrac{\mathrm{d}C^{*}}{\mathrm{d}t}\ .$$

Case 3: $s_{\mathbf{a}}=0, s_{\mathbf{o}}>0$**.** In this case, $\frac{\mathrm{d}G_{\mathrm{a}}}{\mathrm{d}t}=0$, $\frac{\mathrm{d}C^{*}}{\mathrm{d}t}=s_{\mathrm{o}}^{\alpha}+\sigma+n_{\mathrm{o}}$, $\frac{\mathrm{d}F_{\mathrm{i}}}{\mathrm{d}t}=n_{\mathrm{a}}$, and $\frac{\mathrm{d}E_{\mathrm{ws}}}{\mathrm{d}t}=0$. Since $s_{\mathrm{o}}=0$, it is clear that $\frac{\mathrm{d}\Phi_{1}}{\mathrm{d}t}=0$. Thus, by Claim 5.17 (ii), we have

$$\tfrac{\mathrm{d}G_{\mathrm{a}}}{\mathrm{d}t}+\tfrac{\mathrm{d}\Phi}{\mathrm{d}t}\leq\gamma s_{\mathrm{o}}^{\alpha}+(\alpha-1)\gamma(n_{\mathrm{a}}+\sigma)\leq(\alpha-1)\gamma(s_{\mathrm{o}}^{\alpha}+\sigma)+(\alpha-1)\gamma n_{\mathrm{a}}\leq 4\alpha^{3}\gamma\tfrac{\mathrm{d}C^{*}}{\mathrm{d}t}+(\alpha-1)\gamma\tfrac{\mathrm{d}F_{\mathrm{i}}}{\mathrm{d}t}\ .$$

Case 4: $s_{\mathbf{a}}=0, s_{\mathbf{o}}=0$**.** In this case, $\frac{\mathrm{d}G_{\mathrm{a}}}{\mathrm{d}t}=0$, $\frac{\mathrm{d}C^{*}}{\mathrm{d}t}=n_{\mathrm{o}}$, $\frac{\mathrm{d}F_{\mathrm{i}}}{\mathrm{d}t}=n_{\mathrm{a}}$, and $\frac{\mathrm{d}E_{\mathrm{ws}}}{\mathrm{d}t}=0$. Since $s_{\mathrm{a}}=0$ and $s_{\mathrm{o}}=0$, it is clear that $\frac{\mathrm{d}\Phi}{\mathrm{d}t}=0$. Therefore, $\frac{\mathrm{d}G_{\mathrm{a}}}{\mathrm{d}t}+\frac{\mathrm{d}\Phi}{\mathrm{d}t}=0\leq 4\alpha^{3}\gamma\frac{\mathrm{d}C^{*}}{\mathrm{d}t}$.

To sum up, we have $\frac{\mathrm{d}G_{\mathrm{w}}}{\mathrm{d}t}+\frac{\mathrm{d}\Phi}{\mathrm{d}t}\leq 4\alpha^{3}\gamma\frac{\mathrm{d}C^{*}}{\mathrm{d}t}+(\alpha-1)\gamma\frac{\mathrm{d}F_{\mathrm{i}}}{\mathrm{d}t}+\gamma\frac{\mathrm{d}E_{\mathrm{ws}}}{\mathrm{d}t}$ for all the four cases, completing the proof of Lemma 5.16. Lemma 5.16 immediately implies that $G_{\mathrm{a}}\leq 4\alpha^{3}\gamma C^{*}+(\alpha-1)\gamma F_{\mathrm{i}}+\gamma E_{\mathrm{ws}}$. We further observe $E_{\mathrm{ws}}\leq C^{*}$. Then Theorem 5.14 follows.

Lemma 5.18. *With respect to* Slp(SLS)*,* $E_{\mathrm{ws}}\leq G^{*}$.

Proof. Let x be the total size of all jobs. When Slp(SLS) is working, its speed is at least $(1+\frac{3}{\alpha})(\sigma+1)^{1/\alpha}\geq\sigma^{1/\alpha}$ and thus $E_{\mathrm{ws}}\leq\sigma\cdot(x/\sigma^{1/\alpha})=x\sigma^{1-1/\alpha}$. Recall that running a job at the critical speed $s_{\mathrm{crit}}=(\sigma/(\alpha-1))^{1/\alpha}$ minimizes the energy usage of the job (see Chapter 4). Therefore, the energy usage of any schedule and hence C^{*} is at least $(x/s_{\mathrm{crit}})\cdot(s_{\mathrm{crit}}^{\alpha}+\sigma)=(\alpha/(\alpha-1)^{1-1/\alpha})\cdot(x\sigma^{1-1/\alpha})\geq(\alpha/(\alpha-1)^{1-1/\alpha})E_{\mathrm{ws}}$. For any $\alpha>1$, $(\alpha/(\alpha-1)^{1-1/\alpha})\geq 1$, and hence $C^{*}\geq E_{\mathrm{ws}}$. □

Chapter 6

Space-efficient Data Stream Algorithms

In this chapter, we study space-efficient data stream algorithms. To answer a statistics on a data stream, we need to handle a huge volume of items from the stream, usually in the order of gigabytes a second, and as the stream passes we have only a few nanoseconds to react to each item. Furthermore, we usually have stringent memory, e.g., the routers used in network monitoring are relatively cheap and have small main memory. Thus, any algorithm for answering the statistics on a data stream must have *extremely fast update and query time*, and use *small memory space*. This chapter focuses on the count-based sliding window model. Precisely, let W be the size of the sliding window. The count-based sliding window covers the most recent W data items in the data stream.

We first consider the following new problem, called *Significant One Counting*, which is more flexible in space-accuracy tradeoff than the basic counting problem (see Section 1.2 for its formal definition).

> **Significant One Counting.** For a stream of *bits*, let c be the number of 1-bits in the current sliding window (of size W). Given a *threshold* $0 < \theta < 1$, and a *relative error bound* $0 < \varepsilon < 1$, the problem is to return an estimate $\hat{c}$ of the number c of 1-bits such that if $c \geq \theta W$, we have $|\hat{c} - c| \leq \varepsilon c$.

We prove a lower bound on the memory of any algorithm for the significant one

counting problem, showing that any algorithm for the problem must have memory at least $\Omega(\frac{1}{\varepsilon}\log^2(\frac{1}{\theta}) + \log(\varepsilon\theta W))$ bits. Then, we give an algorithm that has constant update and query time, and uses memory matching the lower bound, i.e., the algorithm has the optimal time and space complexity. This algorithm returns an estimate $\hat{c}$ of the number c of 1-bits in the sliding window such that if $c \geq \theta W$, then $c \leq \hat{c} \leq c + \varepsilon c$, and if $c < \theta W$, then $c \leq \hat{c} \leq c + \varepsilon\theta W$. Note that the algorithm becomes an optimal algorithm for basic counting when we set θ to $1/W$. On the other hand, for any fixed θ, its memory usage is only $O(\frac{1}{\varepsilon} + \log(\varepsilon W))$ bits, which is much smaller than $\Theta(\frac{1}{\varepsilon}\log^2(\varepsilon W))$ bits.

We also study finding approximate frequent items over a count-based sliding window. More precisely, we consider a stream of data items from a set U. Let $0 < \varepsilon < 1$ be the user-specified error bound. For any item $j \in U$, let c_j be the number of j in the most recent W data items in the stream. The problem is defined as follows.

> *ε-Approximate Frequent Items.* Given any user-specified threshold $0 < \phi < 1$, return a set $F \subseteq U$ which includes all items j with $c_j \geq \phi c$ and possibly some items j' with $c_{j'} \geq \phi c - \varepsilon c$.

We give a deterministic algorithm for the ε-approximate frequent items problem over sliding window. This algorithm supports $O(\frac{1}{\varepsilon})$ update and query time, and uses $O(\frac{1}{\varepsilon})$ words of space. This substantially improves the previous result by Arasu and Manku [5], where their algorithm takes $O(\frac{1}{\varepsilon}\log\frac{1}{\varepsilon})$ query and update time and uses $O(\frac{1}{\varepsilon}\log^2(\frac{1}{\varepsilon}))$ words of space. Note that our memory usage is essentially optimal; the number of frequent items can be $\Omega(\frac{1}{\varepsilon})$ in the worst case and thus any algorithm must use $\Omega(\frac{1}{\varepsilon})$ words of memory. More importantly, this algorithm is much simpler than Arasu and Manku's result; our algorithm has about twenty lines of codes. It uses only $O(\frac{1}{\varepsilon})$ simple variables and $O(\frac{1}{\varepsilon})$ queues, and the total length of these queues is $O(\frac{1}{\varepsilon})$. To update these data structures when the window slides, we need only to increment/decrement some of the variables for most cases, and we seldom need to insert or delete entries in the queues. By adapting a technique in [5], we also extend our algorithm to identify frequent items in a sliding window whose size can be changed by the user.

Remark. Besides frequent items, our algorithm can also find top items. We say that an item is the *top 1st* item if it is the most frequent item, and in general, an item is the *top ith* item if it is the ith most frequent item. Charikar et al. [25] introduced the

following Approximate Top-k problem: Let k and δ be two user-specified parameters. For any integer $i > 0$, let F_i be the frequency of the top ith item. The problem asks for a set S of k items such that every item $s \in S$ has frequency at least $(1-\delta)F_k$. Note that this problem is proposed for whole data stream originally, but it is straightforward to generalize the problem for sliding windows. We remark that our algorithm for frequent items can be used to solve the approximate top-k problem using $O(\frac{k}{\delta})$ words of space.

Section 6.1 gives our results on significant one counting. Section 6.2 gives our results on approximate frequent items problem over sliding window.

6.1 Significant One Counting

This section considers the significant one counting problem. First, Section 6.1.1 gives a lower bound of $\Omega(\frac{1}{\varepsilon}\log^2\frac{1}{\theta}+\log\varepsilon\theta W)$ bits of memory for any (deterministic) data structure for the significant one counting problem. Then, we introduce our data structure in a sequence of steps, as follows. In Section 6.1.2, we introduce an one-level data structure. In Section 6.1.3, we improve the memory usage by using a multi-level data structure. Finally, Section 6.1.4 presents our data structure, that is optimal in space, query time and update time.

6.1.1 Lower bound

In this section, we derive two lower bounds on the size of any data structure for the significant one counting problem; one is $\Omega(\log\varepsilon\theta W)$ and the other is $\Omega(\frac{1}{\varepsilon}\log^2\frac{1}{\theta})$. Combining these two results, we conclude a lower bound of $\Omega(\frac{1}{\varepsilon}\log^2\frac{1}{\theta}+\log\varepsilon\theta W)$. Note that our results have assumed some bounds on the values of the threshold θ and the relative error bound ϵ.

Lemma 6.1. *Suppose that $\theta \le \frac{1}{2}$ and $\varepsilon \le \frac{1}{5}$. Then, any data structure for significant one counting with threshold θ and relative error bound ε has size $\Omega(\log\varepsilon\theta W)$ bits.*

Proof. Suppose that the data stream has a sequence of W 0-bits followed by a sequence of $\varepsilon\theta W$ 1-bits, where the latter sequence of 1-bits is denoted by P. For the sake of contradiction, assume that the data structure uses $o(\log\varepsilon\theta W)$ bits of memory. Thus,

there are two 1-bits in sequence P, say the x-th 1-bit and the $(x+y)$-th 1-bit where $x, y \in [1, \varepsilon\theta W - 1]$, such that when they are received, the memory states are the same, which is denoted by M.

When the x-th 1-bit is received, since $x < \varepsilon\theta W < W$, the actual number of 1-bits in the sliding window, denoted by c_1, is $c_1 = x$. Now, we modify the bit stream such that following this x-th 1-bit is a sequence of $y\delta$ 1-bits, where $\delta = \lceil (4\varepsilon\theta W)/y \rceil$. Note that after the arrivals of each subsequence of y 1-bits, the memory state is also equal to M. Thus, after the arrivals of all the $y\delta$ 1-bits, the memory state is M. Since $\theta \leq \frac{1}{2}$ and $\varepsilon \leq \frac{1}{5}$, it follows that $x + y\delta \leq x + (4\varepsilon\theta W + y) < 6\varepsilon\theta W < W$, which implies the actual number of 1-bits in the sliding window, denoted by c_2, is $c_2 = x + y\delta$.

Consider the following two cases: *(Case 1)* the xth 1-bit is received, and *(Case 2)* the $y\delta$ 1-bits following the x-th 1-bit are received. Note that in both cases, the memory state is M. Thus, when using the data structure to estimate the number of 1-bits, both cases return the same estimate. That is, the data structure will give an estimate with absolute error at least $|c_1 - c_2|/2 \geq (4\varepsilon\theta W)/2 = 2\varepsilon\theta W$ for one of the streams in the two cases. Consequently, the relative error is at least

$$\begin{aligned} \frac{2\varepsilon\theta W}{\max\{c_1, c_2\}} &= \frac{2\varepsilon\theta W}{x + y\lceil \frac{4\varepsilon\theta W}{y} \rceil} > \frac{2\varepsilon\theta W}{\theta W + 4\varepsilon\theta W + y} \\ &> \frac{2\varepsilon\theta W}{\theta W + 5\varepsilon\theta W} \geq \frac{2\varepsilon\theta W}{2\theta W} = \varepsilon \ , \end{aligned}$$

which contradicts that the relative error of the estimate is no more than ε. Therefore, $\Omega(\log \varepsilon\theta W)$ bits of memory is required. □

The proof of the following lemma adapts the proof techniques given in [35].

Lemma 6.2. *Suppose that $\theta \leq \frac{1}{16}$ and $\varepsilon \geq \frac{1}{4\theta W}$. Any data structure for significant one counting with threshold θ and relative error bound ε uses $\Omega(\frac{1}{\varepsilon}\log^2 \frac{1}{\theta})$ bits of memory.*

Proof. We first describe the data stream under consideration. Let $\varepsilon = \frac{1}{k}$ for some constant k. Since $\varepsilon \geq \frac{1}{4\theta W}$, we have $k \leq 4\theta W$. We list the bits in the stream from left to right, where the rightmost bit is the most recent bit. A window of size W is divided from right to left into blocks of size B, $2B$, $4B$, $8B$, ..., $2^j B$, where $B \geq \frac{k}{4}$ is some constant (which will be later set to $\frac{k}{16}\sqrt{\frac{1}{\theta}}$) and $j = \lfloor \log \frac{W}{B} \rfloor - 1$. To simplify the discussion, we consider

<table>
<tr><td>block of 2^jB bits</td><td>block of $2^{j-1}B$ bits</td><td>$\cdots$</td><td colspan="3">block of 2^iB bits</td><td>$\cdots$</td><td>block of B bits</td></tr>
<tr><td colspan="3">$\cdots$</td><td>sub-blocks of 2^i bits</td><td>$\cdots$</td><td>sub-blocks of 2^i bits</td><td colspan="2">$\cdots$</td></tr>
</table>

Figure 6.1: The bits in the sliding window are listed from left to right, where the rightmost bit is the most recent bit, and divided into different blocks. A block of size 2^iB is further divided into B contiguous sub-blocks of size 2^i bits.

the case that k and B are integers; the proof can be adapted to handle other cases. For a block of size 2^iB, where $0 \leq i \leq j$, it is further divided into B contiguous sub-blocks of size 2^i bits, as shown in Figure 6.1.

Among the B sub-blocks, $k/4$ of them are all 1-bits, while the rest are all 0-bits. Let $d = \lceil \log(\frac{4\theta W}{k} + 1) \rceil$. Note that $d = \lceil \log(\frac{4\theta W + k}{k}) \rceil \leq \lceil \log(\frac{8\theta W}{k}) \rceil$. For a block of size $s > 2^dB$, there are $\binom{B}{k/4}$ possible arrangements for the $k/4$ sub-blocks of 1-bits in the B sub-blocks. For a block of size $s \leq 2^dB$, $k/4$ sub-blocks of the B sub-blocks are fixed to be the sub-blocks of 1-bits. Let b denote the number of blocks of size larger than 2^dB. Therefore, there are $\binom{B}{k/4}^b$ possible arrangements for the sub-blocks of 1-bits in those blocks of size $s > 2^dB$. Also, we have $b \geq j - d \geq (\lfloor \log \frac{n}{B} \rfloor - 1) - \lceil \log(\frac{8\theta n}{k}) \rceil > ((\log \frac{n}{B} - 1) - 1) - (\log \frac{8\theta n}{k} + 1) = \log(\frac{k}{8\theta B}) - 3$.

Consider any two of the $\binom{B}{k/4}^b$ arrangements, say x and y. From the right to the left, we can find the first sub-block (the z-th sub-block in the block of size 2^fB, where $z \in [1, B]$ and $f \in [d+1, j]$) such that without loss of generality, it is the c-th sub-block of 1-bits, where $c \in [1, k/4]$, in the block of size 2^fB in arrangement x, and it is a sub-block of 0-bits in arrangement y. Assume, for the sake of contradiction, that after receiving arrangements x and y respectively, the memory states are the same, denoted by M. Then, $W - ((2^f - 1)B + 2^fz)$ 0-bits are given as input such that the z-th sub-block in the block of size 2^fB becomes the leftmost sub-block in the sliding window. After receiving this sequence of 0-bits, for arrangement x, the number of 1-bits in the sliding window is $c_1 = (2^f - 1)(k/4) + 2^fc$, while for arrangement y, the number of 1-bits in the sliding window is $c_2 = (2^f - 1)(k/4) + 2^f(c-1)$. As the memory states in both cases are also M before receiving the sequence of 0-bits, we should give the same estimate after receiving the sequence of 0-bits. Thus, the estimate given has an absolute error of at least

$|c_1 - c_2|/2 = 2^{f-1}$ for one of the two cases. Consequently, the relative error is at least

$$\begin{aligned}\frac{2^{f-1}}{\max\{m_1, m_2\}} &= \frac{2^{f-1}}{(2^f-1)(\frac{k}{4}) + 2^f c} \geq \frac{2^{f-1}}{2^{f+1}(\frac{k}{4}) - \frac{k}{4}} \\ &> \frac{2^{f-1}}{2^{f-1} \times k} = \frac{1}{k} = \varepsilon\end{aligned}$$

which contradicts that the relative error of the estimate is no more than ε. Thus, we have to differentiate between any two of the $\binom{B}{k/4}^b$ arrangements. It implies that the required memory is at least

$$\begin{aligned}\log \binom{B}{\frac{k}{4}}^b &\geq \log \left(\frac{B}{\frac{k}{4}}\right)^{kb/4} \\ &> \frac{k}{4}\left(\log\left(\frac{k}{8\theta B}\right) - 3\right)\left(\log \frac{4B}{k}\right) \\ &= \frac{k}{4}\left(\log \frac{k}{64\theta B}\right)\left(\log \frac{4B}{k}\right) \quad .\end{aligned}$$

Since $\theta \leq \frac{1}{16}$ and $B \geq \frac{k}{4}$, by choosing $B = \frac{k}{16}\sqrt{\frac{1}{\theta}}$, the required memory is more than

$$\frac{k}{4}\log^2\left(\frac{1}{4}\sqrt{\frac{1}{\theta}}\right) = \frac{k}{16}\log^2\left(\frac{1}{16\theta}\right) \quad .$$

Therefore, $\Omega(\frac{1}{\varepsilon}\log^2\frac{1}{\theta})$ bits of memory is required. □

Combining the above two lemmas, we have the following theorem.

Theorem 6.3. *Suppose that $\theta \leq \frac{1}{16}$ and $\frac{1}{4\theta W} \leq \varepsilon \leq \frac{1}{5}$. Any data structure for significant one counting with threshold θ and relative error bound ε requires $\Omega(\frac{1}{\varepsilon}\log^2\frac{1}{\theta} + \log \varepsilon\theta W)$ bits of memory.*

6.1.2 An one-level data structure

In this section, we describe a simple data structure for significant one counting. First, we give some definitions. Consider any bit stream. Each bit in the stream has a *stream position* (or simply *sp*), the first bit of the stream has sp = 1, the second one has sp = 2, and etc. Each 1-bit has a *1-rank*, which is its rank among all the 1-bits. Figure 6.2

bit stream	1	0	1	1	1	1	1	1	1	1	0	0
stream position (sp)	1	2	3	4	5	6	7	8	9	10	11	12
1-rank	1		2	3	4	5	6	7	8	9		
interesting 1-bit label				1			2			3		
anchor label			1			2			3			4
	Expired								Sliding Window			

bit stream	0	1	0	0	1	0	1	1	1	1	1
stream position (sp)	13	14	15	16	17	18	19	20	21	22	23
1-rank		10			11		12	13	14	15	16
interesting 1-bit label							4			5	
anchor label			5			6			7		
	Sliding Window										

Figure 6.2: An example bit stream with $\omega = 3$ and sliding window of size $W = 15$.

shows an example of a bit stream. Given a positive integer ω, a 1-bit with its 1-rank divisible by ω is called an *interesting 1-bit* (with respect to ω). Each interesting 1-bit has an *interesting 1-bit label* to denote its rank among all the interesting 1-bits. In Figure 6.2, we set $\omega = 3$. The 1-bit with sp 10 has an 1-rank of 9, which is divisible by ω. Thus, it is an interesting 1-bit. As it is the third interesting 1-bit, its interesting 1-bit label is 3. Recall that W is the size of the sliding window. If p is the current sp, i.e., the most recent sp, those bits with sp in $[p - W + 1, p]$ are in the sliding window while other bits with sp less than $p - W + 1$ are expired.

Basic idea

Given the bit stream, the window size W and the positive integer ω, we can estimate the number of 1-bits in the sliding window with a bounded absolute error as follows:

> We store the (interesting 1-bit label, sp) pairs of all interesting 1-bits (with respect to ω) in the sliding window in a linked list and keep a variable *count* to count the number of 1-bits arrived after the last interesting 1-bit (or the beginning of the stream if no such interesting 1-bit exists). Let s denote the size of the linked list. The estimate $\hat{c}$ of the number of 1-bits is computed by $\hat{c} = s \times \omega + count$.

For the example in Figure 6.2, the linked list contains the (interesting 1-bit label, sp) pairs of all interesting 1-bits in the sliding window, i.e., $\langle (3, 10), (4, 19), (5, 22) \rangle$. The size s of the linked list is 3. As there is one 1-bit arrived after the last interesting 1-bit (with label 5), the variable *count* is 1. Therefore, the estimate $\hat{c} = s \times \omega + count = 3 \times 3 + 1 = 10$.

Bounded absolute error. Let c be the actual number of 1-bits in the sliding window. Intuitively, an interesting 1-bit b in the linked list represents ω 1-bits between its previous interesting 1-bit b_p (excluding the 1-bit b_p) and the interesting 1-bit b (including the 1-bit b). A 1-bit in the sliding window will either be counted by $s \times \omega$ or *count*, and this implies $\hat{c} \geq c$. To get an upper bound on $\hat{c}$, we note that if the variable *count* counts an expired 1-bit, there is no interesting 1-bit in the sliding window and thus we have $s = 0$. As $count \leq \omega - 1$, $\hat{c}$ counts at most $\omega - 1$ expired 1-bits, so that $\hat{c} < c + \omega$. Otherwise, *count* does not count any expired 1-bit. Since only the least recent interesting 1-bit b_l in the sliding window can represent expired 1-bits and the 1-bit b_l itself is not expired, $s \times \omega$ counts at most $\omega - 1$ expired 1-bits. Therefore, we also have $\hat{c} < c + \omega$. In conclusion, we have $c \leq \hat{c} < c + \omega$ and the absolute error of the estimate $\hat{c}$ is at most ω.

Linked list with dilution. The above method can be generalized: For any positive integer i, a *linked list with dilution i* stores the (interesting 1-bit label, sp) pairs of all interesting 1-bits in the sliding window with its interesting 1-bit label divisible by 2^i. We keep three auxiliary variables, *pos*, *nb* and *count*, where *pos* is the current sp, *nb* is the number of interesting 1-bits in the stream and *count* is the number of 1-bits arrived after the last interesting 1-bit (or the beginning of the stream if no such interesting 1-bit exists). For a linked list with dilution i, let s be the size of the linked list and nb' be the interesting 1-bit label in the tail of the linked list, i.e., the most recent interesting 1-bit in the linked list. If there is no such interesting 1-bit, we set $nb' = 0$. The estimate $\hat{c}$ is computed by $\hat{c} = s \times 2^i\omega + (nb - nb') \times \omega + count$.

For the example in Figure 6.2, the linked list with dilution 2 contains the (interesting 1-bit label, sp) pairs of all interesting 1-bits in the sliding window with its interesting 1-bit label divisible by 2, i.e., $\langle(4, 19)\rangle$. Thus, we have $s = 1$ and $nb' = 4$. Also, the auxiliary variables are $pos = 23$, $nb = 5$ and $count = 1$. Therefore, the estimate is $\hat{c} = s \times 2\omega + (nb - nb') \times \omega + count = 1 \times 6 + (5 - 4) \times 3 + 1 = 10$.

Lemma 6.4. *Using the above procedure, the linked list with dilution i returns an estimate $\hat{c}$ of the number c of 1-bits in the sliding window such that $c \leq \hat{c} \leq c+2^i\omega$, i.e., the absolute error of the estimate is at most $2^i\omega$.*

Proof. Intuitively, an interesting 1-bit b in a linked list with dilution i represents $2^i\omega$ 1-bits between its previous interesting 1-bit b_p with interesting 1-bit label divisible by 2^i (excluding the 1-bit b_p) and the interesting 1-bit b (including the 1-bit b). Let *count'*

denote the number of 1-bits arrived after the interesting 1-bit with label nb' (or the beginning of the stream if $nb' = 0$). We have $count' = (nb - nb') \times \omega + count$.

A 1-bit in the sliding window will either be counted by $s \times 2^i\omega$ or $count'$, and this implies $\hat{c} \geq c$. To get an upper bound on $\hat{c}$, we note that if $count'$ counts an expired 1-bit, there is no interesting 1-bit in the sliding window and $s = 0$. Since $count' \leq (2^i - 1)\omega + (\omega - 1) < 2^i\omega$, we have $\hat{c} < c + 2^i\omega$. Suppose that $count'$ does not count any expired 1-bit. As only the least recent interesting 1-bit b_l in the linked list can represent expired 1-bits, and b_l itself is not expired, $s \times 2^i\omega$ counts at most $2^i\omega - 1$ expired 1-bits. Therefore, we also have $\hat{c} < c + 2^i\omega$. In conclusion, we have $c \leq \hat{c} < c + 2^i\omega$ and the absolute error is at most $2^i\omega$. □

Memory usage. Since the sliding window is of size W, we can represent a sp in the sliding window by a modulo $2W$ number without ambiguity. Also, there are at most $\lceil W/\omega \rceil$ interesting 1-bits in the sliding window so that an interesting 1-bit label can be represented by a modulo $\lceil 2W/\omega \rceil$ number. Therefore, a sp needs $O(\log W)$ bits and an interesting 1-bit label needs $O(\log(W/\omega))$ bits, respectively. Hence, each (interesting 1-bit label, sp) pair in the linked list needs $O(\log(W/\omega) + \log W) = O(\log W)$ bits of memory.

To further reduce the memory usage, notice that the difference of the sp of two consecutive interesting 1-bits is no less than the difference of their 1-ranks, i.e., ω. If there is an interesting 1-bit with sp p, there is no other interesting 1-bit in the sp range $[(\lceil \frac{p}{\omega} \rceil - 1)\omega + 1, (\lceil \frac{p}{\omega} \rceil)\omega]$. Therefore, for an interesting 1-bit with sp p, we can use $p' = \lceil \frac{p}{\omega} \rceil$ to denote its position in the stream without ambiguity, and we will define the set of all possible p' as *anchors*. As p is a modulo $2W$ number, the memory to represent the position of an interesting 1-bit can then be reduced from $O(\log W)$ bits to $O(\log(W/\omega))$ bits and each pair in the linked list needs only $O(\log(W/\omega))$ bits of memory.

Anchor and anchor label. Given a positive integer ω, if a sp is divisible by ω, this sp is defined as an *anchor*. Each anchor has an *anchor label* for identification of different anchors. In Figure 6.2, when $\omega = 3$, those sp's divisible by ω (e.g. $3, 6, 9$) are anchors, and the first anchor (sp 3) has an anchor label of 1.

For an interesting 1-bit with sp p, we use the anchor label $\lceil \frac{p}{\omega} \rceil$ to denote its position. For instance, in Figure 6.2, we use the anchor with anchor label $\lceil \frac{4}{3} \rceil = 2$ to denote the position of the interesting 1-bit with sp 4. Intuitively, we shift the sp p of an interesting

1-bit to the next anchor p_a, i.e., sp $p_a = \lceil \frac{p}{\omega} \rceil \omega$. Let p_c be the current sp and assume that p_c is not an anchor. If an interesting 1-bit b arrives after the last anchor, the position of b is the anchor label $p' = \lceil \frac{p_c}{\omega} \rceil$. Since $p_c < \lceil \frac{p_c}{\omega} \rceil \omega$, we denote the position of this interesting 1-bit by the anchor $p'\omega$ with anchor label p', which does not occur yet and is not considered expired. There are at most $\lceil \frac{W}{\omega} \rceil + 1$ anchors which are not expired, so we can use a modulo $(\lceil \frac{2W}{\omega} \rceil + 2)$ number to represent the anchor label.

The one-level data structure

Now, we are ready to describe the one-level data structure, which includes a linked list and four variables. The four auxiliary variables are the followings: na is the number of anchors in the stream, dp is the difference of the last anchor and the current sp, nb is the number of interesting 1-bits in the stream and *count* is the number of 1-bits arrived after the last interesting 1-bit (or the beginning of the stream if no such interesting 1-bit exists). The linked list is a linked list with dilution i, for some integer i, which now stores the (interesting 1-bit label, anchor label) pairs of all interesting 1-bits in the sliding window with its interesting 1-bit label divisible by 2^i. For a linked list with dilution i, we compute the estimate in the same way as before. More precisely, let s be the size of the linked list and nb' be the interesting 1-bit label in the tail of the linked list. If there is no such interesting 1-bit, we set $nb' = 0$. The estimate $\hat{c}$ is computed by $\hat{c} = s \times 2^i\omega + (nb - nb') \times \omega + count$.

For instance, in Figure 6.2, a linked list with dilution 2 contains the (interesting 1-bit label, anchor label) pair of all interesting 1-bits in the sliding window with the interesting 1-bit label divisible by 2, i.e., $\langle (2,3),(4,7) \rangle$. (The interesting 1-bit with interesting 1-bit label 2 is included since the anchor with anchor label 3, i.e., sp 9, is in the sliding window.)

Removing expired items in the linked list. Since the variable *pos* is replaced by variables na and dp, to determine whether the anchor with anchor label na_h in the head of the linked list is expired, the current position p is first computed by $p = na \times \omega + dp$. If $na_h \times \omega \leq p - W$, the head is expired and is removed. (Since the anchor label is represented by a modulo $(\lceil \frac{2W}{\omega} \rceil + 2)$ number, the actual operation to determine whether na_h is expired is: if $na_h \times \omega > p$, the head is expired when $(na_h - (\lceil \frac{2W}{\omega} \rceil + 2)) \times \omega \leq p - W$; otherwise, the head is expired when $na_h \times \omega \leq p - W$. To simplify the discussion, such operation for modulo numbers is not explicitly used.)

Theorem 6.5. *For a linked list with dilution i, the above procedure returns the estimate $\hat{c}$ of the number c of 1-bits in the sliding window such that $c \leq \hat{c} \leq c + 2^{i+1}\omega$, i.e., the absolute error of the estimate is at most $2^{i+1}\omega$.*

Proof. If the current sp is p and the least recent anchor a in the sliding window is sp p_a, we have $p_a \geq p - W + 1$. Thus, only expired interesting 1-bits with sp at least $p-W+1-(\omega-1) = p-W-\omega+2$ can have its position shifted to anchor a. As there is at most one interesting 1-bit in the sp range $[p-W-\omega+2, p-W]$, at most one expired interesting 1-bit is treated as in the sliding window.

In a linked list with dilution i, at most one expired interesting 1-bit is stored in the linked list. Let s_0 be the number of interesting 1-bits in the sliding window with label divisible by 2^i, and $\hat{c}_0$ be the estimate obtained from a linked list with dilution i by the procedure for Lemma 6.4. Therefore, we have $s_0 \leq s \leq s_0 + 1$. Since $\hat{c} = s \times 2^i\omega + (nb - nb') \times \omega + count \geq s_0 \times 2^i\omega + (nb - nb') \times \omega + count = \hat{c}_0$, by Lemma 6.4, $\hat{c} \geq \hat{c}_0 \geq c$. On the other hand, $\hat{c} = s \times 2^i\omega + (nb - nb') \times \omega + count \leq (s_0 + 1) \times 2^i\omega + (nb - nb') \times \omega + count = \hat{c}_0 + 2^i\omega$. By Lemma 6.4, it follows that $\hat{c} \leq \hat{c}_0 + 2^i\omega \leq c + 2^i\omega + 2^i\omega = c + 2^{i+1}\omega$. Therefore, the absolute error of the estimate is at most $2^{i+1}\omega$. □

Definition of $\omega_{l,\varepsilon}$. By Theorem 6.5, the estimate obtained from a linked list with dilution 0 has absolute error at most 2ω. Given some fixed constant l, for $c \geq W/2^l$, the relative error is at most $2\omega/c \leq 2\omega/(W/2^l) = 2^{l+1}\omega/W$. Let $\omega_{l,\varepsilon}$ be the value of ω such that the relative error of the estimate is at most ε for $c \geq W/2^l$. Therefore, we have $2^{l+1}\omega_{l,\varepsilon}/W = \varepsilon$, i.e., $\omega_{l,\varepsilon} = \varepsilon W/2^{l+1}$.

Memory usage for significant one counting. To solve the significant one counting problem, we use a linked list with dilution 0. We set $l = \log(1/\theta)$ such that $W/2^l = \theta W$ and $\omega = \omega_{l,\varepsilon} = \varepsilon W/2^{l+1} = \varepsilon\theta W/2$. For the auxiliary variables, na and nb need $O(\log \frac{W}{\omega}) = O(\log \frac{2}{\varepsilon\theta}) = O(\log \frac{1}{\varepsilon\theta})$ bits, and dp and $count$ need $O(\log \omega) = O(\log \varepsilon\theta W)$ bits. Since each (interesting 1-bit label, anchor label) pair in the linked list needs $O(\log \frac{W}{\omega}) = O(\log \frac{1}{\varepsilon\theta})$ bits and the size of the linked list is at most $\lceil \frac{W}{\omega} \rceil = O(\frac{1}{\varepsilon\theta})$, the linked list requires $O(\frac{1}{\varepsilon\theta} \log \frac{1}{\varepsilon\theta})$ bits of memory. By storing the differences of (interesting 1-bit label, anchor label) pairs in the linked list except for the head of the list, the memory can be reduced to $O(\log \frac{1}{\varepsilon\theta} + \frac{1}{\varepsilon\theta} \log(\frac{2/\varepsilon\theta}{1/\varepsilon\theta})) = O(\log \frac{1}{\varepsilon\theta} + \frac{1}{\varepsilon\theta}) = O(\frac{1}{\varepsilon\theta})$ bits. Therefore, the total memory needed is $O(\frac{1}{\varepsilon\theta} + \log \frac{1}{\varepsilon\theta} + \log \varepsilon\theta W) = O(\frac{1}{\varepsilon\theta} + \log \varepsilon\theta W)$.

6.1.3 Improvement in memory: a multilevel data structure

In the one-level data structure, we set $l = \log(1/\theta)$ and the absolute error of the estimate is $2\omega_{l,\varepsilon} = \varepsilon\theta n$ regardless of the number c of 1-bits in the sliding window. In fact, if the number c of 1-bits in the sliding window is large, i.e., $c > \theta W$, the absolute error allowed is εc, which is larger than $\varepsilon\theta W$. In this section, we make use of this observation to improve the memory usage. Throughout our discussion, we assume that $\omega = \omega_{l,\varepsilon}$, which is equal to $\varepsilon W/2^{l+1}$. Recall that this value of ω guarantees that the relative error of the estimate given by the one-level data structure (with dilution 0) is at most ε for $c \geq W/2^l$.

Lemma 6.6. *Let c be the number of 1-bits in the sliding window. If $c \geq W/2^{l-i}$, then the estimate $\hat{c}$ of c obtained from a linked list with dilution i has relative error at most ε.*

Proof. By Theorem 6.5, the estimate $\hat{c}$ has an absolute error of at most $2^{i+1}\omega = 2^{i+1}\omega_{l,\varepsilon} = (2^{i+1})(\varepsilon W/2^{l+1}) = \varepsilon W/2^{l-i}$. Therefore, if $c \geq W/2^{l-i}$, the relative error is at most $(\varepsilon W/2^{l-i})/c \leq \varepsilon$. □

Lemma 6.7. *Let c be the number of 1-bits in the sliding window. If a linked list with dilution i has a size of at least $\lceil W/(2^{l-1}\omega_{l,\varepsilon})\rceil + 2$, then $c \geq W/2^{l-i-1}$.*

Proof. Let s be the size of the linked list. Recall that $\hat{c} = s \times 2^i\omega + (nb - nb') \times \omega + count$. By Theorem 6.5, if an estimate $\hat{c}$ of c is obtained from a linked list with dilution i, we have $c \leq \hat{c} \leq c + 2^{i+1}\omega_{l,\varepsilon}$. It follows that $c \geq \hat{c} - 2^{i+1}\omega_{l,\varepsilon} = s \times 2^i\omega_{l,\varepsilon} + (nb - nb') \times \omega_{l,\varepsilon} + count - 2^{i+1}\omega_{l,\varepsilon} \geq (\lceil W/(2^{l-1}\omega_{l,\varepsilon})\rceil + 2) \times 2^i\omega_{l,\varepsilon} + count - 2^{i+1}\omega_{l,\varepsilon} \geq W/2^{l-i-1}$. □

Multi-level data structure. We use the same set of auxiliary variables na, nb, dp, $count$, and we set $\omega = \omega_{l,\varepsilon}$ as before. However, instead of keeping a linked list with dilution 0, we keep l levels of linked list, numbered 0 to $l-1$. (Recall that $l = \log(1/\theta)$.) The level k linked list is a linked list with dilution k of size at most $\lceil W/(2^{l-1}\omega_{l,\varepsilon})\rceil + 2$, which keeps the (interesting 1-bit label, anchor label) pairs of the $\lceil W/(2^{l-1}\omega_{l,\varepsilon})\rceil + 2$ most recent interesting 1-bits in the sliding window with interesting 1-bit label divisible by 2^k. For each level $i \in [0, l-1]$ linked list, we introduce the variables pair (nb_i, na_i), which is set to be the (interesting 1-bit label, anchor label) pair of most recent interesting 1-bit removed from the linked list. Initially, (nb_i, na_i) is set to $(0, 0)$. To return an estimate $\hat{c}$, we first find the smallest level j which contains the most recent *expired* interesting 1-bit in (nb_j, na_j) among all levels, i.e., na_j is the anchor label of the most recent expired

anchor among all levels. The estimate is then obtained from the level j linked list as follows. Let s be the size of the level j linked list and nb' be the interesting 1-bit label in the tail of the level j linked list. If there is no such interesting 1-bit, we set $nb' = 0$. The estimate $\hat{c}$ is computed by $\hat{c} = s \times 2^j\omega + (nb - nb') \times \omega + count$.

Theorem 6.8. *Let $l = \log \frac{1}{\theta}$ (or equivalently $\omega_{l,\varepsilon} = \varepsilon\theta W/2$). The above procedure returns an estimate $\hat{c}$ of the number c of 1-bits in the sliding window with a relative error at most ε for $c \geq \theta W$ and an absolute error at most $\varepsilon\theta W$ for $c < \theta W$, using $O(\frac{1}{\varepsilon}\log^2\frac{1}{\theta} + \log \varepsilon\theta W)$ bits of memory.*

Proof. In the above procedure, if a level $i \in [0, l-1]$ linked list has (nb_i, na_i) where na_i is the anchor label of an anchor in the sliding window, then the level i linked list is truncated to have a size at most $\lceil W/(2^{l-1}\omega_{l,\varepsilon})\rceil + 2$. Note that when a linked list with dilution $i \in [0, l-1]$ is truncated, we cannot apply Theorem 6.5 and Lemma 6.6 on the estimate from this linked list. The number of interesting 1-bits in the sliding window with interesting 1-bit label divisible by 2^{l-1} is at most $\lceil W/2^{l-1}\omega_{l,\varepsilon}\rceil < \lceil W/2^{l-1}\omega_{l,\varepsilon}\rceil + 2$. Therefore, the level $l-1$ linked list can store all the interesting 1-bits in the sliding window with label divisible by 2^{l-1}. Since the level $l-1$ linked list is not truncated, among all the levels, there is at least one level (i.e., level $l-1$) that Theorem 6.5 and Lemma 6.6 can be applied on the estimate from that level.

Note that the interesting 1-bit with label nb_j is the most recent expired 1-bit in the data structure. Furthermore, nb_j is divisible by 2^j since it is removed from the level j linked list. As na_j is the anchor label of an expired anchor, the level j linked list is not truncated, and thus Theorem 6.5 and Lemma 6.6 can be applied on the estimate from the level j linked list. If $j > 0$, as nb_j is divisible by 2^j, nb_j is also divisible by 2^{j-1}. Since level j is the smallest level containing this interesting 1-bit, level $j-1$ must keep $\lceil W/2^{l-1}\omega_{l,\varepsilon}\rceil + 2$ interesting 1-bits in the sliding window. By Lemma 6.7, we have $c \geq W/2^{l-(j-1)-1} = W/2^{l-j}$. Therefore, by Lemma 6.6, the estimate $\hat{c}$ obtained from the level j linked list has a relative error of at most ε. Note that for $j > 0$, we have $c \geq W/2^{l-j} \geq W/2^{l-1} = 2(W/2^l) = \theta W$. If $j = 0$, since $\omega = \omega_{l,\varepsilon}$, the estimate obtained from the level $j = 0$ linked list has relative error at most ε for $c \geq W/2^l$. Therefore, for $c \geq W/2^l = \theta W$, the procedure returns an estimate $\hat{c}$ with a relative error at most ε. Also, for $c < \theta W$, the estimate is obtained from the level 0 linked list, i.e., a linked list with dilution 0. By Theorem 6.5, the absolute error of the estimate is at most $2\omega = 2\omega_{l,\varepsilon} = \varepsilon\theta W$.

Memory usage. The auxiliary variables still needs $O(\log \frac{1}{\varepsilon\theta} + \log \varepsilon\theta W)$ bits of memory. Since each (interesting 1-bit label, anchor label) pair in the linked list and (nb_i, na_i) for $i \in [0, l-1]$ need $O(\log \frac{W}{\omega}) = O(\log \frac{1}{\varepsilon\theta})$ bits and the size of a linked list is at most $\lceil W/2^{l-1}\omega_{l,\varepsilon} \rceil + 2 = \lceil \frac{4}{\varepsilon} \rceil + 2 = O(\frac{1}{\varepsilon})$, a level i linked list with (nb_i, na_i) requires $O((\frac{1}{\varepsilon} + 1) \log \frac{1}{\varepsilon\theta}) = O(\frac{1}{\varepsilon} \log \frac{1}{\varepsilon\theta})$ bits of memory. By storing the differences of (interesting 1-bit label, anchor label) pairs in the linked list except for the head of the linked list, the memory can be reduced to $O(2 \log \frac{1}{\varepsilon\theta} + \frac{1}{\varepsilon} \log(\frac{2/\varepsilon\theta}{4/\varepsilon})) = O(\frac{1}{\varepsilon} \log \frac{1}{\theta})$ bits. Therefore, the total memory needed is $O((\frac{1}{\varepsilon} \log \frac{1}{\theta})l + \log \frac{1}{\varepsilon\theta} + \log \varepsilon\theta W + 1) = O(\frac{1}{\varepsilon} \log^2 \frac{1}{\theta} + \log \varepsilon\theta W)$ bits. □

6.1.4 Improvement in time: the optimal data structure

Note that by setting $l = \log \frac{1}{\theta}$, the data structure described in the last section takes $O(l) = O(\log \frac{1}{\theta})$ time to answer a query because it takes $O(l)$ time to find the smallest level j containing the most recent expired interesting 1-bit with label nb_j among all the levels. Also, for the per-item processing, we have to insert and remove an interesting 1-bit to/from at most l levels, which also costs $O(l) = O(\log \frac{1}{\theta})$ time. In this section, we improve the per-item processing time and query time to $O(1)$.

In our optimal data structure, we let $l = \log \frac{1}{\theta}$ and $\omega = \omega_{l,\varepsilon}$; the latter is $\varepsilon\theta W/2$. We still keep the set of the four auxiliary variables *na*, *nb*, *dp*, *count* as in Section 6.1.2 and keep the l levels of linked list as in Section 6.1.3. In addition, we use one more auxiliary variable *lb* to store the interesting 1-bit label of the most recent expired interesting 1-bit removed from the l levels, which is 0 initially. The estimate $\hat{c}$ is then computed by $\hat{c} = (nb - lb) \times \omega + count$. Also, we insert an interesting 1-bit only to the maximum level containing it. Thus, the size of the level $i \in [0, l-2]$ linked list is halved, i.e., $\frac{1}{2}(\lceil W/2^{l-1}\omega_{l,\varepsilon} \rceil + 2)$, while the size of the level $l-1$ linked list is not changed. Furthermore, we keep one more linked list L of (interesting 1-bit label, anchor label) pairs of all the interesting 1-bit stored in all the l levels so that only $O(1)$ time is needed to remove an interesting 1-bit. The updating algorithm is shown as follows, where $\omega = \omega_{l,\varepsilon} = \varepsilon\theta W/2$:

Initially, variables *na*, *nb*, *dp*, *count* and *lb* are set to 0.

For per-item processing (when a bit b arrives):

1. Increment *dp*.

2. If $dp = \omega$ (i.e., $0 \mod \omega$), increment na.

3. Check for expiration: If the head $(nb_h,\ na_h)$ of the linked list L has expired, i.e., $na_h \times \omega \leq na \times \omega + dp - W$, set $lb = nb_h$ and remove $(nb_h,\ na_h)$ from L and thus the corresponding level.

4. If $b = 1$, increment *count*.

5. If $b = 1$ and $count = \omega$ (i.e., $count = 0 \mod \omega$),

   - Increment nb.
   - Determine the largest level k such that nb is a multiple of 2^k.
   - If the level k linked list reaches its the maximum size, remove its head from the level k linked list and thus the linked list L at the same time.
   - If $dp = 0$, add (nb, na) to the tail of the level k linked list and the tail of L; else, add $(nb, na + 1)$ to the tail of the level k linked list and the tail of L.

For answering a query:

Return an estimate $\hat{c} = (nb - lb) \times \omega + count = (nb - lb) \times (\varepsilon\theta W/2) + count$.

Theorem 6.9. *The above algorithm returns an estimate $\hat{c}$ of the number c of 1-bits in the sliding window with a relative error at most ε, i.e., $c \leq \hat{c} \leq c + \varepsilon c$, for $c \geq \theta W$ and an absolute error at most $\varepsilon\theta W$, i.e., $c \leq \hat{c} \leq c + \varepsilon\theta W$, for $c < \theta W$ using $O(\frac{1}{\varepsilon}\log^2\frac{1}{\theta} + \log\varepsilon\theta W)$ bits of memory with $O(1)$ per-item processing and $O(1)$ query time.*

Proof. Let j be the minimum level containing the interesting 1-bit with label lb when it is expired. Let nb_t be the interesting 1-bit label in the tail of the level j linked list. The level j linked list thus has size $s = (nb_t - lb)/2^j$. By the procedure for Theorem 6.5, the estimate from the linked list with dilution j is computed by $\hat{c} = s \times 2^j\omega + (nb - nb_t) \times \omega + count = ((nb_t - lb)/2^j) \times 2^j\omega + (nb - nb_t) \times \omega + count = (nb - lb) \times \omega + count$. Therefore, the estimate is equal to that obtained from the level j linked list. Note that in the proof of Theorem 6.8, the interesting 1-bit with label nb_j is the most recent expired 1-bit in the data structure and it is removed from the level j linked list. Thus, we have $lb = nb_j$. By Theorem 6.8, the above algorithm returns an estimate $\hat{c}$ of the number c of 1-bits in the sliding window with a relative error at most ε, i.e., $c \leq \hat{c} \leq c + \varepsilon c$, for $c \geq \theta W$ and an absolute error at most $\varepsilon\theta W$, i.e., $c \leq \hat{c} \leq c + \varepsilon\theta W$, for $c < \theta W$.

As the variable lb needs $O(\log \frac{W}{\omega}) = O(\log \frac{1}{\varepsilon\theta})$ bits of memory, the set of auxiliary variables still needs $O(\log \frac{1}{\varepsilon\theta} + \log \varepsilon\theta W)$ bits of memory. Since each (interesting 1-bit label, anchor label) pair in the linked list needs $O(\log \frac{1}{\varepsilon\theta})$ bits and the size of a linked list remains $O(\frac{1}{\varepsilon})$, a linked list requires $O(\frac{1}{\varepsilon} \log \frac{1}{\varepsilon\theta})$ bits of memory. By storing differences of (interesting 1-bit label, anchor label) pairs in the linked list except for the head of the linked list, the memory can be reduced to $O(\log \frac{1}{\varepsilon\theta} + \frac{1}{\varepsilon} \log(\frac{2/\varepsilon\theta}{2/\varepsilon})) = O(\frac{1}{\varepsilon} \log \frac{1}{\theta})$ bits. Therefore, the total memory needed is still $O(\frac{1}{\varepsilon} \log^2 \frac{1}{\theta} + \log \varepsilon\theta W)$. Moreover, in the algorithm, all operations in per-item processing and query can be done in $O(1)$ time. □

6.2 Finding Frequent Items over Sliding Window

This section considers finding ε-approximate frequent items over a count-based sliding window, where $\varepsilon \in (0, 1)$ is the error bound. To ease discussion, a set of items is said to be a (θ, ε)-*frequent item set* if it is an answer to the ε-approximate frequent items problem with threshold θ. We give an algorithm for identifying (θ, ε)-frequent item set over a sliding window, which improves the previous work by Arasu and Manku [5]. Both Arasu and Manku's algorithm and our algorithm are based on an algorithm of Misra and Gries [73]. This algorithm, which we call the MG algorithm, finds (θ, ε)-frequent item set over the whole data stream using $O(\frac{1}{\varepsilon})$ space and supporting $O(\frac{1}{\varepsilon})$ update and query time. However, unlike Arasu and Manku's algorithm, which uses the MG algorithm as a black box, our algorithm applies a technique, which we call the *batch-decrement* technique, introduced in the MG algorithm. To apply this technique to find (θ, ε)-frequent item set over the entire data stream, the MG algorithm uses $O(\frac{1}{\varepsilon})$ counters to count the frequency of $O(\frac{1}{\varepsilon})$ items. To find (θ, ε)-frequent item set over the sliding window, our idea is to replace the simple counter by some data structure called λ-counters that counts the items over the sliding window. Throughout this section, we let n be the window size.

In Section 6.2.1, we give details on the λ-counters and prove some important properties about them. In Section 6.2.2, we describe our algorithm for identifying (θ, ε)-frequent item set and prove its correctness. Finally Section 6.2.3 extends our algorithm to sliding window with variable size.

position	1	2	3	4	5	6	7	8	9	10	11	12
bit stream	1	0	1	1	1	1	1	1	1	1	0	0
λ-sampled 1-bit				√			√			√		
									window W_{23}			

position	13	14	15	16	17	18	19	20	21	22	23
bit stream	0	1	0	0	1	0	1	1	1	1	1
λ-sampled 1-bit							√			√	
	window W_{23}										

Figure 6.3: An example with $\lambda = 3$ and window size $n = 15$.

6.2.1 λ-snapshot and λ-counter

In this section, we introduce a sampling technique for estimating the number of 1-bits of a bit stream over a sliding window. Then, we give its implementation. Note that λ-counter is a simple variant of the window counter given in Section 6.1.

Consider any bit stream $\delta = b_1 b_2 b_3 \cdots$. Recall that n is the sliding window size. For any $1 \leq i \leq j$, let $[i..j]$ denote the window from positions i to j of the stream; in other words, $[i..j]$ covers the bits $b_i, b_{i+1}, \ldots, b_j$. For any $p \geq n$, let W_p denote the window $[(p-n+1)..p]$, which ends at position p and has size n. We let $W_p = [1..p]$ if $p < n$. Let λ be any positive integer. A 1-bit in δ is said to be a *λ-sampled 1-bit*, or simply a sampled 1-bit, if it is the $(i\lambda)$th 1-bit in δ for some integer $i \geq 1$. Thus, the (λ)th, (2λ)th and (3λ)th 1-bits in δ are all sampled 1-bits. Figure 6.3 shows the sampled 1-bits of a bit stream at or before position 23. Note that for any two consecutive sampled 1-bits b_i and b_j $(i < j)$, there are exactly $\lambda - 1$ (non-sampled) 1-bits between them. We say that these $\lambda - 1$ bits, as well as the bit b_j, are *covered* by b_j. For example, in Figure 6.3, the 1-bits at positions 14, 17 and 19 are covered by the 1-bit at position 19. Our sampling technique for estimating the number of 1-bits over a sliding window is based on *λ-snapshot*, which is defined as follows.

Definition 6.10. *The λ-snapshot of a bit stream δ over window W_p is the pair (Q, ℓ) where*

- *Q is a queue storing the (sorted) positions of the sampled 1-bits of δ over W_p, and*
- *ℓ is the number of 1-bits at or before position p that are not covered by any sampled 1-bit at or before p.*

For example, in Figure 6.3, the λ-snapshot of the bit stream over window W_{23} is $(Q, \ell) = (\langle 10, 19, 22 \rangle, 1)$. (Note that W_{23} has size 15 and the bits at positions from 1

to 8 are expired.) It should be clear that $\ell \leq \lambda - 1$ and the ℓ 1-bits mentioned in the definition must be the last ℓ 1-bits over $[1..p]$ (and some of them may be outside W_p and at positions smaller than $p - n + 1$). The following lemma proves that λ-snapshot gives good estimate of the number of 1-bits over a sliding window.

Lemma 6.11. *Let (Q, ℓ) be the λ-snapshot of bit stream $\delta = b_1 b_2 b_3 \ldots$ over window W_p. Let m be the number of 1-bits of δ over W_p. Then,*

$$m \leq |Q|\lambda + \ell < m + \lambda \tag{6.1}$$

where $|Q|$ denotes the length of Q.

Proof. If $|Q| = 0$, all the m 1-bits over W_p are not covered and thus $m \leq \ell$. Together with the fact that $\ell \leq \lambda - 1$, we have $m \leq \ell \leq \lambda - 1 < m + \lambda$ and (6.1) holds. Suppose that $|Q| \geq 1$ and $Q = \langle i_1, i_2, \ldots, i_{|Q|} \rangle$. Then, the sampled 1-bits over W_p are $b_{i_1}, b_{i_2}, \ldots, b_{i_{|Q|}}$, which cover a total of $|Q|\lambda$ 1-bits. Note that among the m 1-bits over W_p, the first $m-\ell$ of them are covered by $b_{i_1}, b_{i_2}, \ldots, b_{i_{|Q|}}$. It follows that $m - \ell \leq |Q|\lambda$ and the first inequality of (6.1) holds. Note also that all of $b_{i_1}, b_{i_2}, \ldots, b_{i_{|Q|}}$, as well as the $(\lambda - 1)(|Q| - 1)$ non-sampled 1-bits covered by $b_{i_2}, b_{i_3}, \ldots, b_{i_{|Q|}}$, must be over W_p. Together with the last ℓ 1-bits, we have $m \geq |Q| + (\lambda - 1)(|Q| - 1) + \ell = \lambda|Q| + \ell - (\lambda - 1)$, which implies the second inequality of (6.1). □

We now describe the λ-*counter* for maintaining the λ-snapshot over a sliding window. The counter has a queue Q and a variable ℓ. To speed up the update time, Q is implemented as a deque, which allows us to remove an entry from both ends of Q (using the operations *pop_head* and *pop_tail*), and to append an entry to the end (using the operation *push_tail*). We also have a global variable p for remembering the current position, i.e., the position of the most recently read bit. Initially, Q is empty and ℓ is equal to 0. In the following, we implement the operation *shift*( ) for a λ-counter C: when a new bit b arrives, we execute *shift*(b) to update the λ-snapshot stored in C.

The following lemma is for the correctness of λ-counter and its proof is straightforward.

Lemma 6.12. *Let $\delta = b_1 b_2 b_3 \cdots$ be a bit stream and C be a λ-counter. Suppose that initially, C's queue Q is empty and its variable $\ell = 0$. Then, after executing*

[1]Note that a data stream is potentially infinite. To ensure that the current position can be stored in the variable p, we may store it as a modulo $2n$ integer without any ambiguity.

C.shift (b) :
1: $p \leftarrow p + 1$ [1]
2: **if** $|Q| > 0$ and $head[Q] < p - n + 1$ **then**
3: $\quad$ *pop_head*(Q) {remove the expired head of Q}
4: **if** $b = 1$
5: $\quad \ell \leftarrow (\ell + 1) \bmod \lambda$
6: $\quad$ **if** $\ell = 0$ **then** {this bit is a sampled 1-bit}
7: $\quad\quad$ *push_tail*(Q, p) {append p to Q}

C.shift (b_1), *C.shift* $(b_2), \ldots,$ *C.shift* (b_p) *for any* $p \geq 1$*, the queue* Q *and variable* ℓ *form a* λ*-snapshot* (Q, ℓ) *of* δ *over* W_p*.*

To estimate the number of 1-bits over a sliding window, we define the value of the λ-counter C to be $v(C) = |Q|\lambda + \ell$. Let $N_\delta(1,W_p)$ denote the number of 1-bits of bit stream δ over W_p. The lemma below gives the accuracy of $v(C)$ and the space requirement of C.

Lemma 6.13. *Given a* λ*-counter* C*, if its queue* Q *and variable* ℓ *form a* λ*-snapshot* (Q, ℓ) *of bit stream* δ *over window* W_p*, then*

$$N_\delta(1,W_p) \leq v(C) < N_\delta(1,W_p) + \lambda. \tag{6.2}$$

Furthermore, the size of C *is at most* $\lceil N_\delta(1,W_p)/\lambda \rceil + 1$*.*

Proof. The inequality (2) follows directly from Lemma 6.11. For the size of C, note that there are at most $\lceil N_\delta(1,W_p)/\lambda \rceil$ entries in Q. Together with the variable ℓ, we have the bound. □

In the rest of this section, we describe how to decrement a positive λ-counter (i.e., decrease its value by 1); we need this operation to implement the technique of batch-decrement (see Section 6.2.2). Suppose that we have used a λ-counter C to count the bit stream $\delta = b_1 b_2 \ldots b_p$ by executing *C.shift* (b_1), *C.shift* (b_2), $\ldots$, *C.shift* (b_p), and afterward the value $v(C)$ of C is greater than zero. Note that $v(C) = |Q|\lambda + \ell > 0$ implies that at least one of the following cases is true: (i) the queue Q is not empty, or (ii) ℓ is greater than 0. To decrement C, we modify Q and ℓ such that C becomes the λ-snapshot of the stream obtained from δ by replacing its last 1-bit with a 0-bit. Following is the implementation.

C.decrement():
1: **if** $v(C) = 0$ **then** return error
{We will see later that we execute the operation only when $v(C) > 0$.}
2: **else if** $\ell > 0$ **then**
3: $\ell \leftarrow \ell - 1$
4: **else** {i.e., $\ell = 0$, and this implies Q is not empty}
5: *pop_tail*(Q) {delete the last entry from Q}
6: $\ell \leftarrow \lambda - 1$

For any sequence σ of *decrement*() and *shift*() operations, we say that σ is *associated with* the bit stream $b_1 b_2 \dots b_p$ if *shift*(b_1)*shift*$(b_2) \cdots$ *shift*(b_p) is the subsequence of all shift operations in σ. For any bit streams $b'_1 b'_2 \dots b'_p$ and $b_1 b_2 \dots b_p$, we say that $b'_1 b'_2 \dots b'_p \leq b_1 b_2 \dots b_p$ if $b'_i \leq b_i$ for every $1 \leq i \leq p$. The following lemma generalizes Lemma 6.12.

Lemma 6.14. *Let $\sigma = \sigma_1 \sigma_2 \cdots \sigma_k$ be a sequence of shift() and decrement() operations on the λ-counter C, whose initial value is zero. Suppose that σ is associated with $b_1 b_2 \cdots b_p$ and it executes C.decrement() only when $v(C) > 0$. Let d be the number of decrement() operations in σ. Then, after executing the last operation σ_k, C will be storing the λ-snapshot of some bit stream $b'_1 b'_2 \cdots b'_p$ over window W_p where $b'_1 b'_2 \dots b'_p \leq b_1 b_2 \dots b_p$ and there are d positions $i \in [1..p]$ with $b'_i \neq b_i$.*

Proof. We prove the lemma by induction on the length k of the sequence. The lemma is obviously true for $k = 0$. Suppose that it is true for any sequence with length smaller than k and we now consider the sequence $\sigma = \sigma_1 \sigma_2 \cdots \sigma_k$ given in the lemma. Depending on σ_k, we consider the following two cases.

- If σ_k is a shift operation, then it must be *C.shift*(b_p). This implies $\sigma_1 \sigma_2 \cdots \sigma_{k-1}$ is associated with $b_1 b_2 \cdots b_{p-1}$ and has d decrement operations. The induction hypothesis asserts that after executing σ_{k-1}, C stores the λ-snapshot S of some bit stream $b'_1 b'_2 \dots b'_{p-1}$ over window W_{p-1} where $b'_1 b'_2 \dots b'_{p-1} \leq b_1 b_2 \dots b_{p-1}$ and there are d positions $i \in [1..p-1]$ with $b'_i \neq b_i$. By design, *C.shift*(b_p) transforms S to the λ-snapshot of bit stream $b'_1 b'_2 \cdots b'_{p-1} b'_p$ over W_p where $b'_1 b'_2 \cdots b'_{p-1} b'_p = b'_1 b'_2 \cdots b'_{p-1} b_p$. Since $b'_p = b_p$, we still have $b'_1 b'_2 \cdots b'_p \leq b_1 b_2 \dots b_{p-1} b_p$ and the number of positions $i \in [1..p]$ with $b'_i \neq b_i$ is still d.

- If σ_k is a *C.decrement*() operation, then $\sigma_1 \sigma_2 \cdots \sigma_{k-1}$ must be associated with the bit stream $b_1 b_2 \dots b_p$ and has $d-1$ decrement operations. By the induction hypothesis,

after executing σ_{k-1}, C stores the λ-snapshot S of some bit stream $c_1c_2\cdots c_p$ over window W_p where $c_1c_2\ldots c_p \leq b_1b_2\ldots b_p$ and there are $d-1$ positions with $c_i \neq b_i$. Then, σ_k will transform S to the λ-snapshot of the stream $b'_1b'_2\ldots b'_p$ obtained by replacing the last 1-bit of $c_1c_2\ldots c_p$ with a 0-bit. Therefore, $b'_1b'_2\ldots b'_p \leq c_1c_2\ldots c_p \leq b_1b_2\ldots b_p$, and the number of positions with $b'_i \neq b_i$ is increased to d. □

6.2.2 Finding (θ, ε)-frequent item set

In this section, we describe a simple and efficient algorithm for finding (θ, ε)-frequent item sets. Our algorithm applies the batch-decrement technique introduced in [73]. For ease of discussion, we first describe an implementation that uses $|\Pi|$ λ-counters where Π is the set of all possible items. Then, we note that most of these counters have value 0 and need not be stored physically. We prove that this improved implementation uses $O(\frac{1}{\varepsilon})$ space and supports $O(\frac{1}{\varepsilon})$ query and update times.

Let $\delta = e_1e_2e_3\ldots$ be a stream of items in Π. For any window W and any item e, let $N_\delta(e, W)$ denote the number of occurrences of e in δ over W. Define δ_e to be the bit stream $b_1b_2b_3\ldots$ where

$$b_i = \begin{cases} 1, & \text{if } e_i = e; \\ 0, & \text{otherwise.} \end{cases} \tag{6.3}$$

For example, if $\delta = aaababbca\ldots$, then $\delta_a = 111010001\ldots$. Note that $N_{\delta_e}(1, W)$ is the number of occurrences of 1-bits in δ_e over W, and $N_\delta(e, W) = N_{\delta_e}(1, W)$. To identify δ's frequent items over a sliding window, our algorithm maintains a data structure $\texttt{BD}(n, \lambda, c)$, which has, for each item $e \in \Pi$, a λ-counter C_e to count the 1-bit in δ_e, or equivalently, to count the item e in δ. The parameter c determines when we need to do a batch of *decrement*() operations. In our implementation, we let $\lambda = \varepsilon n/4$ and $c = 4/\varepsilon$. Roughly speaking, the λ-counters we use are $(\varepsilon n/4)$-counters and we perform a batch of *decrement*() operations whenever there are more than $c = 4/\varepsilon$ λ-counters with value greater than 0. The main operation supported by $\texttt{BD}(n, \lambda, c)$ is *Update*(); to process the item stream $\delta = e_1e_2e_3\ldots$, we execute *Update*(e_1), *Update*(e_2), *Update*(e_3), ... The implementation of *Update*() is given in the subrountine *Update*(e).

Note that all the λ-counters are initialized to zero. The following fact gives an invariant preserved by $\texttt{BD}(n, \lambda, c)$.

Update(e) :
1: $C_e.shift(1)$
2: **for** each item $x \neq e$ **do** $C_x.shift(0)$
3: **if** there are more than c items x with $v(C_x) > 0$ **then**
4: **for** each item x with $v(C_x) > 0$ **do** $C_x.decrement()$

Fact 6.15. *There are at most c positive counters (i.e., counters with value > 0) in $\texttt{BD}(n, \lambda, c)$ after each Update() operation.*

Note that if a λ-counter has value 0, its queue Q must be empty and its variable ℓ must store a 0; storing it physically in $\texttt{BD}(n, \lambda, c)$ will not give us extra information. Therefore, in the actual implementation of $\texttt{BD}(n, \lambda, c)$, we will only store the few λ-counters with positive values, with the understanding that any λ-counter that is not in $\texttt{BD}(n, \lambda, c)$ has value 0.

We now analyze the accuracy of $\texttt{BD}(n, \lambda, c)$ for processing $\delta = e_1 e_2 e_3 \ldots$. Note that if Line 4 of *Update*() is executed, we will make a *batch of decrement*() operations. The key step in our analysis is to estimate how many batches of *decrement* we can make (i.e., how many times Line 4 is executed) during a sequence of updates.

Lemma 6.16. *Suppose that given the item stream $\delta = e_1 e_2 \ldots e_p$, we have updated $\texttt{BD}(n, \lambda, c)$ by executing Update(e_1), Update(e_2), ..., Update(e_p). Let D_t be the number of batches of decrement() operations made during the last t updates, that is, during Update(e_{p-t+1}), ..., Update(e_p). Then, $D_t < \frac{n+t}{c} + \lambda$.*

Proof. Recall that for each item $e \in \Pi$, $\texttt{BD}(n, \lambda, c)$ uses λ-counter C_e to count the bit stream δ_e defined in (6.3). We first estimate the total value of these counters just before executing *Update*(e_{p-t+1}). From Lemma 6.14, we conclude that at this moment, C_e is storing the λ-snapshot of some bit stream δ'_e over W_{p-t} where $\delta'_e[1..p-t] \leq \delta_e[1..p-t]$. Together with Lemma 6.13, we have

$$\sum_{e \in \Pi} v(C_e) = \sum_{e \in \Pi, v(C_e) > 0} v(C_e) \leq \sum_{e \in \Pi, v(C_e) > 0} (N_{\delta'_e}(1, W_{p-t}) + \lambda)$$
$$\leq \sum_{e \in \Pi, v(C_e) > 0} (N_{\delta_e}(1, W_{p-t}) + \lambda) = \sum_{e \in \Pi, v(C_e) > 0} (N_{\delta}(e, W_{p-t}) + \lambda).$$

Since $\sum_{e \in \Pi} N_\delta(e, W_{p-t}) = |W_{p-t}| \leq n$, and there are at most c positive counters (Fact 6.15),

the total value of the counters just before executing $Update(e_{p-t+1})$ is at most $n + c\lambda$. Since the t remaining operations $Update(e_{p-t+1})$, $Update(e_{p-t+2})$, ..., $Update(e_p)$ will increase this total value by at most t, there are at most $(n + c\lambda) + t$ units for the D_t batches of *decrement*() made during $Update(e_{p-t+1}), \ldots, Update(e_p)$ to take away. Note that each batch of *decrement*() would take away more than c units from the counters, and since the counters cannot have negative values, we conclude that $cD_t < n + c\lambda + t$. The lemma follows. □

We are now ready to show that the λ-counters maintained by $\texttt{BD}(n, \lambda, c)$ give estimates good enough for us to identify (θ, ε)-frequent item set over a sliding window. Recall that in our implementation, we set $\lambda = \varepsilon n/4$ and $c = 4/\varepsilon$.

Lemma 6.17. *Suppose that in processing the stream $\delta = e_1 e_2 \ldots e_p$, we have made D_n batches of decrement() during the last n updates (i.e., from $Update(e_{p-n+1})$ to $Update(e_p)$). Then, after the last operation $Update(e_p)$, we have the following.*

(i) *For each item e,*

$$N_\delta(e,W_p) - D_n \leq v(C_e) < N_\delta(e,W_p) + \lambda. \tag{6.4}$$

(ii) *The set*

$$S = \left\{ e \;\middle|\; v(C_e) \geq (\theta - \varepsilon)n + \lambda \right\} \tag{6.5}$$

is a (θ, ε)-frequent item set of δ over W_p.

Proof. We first prove (i). As argued in the proof of Lemma 6.16, after executing $Update(e_p)$, C_e stores the λ-snapshot of some bit stream δ'_e where $\delta'_e \leq \delta_e$, and

$$v(C_e) < N_{\delta'_e}(1,W_p) + \lambda \leq N_{\delta_e}(1,W_p) + \lambda = N_\delta(e,W_p) + \lambda,$$

which is the second inequality of (6.4). For the first inequality, recall that δ'_e is obtained from δ_e by replacing some 1-bits of δ_e with 0-bits, and each of these replacements results from a distinct call of $C_e.decrement()$. Note that only those $C_e.decrement()$ called after $Update(e_{p-n})$ can change the 1-bits of δ_e over $W_p = [p - n + 1..p]$. Since there are only D_n batches of *decrement*() made after $Update(e_{p-n})$, we call $C_e.decrement()$ at most D_n times and hence there are at most D_n positions $i \in W_p$ with $\delta'_e[i] < \delta_e[i]$.

It follows that $N_{\delta'_e}(1,W_p) \geq N_{\delta_e}(1,W_p) - D_n$, and together with Lemma 6.13, we have $v(C_e) \geq N_{\delta'_e}(1,W_p) \geq N_{\delta_e}(1,W_p) - D_n = N_\delta(e,W_p) - D_n$.

We now prove (ii). From the second inequality of (6.4) and the definition of S, we conclude that every item $e \in S$ satisfies

$$N_\delta(e,W_p) > v(C_e) - \lambda \geq \big((\theta-\varepsilon)n + \lambda\big) - \lambda = (\theta-\varepsilon)n.$$

By the first inequality of (6.4), Lemma 6.16, and the fact that $\lambda = \varepsilon n/4$ and $c = 4/\varepsilon$, we conclude that any item e with $N_\delta(e,W_p) \geq \theta n$ satisfies

$$v(C_e) \geq N_\delta(e,W_p) - D_n \geq \theta n - D_n \geq \theta n - \tfrac{2n}{c} - \lambda = \theta n - \varepsilon n + \lambda,$$

and hence is in S. Hence, S is a (θ,ε)-frequent item set. □

Lemma 6.17 suggests that we can find (θ,ε)-frequent item sets as follows.

Query() :
1: $S \leftarrow \emptyset$
2: **for** each $e \in \Pi$ **do**
3: **if** $v(C_e) \geq (\theta-\varepsilon)n + \lambda$ **then** $S \leftarrow S \cup \{e\}$
4: output S

The following theorem gives bounds on the size of $\texttt{BD}(n,\lambda,c)$ and the query and update times supported by $\texttt{BD}(n,\lambda,c)$.

Theorem 6.18. $\texttt{BD}(n,\lambda,c)$ *uses* $O(\frac{1}{\varepsilon})$ *space, and it takes* $O(\frac{1}{\varepsilon})$ *time to execute the* *Query*() *and* *Update*() *operations.*

Proof. Suppose that we have executed *Update*(e_1), *Update*(e_2), ..., *Update*(e_p). To estimate the space requirement, recall that we will only store in $\texttt{BD}(n,\lambda,c)$ those λ-counters with positive values. Together with Lemma 6.13 and Fact 6.15, we conclude that the size of $\texttt{BD}(n,\lambda,c)$ is

$$\sum_{e\in\Pi, v(C_e)>0} \Big(1 + \tfrac{N_{\delta'_e}(1,W_p)}{\lambda}\Big) \leq c + \sum_{e\in\Pi} \tfrac{N_{\delta_e}(1,W_p)}{\lambda} = c + \sum_{e\in\Pi} \tfrac{N_\delta(e,W_p)}{\lambda} = c + \tfrac{n}{\lambda},$$

which is $\frac{8}{\varepsilon}$ because $c = \frac{4}{\varepsilon}$ and $\lambda = \frac{\varepsilon n}{4}$.

To estimate the time needed for executing *Query*(), we note that all items e in the set returned by *Query*() have $v(C_e) > 0$. Hence their λ-counters are stored in $\texttt{BD}(n, \lambda, c)$ and we can identify them by scanning the data structure once; the time needed is proportional to the size of $\texttt{BD}(n, \lambda, c)$, which is $O(\frac{1}{\varepsilon})$.

For *Update*(e), we need to execute C_e.*shift*(1) once and *decrement*() at most $c + 1$ times. We also need to execute C_x.*shift*(0) for the other items x. However, note that if $v(C_x) = 0$, then the value is still 0 after executing C_x.*shift*(0). Therefore, we only need to execute C_x.*shift*(0) for the $O(c)$ counters C_x with positive values. In conclusion, we need to execute the *shift*() and *decrement*() operations $O(c)$ times. Since $c = O(\frac{1}{\varepsilon})$ and each of *shift*() and *decrement*() takes $O(1)$ time, the time bound on *Update*(e) follows. □

6.2.3 Algorithm extensions

In this section, we extend our algorithm to handle variable-size windows. We first give some notations for our discussion. For any position p and size s, let $\mathcal{W}_{p,s}$ denote the window $[p - s + 1..p]$, which is of size s and ends at position p. Given any stream δ, let $\delta[i]$ denote its item at position i, and for any $1 \leq p \leq q$, let $\delta[p..q]$ denote the sequence of items $\delta[p]\delta[p + 1] \ldots \delta[q]$.

In Section 6.2.2, we gave a data structure for identifying (θ, ε)-frequent item sets of a data stream δ over window $W_p = \mathcal{W}_{p,n}$. Below, we show that by increasing the "sampling rate", our data structure would be accurate enough for us to identify, for any $\frac{1}{2}n \leq s \leq n$, a (θ, ε)-frequent item set over $\mathcal{W}_{p,s}$ (i.e., an item set S that contains only item e with $N_\delta(e, \mathcal{W}_{p,s}) > (\theta - \varepsilon)s$, and all items e with $N_\delta(e, \mathcal{W}_{p,s}) \geq \theta s$ are in S). Then, we show that by maintaining a collection of $O(\log n)$ such data structures, we are able to handle any size between 1 and n. Finally, we describe how to maintain this collection dynamically so that we can change the size n of a sliding window during the processing. This new algorithm uses $O(\frac{1}{\varepsilon} \log \varepsilon n)$ space and supports $O(\frac{1}{\varepsilon} \log \varepsilon n)$ query and update times.

Variable-size windows

Observe that for any λ-snapshot (Q, ℓ) over window $W_p = \mathcal{W}_{p,n}$, (Q, ℓ) contains, for each $1 \leq s \leq n$, a unique λ-snapshot (Q_s, ℓ_s) over sub-window $\mathcal{W}_{p,s}$: Q_s is simply the queue

of positions in Q that are in $\mathcal{W}_{p,s}$ and $\ell_s = \ell$. For example, in Figure 6.3, the λ-snapshot $(\langle 10, 19, 22\rangle, 1)$ over $\mathcal{W}_{23,15}$ contains the λ-snapshot $(\langle 19, 22\rangle, 1)$ over $\mathcal{W}_{23,9} = [15, 23]$. To estimate the number of 1-bits over $\mathcal{W}_{p,s}$, we extend the definition of the value of a λ-counter C as follows.

> Suppose that C is storing the λ-snapshot (Q, ℓ) over $\mathcal{W}_{p,n}$. For any $1 \leq s \leq n$, define the value of C over $\mathcal{W}_{p,s}$ to be $v(C, s) = \lambda|Q_s| + \ell_s$ where (Q_s, ℓ_s) is the λ-snapshot over $\mathcal{W}_{p,s}$ contained in (Q, ℓ).

The following lemma extends Lemma 6.11.

Lemma 6.19. *If the λ-counter C is storing the λ-snapshot of bit stream δ over window $\mathcal{W}_{p,n}$, then, for each $1 \leq s \leq n$, $N_\delta(1,\mathcal{W}_{p,s}) \leq v(C, s) < N_\delta(1,\mathcal{W}_{p,s}) + \lambda$.*

Proof. Replace n by s in the proof of Lemma 6.11. □

The next lemma, which generalizes Lemma 6.17, shows that by reducing λ from $\frac{\varepsilon}{4}n$ to $\frac{\varepsilon}{16}n$, the data structure $\texttt{BD}(n, \lambda, c)$ is accurate enough to find (θ, ε)-frequent item sets over many sub-windows of $\mathcal{W}_{p,n}$. Its proof is almost identical to that of Lemma 6.17.

Lemma 6.20. *Let $\lambda = \frac{\varepsilon}{16}n$. Suppose that we have used $\texttt{BD}(n, \lambda, c) = \texttt{BD}(n, \frac{\varepsilon}{16}n, c)$ to process the item stream $\delta = e_1 e_2 \ldots e_p$ by calling Update(e_1), Update(e_2), ..., Update(e_p), and we have made D_s batches of decrement() during the last s updates (i.e., from Update(e_{p-s+1}) to Update(e_p)). Then, after the last operation Update(e_p), the followings hold for any $\frac{n}{2} \leq s \leq n$:*

(i) *For each item e, $N_\delta(e,\mathcal{W}_{p,s}) - D_s \leq v(C_e, s) < N_\delta(e,\mathcal{W}_{p,s}) + \lambda$.*

(ii) *The set $S = \left\{e \;\middle|\; v(C_e, s) \geq (\theta - \varepsilon)s + \lambda\right\}$ is a (θ, ε)-frequent item set of δ over $\mathcal{W}_{p,s}$.*

Proof. Consider any item e. Lemma 6.14 asserts that the λ-counter C_e is storing the λ-snapshot of bit stream $\delta'_e \leq \delta_e$ over $\mathcal{W}_{p,n}$. Together with Lemma 6.19, we conclude that $v(C_e, s) \leq N_{\delta'_e}(1,\mathcal{W}_{p,s}) + \lambda \leq N_{\delta_e}(1,\mathcal{W}_{p,s}) + \lambda = N_\delta(e,\mathcal{W}_{p,s}) + \lambda$. On the other hand, if we make D_s batches of *decrement*() during Update$(e_{p-s+1}), \ldots,$ Update(e_p), we change at most D_s 1-bits in $\delta_e[p - s + 1..p]$ to 0-bits, and by Lemma 6.19 again, we have $v(C_e, s) \geq N_{\delta'_e}(1,\mathcal{W}_{p,s}) \geq N_{\delta_e}(1,\mathcal{W}_{p,s}) - D_s = N_\delta(e,\mathcal{W}_{p,s}) - D_s$. Statement (i) follows.

For Statement (ii), note that every item $e \in S$ satisfies $N_\delta(e,\mathcal{W}_{p,s}) > v(C_e,s) - \lambda \geq (\theta-\varepsilon)s$ (the first inequality is from Statement (i) and the second follows from the definition of S.) On the other hand, by Statement (i), $D_s \leq \frac{n+s}{c} + \lambda$ (Lemma 6.16), and the fact that $c = \frac{4}{\varepsilon}$, $\lambda = \frac{\varepsilon}{16}n$ and $n \leq 2s$, we have, for any item e with $N_\delta(e,\mathcal{W}_{p,s}) \geq \theta s$, $v(C_e,s) \geq N_\delta(e,\mathcal{W}_{p,s}) - D_s \geq N_\delta(e,\mathcal{W}_{p,s}) - (\frac{n+s}{c} + \lambda) = N_\delta(e,\mathcal{W}_{p,s}) - (\frac{n+s}{c} + 2\lambda) + \lambda \geq \theta s - \varepsilon s + \lambda$, and hence e is in S. It follows that S is a (θ,ε)-frequent item set over $\mathcal{W}_{p,s}$. □

Note that in Lemma 6.20, we can treat n as a parameter. By substituting n with $n/2$, the lemma asserts that $\mathtt{BD}(\frac{n}{2}, \frac{\varepsilon}{16}\frac{n}{2}, c)$ enables us to find (θ,ε)-frequent item sets over $\mathcal{W}_{p,s}$ for any $\frac{n}{4} \leq s \leq \frac{n}{2}$. Therefore, by maintaining both $\mathtt{BD}(n, \frac{\varepsilon}{16}n, c)$ and $\mathtt{BD}(\frac{n}{2}, \frac{\varepsilon}{16}\frac{n}{2}, c)$, we would be able to find (θ,ε)-frequent item sets over $\mathcal{W}_{p,s}$ for any size $\frac{n}{4} \leq s \leq n$. This leads us to the following implementation for finding (θ,ε)-frequent item sets over $\mathcal{W}_{p,s}$ for any size $1 \leq s \leq n$.

> Let $\hat{\varepsilon} = \frac{\varepsilon}{16}$ and h be an integer with $n \leq 2^h$. Let $\mathtt{BD}^i$ denote the data structure $\mathtt{BD}(n_i, \lambda_i, c) = \mathtt{BD}(2^i, \hat{\varepsilon}2^i, c)$. We maintain the following set of data structures
>
> $$\mathtt{BD}^* = \left\{\mathtt{BD}^i \mid \lceil \log \tfrac{1}{\hat{\varepsilon}} \rceil \leq i \leq h\right\},$$
>
> as well as a queue Q^* storing the last $\lceil \frac{1}{\hat{\varepsilon}} \rceil$ items over the sliding window[2].

Note that each $\mathtt{BD}^i$ enables us to find (θ,ε)-frequent item sets over $\mathcal{W}_{p,s}$ for any $s \in [2^{i-1}..2^i]$ (Lemma 6.20) and hence $\mathtt{BD}^*$ can handle any size $s \in \bigcup_{\lceil \log \frac{1}{\hat{\varepsilon}} \rceil \leq i \leq h}[2^{i-1}..2^i]$, which includes all sizes from $\frac{1}{\hat{\varepsilon}}$ to n. For any $1 \leq s < \frac{1}{\hat{\varepsilon}}$, we can find the frequent item set over $\mathcal{W}_{p,s}$ by scanning Q^* directly. Therefore, by maintaining $\mathtt{BD}^*$ and Q^*, we can find (θ,ε)-frequent items set over $\mathcal{W}_{p,s}$ for any $1 \leq s \leq n$. Note that $\mathtt{BD}^*$ has space $O(\frac{1}{\varepsilon}\log \varepsilon n)$ and supports $O(\frac{1}{\varepsilon}\log \varepsilon n)$ query and update times.

Maintaining $\mathtt{BD}^*$ dynamically

It is straightforward to maintain $\mathtt{BD}^* = \{\mathtt{BD}^i \mid \lceil \log \frac{1}{\hat{\varepsilon}} \rceil \leq i \leq h\}$ if h is fixed; to process the item stream $\delta = e_1 e_2 \ldots$, we simply call $\mathit{Update}(e_1)$, $\mathit{Update}(e_2), \ldots$ for each $\mathtt{BD}^i \in \mathtt{BD}^*$. In this section, we describe how to maintain $\mathtt{BD}^*$ dynamically. In particular, we show

[2]Note that we do not keep $\mathtt{BD}^i$ for any $i < \lceil \lg \frac{1}{\hat{\varepsilon}} \rceil$ because these data structures cover windows of size smaller than $\frac{1}{\hat{\varepsilon}}$, and Q^* is storing all the items in these windows.

how to construct the next data structure $\mathtt{BD}^{h+1}$ for $\mathtt{BD}^*$ during the processing. Obviously, we cannot construct $\mathtt{BD}^{h+1}$ from scratch because most of the items necessary for its construction may have gone. Our idea is to construct $\mathtt{BD}^{h+1}$ from $\mathtt{BD}^h$.

Note that a fundamental difference between $\mathtt{BD}^h$ and $\mathtt{BD}^{h+1}$ is that the counters in $\mathtt{BD}^h$ concern only the last 2^h arrived items, while those in $\mathtt{BD}^{h+1}$ concern the last 2^{h+1} arrived items. In general, a counter C in $\mathtt{BD}(n, \lambda, c)$ is only interested in the last n arrived items; when processing the pth item in the stream, it would delete any entry from its queue whose position falls outside the window $[p-n+1..p]$. To emphasize this behaviour of C, we say that C has *limit* n. Therefore, every counter C_e^h in $\mathtt{BD}^h$ is an $\hat{\varepsilon}2^h$-counter with limit 2^h. To create $\mathtt{BD}^{h+1}$, we need to construct, for each item e, an $\hat{\varepsilon}2^{h+1}$-counter C_e^{h+1} with limit 2^{h+1}. Our construction is based on the following notion of dilution.

Let (Q, ℓ) be a λ-snapshot of some bit stream δ over $\mathcal{W}_{p,n}$. The *dilution* (Q', ℓ') of (Q, ℓ) is a snapshot defined as follows.

- If $Q =$ the empty queue $\langle\ \rangle$, then $(Q', \ell') = (Q, \ell)$.
- If $Q = \langle i_1 \rangle$, then $(Q', \ell') = (\langle\ \rangle, \lambda + \ell)$.
- If $Q = \langle i_1, i_2, \ldots, i_k \rangle$, then $(Q', \ell') = \begin{cases} (\langle i_2, i_4, \ldots, i_k \rangle, & \ell), \quad \text{if } k \text{ is even;} \\ (\langle i_2, i_4, \ldots, i_{k-1} \rangle, & \lambda + \ell), \quad \text{otherwise.} \end{cases}$

It is easy to verify that (Q', ℓ') is a (2λ)-snapshot of δ over $\mathcal{W}_{p,n}$ and

$$2\lambda|Q'| + \ell' = \lambda|Q| + \ell. \tag{6.6}$$

We create $\mathtt{BD}^{h+1}$ from $\mathtt{BD}^h$ by diluting every counter in $\mathtt{BD}^h$ as follows.

Dilute $(\mathtt{BD}^h)$:

1: **for** each counter C_e^h in $\mathtt{BD}^h$ with $v(C_e^h) > 0$ **do**[3]
2: Create an $\hat{\varepsilon}2^{h+1}$-counter C_e^{h+1} with limit 2^{h+1} storing the dilution (Q', ℓ') of the snapshot (Q, ℓ) currently stored in C_e^h;
3: Add C_e^{h+1} to $\mathtt{BD}^{h+1}$.

It is important to note that if (Q, ℓ) is an $\hat{\varepsilon}2^h$-snapshot over $\mathcal{W}_{p,2^h}$, then its dilution (Q', ℓ') is an $\hat{\varepsilon}2^{h+1}$-snapshot over $\mathcal{W}_{p,2^h}$, not over $\mathcal{W}_{p,2^{h+1}}$. Hence, a newly created C_e^{h+1}

[3] If $v(C_e^h) = 0$, then $v(C_e^{h+1}) = 0$ and we do not need to store the counter physically in $\mathtt{BD}^{h+1}$.

cannot give accurate estimate over $\mathcal{W}_{p,2^{h+1}}$ even though its limit is 2^{h+1}. We say that C_e^{h+1} has *coverage* g if it is storing a snapshot over some window of size g. Obviously, $g \leq 2^{h+1}$. The following fact promises that after 2^h shift operations, a newly created C_e^{h+1} will have its coverage increased from 2^h to its full coverage 2^{h+1}.

Fact 6.21. *Suppose that a newly created counter C_e^{h+1} is storing the $\hat{\varepsilon}2^{h+1}$-snapshot of bit stream $\delta[1..p]$ over $\mathcal{W}_{p,2^h}$. Then, after calling shift$(\delta[p+1])$, shift$(\delta[p+2]), \ldots,$ shift$(\delta[p+r])$, the counter will be storing the $\hat{\varepsilon}2^{h+1}$-snapshot of bit stream $\delta[1..(p+r)]$ over window $\mathcal{W}_{p+r,g}$ where $g = \min\{2^h + r, 2^{h+1}\}$.*

To show that $\mathtt{BD}^*$ is accurate enough for finding (θ, ε)-frequent item sets even if it is maintained dynamically, suppose that we are processing the item stream $\delta = e_1 e_2 e_3 \ldots$. Let $i_o = \lceil \log \frac{1}{\hat{\varepsilon}} \rceil$. Recall that for each item e, δ_e is the bit stream where $\delta_e[i] = 1$ if $\delta[i] = e_i = e$. Throughout the processing, we maintain a queue Q^* storing the last $\lceil \frac{1}{\hat{\varepsilon}} \rceil$ arrived items. Suppose that we create $\mathtt{BD}^{i_o}$ from Q^* just after processing the item $e_{p_{i_o}}$, and for any $i_o < i \leq h$, we create $\mathtt{BD}^i$ by calling *Dilute*$(\mathtt{BD}^{i-1})$ just after processing the item e_{p_i}. We assume that when we call *Dilute*$(\mathtt{BD}^{i-1})$, all counters in $\mathtt{BD}^{i-1}$ have full coverage 2^{i-1}. (We will explain later that this is not a serious restriction.) To process a newly arrived item e_p in δ, we update Q^* and call *Update*(e_p) for each $\mathtt{BD}^i$ in $\mathtt{BD}^*$.

Lemma 6.22. *Suppose that we maintain Q^* and $\mathtt{BD}^*$ as described above. Then, for any $i \geq i_o$, after creating $\mathtt{BD}^i$ at p_i and executing Update$(e_{p_i+1}), \ldots,$ Update(e_p), each counter C_e^i in $\mathtt{BD}^i$ will be storing the $\hat{\varepsilon}2^i$-snapshot of some bit stream δ_e^i over $\mathcal{W}_{p,g_i} = \mathcal{W}_{p,\min\{2^{i-1}+p-p_i,2^i\}}$ where*

(i) *$\delta_e^i[1..p] \leq \delta_e[1..p]$, and*

(ii) *for any $1 \leq s \leq g_i$, there are at most $\frac{2^i+s}{c} + \lambda_i = \frac{2^i+s}{c} + \hat{\varepsilon}2^i$ positions $u \in [p-s+1..p]$ with $\delta_e^i[u] \neq \delta_e[u]$.*

Proof. We prove the lemma by induction. For the base case, note that $\mathtt{BD}^{i_o}$ is created from Q^*, which stores all the necessary items for us to construct the counters in $\mathtt{BD}^{i_o}$ from scratch. Thus, $\mathtt{BD}^{i_o}$ is created and maintained normally without using dilution, and as in Section 6.2.2, we can prove that (i) and (ii) hold for $\mathtt{BD}^{i_o}$. Suppose that the lemma is true for $\mathtt{BD}^{i_o}, \ldots, \mathtt{BD}^{i-1}$, and we consider $\mathtt{BD}^i$, which is created by calling *Dilute*$(\mathtt{BD}^{i-1})$ after processing e_{p_i}. Recall that each counter C_e^{i-1} in $\mathtt{BD}^{i-1}$ is assumed to have full coverage at this moment and thus it is storing the $\hat{\varepsilon}2^{i-1}$-snapshot of δ_e^{i-1}

over $\mathcal{W}_{p_i,2^{i-1}}$. It follows that the counter C_e^i in the newly created $\texttt{BD}^i$ is storing the $\hat{\varepsilon}2^i$-snapshot of δ_e^{i-1} over $\mathcal{W}_{p_i,2^{i-1}}$. During *Update*$(e_{p_i+1})$, *Update*$(e_{p_i+2}), \ldots,$ *Update*(e_p), we execute *shift*$(\delta_e[p_i+1])$, *shift*$(\delta_e[p_i+2]), \ldots,$ *shift*$(\delta_e[p])$ on C_e^i. If there is no *decrement*() operation, then we conclude from Fact 6.21 that after these *shift* operations, C_e^i would be storing the $\hat{\varepsilon}2^i$-snapshot of

$$\delta_e^i[1..p] = \delta_e^{i-1}[1..p_i]\delta_e[p_i+1..p] \tag{6.7}$$

over $\mathcal{W}_{p,g_i}$. If there are *decrement*() operations, some bits of the stream in (6.7) may be reset. Together with $\delta_e^{i-1}[1..p_i] \leq \delta_e[1..p_i]$ (by induction hypothesis), (i) follows.

To prove (ii), we consider two cases.

1. $p-s+1 > p_i$: From (i), we conclude that after *Update*(e_{p-s}), each counter C_e^i in $\texttt{BD}^i = \texttt{BD}(2^i, \hat{\varepsilon}2^i, c)$ is storing the $\hat{\varepsilon}2^i$-snapshot of $\delta_e^i[1..p-s] \leq \delta_e[1..p-s]$ over window $\mathcal{W}_{p-s,g} = \mathcal{W}_{p-s,\min\{2^i,2^{i-1}+p-s-p_i\}}$. Together with the fact that there are always no more than c counters in $\texttt{BD}^i$ with positive values, we conclude that the total value of these counters immediately after *Update*(e_{p-s}) is

$$\sum_{v(C_e^i)>0} v(C_e^i) \leq \sum_{v(C_e^i)>0} \big(N_\delta(e, \mathcal{W}_{p-s,g}) + \lambda_i\big) = |\mathcal{W}_{p-s,g}| + \sum_{v(C_e^i)>0} \hat{\varepsilon}2^i \leq 2^i + c\hat{\varepsilon}2^i \ .$$

Together with the s units added to the counters during the processing of $e_{p-s+1}, \ldots, e_p$, there are at most $2^i + c\hat{\varepsilon}2^i + s$ units for the batch of *decrement*() operations to take away during *Update*$(e_{p-s+1}), \ldots,$ *Update*(e_p). It follows that there are at most $\frac{2^i+s}{c} + \hat{\varepsilon}2^i$ batches of *decrement* made during this period, and they will reset at most $\frac{2^i+s}{c} + \hat{\varepsilon}2^i$ bits in $\delta_e^i[p-s+1..p]$, (ii) follows.

2. $p-s+1 \leq p_i$: We first estimate the number of batches of *decrement*() made by $\texttt{BD}^i$ after its creation. Note that

   - Equation (6.6) asserts that each newly created counter C_e^i in $\texttt{BD}^i$ has its value equal to that of C_e^{i-1}, and
   - (i) asserts that C_e^{i-1} is storing the $\hat{\varepsilon}2^{i-1}$-snapshot of some bit stream $\delta_e^{i-1}[1..p_i] \leq \delta_e[1..p_i]$ over $\mathcal{W}_{p_i,2^{i-1}}$ at this moment.

Thus, we have

$$\sum_e v(C_e^i) = \sum_e v(C_e^{i-1}) \leq \sum_{v(C_e^{i-1})>0} \big(N_\delta(e, \mathcal{W}_{p_i,2^{i-1}}) + \lambda_{i-1}\big) \leq 2^{i-1} + c\hat{\varepsilon}2^{i-1}.$$

The operations *Update*$(e_{p_i+1}), \ldots,$ *Update*(e_p) will add another $p-p_i$ units to these counters for the batches of *decrement*() to take away. It follows that there are at most $\frac{2^{i-1}+p-p_i}{c} + \hat{\varepsilon}2^{i-1}$ batches of *decrement*() made to $\texttt{BD}^i$ since its creation.

Consider any counter C_e^i in $\texttt{BD}^i$. Again, if there is no batch of *decrement*, C_e^i will be storing the $\hat{\varepsilon}2^i$-snapshot of the bit stream $\delta_e^i[1..p] = \delta_e^{i-1}[1..p_i]\delta_e[p_i..p]$ after *Update*(e_p). Recall that $s \leq g_i = \min\{2^i, 2^{i-1}+p-p_i\}$, and this implies $s-(p-p_i) \leq 2^{i-1}$. Thus, we can apply the induction hypothesis on $\texttt{BD}^{i-1}$[4] and asserts that in the window $[p-s+1..p_i] = [[p_i-(s-(p-p_i))+1..p_i]$, there are at most $\frac{2^{i-1}+s-(p-p_i)}{c} + \hat{\varepsilon}2^{i-1}$ positions u with $\delta_e^{i-1}[u] \neq \delta_e[u]$. Together with the possible $\frac{2^{i-1}+p-p_i}{c} + \hat{\varepsilon}2^{i-1}$ bits reset by the *decrement* operations made during *Update*$(e_{p_i+1}), \ldots,$ *Update*(e_p), there are at most

$$\tfrac{2^{i-1}+s-(p-p_i)}{c} + \hat{\varepsilon}2^{i-1} + \tfrac{2^{i-1}+p-p_i}{c} + \hat{\varepsilon}2^{i-1} = \tfrac{2^i+s}{c} + \hat{\varepsilon}2^i$$

positions $u \in [p_i - s + 1..p_i]$ with $\delta_e^i[u] \neq \delta_e[u]$; (ii) follows. □

The following lemma generalizes Lemma 6.20; the correctness of our approach follows from the lemma directly.

Lemma 6.23. *Suppose that we have executed* Update$(e_{p_i+1}), \ldots,$ Update(e_p) *after the creation of* $\texttt{BD}^i$ *at* p_i. *Let* $g_i = \min\{2^{i-1}+p-p_i, 2^i\}$. *Then, for any* $2^{i-1} \leq s \leq g_i$, *we have the following:*

(i) *For each item* e, *the counter* $C_e^i \in \texttt{BD}^i$ *satisfies*

$$N_\delta(e,\mathcal{W}_{p,s}) - (\tfrac{2^i+s}{c} + \lambda_i) \leq v(C_e^i, s) \leq N_\delta(e,\mathcal{W}_{p,s}) + \lambda_i.$$

(ii) *The set* $S = \big\{e \;\big|\; v(C_e^i, s) \geq (\theta-\varepsilon)s + \lambda_i\big\}$ *is a* (θ,ε)*-frequent item set of* δ *over* $\mathcal{W}_{p,s}$.

[4]Recall that we assume $\texttt{BD}^{i-1}$ has full coverage 2^{i-1} when we call *Dilute*$(\texttt{BD}^{i-1})$ to create $\texttt{BD}^i$ after processing e_{p_i}.

Proof. Consider any item e. From Lemma 6.22, we conclude that C_e^i is storing the λ_i-snapshot of bit stream $\delta_e^i \leq \delta_e$ over $\mathcal{W}_{p,g_i}$. Together with Lemma 6.19, we conclude that

$$v(C_e, s) \leq N_{\delta_e^i}(1,\mathcal{W}_{p,s}) + \lambda_i \leq N_{\delta_e}(1,\mathcal{W}_{p,s}) + \lambda_i = N_\delta(e,\mathcal{W}_{p,s}) + \lambda_i.$$

On the other hand, Lemma 6.22 also asserts that there are at most $\frac{2^i+s}{c} + \lambda_i$ positions $u \in [p-s+1, p]$ with $\delta_e^i[u] \neq \delta_e[u]$, and with Lemma 6.19 again, we have

$$v(C_e, s) \geq N_{\delta_e^i}(1,\mathcal{W}_{p,s}) \geq N_{\delta_e}(1,\mathcal{W}_{p,s}) - (\tfrac{2^i+s}{c} + \lambda_i) = N_\delta(e,\mathcal{W}_{p,s}) - (\tfrac{2^i+s}{c} + \lambda_i),$$

and (i) follows. Given (i), we can prove (ii) as in the proofs of Lemma 6.20. □

Remark. Recall that we assume that $\texttt{BD}^i$ has full coverage when we call *Dilute*$(\texttt{BD}^i)$. Note that this is not a serious restriction because if $\texttt{BD}^i$ does not have full coverage, then any arrival of a new item will automatically increase its coverage by one; we do not need to dilute the counter to increase the size of sliding window. Furthermore, it is trivial to reduce the sliding window size, and to save space, we can throw away $\texttt{BD}^h$ from $\texttt{BD}^*$ as soon as the sliding window size becomes smaller than 2^{h-1}.

Chapter 7

Communication-efficient Data Stream Algorithms

In this chapter, we study communication-efficient algorithms for continuous monitoring of multiple, distributed data streams. The formal problem is as follows. We have $k \geq 1$ remote sites each monitoring a data stream, and there is a root (or coordinator) responsible for computing some global statistics. The data stream at each remote site is a sequence of items from a totally ordered set U. Each item is associated with an integral time-stamp recording its creation time. A remote site needs to maintain certain statistics itself, and has to communicate with the root often enough so that the root can compute, at any time, the statistics of the union of all data streams within a certain error $\varepsilon \in (0, 1)$. We focus on time-based sliding window, where given a positive integer W as the window size, the statistics is computed on all items whose time-stamps are within the last W time units. Note that algorithms for time-based sliding window are applicable to both the models of whole data stream and count-based sliding window. We study the four classical ε-approximate queries, namely, basic counting, approximate counting, frequent items and quantiles, as defined below. For any stream σ, let $c_{j,\sigma}$ and c_σ be the count of item j and all items, respectively, in the current window. Denote $c_j = \sum_\sigma c_{j,\sigma}$ and $c = \sum_\sigma c_\sigma$ as the total count of item j and all items in the streams from the remote sites, respectively.

- *Basic Counting.* Return an estimate $\hat{c}$ on the total count c such that $|\hat{c} - c| \leq \varepsilon c$. (Note that this query can be generalized to count data items of a fixed subset

$X \subseteq U$; the literature often refers to the special case with $U = \{0, 1\}$ and $X = \{1\}$.)

- *Approximate Counting.* Given any item j, return an estimate $\hat{c}_j$ such that $|\hat{c}_j - c_j| \leq \varepsilon c$. (Note that this query gives estimate for any item, not just the frequent items.)
- *Frequent Items.* Given any $0 < \phi < 1$, return a set $F \subseteq U$ which includes all items j with $c_j \geq \phi c$ and possibly some items j' with $c_{j'} \geq \phi c - \varepsilon c$.
- *Quantiles.* Given any $0 < \phi < 1$, return an item whose rank is in $[\phi c - \varepsilon c, \phi c + \varepsilon c]$ among the c items in the current sliding window.

To handle these queries, we need an algorithm to determine when and how the remote sites communicate with the root, so that the root can answer the queries at any time. The objective is to minimize the worst-case communication cost by any remote site within a window of W time units.

Our results. Consider a sliding window of W time units and let B be the maximum number of items arriving at each stream within a window. We prove that for basic counting, any remote site needs to communicate $\Omega(\frac{1}{\varepsilon}\log(\varepsilon B))$ bits with the root within a window of W time units in the worst case, and $\Omega(\frac{1}{\varepsilon}\log(\varepsilon B))$ words for the other three queries. For upper bounds, our analysis shows that basic counting requires $O(\frac{1}{\varepsilon}\log(\varepsilon B))$ bits within any window, and approximate counting $O(\frac{1}{\varepsilon}\log B)$ words. Note that the estimates given by approximate counting is sufficient to find frequent items, and thus the latter problem has the same communication cost. For quantiles, it takes $O(\frac{1}{\varepsilon^2}\log B)$ words. Note that all our algorithms do not need to know the value of B in advance, it is only needed in the analysis.

Out-of-order streams. Our algorithms can be readily applied to *out-of-order* streams, where items may arrive in any order of their time-stamps. This is in contrast to an *in-order* streams, where items arrive in non-decreasing order of their time-stamps. Recall that for an out-of-order stream, we say that the stream has *tardiness* τ if any item with time-stamp t must arrive within τ time units from t, i.e., at any time in $[t, t+\tau]$. Without loss of generality, we assume that $\tau \in \{0, 1, 2, \ldots, W-1\}$ (if an item time-stamped at t arrives after $t+W-1$, it has already expired and can be ignored). Note that in-order data streams have tardiness 0. The previous lower bounds for in-order streams are all valid in the out-of-order setting. In addition, we obtain a lower bound related to τ, namely,

$\Omega(\frac{W}{W-\tau})$ bits for basic counting and $\Omega(\frac{W}{W-\tau})$ words for the other three problems. Regarding upper bounds, our algorithms when applied to out-of-order streams with tardiness τ will just increase the communication cost by a factor of $\frac{W}{W-\tau}$.

We first present upper bound results. Assuming in-order streams, Sections 7.1 and 7.2 analyzes the algorithm for basic counting and approximate counting, respectively. Section 7.3 discusses frequent items, quantiles, and out-of-order streams. Finally, Section 7.4 gives the lower bound results.

7.1 Basic Counting

This section presents a simple algorithm for each individual stream to communicate to the root so that the root can answer at any time an ε-approximate basic counting query over all streams. Recall that the sliding window includes W time units and B denotes the maximum number of items arriving within a window. Below we let t denote the current time, and let $t_o = t - W + 1$ be the starting time of the current window. We first give an algorithm that requires each stream to send $O(\frac{1}{\varepsilon}\log(\varepsilon B))$ words in a window. Then we exploit an "estimate discretization" technique to reduce the communication to $O(\frac{1}{\varepsilon}\log(\varepsilon B))$ bits, which matches the lower bound.

Let $0 < \lambda < 1/9$ be the local error parameter (to be set to $\varepsilon/9$ later). For each stream σ, we maintain a λ-approximate data structure [35] locally that can report an estimate $\hat{c}(t)$ of the count $c(t)$ of the total items arriving in the current window such that $(1-\lambda)c(t) \le \hat{c}(t) \le (1+\lambda)c(t)$. The following algorithm specifies when and what message the stream σ sends to the root.

Algorithm BC. At any time t, let $p < t$ be the last time when an estimate $\hat{c}(p)$ is sent to the root. The stream σ sends $\hat{c}(t)$ to the root if one of the following events occur.

- **Up**: $\hat{c}(t) > (1 + 4\lambda)\,\hat{c}(p)$.
- **Down**: $\hat{c}(t) < (1 - 4\lambda)\,\hat{c}(p)$.

The root's perspective. Suppose the root is required to estimate the total count in the current window over all streams within an error of ε, where $0 < \varepsilon < 1$. We set

$\lambda = \varepsilon/9$ and make all streams to use the communication algorithm BC. At any time t, let $r_\sigma(t)$ be the last estimate sent by stream σ at or before time t, then the root can estimate within an error of ε, the total count by summing $r_\sigma(t)$ over all streams. This is because for each stream σ, BC ensures that $\frac{1}{1+4\lambda}\hat{c}(t) \leq r_\sigma(t) \leq \frac{1}{1-4\lambda}\hat{c}(t)$; otherwise the stream σ would have sent a new estimate to the root. With respect to σ, $\hat{c}(t)$ is an λ-approximation of $c(t)$, and $r_\sigma(t) \leq \frac{1}{1-4\lambda}\hat{c}(t) \leq \frac{1+\lambda}{1-4\lambda}c(t) \leq (1+9\lambda)c(t) = (1+\varepsilon)c(t)$ (recall that $\lambda = \varepsilon/9 \leq 1/9$). Similarly, we can show that $r_\sigma(t) \geq (1-\varepsilon)c(t)$.

Communication complexity. Below we first show that each stream σ has at most $O(\frac{1}{\lambda}\log(\lambda B))$ ups or downs in a window. Thus the stream sends at most $O(\frac{1}{\lambda}\log(\lambda B))$ words per window. For any time $t_1 \leq t_2$, it is useful to define $\sigma_{[t_1,t_2]}$ (resp. $\sigma_{j,[t_1,t_2]}$) as the multi-set of all items (resp. item j only) arriving at σ within $[t_1, t_2]$, and $|\sigma_{[t_1,t_2]}|$ as the size of this multi-set.

Suppose there are m ups in the current window $[t_o, t]$, occurring at times $t_1 < t_2 < \cdots < t_m$. To upper bound m, the key idea is to associate each t_i with a suitable characteristic set S_i of items. A natural choice would be those items that contribute to $c(t_i)$, i.e., $\sigma_{[t_i-W+1,t_i]}$, yet this is not useful. Instead we define S_i as the items arriving within $[t_o, t_i]$, i.e., $S_i = \sigma_{[t_o,t_i]}$. Then we can show that S_i increases by at least a multiplicative factor of $1+\lambda$ as i increases.

Lemma 7.1. *For any $2 \leq i \leq m$, $|S_i| > (1+\lambda)|S_{i-1}|$.*

Proof. For any $2 \leq i \leq m$, let $p_i < t_i$ be the latest time when an up or down occurs. Since there is an up at t_i, we have $\hat{c}(t_i) > (1+4\lambda)\hat{c}(p_i) \geq (1+4\lambda)(1-\lambda)c(p_i)$. Furthermore, $\hat{c}(t_i) \leq (1+\lambda)c(t_i)$. Thus, $c(t_i) > \frac{(1+4\lambda)(1-\lambda)}{1+\lambda}c(p_i) \geq (1+\lambda)c(p_i)$, and $c(t_i) - c(p_i) > \lambda c(p_i)$. Note that the extra items come from $\sigma_{[p_i+1,t_i]}$. We conclude that $|S_i| - |S_{i-1}| = |\sigma_{[t_o,t_i]}| - |\sigma_{[t_o,t_{i-1}]}| \geq |\sigma_{[p_i+1,t_i]}| \geq c(t_i) - c(p_i) > \lambda c(p_i) = \lambda|\sigma_{[p_i-W+1,p_i]}| \geq \lambda|\sigma_{[t_o,p_i]}| \geq \lambda|\sigma_{[t_o,t_{i-1}]}| = \lambda|S_{i-1}|$. □

As $|S_m| \leq B$, Lemma 7.1 implies that $m = O(\frac{1}{\lambda}\log B)$. Below is a tighter analysis.

Corollary 7.2. $m = O(\frac{1}{\lambda}\log(\lambda B))$.

Proof. Let ℓ be the smallest integer such that $|S_\ell| \geq 1/\lambda$ (if no such ℓ exists, let $\ell = m+1$). Since $|S_i| > (1+\lambda)|S_{i-1}|$ for $2 \leq i \leq m$, we have $0 \leq |S_1| < |S_2| < \cdots < |S_{\ell-1}| < \frac{1}{\lambda}$,

and $B \geq |S_m| > (1+\lambda)|S_{m-1}| > \cdots > (1+\lambda)^{m-\ell}|S_\ell| \geq (1+\lambda)^{m-\ell}(\frac{1}{\lambda})$. This implies that $\ell \leq \frac{1}{\lambda} + 1$ and $m - \ell \leq \log_{1+\lambda}(\lambda B)$, and $m = O(\frac{1}{\lambda} + \frac{1}{\lambda}\log(\lambda B))$. □

The analysis of down events is symmetric. For an item with time-stamp t, we define the *first expiry time* to be $t + W$, and it is said to expire after $t + W - 1$. Suppose there are m' downs in $[t_o, t]$, occurring at times $d_1 < d_2 < \cdots < d_{m'}$. Define the characteristic set H_i of each d_i as those items whose first expiry time is within $[d_i + 1, t]$, or equivalently, $H_i = \sigma_{[d_i - W + 1, t_o - 1]}$. Similar to Lemma 7.1, we can prove that H_i decreases by a constant factor as i increases.

Lemma 7.3. *For any* $2 \leq i \leq m'$, $|H_i| < \frac{1}{1+\lambda}|H_{i-1}|$. *Furthermore,* $m' = O(\frac{1}{\lambda}\log(\lambda B))$.

From $O(\frac{1}{\lambda}\log(\lambda B))$ words to $O(\frac{1}{\lambda}\log(\lambda B))$ bits. When using BC, a message sent at time t is the current estimate $\hat{c}(t)$, which occupies up to a word. Thus BC sends $O(\frac{1}{\lambda}\log(\lambda B))$ words in a window. To reduce the message size, we first observe that the data structure kept for each stream σ can be made to produce a "restricted" estimate that is chosen from a "small" fixed set (with only $O(\frac{1}{\lambda}\log(\lambda B))$ numbers), while the desired error bound can still be maintained. Denote this set as $K = \{k_0, k_1, \ldots, k_h\}$. When BC needs to send an estimate $k_y \in K$ to the root and the last estimate that should be sent is k_x, it simply communicates the difference of the indices $(y - x)$. This requires $\log(y - x)$ bits, reducing the communication cost substantially.

Intuitively, different definitions of K would give different tradeoff between accuracy and communication. The following definition optimizes the communication, while meeting the desired error bound. Let $\alpha = \lceil \frac{3}{\lambda} \rceil$ and let d be the smallest integer such that $(1 + \frac{\lambda}{2})^d \alpha \geq B$. Define $K = \{0, 1, 2, \ldots, \alpha\} \cup \{(1 + \frac{\lambda}{2})\alpha, (1 + \frac{\lambda}{2})^2\alpha, \ldots, (1 + \frac{\lambda}{2})^d\alpha\}$. Note that $|K| = O(\frac{1}{\lambda} + \frac{1}{\lambda}\log(\lambda B))$.

Lemma 7.4. *We can keep a data structure for* σ*, which at any time* t *can give an estimate* $\hat{c}(t)$ *such that* $\hat{c}(t) \in K$ *and* $(1 - \lambda)c(t) \leq \hat{c}(t) \leq (1 + \lambda)c(t)$.

Proof. We keep the data structure of [35] with a better error bound of $\frac{\lambda}{3}$. Hence, it can return an estimate $\tilde{c}(t)$ such that $(1 - \frac{\lambda}{3})c(t) \leq \tilde{c}(t) \leq (1 + \frac{\lambda}{3})c(t)$. We can assume $\tilde{c}(t) \leq B$ and we define $\hat{c}(t)$ from $\tilde{c}(t)$ as follows. If $\tilde{c}(t) \leq \alpha$, we simply let $\hat{c}(t) = \tilde{c}(t)$ and $\hat{c}(t)$ satisfies the requirements obviously. Otherwise, let $\hat{c}(t) = (1 + \frac{\lambda}{2})^i \alpha$, where i is

the smallest integer with $(1+\frac{\lambda}{2})^i\alpha \geq \tilde{c}(t)$. Then $\hat{c}(t) \in K$, and

- $\hat{c}(t) = (1+\frac{\lambda}{2})^i\alpha \geq \tilde{c}(t) \geq (1-\frac{\lambda}{3})c(t) \geq (1-\lambda)c(t)$, and
- $\hat{c}(t) = (1+\frac{\lambda}{2})(1+\frac{\lambda}{2})^{i-1}\alpha < (1+\frac{\lambda}{2})\tilde{c}(t) \leq (1+\frac{\lambda}{2})(1+\frac{\lambda}{3})c(t) \leq (1+\lambda)c(t)$. □

Lemma 7.4 guarantees that the local data structure can maintain the required accuracy when restricting the estimates to some values in K. Thus, using the same up and down thresholds, the root can answer the ε-approximate basic counting queries correctly. To analyze the communication cost, we need the following technical lemma to translate the actual distance between $c(t)$ and $\hat{c}(t)$ to a relative distance with respect to K.

Lemma 7.5. *At any time t, let $\hat{c}(t) = k_i$ for some $k_i \in K$. Then, we have $k_{\max\{0,i-4\}} \leq c(t) \leq k_{\min\{i+4,h\}}$.*

Proof. We consider two cases depending on $i \leq \alpha+2$ or $i > \alpha+2$. First assume $i \leq \alpha+2$. By Lemma 7.4, $c(t) \leq \frac{1}{1-\lambda}k_i = k_i + \frac{\lambda}{1-\lambda}k_i$. Note that $\frac{\lambda}{1-\lambda}k_i \leq \frac{\lambda}{1-\lambda}(1+\frac{\lambda}{2})^2\alpha \leq 4$ (since $\lambda < 1/9$ and $i \leq \alpha+2$). Hence $c(t) \leq k_i + 4$. Since $c(t) \leq B \leq k_h$ and $k_i + 4 \leq k_{i+4}$ if $i+4 \leq h$, we conclude that $c(t) \leq k_{\min\{i+4,h\}}$. Similarly, $c(t) \geq \frac{1}{1+\lambda}k_i = k_i - \frac{\lambda}{1+\lambda}k_i$. Using the same argument above, we have $c(t) \geq k_i - 4$ and $c(t) \geq k_{\max\{0,i-4\}}$.

Then assume $i > \alpha+2$. If $i+4 \leq h$, we have $k_{i+4} = (1+\frac{\lambda}{2})^4 k_i \geq \frac{1}{1-\lambda}k_i \geq c(t)$, where the first inequality follows from $\lambda < 1/9$ and the second inequality from Lemma 7.4. Together with $c(t) \leq B \leq k_h$, we conclude $c(t) \leq k_{\min\{i+4,h\}}$. On the other hand, $c(t) \geq \frac{1}{1+\lambda}k_i \geq \frac{1}{1+\lambda}(1+\frac{\lambda}{2})^2 k_{i-2} \geq k_{i-2}$. Thus, $c(t) \geq k_{i-2} > k_{i-4}$ and $c(t) \geq k_{\max\{0,i-4\}}$. □

Suppose there are m ups in the current window $[t_o, t]$, occurring at times $t_1 < t_2 < \cdots < t_m$. To analyze the total number of bits sent due to ups in a window $[t_o, t]$, we need to relate the growth of the characteristic set S_i at each up event time t_i with the difference of indices sent in each message. Recall that we define S_i as the items arriving within $[t_o, t_i]$, i.e., $S_i = \sigma_{[t_o,t_i]}$. Let ℓ be the smallest integer with $|S_\ell| \geq k_{\alpha+8}$. If no such ℓ exists, let $\ell = m+1$.

Lemma 7.6. *Let k_{y_i} and k_{x_i} be the estimates kept by the stream σ at the up event time t_i and at the last time when an up or down occurs before t_i, respectively.* **(i)** *For any $2 \leq i \leq \ell-1$, $|S_i| - |S_{i-1}| \geq (y_i - x_i - 8)$.* **(ii)** *For any $\ell+1 \leq i \leq m$, $|S_i| \geq (1+\frac{\lambda}{2})^{y_i - x_i - 8}|S_{i-1}|$.*

Proof. For any $2 \leq i \leq m$, let $p_i < t_i$ be the latest time when an up or down occurs. The lemma is obviously true if $y_i - x_i - 8 \leq 0$. Below, we assume $y_i - x_i - 8 > 0$. Also note that $|\sigma_{[t_o,t_i]}| - |\sigma_{[t_o,p_i]}| = |\sigma_{[p_i+1,t_i]}| \geq c(t_i) - c(p_i)$.

For (i), consider any $2 \leq i \leq \ell - 1$. By Lemma 7.5, $c(t_i) - c(p_i) \geq k_{\max\{0,y_i-4\}} - k_{\min\{x_i+4,h\}} \geq y_i - x_i - 8$. Thus, $|S_i| - |S_{i-1}| = |\sigma_{[t_o,t_i]}| - |\sigma_{[t_o,t_{i-1}]}| \geq |\sigma_{[t_o,t_i]}| - |\sigma_{[t_o,p_i]}| \geq y_i - x_i - 8$.

For (ii), consider any $\ell + 1 \leq i \leq m$. Let $\Delta_i = (1 + \frac{\lambda}{2})^{y_i - x_i - 8}$. By Lemma 7.5, $k_{\min\{y_i+4,h\}} \geq c(t_i) \geq |S_i| \geq k_{\alpha+8}$, so we have $y_i \geq \alpha + 4$. Similarly, we have $x_i \geq \alpha + 4$. It implies $k_{y_i-4} = (1 + \frac{\lambda}{2})^{(y_i-4)-(x_i+4)} k_{x_i+4} = \Delta_i k_{x_i+4}$. By Lemma 7.5, $c(t_i) \geq k_{y_i-4} = \Delta_i k_{x_i+4} \geq \Delta_i c(p_i)$. Note that $|\sigma_{[t_o,t_i]}| - |\sigma_{t_o,p_i]}| \geq c(t_i) - c(p_i) \geq (\Delta_i - 1)c(p_i) \geq (\Delta_i - 1)|\sigma_{[t_o,p_i]}|$, and hence $|\sigma_{[t_o,t_i]}| \geq \Delta_i |\sigma_{t_o,p_i]}|$. Finally, $|S_i| = |\sigma_{[t_o,t_i]}| \geq \Delta_i |\sigma_{[t_o,p_i]}| \geq \Delta_i |\sigma_{[t_o,t_{i-1}]}| = \Delta_i |S_{i-1}|$. □

With Lemma 7.6, a simple counting would show that $O(\frac{1}{\lambda} \log(\lambda B))$ bits are sent for the up events in the window $[t_o, t]$. The analysis of the down events is again symmetric. Thus we have the following theorem.

Theorem 7.7. *The number of bits sent due to ups and downs in a window is* $O(\frac{1}{\lambda} \log(\lambda B))$.

Proof. We first consider ups. We use the same definitions of x_i and y_i as in Lemma 7.6. The total number of bits sent due to ups is at most

$$\log y_1 + \textstyle\sum_{i=2}^{m} \log(y_i - x_i) \leq y_1 + \sum_{i=2}^{\ell-1} (y_i - x_i) + (y_\ell - x_\ell) + \sum_{i=\ell+1}^{m} (y_i - x_i).$$

Note that y_1 and $(y_\ell - x_\ell)$ are at most $|K| = O(\frac{1}{\lambda} \log(\lambda B))$. We bound the remaining two terms as follows.

- For $\sum_{i=2}^{\ell-1} (y_i - x_i)$, we consider the non-trivial case that $\ell \geq 3$. By Lemma 7.6(i), $\sum_{i=2}^{\ell-1} (y_i - x_i - 8) \leq \sum_{i=2}^{\ell-1} (|S_i| - |S_{i-1}|) = |S_{\ell-1}| - |S_1| < k_{\alpha+8} = (1 + \frac{\lambda}{2})^8 \alpha \leq 2\alpha$ for $\lambda < 1/9$. Thus, $\sum_{i=2}^{\ell-1} (y_i - x_i) \leq 2\alpha + 8(\ell - 2) = 2\lceil \frac{3}{\lambda} \rceil + 8(\ell - 2) = O(\frac{1}{\lambda} + \ell)$.
- For $\sum_{i=\ell+1}^{m} (y_i - x_i)$, we consider the non-trivial case that $m \geq \ell + 2$. By Lemma 7.6(ii), $B \geq |S_m| \geq \Delta_m |S_{m-1}| \geq \cdots \geq (\prod_{i=\ell+1}^{m} \Delta_i) |S_\ell| \geq (\prod_{i=\ell+1}^{m} \Delta_i) k_{\alpha+8}$ where $\Delta_i = (1 + \frac{\lambda}{2})^{y_i - x_i - 8}$. It follows that

$$\textstyle\prod_{i=\ell+1}^{m} \Delta_i = (1 + \frac{\lambda}{2})^{\sum_{i=\ell+1}^{m} (y_i - x_i - 8)} \leq B / k_{\alpha+8},$$

$$\text{and } \sum_{i=\ell+1}^{m}(y_i - x_i) \leq \log_{1+\lambda/2}(B/k_{\alpha+8}) + 8(m-\ell) = O(\tfrac{1}{\lambda}\log(\lambda B) + m - \ell).$$

Summing up these four bounds, and using the fact that $\ell \leq m+1 = O(\frac{1}{\lambda}\log(\lambda B))$ (by Corollary 7.2), we conclude that the total number of bits sent due to the up events in a window is $O(\frac{1}{\lambda}\log(\lambda B))$.

For downs, we have the following assertion, which is symmetric to Lemma 7.6, and can be proved similarly. Suppose that there are m' downs occurring at time $d_1 < d_2 < \cdots < d'_m$. Recall that the characteristic set H_i of each d_i is those items whose first expiry time is within $[d_i + 1, t]$, or equivalently, $H_i = \sigma_{[d_i - W+1, t_o-1]}$. For any $2 \leq i \leq m'$, let $u_i < d_i$ be the latest time when an up or down occurs. Let ℓ' be the largest integer such that $|H_{\ell'}| \geq k_{\alpha+8}$ (if ℓ' does not exist, let $\ell' = 0$). Let $\hat{c}(u_i) = k_{x'_i}$ and $\hat{c}(d_i) = k_{y'_i}$. Then,

(1) for any $2 \leq i \leq \ell' - 1$, $|H_{i-1}| \geq (1 + \frac{\lambda}{2})^{x'_i - y'_i - 8}|H_i|$, and

(2) for any $\ell' + 1 \leq i \leq m$, $|H_{i-1}| - |H_i| \geq x'_i - y'_i - 8$.

Then, we can count the bits sent as above and conclude that the total number of bits sent due to downs in a window is $O(\frac{1}{\lambda}\log(\lambda B))$. Thus, the theorem follows. □

7.2 Approximate Counting of All Items

This section presents algorithms for the streams to communicate to the root so that the root at any time can approximate the count of each item. As a warm-up, we first consider the simple algorithm in which a stream will inform the root whenever its count of an item increases or decreases by a certain fraction of its total item count. We show that each stream sends at most $O((\Delta + \frac{1}{\varepsilon})\log B)$ words in a window, where Δ is the number of distinct items (Section 7.2.1). We then modify the algorithm so that a stream can "turn off" items whose counts are too small, and we give a more complicated analysis to deal with the case when many such items increase their counts rapidly (Section 7.2.2). The communication cost is reduced to $O(\frac{1}{\varepsilon}\log B)$ words.

7.2.1 A simple algorithm

Consider a stream σ. At any time t, let $c(t)$ and $c_j(t)$ be the number of all items and item j arriving at σ in $[t-W+1,t]$, respectively. Let $\lambda < 1/11$ be a positive constant (which will be set to $\varepsilon/11$ later). We maintain two λ-approximate data structures (the data structures in [35] and in Section 6.2 of Chapter 6, respectively) at σ locally, which can report estimates $\hat{c}(t)$ and $\hat{c}_j(t)$ for $c(t)$ and $c_j(t)$, respectively, such that [1]

$$(1-\lambda/6)c(t) \le \hat{c}(t) \le (1+\lambda/6)c(t); \quad \text{and} \quad c_j(t)-\lambda c(t) \le \hat{c}_j(t) \le c_j(t)+\lambda c(t).$$

Simple algorithm. At any time t, for any item j, let $p < t$ be the last time $\hat{c}_j(p)$ is sent to the root. The stream sends the estimate $\langle j, \hat{c}_j(t)\rangle$ to the root if the following event occurs.

- *Up:* $\hat{c}_j(t) > \hat{c}_j(p) + 9\lambda\hat{c}(t)$.
- *Down:* $\hat{c}_j(t) < \hat{c}_j(p) - 9\lambda\hat{c}(t)$.

The root's perspective. At any time t, let $r_{j,\sigma}(t)$ be the last estimate received from a stream σ for item j (at or before t). The root can estimate the total count of item j over all streams by summing all $r_{j,\sigma}(t)$ received. More precisely, for any $0 < \varepsilon < 1$, we set $\lambda = \varepsilon/11$ and let each stream use the simple algorithm. Then for each stream σ, the approximate data structures for $\hat{c}_j(t)$ and $\hat{c}(t)$ together with the simple algorithm guarantee that $c_j(t)-11\lambda c(t) \le r_{j,\sigma}(t) \le c_j(t)+11\lambda c(t)$. Summing $r_{j,\sigma}(t)$ over all streams would give the root an estimate of the total count of item j within an error of ε of the total count of all items.

Communication complexity. At any time t, we denote the reference window as $[t_o,t]$, where $t_o = t-W+1$. Assume that there are at most Δ distinct items. We first show that a stream σ encounters $O((\frac{1}{\lambda}+\Delta)\log B)$ up events and sends $O((\frac{1}{\lambda}+\Delta)\log B)$ words within $[t_o,t]$. The analysis of down events is similar, to be detailed later. Recall that for any time $t_1 \le t_2$, we define $\sigma_{[t_1,t_2]}$ (resp. $\sigma_{j,[t_1,t_2]}$) as the multi-set of all items (resp. item j only) arriving at σ within $[t_1,t_2]$, and $|\sigma_{[t_1,t_2]}|$ as the size of this multi-set.

[1] The constant 6 in the inequality is arbitrary. It can be replaced with any number provided that other constants in the algorithm and analysis (e.g., the constant 9 in the definition of up events) are adjusted accordingly.

Consider an up event U_j of some item j that occurs at time $v \in [t_o, t]$. Define the *previous event* of U_j to be the latest event (up or down) of item j that occurs at time $p < v$. We call p the *previous-event time* of U_j. The number of up events with previous-event time before t_o is at most Δ. To upper bound the number of up events with previous-event time $p \geq t_o$ is, however, non-trivial; below we call such an up event a *follow-up* (event). Intuitively, a follow-up can be triggered by frequent arrivals of an item, or mainly the relative decrease of the total count. This motivates us to classify follow-ups into two types and analyze them differently. A follow-up U_j is said to be *absolute* if $c(p) \leq \frac{6}{5}c(v)$, and *relative* otherwise. Define *Recent-items*(U_j) to be the multi-set of item j's that arrive after the previous event of U_j, i.e., *Recent-items*$(U_j) = \sigma_{j,[p+1,v]}$.

Absolute follow-ups. To obtain a tight bound of absolute follow-ups, we need a better characteristic-set argument that can consider the growth of different items together. Let $t_1, t_2, ..., t_k$ be the times in $[t_o, t]$ when some absolute follow-ups (of one or more items) occur. Let x_i be the number of items having an absolute follow-up at t_i. Note that for all i, $x_i \leq \min\{1/(7\lambda), \Delta\}$,[2] and $\sum_{i=1}^{k} x_i$ is the number of absolute follow-ups in $[t_o, t]$. We define the characteristic set S_i at each t_i as follows:

S_i = the union of *Recent-items*(U_j) over all absolute follow-ups U_j occurring at $t_1, t_2, \ldots, t_i$.

Lemma 7.8. (i) *For any* $2 \leq i \leq k$, $|S_i| > (1 + 6x_i\lambda)|S_{i-1}|$. **(ii)** *There are* $\sum_{i=1}^{k} x_i = O(\frac{1}{\lambda} \log B)$ *absolute follow-ups within* $[t_o, t]$.

Proof. For (i), consider an absolute follow-up U_j of an item j, occurring at time t_i with previous-event time p_i. Note that the increase in the count of item j from p_i to t_i must be due to the recent items. We have

$$\begin{aligned}
|\textit{Recent-items}(U_j)| &\geq c_j(t_i) - c_j(p_i) \\
&\geq \hat{c}_j(t_i) - \hat{c}_j(p_i) - \lambda c(t_i) - \lambda c(p_i) \quad \text{(guarantee by } \sigma\text{'s data structures)} \\
&> 9\lambda\hat{c}(t_i) - \lambda c(t_i) - \lambda c(p_i) \quad \text{(definition of an up event)} \\
&\geq (9\lambda(1 - \tfrac{\lambda}{6}) - \lambda - \tfrac{6}{5}\lambda)c(t_i) \geq 6\lambda c(t_i) \quad (U_j \text{ is absolute})
\end{aligned}$$

There are x_i absolute follow-ups at t_i, so $|S_i| > |S_{i-1}| + x_i\,(6\lambda c(t_i))$. Since $S_i \subseteq \sigma_{[t_o,t_i]}$,

[2] If an up event of an item j occurs at time t_i, then $c_j(t_i) \geq \hat{c}_j(t_i) - \lambda c(t_i) > 9\lambda\hat{c}(t_i) - \lambda c(t_i) \geq 7\lambda c(t_i)$. Thus the number of up events at time t_i is at most $c(t_i)/(7\lambda c(t_i)) = 1/(7\lambda)$.

$c(t_i) \geq |S_i| \geq |S_{i-1}|$. Therefore, we have $|S_i| > |S_{i-1}| + 6x_i\lambda|S_i| \geq (1 + 6x_i\lambda)|S_{i-1}|$.

For (ii), we note that $B \geq |S_k| > \prod_{i=2}^{k}(1 + 6x_i\lambda)|S_1|$, and $|S_1| \geq 1$. Thus, $\prod_{i=2}^{k}(1 + 6x_i\lambda) < B$, or equivalently, $\ln B > \sum_{i=2}^{k} \ln(1+6x_i\lambda)$. The latter is at least $\sum_{i=2}^{k} \frac{6x_i\lambda}{1+6x_i\lambda} \geq \lambda \sum_{i=2}^{k} x_i$. The last inequality follows from that $x_i \leq 1/(7\lambda)$ for all i. Thus, $\sum_{i=1}^{k} x_i \leq x_1 + \frac{1}{\lambda} \ln B = O(\frac{1}{\lambda} \log B)$. □

Relative follow-ups. A relative follow-up occurs only when a lot of items expire, and relative follow-ups of the same item cannot occur too frequently. Below we define $O(\log B)$ time intervals and argue that no item can have two relative follow-ups within an interval. Recall that for an item with time-stamp t_1, we define the *first expiry time* to be $t_1 + W$. At any time u in $[t_o, t]$, define H_u to be the set of all items whose first expiry time is within $[u + 1, t]$, i.e., $H_u = \sigma_{[u-W+1,t_o-1]}$. $|H_u|$ is non-increasing as u increases. Consider the times $t_o = u_0 < u_1 < u_2 < \cdots < u_\ell \leq t$ such that for $i \geq 1$, u_i is the first time such that $|H_{u_i}| < \frac{5}{6}|H_{u_{i-1}}|$. For convenience, let $u_{\ell+1} = t + 1$. Note that $|H_{u_0}| \leq B$ and $\ell = O(\log B)$.

Lemma 7.9. (i) *Every item j has at most one relative follow-up U_j within each interval $[u_i, u_{i+1} - 1]$.* **(ii)** *There are at most $O(\Delta \log B)$ relative follow-ups within $[t_o, t]$.*

Proof. For (i), assume U_j occurs at time v in $[u_i, u_{i+1} - 1]$, and its previous event occurs at time p. By definition, $c(p) > \frac{6}{5}c(v)$. Thus,

$$|H_p| - |H_v| = |\sigma_{[p-W+1,v-W]}| \geq c(p) - c(v) > \tfrac{1}{5}c(v) \geq \tfrac{1}{5}|\sigma_{[v-W+1,t_o-1]}| = \tfrac{1}{5}|H_v| \ ,$$

and $|H_v| < \frac{5}{6}|H_p|$. On the other hand, $v < u_{i+1}$ and $|H_v| \geq \frac{5}{6}|H_{u_i}|$. Thus, $|H_p| > |H_{u_i}|$ and $p < u_i$. For (ii), since there are Δ distinct items, there are at most Δ relative follow-ups within each interval $[u_i, u_{i+1} - 1]$, and at most $O(\Delta \log B)$ relative follow-ups within $[t_o, t]$. □

Down events. The analysis is symmetric to that of up events. The only non-trivial thing is the definition of the characteristic set for bounding the absolute follow-downs D_j, which is defined in an opposite sense: Assume D_j occurs at time v and its previous event occurs at $p \geq t_o$. D_j is said to be *absolute* if $c(p) \leq \frac{6}{5}c(v)$. Let *Expire*$(D_j)$ be the multi-set of item j's whose first expiry time is within $[p + 1, v]$. I.e., *Expire*$(D_j) = \sigma_{j,[p-W+1,v-W]}$.

It is perhaps a bit tricky that instead of defining the characteristic set of absolute follow-downs at the time they occur, we consider the times of the corresponding *previous events* of these follow-downs. Let $p_1, p_2, ..., p_k$ be the times in $[t_o, t]$ such that there is at least one event E_j (up or down) at p_i which is the previous event of an absolute follow-down D_j occurring after p_i. Let y_i be the number of such previous events at p_i, and let $AD(p_i)$ be the set of corresponding absolute follow-downs. Note that y_i (unlike x_i) only admits a trivial upper bound of Δ. We define the characteristic set T_i for each p_i as follows:

$$T_i = \text{the union of } \mathit{Expire}(D_j) \text{ over all } D_j \in AD(p_i), AD(p_{i+1}), \ldots, AD(p_k).$$

Similar to Lemma 7.8, we can show that $|T_i| > (1+5y_i\lambda)|T_{i+1}|$. Owing to a weaker bound of individual y_i, the number of absolute follow-downs, which equals $\sum_{i=1}^{k} y_i$, is shown to be $O((\frac{1}{\lambda} + \Delta)\log B)$.

Combining the analyses on up and down events, we have the following.

Theorem 7.10. *For approximate counting, each individual stream can use the simple algorithm with $\lambda = \varepsilon/11$ and it sends at most $O((\frac{1}{\varepsilon} + \Delta)\log B)$ words to the root within a window.*

7.2.2 The full algorithm

In this section, we extend the previous algorithm and give a new characteristic-set analysis that is based on future events (instead of the past events) to show that the communication cost per window can be reduced to $O(\frac{1}{\lambda}\log B)$ words. Intuitively, when the estimate $\hat{c}_j(t)$ of an item j is too small, say, less than $3\lambda\hat{c}(t)$, the algorithm treats this estimate as 0 and set the off_j flag of j to be true. This restricts the number of items with a positive estimate to $O(\frac{1}{\lambda})$. Initially, the off_j flag is true for all items j. Given $0 < \lambda < 1/11$, the stream communicates with the root as follows.

Algorithm AC. At any time t, for any item j, let $\ell_j(p)$ be the last estimate of j sent to the root at some time $p < t$. The stream sends the estimate of j to the root if the following event occurs.

- *Up:* If $\hat{c}_j(t) > \ell_j(p) + 9\lambda\hat{c}(t)$, send $\langle j, \hat{c}_j(t)\rangle$ and set $\mathit{off}_j = \mathit{false}$.
- *Off:* If $\mathit{off}_j = \mathit{false}$ and $\hat{c}_j(t) < 3\lambda\hat{c}(t)$, send $\langle j, 0\rangle$ and set $\mathit{off}_j = \mathit{true}$.
- *Down:* If $\mathit{off}_j = \mathit{false}$ and $\hat{c}_j(t) < \ell_j(p) - 9\lambda\hat{c}(t)$, send $\langle j, \hat{c}_j(t)\rangle$.

It is straightforward to check that the root can answer the approximate counting query for any item. We analyze the communication complexity by considering different events, as follows.

Fact 7.11. *At any time v, the number of items j with $\mathit{off}_j = \mathit{false}$ is at most $\frac{1}{\lambda}$.*[3]

Off events. At any time t, consider the current window $[t_o, t]$. By Fact 7.11, just before t_o, there are at most $\frac{1}{\lambda}$ items with $\mathit{off}_j = \mathit{false}$. Within $[t_o, t]$, only an up event can set the *off* flag to false. Thus the number of off events within $[t_o, t]$ is bounded by $\frac{1}{\lambda}$ plus the number of up events.

Up and Down events. The assumption of Δ gives a trivial bound on those events involving items with very small counts and in particular, those up events immediately following the off events. Such up events are called *poor-up* events or simply *poor-ups*. Using the *off* flag, we can easily adapt the analysis of the simple algorithm to bound all the down and up events of the full algorithm, but except the poor-ups. The following simple observations, derived from Fact 1, allow us to replace Δ with $1/\lambda$ in the previous analysis to obtain a tighter upper bound of $O(\frac{1}{\lambda}\log B)$. Let v by any time in $[t_o, t]$.

- There are at most $1/\lambda$ items whose first event after v is a down event.
- There are at most $1/\lambda$ non-poor-up events after v such that its previous event is before v.

It remains to analyze the poor-ups. Consider a poor-up U_j at time v in $[t_o, t]$. By definition, $\mathit{off}_j = \mathit{false}$ at time v. The trick of analyzing U_j's is to consider when the corresponding items will be "off" again instead of what items constitute the up events. Then a characteristic set argument can be formulated easily. Specifically, we first observe

[3]For any item j, if $\mathit{off}_j = \mathit{false}$, then $\hat{c}_j(v) \geq 3\lambda\hat{c}(v)$ and $c_j(v) \geq \hat{c}_j(v) - \lambda c(v) \geq (3\lambda(1-\lambda) - \lambda)c(v) \geq \lambda c(v)$. Thus the number of items j with $\mathit{off}_j = \mathit{false}$ is at most $c(v)/\lambda c(v) = \frac{1}{\lambda}$.

that, by Fact 1, there are at most $\frac{1}{\lambda}$ poor-ups whose *off* flags remain false up to time t. Then it remains to consider those U_j whose *off* flags will be set to true at some time $d \leq t$. Below we refer to d as the *first off time* of U_j.

Poor-up with early off. Consider a poor-up U_j that occurs at time v in $[t_o, t]$ and has its first off time at d in $[v+1, t]$. Let *F-Expire*(U_j) be all the item j whose first expiry time is within $[v+1, d]$. I.e., *F-Expire*$(U_j) = \sigma_{j,[v+1-W,d-W]}$. As an early off can be due to the expiry of many copies of item j or the arrival of a lot of items, it is natural to divide the poor-ups into two types: with an *absolute* off if $c(d) \leq \frac{6}{5}c(v)$, and *relative* off otherwise. For the case with absolute off, we consider the distinct times $t_1, t_2, \ldots, t_k$ in $[t_o, t]$ when such poor-ups occur. Let x_i be the number of such poor-ups at time t_i. Note that $x_i \leq 1/(7\lambda)$. For each time t_i, we define the characteristic set

$$F_i = \text{the union of } \textit{F-Expire}(U_j) \text{ over all } U_j \text{ occurring at } t_i, t_{i+1}, \ldots, t_k.$$

Lemma 7.12. (i) *For any* $1 \leq i \leq k-1$, $|F_i| > (1 + x_i\lambda)|F_{i+1}|$. **(ii)** *Within* $[t_o, t]$, *there are* $\sum_{i=1}^{k} x_i = O(\frac{1}{\lambda}\log B)$ *poor-ups each with an absolute off.*

Proof. For (i), let U_j be a poor-up with an absolute off of an item j, where U_j occurs at time t_i with first off time d_i. Note that the decrease in c_j must be due to expiry of item j. Then

$$\begin{aligned}
|\textit{F-Expire}(U_j)| &\geq c_j(t_i) - c_j(d_i) \geq \hat{c}_j(t_i) - \hat{c}_j(d_i) - \lambda c(t_i) - \lambda c(d_i) \\
&> 9\lambda\hat{c}(t_i) - 3\lambda\hat{c}(d_i) - \lambda c(t_i) - \lambda c(d_i) \qquad \text{(definition of up and off)} \\
&\geq (9\lambda(1-\tfrac{\lambda}{6}) - \lambda)c(t_i) - (3\lambda(1+\tfrac{\lambda}{6}) + \lambda)c(d_i) \geq 7\lambda c(t_i) - 5\lambda c(d_i) \\
&\geq (7 - 5(\tfrac{6}{5}))\lambda c(t_i) = \lambda c(t_i) \qquad \text{(definition of absolute off)}
\end{aligned}$$

Therefore, $|F_i| > |F_{i+1}| + x_i\,(\lambda c(t_i))$. Since $F_i \subseteq \sigma_{[t_i-W+1,t-W]}$, $|F_i| \leq c(t_i)$. This implies $|F_i| > |F_{i+1}| + x_i\lambda|F_i| > (1+x_i\lambda)|F_{i+1}|$. We can prove (ii) similarly to Lemma 7.8 (ii). □

Analyzing poor-ups with a relative off is again based on an isolating argument. We divide $[t_o, t]$ into $O(\log B)$ intervals according to how fast the total item count starting from t_o grow; specifically, we want two consecutive time boundaries u_{i-1} and u_i to satisfy $|\sigma_{[t_o,u_i]}| > \frac{6}{5}|\sigma_{[t_o,u_{i-1}]}|$. Then we show that for any poor-up within $[u_{i-1}, u_i - 1]$, its relative off, if exists, occurs at or after u_i. Thus there are at most $\frac{1}{\lambda}$ such poor-ups within each

interval and a total of $O(\frac{1}{\lambda}\log B)$ within $[t_o, t]$.

Lemma 7.13. (i) *Consider a poor-up U_j with a relative off. Suppose it occurs at time v in $[t_o, t]$, and its first off time is at d in $[v+1, t]$. Then $|\sigma_{[t_o,d]}| > \frac{6}{5}|\sigma_{[t_o,v]}|$.* **(ii)** *Within $[t_o, t]$, there are at most $O(\frac{1}{\lambda}\log B)$ poor-ups each with a relative off.*

Proof. For (i), by the definition of a relative off, $c(v) < \frac{5}{6}c(d)$. Thus, $|\sigma_{[t_o,d]}| - |\sigma_{[t_o,v]}| = |\sigma_{[v+1,d]}| \geq c(d) - c(v) > \frac{1}{6}c(d) \geq \frac{1}{6}|\sigma_{[t_o,d]}|$. This implies $|\sigma_{[t_o,d]}| > \frac{6}{5}|\sigma_{[t_o,v]}|$.

For (ii), consider the times $t_o = u_0 < u_1 < u_2 < \cdots < u_\ell \leq t$ such that for $i \geq 1$, u_i is the first time such that $|\sigma_{[t_o,u_i]}| > \frac{6}{5}|\sigma_{[t_o,u_{i-1}]}|$. For convenience, let $u_{\ell+1} = t+1$. Note that $|\sigma_{[t_o,t]}| \leq B$ and $\ell = O(\log B)$. Furthermore, for any time $v \in [u_{i-1}, u_i - 1]$, $|\sigma_{[t_o,v]}| \leq \frac{6}{5}|\sigma_{[t_o,u_{i-1}]}|$. Therefore, by (i), for any poor-up of an item j within $[u_{i-1}, u_i - 1]$, its relative off, if exists, occurs at or after u_i, which implies at time $u_i - 1$, $c_j(u_i - 1) \geq \lambda c(u_i - 1)$. Then within each interval $[u_{i-1}, u_i - 1]$, the number of such j as well as the number of poor-ups with a relative off are at most $\frac{1}{\lambda}$. Within $[t_o, t]$, there are $\ell = O(\log B)$ intervals and hence $O(\frac{1}{\lambda}\log B)$ poor-ups each with a relative off. □

To sum up, the following is our main result.

Theorem 7.14. *For approximate counting, each individual stream can use the algorithm AC with $\lambda = \varepsilon/11$ and it sends at most $O(\frac{1}{\varepsilon}\log B)$ words to the root within a window.*

Memory usage on each remote site. Recall that we use two λ-approximate data structures (the data structures in [35] and in Section 6.2 of Chapter 6, respectively) for the total item count and individual item counts, which respectively require $O(\frac{1}{\lambda}\log^2(\lambda B))$ bits and $O(\frac{1}{\lambda})$ words. Note that $O(\frac{1}{\lambda}\log^2(\lambda B))$ bits is equivalent to $O(\frac{1}{\lambda}\log(\lambda B))$ words. Furthermore, at any time, we only need to keep track of the last estimate sent to the root of all item j with $off_j = false$, which by Fact 7.11, requires $O(\frac{1}{\lambda})$ words. By setting $\lambda = \varepsilon/11$ (see Theorem 7.14), the total memory usage of a remote site is $O(\frac{1}{\lambda}\log(\lambda B)) = O(\frac{1}{\varepsilon}\log(\varepsilon B))$ words.

7.3 Other Extensions

This section shows how to extend the previous algorithms and techniques to the problems of frequent items and quantiles. We also explain how to handle out-of-order streams.

7.3.1 Frequent items

Using the algorithms BC and AC, the root can answer the ε-approximate frequent items as follows. Each stream σ communicates with the root using BC with error parameter $\varepsilon/24$ and AC with error parameter $11\varepsilon/24$. Hence, σ sends at most $O(\frac{1}{\varepsilon}\log B)$ words within any time window. At any time t, let $r_\sigma(t)$ and $r_{j,\sigma}(t)$ be the latest estimates of the numbers of all items and item j, respectively, received by the root from σ. In the root's perspective, to answer a query of frequent items with threshold $\phi \in (0,1]$ at time t, it can return all items j with $\sum_\sigma r_{j,\sigma}(t) \geq (\phi - \frac{\varepsilon}{2})\sum_\sigma r_\sigma(t)$ as the set of frequent items.

To see the correctness, let $c_\sigma(t)$ and $c_{j,\sigma}(t)$ be the number of all items and item j in σ at time t, respectively. Algorithm BC guarantees $|r_\sigma(t) - c_\sigma(t)| \leq \frac{\varepsilon}{24}c_\sigma(t)$, and algorithm AC guarantees $|r_{j,\sigma}(t) - c_{j,\sigma}(t)| \leq \frac{11\varepsilon}{24}c_\sigma(t)$. Therefore, if an item j is returned by the root, then $\sum_\sigma c_{j,\sigma}(t) \geq \sum_\sigma r_{j,\sigma}(t) - \frac{11\varepsilon}{24}\sum_\sigma c_\sigma(t) \geq (\phi - \frac{\varepsilon}{2})\sum_\sigma r_\sigma(t) - \frac{11\varepsilon}{24}\sum_\sigma c_\sigma(t) \geq (\phi - \frac{\varepsilon}{2})(1 - \frac{\varepsilon}{24})\sum_\sigma c_\sigma(t) - \frac{11\varepsilon}{24}\sum_\sigma c_\sigma(t) \geq (\phi - \frac{\varepsilon}{2} - \phi\frac{\varepsilon}{24} - \frac{11\varepsilon}{24})\sum_\sigma c_\sigma(t)$ where the second inequality comes from the definition of the algorithm. The last term above is at least $(\phi - \varepsilon)\sum_\sigma c_\sigma(t)$, so j is a frequent item. If an item j is not returned by the root, then $\sum_\sigma r_{j,\sigma}(t) < (\phi - \frac{\varepsilon}{2})\sum_\sigma r_\sigma(t)$ and we can show similarly that $\sum_\sigma c_{j,\sigma}(t) < \phi\sum_\sigma c_\sigma(t)$.

7.3.2 Quantiles

We give an algorithm where each stream sends at most $O(\frac{1}{\varepsilon^2}\log B)$ words per window. Let $\lambda = \varepsilon/20$. For each stream, we keep track of the λ-approximate ϕ-quantiles for $\phi = 5\lambda, 10\lambda, 15\lambda, \ldots, 1$. We update the root for all these ϕ-quantiles when one of the following two events occurs: (i) for any k, the value of the $(5k\lambda)$-quantile is larger than the value of the $(5(k+1)\lambda)$-quantile last reported to the root, or (ii) for any k, the value of the $(5k\lambda)$-quantile is smaller than the value of the $(5(k-1)\lambda)$-quantile last reported to the root. The stream also communicates with the root using BC with error parameter λ. In the root's perspective, at any query time t, let $\phi \in (0,1]$ be the query given and let $r_\sigma(t)$ be the last estimate sent by σ for the number of all items. The root sorts the quantiles last reported by all streams and for each stream σ, gives a weight of $5\lambda r_\sigma(t)$ to each quantile of σ. Then the root returns the smallest item j in the sorted sequence such that the sum of weights for all items no greater than j is at least $\lceil\phi\sum_\sigma r_\sigma(t)\rceil$. Careful counting can show that j is an ε-approximate ϕ-quantile.

For the communication cost, we observe that when an event occurs, many items have either arrived or expired after the previous event. Using similar analysis as before, we can show that within a window, there are at most $O(\frac{1}{\varepsilon}\log B)$ such events and thus each stream sends $O(\frac{1}{\varepsilon^2}\log B)$ words. Note that the lower bound of $O(\frac{1}{\varepsilon}\log(\varepsilon B))$ words per window for approximate frequent items carries to approximate quantiles, as we can answer approximate frequent items using approximate quantiles, as follows. The root poses ε-approximate ϕ-quantile queries for $\phi = \varepsilon, 2\varepsilon, \ldots, 1$. Given the threshold ϕ' for frequent items, the root returns all items that repeatedly occur as $\frac{\phi'}{\varepsilon} - 2$ (or more) consecutive quantiles, and we can show that these items are (4ε)-approximate frequent items.

7.3.3 Out-of-order streams

All our algorithms can be extended to out-of-order stream with a communication cost increased by a factor of $\frac{W}{W-\tau}$, as follows. Each stream uses the data structures for out-of-order streams (e.g., [22, 30]) to maintain the local estimates. Then each stream uses our communication algorithms for in-order streams. It is obvious that the root can answer the corresponding queries. For the communication cost, consider any time interval $P = [t - (W - \tau) + 1, t]$ of size $W - \tau$. Items arriving in P must have time-stamps in $[t-W+1, t]$, so at most B items arrive in P. Also, at most B items expire in P. Using the same arguments as before, we can show the same communication cost of each algorithm, but only for a window of size $W - \tau$ instead of W. Equivalently, in any window of size W, the communication cost is increased by a factor of $O(\frac{W}{W-\tau})$.

7.4 Lower Bounds

This section presents the lower bounds. We assume the two-way communication model, and the results also hold when only one-way communication is allowed. We start with the sliding window setting, and consider the whole data stream and out-of-order stream settings afterwards.

Theorem 7.15. *For the ε-approximate basic counting problem on a sliding window, the communication cost between any stream and the root is $\Omega(\frac{1}{\varepsilon}\log(\varepsilon B))$ bits in every $2W$ time units in the worst case.*

Proof. Consider any $\varepsilon < 1/3$. Let $x_0 < x_1 < x_2 < \dots$ be an increasing sequence such that $x_0 = 0$, $x_1 = \lceil \frac{1}{\varepsilon} \rceil$ and for $i \geq 2$, $x_i = \lceil (1+3\varepsilon)x_{i-1} \rceil$. Consider the time window $[1, W]$ and any stream σ. At each time $i \geq 1$, we release $x_i - x_{i-1}$ items to σ and no item to other streams[4]. Note that there are exactly x_i items in the window of σ at time i. The last item arrives at time m where m is the largest integer such that $x_m \leq B$. Note that $x_1 = \lceil \frac{1}{\varepsilon} \rceil < \frac{2}{\varepsilon}$, and for any $i = 2, \dots m$, $x_i = \lceil (1+3\varepsilon)x_{i-1} \rceil \leq (1+3\varepsilon)x_{i-1} + \varepsilon x_{i-1} = (1+4\varepsilon)x_{i-1}$. Since $B < x_{m+1}$, we have $m = \Omega(\log_{1+4\varepsilon}(\varepsilon B/2)) = \Omega(\frac{1}{\varepsilon}\log(\varepsilon B))$.

At any time i, the root can return an estimate $r(i)$ such that $(1-\varepsilon)x_i < r(i) < (1+\varepsilon)x_i$. Note that $(1-\varepsilon)x_i \geq (1-\varepsilon)(1+3\varepsilon)x_{i-1} = (1+2\varepsilon-3\varepsilon^2)x_{i-1} > (1+\varepsilon)x_{i-1}$ (since $\varepsilon < 1/3$). Thus, at each time i, the root must have a different estimate. It implies that σ communicates at least 1 bit with the root at each time $i = 1, \dots, m$, and hence the communication cost is $\Omega(\frac{1}{\varepsilon}\log(\varepsilon B))$ bits in $[1, W]$. We do not release items during $[W+1, 2W]$, so all items expire no later than $2W$. We can repeat the sequence above, and the theorem follows. □

Theorem 7.16. *For the ε-approximate frequent items problem on a sliding window, the communication cost between any stream and the root is $\Omega(\frac{1}{\varepsilon}\log(\varepsilon B))$ words in every $2W$ time units in the worst case.*

Proof. Consider any $\varepsilon < 1/12$. Let $x_0 < x_1 < x_2 < \dots$ be an increasing sequence such that $x_0 = 0$, $x_1 = \lceil \frac{1}{\varepsilon} \rceil$ and for any $i \geq 2$, $x_i = \lceil (1+4\varepsilon)x_{i-1} \rceil$. Consider the time window $[1, W]$ and any stream σ. Let $\{j_1, j_2, \dots\}$ be a set of distinct item types. At each time $i \geq 1$, we release $x_i - x_{i-1}$ item j_i's to σ and no items to other streams. Note that there are totally x_i items in the window of σ at time i. The last item arrives at time m where m is the largest integer such that $x_m \leq B$. Note that $x_1 = \lceil \frac{1}{\varepsilon} \rceil < \frac{2}{\varepsilon}$, and for any $i = 2, \dots m$, $x_i = \lceil (1+4\varepsilon)x_{i-1} \rceil \leq (1+4\varepsilon)x_{i-1} + \varepsilon x_{i-1} = (1+5\varepsilon)x_{i-1}$. Since $B < x_{m+1}$, we have $m = \Omega(\log_{1+5\varepsilon}(\varepsilon B/2)) = \Omega(\frac{1}{\varepsilon}\log(\varepsilon B))$.

Let $\phi = 2\varepsilon$. At any time i, the number of item j_i is $x_i - x_{i-1} \geq 4\varepsilon x_{i-1}$. Note that $x_i = \lceil (1+4\varepsilon)x_{i-1} \rceil < (1+4\varepsilon)x_{i-1} + 1$. Hence, $4\varepsilon x_{i-1} > 4\varepsilon(x_i - 1)/(1+4\varepsilon)$, which is at least $2\varepsilon x_i = \phi x_i$ for $\varepsilon < 1/12$ and $x_i \geq 1/\varepsilon$. Thus, for each $i = 1, \dots, m$, j_i becomes a new frequent item at time i. It implies that σ communicates at least 1 word with the root (to determine the label of j_i) at each time $i = 1, \dots, m$, and hence the communication

[4]Our argument can be extended such that items arrive in more than one stream and each of them needs to communicate with the root.

cost is $\Omega(\frac{1}{\varepsilon}\log(\varepsilon B))$ words in $[1, W]$. We do not release items during $[W+1, 2W]$, so all items expire no later than $2W$. We can repeat the sequence above, and the theorem follows. □

As shown in Section 7.3, we can solve the frequent items problem using the approximate counting or quantiles queries, together with the basic counting queries. The basic counting problem requires only $O(\frac{1}{\varepsilon}\log(\varepsilon B))$ bits communication. Thus, by Theorem 7.16, we obtain the lower bounds for approximate counting or quantiles.

Corollary 7.17. *For the ε-approximate counting and ε-approximate quantiles problems on a sliding window, the communication cost between any stream and the root is $\Omega(\frac{1}{\varepsilon}\log(\varepsilon B))$ words in every $2W$ time units in the worst case.*

Whole data stream. For any stream σ, let n be the total number of items that arrive in σ. Using the same arguments as that for the sliding window setting, we can obtain the same lower bounds with B replaced by n. Specifically, the communication cost between σ and the root is $\Omega(\frac{1}{\varepsilon}\log(\varepsilon n))$ bits for basic counting, and $\Omega(\frac{1}{\varepsilon}\log(\varepsilon n))$ words for frequent items, approximate counting and quantiles in the worst case.

Out-of-order streams. An in-order stream is a special case of an out-of-order stream, so the previous lower bounds still hold for out-of-order streams with any tardiness τ. The following theorem gives stronger lower bounds of communication for the four queries.

Theorem 7.18. *Consider the out-of-order streams setting with tardiness $0 \leq \tau \leq W-1$.*

- *For the ε-approximate basic counting problem, the communication cost between any stream and the root is $\Omega(\max\{\frac{W}{W-\tau}, \frac{1}{\varepsilon}\log(\varepsilon B)\})$ bits in every $2W$ time units in the worst case.*
- *For the ε-approximate frequent items, ε-approximate counting and ε-approximate quantiles problems, the communication cost between any stream and the root is $\Omega(\max\{\frac{W}{W-\tau}, \frac{1}{\varepsilon}\log(\varepsilon B)\})$ words in every $2W$ time units in the worst case.*

Proof. Consider any stream σ. We want to show that in every $2W$ time units, σ and the root need to communicate $\Omega(\frac{W}{W-\tau})$ bits for the basic counting problem, and $\Omega(\frac{W}{W-\tau})$ words

for the frequent items, approximate counting and quantiles problems. Then, together with Theorems 7.15 and 7.16 and Corollary 7.17, the theorem follows.

Consider the sequence that at each time $t_i = i(W - \tau + 1)$, $i = 1, 2, \ldots$, a distinct item arrives with time-stamp $t_i - \tau$, and the last item arrives at time t_m where m is the largest integer such that $t_m \leq W$. Note that the item arriving at t_i expires at time $t_i - \tau + W = t_{i+1} - 1$. Thus, there are exactly one item in the sliding window during $[t_i, t_{i+1} - 1)$ and exactly zero item at $t_{i+1} - 1$. At each $t_{i+1} - 1$, the stream σ needs to communicate with the root at least 1 bit for basic counting and at least 1 word (an item label) for the other queries. Thus, during $[1, W]$, the communication cost is $\Omega(\frac{W}{W-\tau})$ bits the basic counting problem, and $\Omega(\frac{W}{W-\tau})$ words for the frequent items, approximate counting and quantiles problems. We do not release items during $[W + 1, 2W]$, so all items expire no later than $2W$. We can repeat the sequence above to obtain the required results. □

Chapter 8

Conclusion

In this thesis, we have presented several new results on online job scheduling and data stream algorithms. For online job scheduling, we studied four problems on online flow-energy scheduling, namely, flow-energy scheduling on a single processor, flow-energy scheduling with sleep states, non-migratory multi-processor flow-energy scheduling, and non-clairvoyant flow-energy scheduling. For data stream algorithms, we studied both space-efficient and communication-efficient data stream algorithms.

For flow-energy scheduling on a single processor, we introduce a new speed function AJC that depends on the number of active jobs. AJC is a more stable speed function than existing speed functions [12,16]; existing speed functions depend on the remaining work of active jobs and therefore change continuously, which is undesirable practically, while AJC changes only at job arrival or completion. Using AJC leads to an algorithm SRPT-AJC, which is $O(1)$-competitive for total flow time plus energy in both the infinite and bounded speed models. These results improve the best competitive ratios in the infinite speed model [16] and the bounded speed model [12], respectively. More importantly, under the bounded speed model, SRPT-AJC does not require extra maximum processor speed as the existing algorithm [12] does.

Nowadays, energy saving can be achieved not only by speed scaling of the processors, but also by allowing a processor to enter a low-power sleep state. We initiate the study of flow-energy scheduling that exploits both speed scaling and sleep states. We give a sleep management algorithm called IdleLonger, which determines when the processor should sleep, idle, and work. IdleLonger works for a processor with one or multiple levels of sleep

states and works in both infinite and bounded speed models. We adapt the speed scaling algorithm SRPT-AJC and show that this adapted algorithm together with IdleLonger is $O(1)$-competitive for total flow plus energy in both infinite and bounded speed models.

We extend the study of flow-energy scheduling to the setting with $m \geq 2$ processors. This extension is not only of theoretical interest, as modern processors adopt multi-core technology (dual-core and quad-core are getting common). A multi-core processor is essentially a pool of parallel processors. We aim at schedules that do not require job migration among processors; in practice, migrating jobs requires overheads and is avoided in many applications. We introduce the job dispatching policy CRR, which leads to a online non-migratory algorithm CRR-A. We analyze CRR-A under the bounded speed model (the results also hold in the infinite speed model). When job size is restricted to power-of-2, CRR-A is $O(\log P)$-competitive for flow plus energy, where P is the ratio of the maximum job size to the minimum job size. For jobs of arbitrary size, CRR-A is $O(1)$-competitive for flow plus energy, using slightly higher maximum processor speed. We also show that any online scheduling algorithm (without extra maximum speed) is $\Omega(\log P)$-competitive. When jobs are all power-of-2 size, this lower bound still holds and CRR-A is therefore optimal (up to a constant factor) for such jobs.

In some applications like operating systems, job size is only known when the job finishes, which is referred to as the *non-clairvoyant* model. We initiate the study of non-clairvoyant flow-energy scheduling. When all jobs are released at the same time, we give an algorithm that is $(2 - \frac{1}{\alpha})$-competitive and 2-competitive in the infinite and bounded speed model, respectively. The latter inherits a lower bound of 2 from flow-time scheduling [74] and is therefore optimal. We can further generalize this result for minimizing weighted flow time plus energy, and the corresponding ratios become $(2 - \frac{1}{\alpha})^2$ and 4, respectively. For jobs with arbitrary release times, we focus on the infinite speed model. We give an algorithm that is $O(1)$-competitive for flow plus energy. This algorithm can be adapted to the model with sleep states and the adapted algorithm together with the sleep management algorithm IdleLonger remains $O(1)$-competitive. It is open whether an online algorithm can be $O(1)$-competitive in the bounded speed model, and our work serves as a step towards this more difficult problem.

For space-efficient data stream algorithms, we focus on count-based sliding window of size W and consider two problems. The first problem is about counting the number of 1-bits in a bit stream, which is a fundamental data stream problem. We introduce

the Significant One Counting problem, which asks for an estimate with bounded relative error $\varepsilon \in (0,1)$ only when the number of 1-bits in the window is at least θW for some threshold $\theta \in (0,1)$. This problem provides more flexibility in space-accuracy tradeoff than the basic counting problem, which asks for an estimate with bounded relative error in any case. In particular, by setting $\theta = 1/W$, the new problem becomes the basic counting problem. We show that any algorithm for significant one counting must use at least $\Omega(\frac{1}{\varepsilon}\log^2(\frac{1}{\theta}) + \log(\varepsilon\theta W))$ bits of memory. Then, we give an algorithm that has constant update and query time, and uses memory matching the lower bound, i.e., the algorithm has the optimal time and space complexity. The second problem is finding ε-approximate frequent items over sliding window. We give an algorithm which supports $O(\frac{1}{\varepsilon})$ update and query time, and uses $O(\frac{1}{\varepsilon})$ words of memory which is essentially optimal. This substantially improves the previous result by Arasu and Manku [5]. We also extend our algorithm to the setting where the window size W can be changed by the user. This extended algorithm uses $O(\frac{1}{\varepsilon}\log W)$ space.

Finally, we study communication-efficient algorithms for continuous monitoring of multiple, distributed data streams. We initiate a mathematical study of algorithms for monitoring distributed data streams over a time-based sliding window (which contains a variable number of items and possibly out-of-order items). The concern is how to minimize the communication between individual streams and the root (or the coordinator), while allowing the root, at any time, to be able to report the global statistics of all streams within a given error bound. We present communication-efficient algorithms for four classical statistics, namely, basic counting, approximate counting, frequent items and quantiles. The worst-case communication cost over a window is $O(\frac{1}{\varepsilon}\log(\varepsilon B))$ bits for basic counting and $O(\frac{1}{\varepsilon}\log B)$ words for the remaining, where B is the maximum number of items that can arrive in a time window, and $\varepsilon < 1$ is the desired error bound. Matching and almost matching lower bounds are also presented. The upper bound results directly imply those for the whole data stream and count-based sliding window.

8.1 Future Work

Our work gives a better understanding of different flow-energy scheduling problems and different space-efficient and communication-efficient data stream algorithms. Yet, there are still many interesting directions that deserve investigation, as shown below.

Non-migratory multi-processor flow-energy scheduling. We have shown that under the bounded speed model, any online scheduling algorithm is $\Omega(\log P)$-competitive for flow plus energy, where P is the ratio of maximum job size to minimum job size. The online non-migratory algorithm CRR-A achieves this optimal competitive ratio only when job sizes are restricted to power-of-2. It would be interesting to see if CRR-A remains $O(\log P)$-competitive for flow plus energy when scheduling arbitrary jobs.

Non-clairvoyant flow-energy scheduling. When jobs have arbitrary release time, we have given an online non-clairvoyant algorithm that is $O(1)$-competitive for flow plus energy in the infinite speed model, yet not much is known for the bounded speed model. Related results are on flow-time scheduling (without energy concern), where it has been known that any online non-clairvoyant algorithm has a competitive ratio of $\Omega(n^{1/3})$ for total flow time [74], where n is the number of jobs, and given extra processor speed, for any $\epsilon > 0$, SETF is $(1+\epsilon)$-speed $O(1+\frac{1}{\epsilon})$-competitive for total flow time [61]. Therefore, extra maximum processor speed is necessary to achieve $O(1)$-competitiveness for flow plus energy. It remains open if extra speed is sufficient to achieve $O(1)$-competitiveness.

Space-efficient data stream algorithms. We have given space-optimal algorithm for finding approximate frequent items over sliding windows. Recently, Cormode et al. [30] has extended the study of this problem to out-of-order streams, yet there is a huge gap on the upper bound and lower bound. It is interesting to close the gap and discover the real memory requirement of the problem. Another direction is to study the quantiles statistics (for in-order stream). It has been long open whether there is an algorithm with memory usage matching the trivial lower bound of $\Omega(1/\varepsilon)$ words, even in the whole data stream setting.

Communication-efficient data stream algorithms. We have shown communication-efficient algorithms for four classical statistics, namely, basic counting, approximate counting, frequent items and quantiles. Assuming in-order streams, our upper bounds and lower bounds on the communication cost for the problems match or almost match with each other. Yet, for out-of-order stream, our simple adaptation of the algorithms lead to a relatively larger gap on the upper bound and lower bound (see Table 1.3 in Section 1.2.2). It is interesting to design more sophisticated algorithms for out-of-order streams in order to close the gap. Another direction is to extend the study of communication-efficient algorithms to more complicated statistics, e.g., the join aggregates and L_2-norm queries.

Bibliography

[1] C. Aggarwal. *Data streams: models and algorithms*. Springer, 2006.

[2] S. Albers and H. Fujiwara. Energy-efficient algorithms for flow time minimization. *ACM Transactions on Algorithms*, 3(4):49, 2007.

[3] S. Albers, F. Muller, and S. Schmelzer. Speed scaling on parallel processors. In *Proceedings of ACM Symposium on Parallelism in Algorithms and Architectures (SPAA)*, pages 289–298, 2007.

[4] N. Alon, Y. Matias, and M. Szegedy. The space complexity of approximating the frequency moments. *Journal of Computer and System Sciences*, 58(1):137–147, 1999.

[5] A. Arasu and G. Manku. Approximate counts and quantiles over sliding windows. In *Proceedings of ACM SIGMOD-SIGACT-SIGART Symposium on Principles of Database Systems (PODS)*, pages 286–296, 2004.

[6] J. Augustine, S. Irani, and C. Swany. Optimal power-down strategies. In *Proceedings of IEEE Symposium on Foundations of Computer Science (FOCS)*, pages 530–539, 2004.

[7] N. Avrahami and Y. Azar. Minimizing total flow time and total completion time with immediate dispatching. In *Proceedings of ACM Symposium on Parallelism in Algorithms and Architectures (SPAA)*, pages 11–18, 2003.

[8] B. Awerbuch, Y. Azar, S. Leonardi, and O. Regev. Minimizing the flow time without migration. *SIAM Journal on Computing*, 31(5):1370–1382, 2002.

[9] B. Babcock, M. Datar, and R. Motwani. Sampling from a moving window over streaming data. In *Proceedings of ACM-SIAM Symposium on Discrete Algorithms (SODA)*, pages 633–634, 2002.

[10] B. Babcock and C. Olston. Distributed top-k monitoring. In *Proceedings of ACM SIGMOD International Conference on Management of Data (SIGMOD)*, pages 28–39, 2003.

[11] K. R. Baker. *Introduction to Sequencing and Scheduling.* Wiley, New York, 1974.

[12] N. Bansal, H. L. Chan, T. W. Lam, and L. K. Lee. Scheduling for speed bounded processors. In *Proceedings of International Colloquium on Automata, Languages and Programming (ICALP)*, pages 409–420, 2008.

[13] N. Bansal, H. L. Chan, and K. Pruhs. Speed scaling with an arbitrary power function. In *Proceedings of ACM-SIAM Symposium on Discrete Algorithms (SODA)*, pages 693–701, 2009.

[14] N. Bansal and K. Dhamdhere. Minimizing weighted flow time. *ACM Transactions on Algorithms*, 3(4):39, 2007.

[15] N. Bansal, T. Kimbrel, and K. Pruhs. Speed scaling to manage energy and temperature. *Journal of the ACM*, 54(1):3, 2007.

[16] N. Bansal, K. Pruhs, and C. Stein. Speed scaling for weighted flow time. In *Proceedings of ACM-SIAM Symposium on Discrete Algorithms (SODA)*, pages 805–813, 2007.

[17] L. Becchetti and S. Leonardi. Nonclairvoyant scheduling to minimize the total flow time on single and parallel machines. *Journal of the ACM*, 51(4):517–539, 2004.

[18] L. Benini, A. Bogliolo, and G. de Micheli. A survey of design techniques for system-level dynamic power management. *IEEE Transactions on VLSI Systems*, 8(3):299–316, 2000.

[19] A. Borodin and R. El-Yaniv. *Online Computation and Competitive Analysis.* Cambridge University Press, 1998.

[20] D. M. Brooks, P. Bose, S. E. Schuster, H. Jacobson, P. N. Kudva, A. Buyuktosunoglu, J. D. Wellman, V. Zyuban, M. Gupta, and P. W. Cook. Power-aware microarchitecture: Design and modeling challenges for next-generation microprocessors. *IEEE Micro*, 20(6):26–44, 2000.

[21] D. P. Bunde. Power-aware scheduling for makespan and flow. In *Proceedings of ACM Symposium on Parallel Algorithms and Architectures (SPAA)*, pages 190–196, 2006.

[22] C. Busch and S. Tirthapua. A deterministic algorithm for summarizing asynchronous streams over a sliding window. In *Proceedings of International Symposium on Theoretical Aspects of Computer Science (STACS)*, pages 465–476, 2007.

[23] H. L. Chan, W. T. Chan, T. W. Lam, L. K. Lee, K. S. Mak, and P. W. H. Wong. Energy efficient online deadline scheduling. In *Proceedings of ACM-SIAM Symposium on Discrete Algorithms (SODA)*, pages 795–804, 2007.

[24] H. L. Chan, T. W. Lam, and K. K. To. Nonmigratory online deadline scheduling on multiprocessors. *SIAM Journal on Computing*, 34(3):669–682, 2005.

[25] M. Charikar, K. Chen, and M. Farach-Colton. Finding frequent items in data streams. In *Proceedings of International Colloquium on Automata, Languages and Programming (ICALP)*, pages 693–703, 2002.

[26] C. Chekuri, A. Goel, S. Khanna, and A. Kumar. Multi-processor scheduling to minimize flow time with ϵ resource augmentation. In *Proceedings of ACM Symposium on Theory of Computing (STOC)*, pages 363–372, 2004.

[27] C. Chekuri, S. Khanna, and A. Zhu. Algorithms for minimizing weighted flow time. In *Proceedings of ACM Symposium on Theory of Computing (STOC)*, pages 84–93, 2001.

[28] G. Cormode and M. Garofalakis. Sketching streams through the net: distributed approximate query tracking. In *Proceedings of International Conference on Very Large Data Bases (VLDB)*, pages 13–24, 2005.

[29] G. Cormode, M. Garofalakis, S. Muthukrishnan, and R. Rastogi. Holistic aggregates in a networked world: distributed tracking of approximate quantiles. In *Proceedings of ACM SIGMOD International Conference on Management of Data (SIGMOD)*, pages 25–36, 2005.

[30] G. Cormode, F. Korn, and S. Tirthapura. Time-decaying aggregates in out-of-order streams. In *Proceedings of ACM SIGMOD-SIGACT-SIGART Symposium on Principles of Database Systems (PODS)*, pages 89–98, 2008.

[31] G. Cormode and S. Muthukrishnan. What's hot and what's not: Tracking most frequent items dynamically. *ACM Transactions on Database Systems*, 30(1):249–278, 2005.

[32] G. Cormode, S. Muthukrishnan, and K. Yi. Algorithms for distributed functional monitoring. In *Proceedings of ACM-SIAM Symposium on Discrete Algorithms (SODA)*, pages 1076–1085, 2008.

[33] G. Cornuejos, G. L. Nemhauser, and L. Wosley. The uncapacitated facility location problem. In P. Mirchandani and R. Francis, editors, *Discrete Location Theory*, pages 119–171. John Wiley & Sons, 1990.

[34] A. Das, S. Ganguly, M. Garofalakis, and R. Rastogi. Distributed set-expression cardinality estimation. In *Proceedings of International Conference on Very Large Data Bases (VLDB)*, pages 312–323, 2004.

[35] M. Datar, A. Gionis, P. Indyk, and R. Motwani. Maintaining stream statistics over sliding windows. *SIAM Journal on Computing*, 31(6):1794–1813, 2002.

[36] M. Datar and S. Muthukrishnan. Estimating rarity and similarity over data stream windows. In *Proceedings of European Symposium on Algorithms (ESA)*, pages 323–334, 2002.

[37] E. D. Demaine, A. Lopez-Ortiz, and J. I. Munro. Frequency estimation of internet packet streams with limited space. In *Proceedings of European Symposium on Algorithms (ESA)*, pages 348–360, 2002.

[38] D. Dilman and D. Raz. Efficient reactive monitoring. *IEEE Journal on Selected Areas in Communications*, 20(4):668–676, 2002.

[39] D. R. Dooly, S. A. Goldman, and S. D. Scott. On-line analysis of the TCP acknowledgment delay problem. *Journal of the ACM*, 48(2):243–273, 2001.

[40] J. Edmonds. Scheduling in the dark. *Theoretical Computer Science*, 235(1):109–141, 2000.

[41] J. Edmonds and K. Pruhs. Scalably scheduling processes with arbitrary speedup curves. In *Proceedings of ACM-SIAM Symposium on Discrete Algorithms (SODA)*, pages 685–692, 2009.

[42] C. Estan and G. Varghese. New directions in traffic measurement and accounting. *ACM SIGCOMM Computer Communication Review*, 32(4):323–336, 2002.

[43] A. Fabrikant, A. Luthra, E. Maneva, C. H. Papadimitriou, and S. Shenker. On a network creation game. In *Proceedings of ACM Symposium on Principles of Distributed Computing (PODC)*, pages 347–351, 2003.

[44] P. B. Gibbons and Y. Matias. New sampling-based summary statistics for improving approximate query answers. In *Proceedings of ACM SIGMOD International Conference on Management of Data (SIGMOD)*, pages 331–342, 1998.

[45] P. B. Gibbons and S. Tirthapura. Estimating simple functions on the union of data streams. In *Proceedings of ACM Symposium on Parallel Algorithms and Architectures (SPAA)*, pages 281–291, 2001.

[46] P. B. Gibbons and S. Tirthapura. Distributed streams algorithms for sliding windows. In *Proceedings of ACM Symposium on Parallel Algorithms and Architectures (SPAA)*, pages 63–72, 2002.

[47] L. Golab, D. DeHaan, E. D. Demaine, A. López-Ortiz, and J. I. Munro. Identifying frequent items in sliding windows over on-line packet streams. In *Proceedings of ACM SIGCOMM Conference on Internet Measurement*, pages 173–178, 2003.

[48] L. Golab, D. DeHaan, A. López-Ortiz, and E. D. Demaine. Finding frequent items in sliding windows with multinomially-distributed item frequencies. In *Proceedings of International Conference on Scientific and Statistical Database Management*, pages 425–426, 2004.

[49] M. B. Greenwald and S. Khanna. Power-conserving computation of order-statistics over sensor networks. In *Proceedings of ACM SIGMOD-SIGACT-SIGART Symposium on Principles of Database Systems (PODS)*, pages 275–285, 2004.

[50] D. Grunwald, P. Levis, K. I. Farkas, C. B. Morrey, and M. Neufeld. Policies for dynamic clock scheduling. In *Proceedings of Symposium on Operating System Design and Implementation (OSDI)*, pages 73–86, 2000.

[51] S. Guha, N. Koudas, and K. Shim. Data-streams and histograms. In *Proceedings of ACM Symposium on Theory of Computing (STOC)*, pages 471–475, 2001.

[52] G. H. Hardy, J. E. Littlewood, and G. Polya. *Inequalities*. Cambridge University Press, 1952.

[53] L. Huang, X. Nguyen, M. Garofalakis, J. Hellerstein, A.D. Joseph, M. Jordan, and N. Taft. Communication-efficient online detection of network-wide anomalies. In *Proceedings of IEEE International Conference on Computer Communications (INFOCOM)*, pages 134–142, 2007.

[54] P. Indyk. Stable distributions, pseudorandom generators, embeddings and data stream computation. In *Proceedings of IEEE Symposium on Foundations of Computer Science (FOCS)*, pages 148–155, 2000.

[55] S. Irani and K. Pruhs. Algorithmic problems in power management. *ACM SIGACT News*, 32(2):63, 2005.

[56] S. Irani, S. Shukla, and R. Gupta. Online strategies for dynamic power management in systems with multiple power-saving states. *ACM Transactions on Embedded Computing Systems*, 2(3):325–346, 2003.

[57] S. Irani, S. Shukla, and R. K. Gupta. Algorithms for power savings. *ACM Transactions on Algorithms*, 3(4):41, 2007.

[58] N. Jain, P. Yalagandula, M. Dahlin, and Y. Zhang. INSIGHT: A distributed monitoring system for tracking continuous queries. In *Proceedings of ACM Symposium on Operating Systems Principles (SOSP)*, pages 1–7, 2005.

[59] C. Jin, W. Qian, C. Sha, J. X. Yu, and A. Zhou. Dynamically maintaining frequent items over a data stream. In *Proceedings of ACM Conference on Information and Knowledge Management (CIKM)*, pages 287–294, 2003.

[60] B. Kalyanasundaram and K. Pruhs. Eliminating migration in multi-processor scheduling. *Journal of Algorithms*, 38:2–24, 2001.

[61] B. Kalyanasundaram and K. Pruhs. Minimizing flow time nonclairvoyantly. *Journal of the ACM*, 50(4):551–567, 2003.

[62] A. Karlin, M. Manasse, L. McGeoch, and S. Owicki. Competitive randomized algorithms for non-uniform problems. In *Proceedings of ACM-SIAM Symposium on Discrete Algorithms (SODA)*, pages 301–309, 1990.

[63] A. R. Karlin, C. Kenyon, and D. Randall. Dynamic TCP acknowledgement and other stories about $e/(e-1)$. In *Proceedings of ACM Symposium on Theory of Computing (STOC)*, pages 502–509, 2001.

[64] R. M. Karp, S. Shenker, and C. H. Papadimitriou. A simple algorithm for finding frequent elements in streams and bags. *ACM Transactions on Database Systems*, 28(1):51–55, 2003.

[65] R. Keralapura, G. Cormode, and J. Ramamirtham. Communication-efficient distributed monitoring of thresholded counts. In *Proceedings of ACM SIGMOD International Conference on Management of Data (SIGMOD)*, pages 289–300, 2006.

[66] J. H. Kim and K. Y. Chwa. Non-clairvoyant scheduling for weighted flow time. *Information Processing Letters*, 87(1):31–37, 2003.

[67] T. W. Lam, L. K. Lee, I. K. K. To, and P. W. H. Wong. Speed scaling functions for flow time scheduling based on active job count. In *Proceedings of European Symposium on Algorithms (ESA)*, pages 647–659, 2008.

[68] L. K. Lee and H. F. Ting. Maintaining significant stream statistics over sliding windows. In *Proceedings of ACM-SIAM Symposium on Discrete Algorithms (SODA)*, pages 724–732, 2006.

[69] S. Leonardi and D. Raz. Approximating total flow time on parallel machines. In *Proceedings of ACM Symposium on Theory of Computing (STOC)*, pages 110–119, 1997.

[70] J. Y. T. Leung, editor. *Handbook of Scheduling: Algorithms, Models, and Performance Analysis*. CRC Press, 2004.

[71] A. Manjhi, V. Shkapenyuk, K. Dhamdhere, and C. Olston. Finding (recently) frequent items in distributed data streams. In *Proceedings of International Conference on Data Engineering (ICDE)*, pages 767–778, 2005.

[72] J. McCullough and E. Torng. SRPT optimally utilizes faster machines to minimize flow time. In *Proceedings of ACM-SIAM Symposium on Discrete Algorithms (SODA)*, pages 350–358, 2004.

[73] J. Misra and D. Gries. Finding repeated elements. *Science of Computer Programming*, 2(2):143–152, 1982.

[74] R. Motwani, S. Phillips, and E. Torng. Nonclairvoyant scheduling. *Theoretical Computer Science*, 130(1):17–47, 1994.

[75] K. Mouratidis, S. Bakiras, and D. Papadias. Continuous monitoring of top-k queries over sliding windows. In *Proceedings of ACM SIGMOD International Conference on Management of Data (SIGMOD)*, pages 635–646, 2006.

[76] T. Mudge. Power: A first-class architectural design constraint. *Computer*, 34(4):52–58, 2001.

[77] S. Muthukrishnan. *Data streams: algorithms and applications*. Now Publishers, 2005.

[78] C. Olston, J. Jiang, and J. Widom. Adaptive filters for continuous queries over distributed data streams. In *Proceedings of ACM SIGMOD International Conference on Management of Data (SIGMOD)*, pages 563–574, 2003.

[79] C. A. Phillips, C. Stein, E. Torng, and J. Wein. Optimal time-critical scheduling via resource augmentation. In *Proceedings of ACM Symposium on Theory of Computing (STOC)*, pages 140–149, 1997.

[80] P. Pillai and K. G. Shin. Real-time dynamic voltage scaling for low-power embedded operating systems. In *Proceedings of ACM Symposium on Operating Systems Principles (SOSP)*, pages 89–102, 2001.

[81] K. Pruhs. Competitive online scheduling for server systems. *SIGMETRICS Performance Evaluation Review*, 34(4):52–58, 2007.

[82] K. Pruhs, J. Sgall, and E. Torng. Online scheduling. In J. Y. T. Leung, editor, *Handbook of Scheduling: Algorithms, Models, and Performance Analysis*, chapter 15.1–15.41. CRC Press, 2004.

[83] K. Pruhs, P. Uthaisombut, and G. Woeginger. Getting the best response for your erg. *ACM Transactions on Algorithms*, 4(3):38, 2008.

[84] K. Pruhs, R. van Stee, and P. Uthaisombut. Speed scaling of tasks with precedence constraints. In *Proceedings of International Workshop on Approximation and Online Algorithms (WAOA)*, pages 307–319, 2005.

[85] N. Rohrer. The IBM PowerPC 970FX power envelope and power management. http://www.ibm.com/developerworks/library/pa-powerenv/.

[86] L. Schrage. A proof of the optimality of the shortest remaining processing time discipline. *Operations Research*, 16(3):687–690, 1968.

[87] J. Sgall. On-line scheduling. In A. Fiat and G. J. Woeginger, editors, *Online Algorithms: The State of Art*, pages 196–231. Springer, 1998.

[88] I. Sharfman, A. Schuster, and D. Keren. A geometric approach to monitoring threshold functions over distributed data streams. *ACM Transactions on Database Systems*, 32(4):23, 2007.

[89] D. D. Sleator and R. E. Tarjan. Amortized efficiency of list update and paging rules. *Communications of the ACM*, 28(2):202–208, 1985.

[90] M. Weiser, B. Welch, A. Demers, and S. Shenker. Scheduling for reduced CPU energy. In *Proceedings of Symposium on Operating System Design and Implementation (OSDI)*, pages 13–23, 1994.

[91] A. C. Yao. Some complexity questions related to distributive computing. In *Proceedings of ACM Symposium on Theory of Computing (STOC)*, pages 209–213, 1979.

[92] F. Yao, A. Demers, and S. Shenker. A scheduling model for reduced CPU energy. In *Proceedings of IEEE Symposium on Foundations of Computer Science (FOCS)*, pages 374–382, 1995.

[93] K. Yi and Q. Zhang. Optimal tracking of distributed heavy hitters and quantiles. To appear in *Proceedings of ACM SIGMOD-SIGACT-SIGART Symposium on Principles of Database Systems (PODS)*, 2009.