Stony Brook University

Using Locality to Tackle Modern Algorithmic Challenges

A Dissertation presented

by

Samuel McCauley

to

The Graduate School

in Partial Fulfillment of the

Requirements

for the Degree of

Doctor of Philosophy

in

Computer Science

Stony Brook University

August 2016

Stony Brook University

The Graduate School

Samuel McCauley

We, the dissertation committe for the above candidate for the

Doctor of Philosophy degree, hereby recommend

acceptance of this dissertation

Michael Bender - Dissertation Advisor
Professor, Computer Science

Rob Johnson - Chairperson of Defense
Research Assistant Professor, Computer Science

Jing Chen
Assistant Professor, Computer Science

Jeremy Fineman
Department of Computer Science, Georgetown University

This dissertation is accepted by the Graduate School

Nancy Goroff
Interim Dean of the Graduate School

Abstract of the Dissertation

Using Locality to Tackle Modern Algorithmic Challenges

by

Samuel McCauley

Doctor of Philosophy

in

Computer Science

Stony Brook University

2016

As computers become faster and more parallel, the number of computations a process executes is becoming too simplistic a predictor of performance. In many circumstances, modern algorithmic research can no longer rely on the assumption that each computation roughly corresponds to a set amount of time. For example, large-scale computers may spend far more time moving data around than performing computation, in which case minimizing data transfers is the best way to improve wall-clock performance. Similarly, manufacturing plants may need to focus on minimizing maintenance costs. While they must still guarantee good throughput, the maintenance costs are what most impact their profits. Under such circumstances, algorithms must be analyzed using models that incorporate these new costs.

This work focuses on using locality as a tool to improve performance across different models and objectives. We discuss three specific areas: scheduling, external memory, and high-performance computing.

Our scheduling research deals with machines that need to be calibrated at large cost. Since jobs can only be scheduled on calibrated machines, we want to group jobs as much as possible—while retaining good throughput—to minimize the number of calibrations. We give algorithms and results for the offline setting (where jobs are known ahead of time), and the online setting (where jobs must be handled as they arrive).

The external-memory model measures the number of hard disk accesses made by an algorithm. This model is applicable for I/O-bound algorithms (such as large-

scale sorting) whose performance is limited by hard drive access time rather than computation time. In this work, we focus on the packed memory array, a data structure used as a building block for external-memory algorithms. We show that we can retain optimal packed memory array performance while also providing history independence, a security guarantee.

Finally, we discuss a new type of memory, High-Bandwidth Memory, which has been proposed for the next generation of supercomputers. We modify the external-memory model to take this new kind of memory into account, and give theoretical analysis. We also give experimental results which show that this memory has the potential to significantly improve sorting performance.

Dedication Page

This thesis is dedicated to my parents,
Tom and Jamie McCauley, and to my
wife, Shikha Singh.

Table of Contents

Acknowledgments

Writing a thesis acknowledgment is a task that can never be fully completed. I can't hope to list every person who's helped me during my doctoral studies. There have been innumerable people who have helped me, who I've talked to, and who have helped me over the last six years. So I would like to begin by acknowledging the myriad people who have helped me—in so many ways—during this time, but who I am not including in this acknowledgment explicitly, from those whom I've spent time with at conferences, to those I've discussed problems with during seminars at Stony Brook, to the taxi driver who finished driving me home even though I'd realized I'd run out of cash after landing at the Albuquerque airport at 2 AM. I'm truly thankful to so many people around me, and I can only apologize to those who are not listed here by name.

To begin, I would like to acknowledge my advisor, Michael Bender. His passion for teaching and research has served as an inspiration and will contin ue to in my future career. It is due to this passion that in my studies I've not just learned about how to create new ideas, but how to effectively communicate them. These complementary aspects are fundamental to me as a person and an academic, and it's been a uniquely rewarding experience to have an advisor who emphasizes them.

I want to give special thanks to my family for their help and support: my parents Tom and Jamie, and my siblings Spencer, Max, and Tanya. I would like to thank my now-wife Shikha, who has provided irreplacable support over the last three years—even as I write this sentence, in fact. All of my family has given me food and coffee, helped me when I've messed up paperwork, and provided a home I could count on whenever I was in need.

I've been fortunate to visit several other researchers, which has broadened my horizons and allowed me to work with new people in new environments. In addition to the technical knowledge I've gleaned, I've come to see how others approach research. In order of visitation, I'd like to thank Vitus Leung, Seth Gilbert, Minming Li, and Frederic Vivien. I was incredibly fortunate to find four hosts so welcoming and inclusive to a new student in their mists—and to visit four places with such excellent food.

I would like to thank my friends who have made my stay at Stony Brook so enjoyable. Thanks to Rishab, Hau, Bill, Jiemin, Rik, Polina, and Rami for bringing a smile to my face during the first year. And thanks to the many friends I've made in the years after that: Mike, Bryan, Connor, Sean, and Amogh. I would like to thank my labmates for providing invaluable discussion and commisseration: Dzejla, Pablo,

Roozbeh, Mayank (honorary labmate), Shikha, Prashant, Tyler, and Yuren.

I would like to thank my defense committee, Rob Johnson, Jing Chen, and Jeremy Fineman, for their advice, for helping in my graduation, and their patience through this process.

One of the crucial parts of graduate student research is funding, and I've been fortunate to receive funding from many generous institutions. My funding has included NSF grants CCF 1114809, CCF 1217708, IIS 1247726, IIS 1251137, CNS 1408695, and CCF 1439084, a Chateaubriand Fellowship, an EAPSI fellowship, and funding from the City University of Hong Kong. I also spent over 14 months as a Graduate Student Intern at Sandia National Laboratories, a multi-program laboratory managed and operated by Sandia Corporation, a wholly owned subsidiary of Lockheed Martin Corporation, for the U.S. Department of Energy's National Nuclear Security Administration under contract DE-AC04-94AL85000.

Finally, I should specifically thank my coauthors. I've learned much more writing papers with others than I have reading and studying myself. Much of what I've learned over the last six years is due to their contributions. In reverse alphabetical order, my coauthors include David Zage, Hòa Vũ, Frédéric Vivien, Yuan Tang, Shikha Singh, Bertrand Simon, Arun Rodrigues, David Resnick, Cindy Phillips, Ben Moseley, Branden Moore, Jie Meng, Andrew McGregor, Jayson Lynch, Andrea Lincoln, Vitus Leung, Tom Kroger, Fulya Kaplan, Rob Johnson, Scott Hemmert, Simon Hammond, Golnaz Ghasemiesfeh, Pramod Ganapathi, Martin Farach-Colton, Jeremy Fineman, Roozbeh Ebrahimi, Erik Demaine, Ayse Coskun, Alex Conway, Rezaul Chowdhury, Jing Chen, David Bunde, Ritwik Bose, Jon Berry, and Michael Bender. As is clear from this large list, collaboration has been a cornerstone of my thesis work, and I cannot thank these coauthors enough for what I have learned.

Chapter 1

Introduction

Algorithms is the study of how to accomplish a goal (in particular, to satisfy a set of constraints) while minimizing a cost. As computing evolves as a field—as computers become larger, more distributed, and more parallel, for example—these goals and costs have evolved with it.

Classical time-based algorithm analysis is usually done in the RAM model [47], which essentially counts the number of operations if the algorithm was written in a common programming language like C. Algorithms are generally analyzed asymptotically, so the precise list of operations often does not affect the final result. There are many exceptions where the exact operations do matter: for example, important past works have made use of word-level parallelism to improve their results [5, 34, 63], or restricted themselves to fast operations like addition and subtraction (eschewing multiplication and division) [35]. However, the overall focus has been on reducing strain on the processor to improve overall running time.

Much of this work focuses on scheduling, which has a slightly broader history of algorithmic costs. In particular, a schedule often tries to minimize some scheduling-related cost (like throughput, or the time required to complete the whole schedule). The schedule itself must be computable efficiently. Research in scheduling has examined many different machine-related costs like energy usage [14, 124], the number of required machines [44, 46, 76], or machine startup costs [13].

Modern machines require new algorithmic considerations. For example, a computer processing more data may be *I/O bound*, spending more time accessing data than actually performing computation on it. In this case, the most efficient algorithm is one that minimizes data accesses. This has led to a long line of research on creating efficient algorithms that minimize the number of I/Os, while still retaining good computation time [3, 64, 114]. These ideas are particularly applicable to research on data structures and databases, where large amounts of data must be updated and accessed as quickly as possible.

Modern theoretical computer science has research considering further, diverse costs, which are not considered here. Some common examples are energy costs [91, 93, 100], for algorithms run on expensive machines or data centers which spend most of their money on powering (and particularly cooling) their machines, and communication costs [43, 110], for extremely large and parallel high-performance computers which must move the results of computations large distances. These costs can be

extremely complicated and difficult to deal with for algorithms researchers, so they are often condensed into simplified models. One example of such a model is MapReduce [1, 49, 101], which gives a model that captures much of the power of a large-scale computer without getting too bogged down in parameterization. MapReduce is a further example of a modern cost metric: the algorithm designer must reduce the number of rounds of pairwise communication.

These new costs are diverse, and yet many of them share a common trait: we want to group similar things close together to reduce the cost. This may mean storing two data points in the same page if we need them at similar times in the future, assigning processors on the same rack to run a large-scale job on a high-performance computer to reduce communication costs, or grouping jobs together on a machine to reduce its startup costs. We refer to this trait as *locality*.

This is the unifying idea behind the results presented in this thesis. By improving the locality of algorithms and data structures, we can improve their performance with respect to these modern cost functions, even in drastically different situations.

We discuss three topics. The first is scheduling with calibrations, a scheduling problem where the cost is the number of times the machines need to be calibrated to complete the schedule. We consider several variants, particularly an offline variant (where the entire workload that must be scheduled is known ahead of time), and an online variant (where tasks may come up suddenly and must be worked into the existing schedule). The second is the history-independent PMA, which stores items relatively densely so that they can be efficiently accessed on a disk, while retaining both good speed against adversarial inserts, and history-independence, a security parameter. Finally, we discuss high-bandwidth memory, a new kind of architecture which speeds up memory-bandwidth-bound computation in high-performance computing applications.

These topics are discussed one by one in the remaining chapters. For each, we give a more detailed motivation and related work before defining the model and giving our results.

Chapter 2

Scheduling With Calibrations

Scheduling is an area of computer science which studies how to efficiently execute a series of tasks, or *jobs*. A scheduling algorithm decides when and where to run the jobs (possibly with some constraints) to minimize an objective function.

Recent work has considered scheduling on machines that need to be calibrated [7, 19, 60]. These machines are assumed to be high-performance and high-maintenance, and perform very precise tasks, which is why they need to be calibrated before they can be trusted to perform a job. Performing these calibrations can be very expensive, possibly much more than the cost of running the machines. The calibrations are only effective for a set period of time, after which the machine may no longer be accurate, and must be (expensively) recalibrated before running more jobs.

The goal of algorithms in this framework is to minimize the number of calibrations. This objective function has an interesting property: in most objective functions (say, minimizing total waiting time, or maximizing number of jobs completed by a deadline), it is generally profitable to schedule a job as early as possible. However, to minimize calibrations, the algorithm wants to group jobs together as much as possible, possibly delaying some jobs considerably. The known algorithms in this framework are designed to effectively handle this tradeoff: grouping jobs for more efficient calibrations, while maintaining a reasonable schedule where jobs are not delayed too long.

2.1 Motivation

The original motivation for scheduling with calibrations came directly from the Integrated Stockpile Evaluation (ISE) program to test nuclear weapons periodically. The tests for these weapons require calibrations that are expensive in a monetary (though not necessarily temporal) sense. This motivation is specified further in [19, 38, 97].

However, this motivation can extend to any context where machines performing jobs must be calibrated periodically. High-precision machines require periodic calibration to ensure precision. Methods for calibrating these machines is itself an area of research; some examples include [103, 116, 126, 127]. Oftentimes, machines are no longer considered to be accurately calibrated after a set period of time. There are many guidelines to determine this *calibration interval*, or period of time between calibrations, both from industry and academia [15, 17, 85, 88, 99, 122].

Calibrations have applications in many areas, including robotics [25, 58, 98], pharmaceuticals [12, 17, 61], and digital cameras [11, 16, 24, 128].

2.2 Background on Scheduling

In this section we give background on scheduling. In particular we provide some formalism for scheduling problems, and some (very brief) related work on past research in scheduling.

The input to a scheduling problem is a set of n jobs $\mathcal{J}$, and possibly a bound on the number of *processors* P. We refer to processors as *machines* interchangeably. Each job $j \in J$ has a release time r_j, a processing time p_j, a weight w_j, and possibly a deadline d_j. A schedule is an assignment of each job in $\mathcal{J}$ to a set of time intervals. Each time interval consists of a half-open interval $[t_1, t_2)$, and a machine m.

The schedule must meet certain constraints. A job must be scheduled for sufficient time to complete its processing: the sum of all assigned time intervals must be at least p_j.[1]

A single job cannot be scheduled concurrently, so all time intervals $[t_1, t_2)$ for a given job must be disjoint regardless of what machine they are assigned to. Similarly, a machine can only process one job at a time, so the set of all time intervals $[t_1, t_2)$ assigned to a given machine m must also be disjoint. All jobs must be scheduled after their release time ($t_1 > r_j$ for all time intervals $[t_1, t_2)$ that j is assigned to). If jobs have deadlines, they must be completed before their deadlines ($t_2 \leq d_j$ for all time intervals $[t_1, t_2)$ that j is assigned to). A schedule is *preemptive* if some jobs are assigned to multiple, noncontiguous time intervals, or a job is split among multiple machines. Otherwise, if each job is assigned to a single contiguous time interval on a single machine, the schedule is called *nonpreemptive*.

There is a standard notation to denote the constraints and objectives of scheduling problems. After all, even the basic notions above (before objective functions are considered) can lead to many different variants. The notation developed in [71] gives a 3-field classification $\alpha|\beta|\gamma$. The first field, α, gives the machine environment, i.e. a bound P on the number of machines, or information about the machines such as whether or not they all run jobs at the same speed. The second field, β, gives information about the jobs themselves: do they have deadlines, are they preemptable, etc. Finally, the third field γ is the cost function used. An example cost function is $C_{\max}$, the maximum completion time of a job.

Scheduling is an extremely broad subarea, and there has been extensive work on many variants on both constraints and objectives. A fairly extensive survey of classic results in offline scheduling is [71]. A more recent survey that focuses on online

[1] With speed scaling s, as in [60], the sum must be at least p_j/s.

scheduling is [105].

2.3 Modeling Calibrations

In this work, we assume unit processing times, i.e., $p_j = 1$ for all j. We further assume that all release times and deadlines are integral. This somewhat simplifies the problem: we can assume that all t_1, t_2 are integral, and preemption is unnecessary.

Thus, in this setting, we want to find an assignment of jobs to unit time intervals (and associated machines). For simplicity, we refer to a unit time interval $[t_1, t_2)$ (where $t_2 - t_1 = 1$) as a *time slot*. In particular, we generally refer to a time slot by its starting time t_1.

We formally model calibrations as follows. We can *calibrate* a machine for unit cost. The machine stays *calibrated* for $T \geq 2$ time steps, after which it must remain idle until it is recalibrated. We refer to these T time steps following a calibration as an *interval*. Calibrating a machine is instantaneous, meaning that a machine can be recalibrated between two job executions running in successive time steps. Minimizing the number of calibrations helps minimize the total cost of an ISE instance.

A job can only be scheduled when a machine is calibrated. Thus if a job is assigned to a time slot beginning at t_1 on machine m, then m must be calibrated at some point in $(t_1 - T, t_1]$. Since all jobs are scheduled at integer times, all machines are calibrated at integer times without loss of generality. We assume that machines are calibrated instantaneously; we may calibrate a machine at t and $t + T$, assigning jobs to time slots $t + T - 1$ and $t + T$.

We refer to the T time steps following the calibrations as an *interval*. In this view, the goal is to find a small set of intervals during which we can schedule all the jobs. Our analysis generally uses this view, manipulating intervals (and the jobs scheduled in the intervals) to minimize the cost.

Our objective differs between the online (Chapter 2.6) and offline (Chapter 2.5) cases. In the offline case we want to schedule unit-sized jobs (with deadlines) while minimizing the number of calibrations. This may be denoted $P|r_j, d_j, p_j = 1|\#$ calibrations. In the online case, we wish to minimize a tradeoff between flow and calibrations. This may be denoted as $P|\textit{online-time}, p_j = 1|\sum F_j + G(\#$ calibrations$)$.[2]

[2]See [105] for more details about this notation for online problems. In short, online-time denotes online algorithms as described above.

2.4 Related Work

Scheduling with calibrations was originally introduced in [19], which contains roughly the same contents as Chapter 2.5.

Later, Fineman and Sheridan developed results for jobs with non-unit processing times [60]. In particular, they note that scheduling with calibrations generalizes the machine minimization problem: given a set of jobs, what is the smallest number of machines that the jobs can be run on so that they all meet their deadline? This problem is trivial with unit-sized jobs, but difficult with generalized processing times. In particular, the best known approximation for machine minimization is $O(\sqrt{\log n / \log \log n})$ [44]. While scheduling with calibrations is at least as difficult—an approximation algorithm for calibrations leads to an approximation algorithm for machine minimization—Fineman and Sheridan show that they can meet nearly the same bounds. Given a machine minimization approximation algorithm which uses s-speed augmentation and αm machines, there is a calibrations algorithm that gives an $O(\alpha)$ approximation for the nuber of calibrations using αm machines and s-speed augmentation.

More recently, Angel et al. developed dynamic programming algorithms for further generalizations—where for example there are multiple kinds of calibrations, or jobs have nonunit processing time, but are preemptible [7]. These results are polynomial time (though often with a moderately large exponent), giving some further evidence that many calibrations scheduling problems may be polynomial-time solvable. On the other hand, they show that if there are several types of calibrations, and jobs have arbitrary processing times, the problem is NP-hard even if preemption is allowed.

Relationship to Classic Scheduling Problems

While scheduling with calibrations is relatively new, previous work has examined problems with a similar flabor. Unfortunately, even some simple variants like $1|r_j|L_{\max}$ (minimize max lateness on one processor with release times) and $1|r_j|\sum C_j$ (minimize average completion time on one processor with release times) are NP-hard [78]. Some others, such as $P|r_j, d|(\sum w_j|d - C_j|)$ (multiprocessor scheduling trying to schedule jobs as close as possible to a common deadline) are NP-hard but can be solved with a pseudopolynomial time algorithm [73].

Many related uniprocessor algorithms can be solved efficiently, such as $1||f_{\max}$, which does not have release times for jobs and has any minimization function that decreases with job completion time [78]. This class of objective functions is explored further for the multiprocessor case with unit jobs in [51]. Minimizing the number of calibrations does not have this property; there are cases where scheduling a job later

reduces the total cost. But possibly more importantly, there is a further assumption that $f_{\max}$ only depends on the values of individual completion times c_j, not on the relationship between multiple completion times. We show in Chapter 2.5.1 that there is in fact an optimal algorithm for the uniprocessor case with unit processing times that runs in polynomial time nonetheless.

Another similar problem that can be solved efficiently is $P|pmtn|C_{\max}$, multiprocessor scheduling with preemption to minimize maximum completion time [78]. While we do not have preemption in our model, unit job lengths are similar to allowing preemption, as there are no intermediary timesteps while a job runs where another job can be started. Furthermore, the hardness of the problem without preemption is due to a reduction to the Partition problem, which depends on large, indivisible jobs; unit jobs avoid hardness in a similar way to preemption.

Interval Scheduling

Interval scheduling is similar to our current problem definition, but all jobs run for the entire time between their release and due dates. In other words, $p_i = d_i - r_i$ using the notation we used in Chapter 2.2. For this reason, there is no choice about when to run a job. If a job does not run at its release time, it is skipped, is late, or is only partially run. The goal, then, is not to efficiently determine when to schedule a job, but to determine which jobs to schedule, and what machines to schedule them on [82]. This area has been studied in both an offline and online [59, 89, 107] context. There exist variants of the problem, such as allowing each job to be scheduled in any of k intervals (i.e. [57,109]), and a variant closer to real-world applications where each job may have a priority [83]. However, these variations seem less applicable to scheduling with calibrations.

Much like scheduling with calibrations, the interval scheduling problem initially came from a real-world Operations Research problem—namely, Dantzig and Fulkerson described a tanker scheduling problem in [48] that would now be considered an instance of interval scheduling [82]. Since then many practical problems have been solved using the guarantees of interval scheduling. In all cases, the scheduling requirements are very rigid; jobs must be scheduled at their release time exactly. Note that some of these, particularly aircraft maintenance, are similar to scheduling with calibrations, but with less time flexibility:

- aircraft maintenance – when to perform aircraft maintenance on a given plane (this is an example where jobs may have priority, as some maintenance jobs can be delayed and some cannot; this problem specifically motivated [83]),
- classroom assignment [40],
- crew scheduling (i.e. assigning bus drivers to routes [90,121]– in fact the interval scheduling problem is sometimes referred to as "The Bus Driver Scheduling

Problem"),

- sattelite photography [65]– given a requested set of photographs for the Earth and a set of satellite orbits, how can the photographs be efficiently taken?
- cottege rental [8]– efficiently assigning cottages to renters given requests for certain vacation times,
- channel assignment [72, 75]– a problem in VLSI layout minimizing ''channels" connecting two different components,
- replacement policies for caches [30],
- identifying protein segments in chains of amino acids [42]. To be more specific, this deals with an intermediate step in Nuclear Magnetic Resonance wherein resonance peaks must be matched with specific amino acids in a sequence.

In general, Interval Scheduling on P processors can be solved by finding a maximum weight P-colorable subgraph on the interval graph representing the job intervals [57]. This was shown to be solvable in polynomial time in [123]. Similarly, a solution using max flow was presented in [10], which was extended to a more general greedy algorithm in [29] (note that this solution assumes identical processors).

Not all interval scheduling problems can be solved so easily, however. Most notably, differences between machines can result in NP-hardness (if there are at least 3 types of machines [26, 56]) and even approximation hardness if the machines have differing availabilities or jobs are constrained to certain machines [82] (to be specific, there is no constant-factor approximation unless P = NP). In this thesis, however, we assume identical machines, and so far we have not seen any algorithmic advantage to assigning additional constraints that would distinguish machines. Interval scheduling has been studied in this context in, for example [29], and as mentioned it is solvable in polynomial time with this restriction. Therefore, interval scheduling could be used to develop an efficient algorithm for a solution (whether optimal or approximate) for scheduling with calibrations if an appropriate reduction is found.

Scheduling with Unit Jobs

As mentioned in Chapter 2.4, many scheduling problems with different processing lengths are NP-hard, and we show in Theorem 2.1 that this is true for scheduling with calibrations as well. Concentrating on unit-sized jobs gives the opportunity to research efficient, optimal solutions to problems that would otherwise be NP-hard.

However, not all unit-sized job scheduling problems give efficient results. $P|r_j, d_j, p_j = 1, assign|L_{\max}$ was shown to be strongly NP-hard in [50] (deadlines may be assigned to jobs in that problem). A more complex example of strongly NP-hard scheduling problem was given by Lee et al. in [86]. For $P|r_j, d_j, p_i = 1|L_{\max}$, identical to our problem other than the objective function, Baker recently gave a polynomial-time algorithm [36].

Machine Minimization

Machine minimization is a classic scheduling problem, in which we want to schedule a set of jobs $\mathcal{J}$ with deadlines using the smallest possible number of machines. Previous work features a large number of clever techniques in order to achieve a given bound. In this section we focus on the continuous version of machine minimization, in which jobs have release times and deadlines as described above (the discrete version limits jobs to be scheduled during particular intervals).

As discussed in [60], scheduling with calibrations generalizes the machine minimization problem. The idea behind the proof is that if T becomes arbitrarily large, we need only calibrate each machine once to process all jobs assigned to that machine. Thus, the machine minimization problem can be regarded as scheduling with calibrations with extremely large T.

The implications of this are twofold. First, lower bounds for machine minimization apply to scheduling with calibrations directly. Upper bounds for machine minimization can be regarded as an informal barrier as well: if we want to beat, for example, the $O(\sqrt{\log n / \log \log n})$ offline machine minimization bound, we must improve on the clever machine minimization techniques used in previous papers. Second, it is possible that the techniques used in for machine minimization can help us improve calibrations results. The work in [60] could be considered a strong result along these lines.

For offline machine minimization, the best known non-preemptive algorithm has an approximation ratio of $O(\sqrt{\log n / \log \log n})$ [44]. However, even with only $(1+\epsilon)$-speed augmentation, an approximation ratio of 2 can be obtained [76] (the running time of the algorithm is a function of ϵ). There exist special cases with small approximation ratios, for example a 2-approximation if the jobs have equal release dates, or a 6-approximation if the jobs have equal processing times [125]. A classic result shows that the problem is strongly NP-complete [66], and there is a lower bound on the approximation ratio of $2-\epsilon$ [46]. Closing this substantial gap, between $2-\epsilon$ and $O(\sqrt{\log n / \log \log n})$, remains a major open problem in scheduling theory.

Online machine minimization is of particular importance to scheduling with calibrations, as it has key connections to extending the online case to allow job deadlines. (The results in this thesis give throughput guarantees rather than meeting deadlines for individual jobs.) Recently, Chen et al. gave an $O(m^2 \log m)$-competitive algorithm [41] when preemption is allowed, where m is the optimal number of machines. In particular, this is a constant approximation if the optimal number of machines is also constant. For laminar instances (where if $[r_j, d_j]$ and $[r_{j'}, d_{j'}]$ overlap, one of these intervals must contain the other), they give a $O(\log m)$-competitive schedule. For agreeable instances (where $r_j < r_{j'}$ implies $d_j \leq d_{j'}$ for all $j, j' \in \mathcal{J}$), they give a $O(1)$-competitive schedule. Previous work gave an $O(\log p_{\max}/p_{\min})$-competitive al-

gorithm, where $p_{\max}$ and $p_{\min}$ are the maximum and minimum processing requirement for jobs, respectively [102]. This paper also gave a lower bound of 5/4 for the online preemptive case, the best currently known.

2.5 Offline Scheduling with Calibrations

In offline scheduling with calibrations, $\mathcal{J}$ is known to the algorithm. The algorithm must schedule jobs on machines to minimize the number of calibrations, while ensuring all jobs meet their deadlines. As mentioned, we assume that all jobs have unit size. We begin by motivating this somewhat by giving an NP-hardness proof.

Theorem 2.1. *Scheduling with calibrations is (weakly)* NP*-hard for any number of machines $P \geq 1$.*

Proof. We reduce the Partition problem to scheduling with calibrations, shown NP-complete by Karp in [79]. The Partition problem asks, given a multiset of integers S, if S can be partitioned into two subsets S_1, S_2 such that the sum of the elements in S_1 is equal to the sum of the elements of S_2.

Given a set S as specified in the Partition problem, we set $T = \frac{1}{2}\sum_{s_i \in S} s_i$, so S_1 (and therefore S_2) has $\sum_{s \in S_1} s = T$. For each $s_i \in S$ we set $p_i = s_i$. Our construction does not take advantage of release and deadlines, so all jobs have $r_i = 0, d_j = 2T$.

If there exists a schedule with a cost of two, all jobs can fit in two time intervals. Then S_1 and S_2 consist of the elements of the set corresponding to the jobs in the first and second interval respectively. If there does not exist a schedule with a cost of two, the set cannot be partitioned into two equal subsets (if it could, we would have been able to use S_1 and S_2 as two different time intervals). □

2.5.1 Single-Machine Algorithm

In this section, we give an optimal algorithm for single-machine scheduling with calibrations. We further show that if a scheduling instance is feasible on a single machine, then the best single-machine schedule is optimal, even if more machines are available.

EDF Scheduling with Intervals

Our algorithms schedule jobs based on the *Earliest Deadline First* (*EDF*) scheduling policy. Although this section primarily focuses on single-machine scheduling with calibrations, we also explain here how EDF works on multiple machines.

The EDF scheduler maintains a priority queue of jobs ordered by increasing deadline. If two jobs have the same deadline, the priority queue breaks ties arbitrarily but consistently.

EDF schedules jobs by iterating over time slots from earliest to latest. When EDF reaches a time slot t during which a new job j arrives ($r_j = t$), EDF adds j to the priority queue. Then EDF selects up to P jobs to run in time step t by popping these jobs from the priority queue.

Lemma 2.2 (FOLKLORE)**.** *If jobs have unit size, EDF finds a feasible schedule on single or multiple machines if one exists.*

EDF also schedules optimally in the multi-machine setting where machines can run only during specific active length-T intervals and otherwise must remain idle. To see why, create "dummy" jobs for each time step of each inactive machine, where each dummy job is constrained to run in its time step. Given the EDF schedule, we can permute the jobs among machines without changing their running time to create the proper intervals.

Optimal Single-Machine Algorithm

Although EDF always finds a feasible schedule, it may not minimize calibrations. We use EDF as a subroutine in another algorithm that we call *Lazy-Binning*. Lazy-Binning delays the start of an interval for as long as possible, until delaying it further would make it impossible to find a feasible schedule.

Before giving the algorithm, we define the notion of a *push*.

Definition 2.3. *A* push *is a move of an interval from time t to $t + 1$. This may mean that later intervals on the same machine are recursively pushed, so that no two intervals overlap.*

Lazy-Binning is the algorithm we obtain by performing all feasible pushes.

Theorem 2.4. *Lazy-Binning is optimal.*

Proof. To obtain a contradiction, assume that Lazy-Binning is not optimal. Consider the time $t \geq 0$ when Lazy-Binning first differs from every optimal schedule. In other words, from time 0 to $t - 1$, there exists at least one optimal schedule OPT that matches Lazy-Binning exactly, but no optimal schedule matches Lazy-Binning through time t. Furthermore, assume without loss of generality that OPT uses EDF to schedule jobs within an interval, as Lazy-Binning does. Thus, there are two ways for Lazy-Binning to differ from OPT:

Algorithm 1 Lazy-Binning

```
for each time step t until all jobs are scheduled do
    if no working interval that contains time t then
        Run EDF starting at time step t + 1
        if EDF cannot find a feasible schedule then
            Begin a working interval at time t.
        end if
    else
        Schedule jobs in the current interval using EDF
    end if
end for
```

- In Case A, Lazy-Binning starts an interval at time t while OPT remains idle.
- In Case B, Lazy-Binning does not start an interval at time t, OPT does.

We argue that neither of these cases is possible, leading to a contradiction.

Case A: Lazy-Binning starts an interval at time t, while OPT remains idle. This means that EDF failed to find a feasible schedule starting the next interval at time $t+1$ or later. But OPT is a feasible schedule that starts the next interval at time $t+1$.

Case B: OPT starts an interval at time t, whereas Lazy-Binning remains idle. Consider a schedule OPT$'$ which has identical intervals to OPT, except that the interval that OPT starts at t is pushed.

OPT$'$ matches Lazy-Binning until time $t+1$ so it cannot be an optimal schedule by our definition of t. Since OPT$'$ uses the same number of intervals as OPT, the cost for the schedule is the same, so it must be that OPT$'$ is infeasible; it cannot be suboptimal.

Let j be the first job that misses its deadline when EDF schedules jobs in OPT$'$. Any subsequent intervals that we add will be after j's deadline and cannot make OPT$'$ feasible. Therefore, j cannot be feasibly scheduled after the push, which means that Lazy-Binning starts an interval at t. □

We thus have an optimal algorithm for the single-machine case, as well as a test to see if a feasible schedule exists.

Lazy-Binning, as stated in Algorithm 1, runs in pseudopolynomial time, but the algorithm is easily modified to run in polynomial time. The issue is that if release

times and deadlines have exponential values, Lazy-Binning may iterate through an exponential number of time steps. However, an interval need never start more than n time steps before the deadline of any job, giving a polynomial solution.

2.5.2 Multiple-Machine Algorithm

We first characterize the structure of optimal solutions on multiple machines. We then give a 2-approximation algorithm. We describe conditions under which our algorithms find optimal solutions.

Round Robin Machine Assignment

Let $\mathcal{I} = (I_1, I_2, \ldots, I_N)$ be a sequence of intervals, a partial solution for a set of jobs, ordered by increasing start time. Ties are broken consistently. For our multiple-machine algorithm, ties can be broken by interval creation time, since the algorithm is incremental. Without loss of generality, no two intervals in $\mathcal{I}$ on the same machine have start times within T of each other.

Each interval $I_j \in \mathcal{I}$ is scheduled on some machine P_j. Assigning intervals to machines can be done using a round-robin method for any feasible schedule.

Lemma 2.5. *Given a feasible schedule S on P machines, there exists a schedule having the same cost and the same assignments of jobs to intervals where $P_j \equiv j \pmod{P}$.*

Proof. We use an inductive argument on $N = |\mathcal{I}|$, the number of intervals. The lemma is true for $N = 1$, where we can place the first interval on the first machine.

Suppose we have legally placed the first k intervals. Let t denote the time when I_{k+1} begins. If scheduling I_{k+1} on machine $(k+1) \pmod{P}$ is not feasible, there exists another interval I_c scheduled at time t_c on P_{k+1} such that $t - T < t_c \leq t$. Since all previous intervals are scheduled using round-robin, there are $P - 1$ intervals starting between times t_c and t. Each of these must also be scheduled at or before t and after $t - T$, so in total there are $P + 1$ intervals that must be scheduled in that range. Since there are P processors, two of the intervals must be scheduled concurrently on the same machine, which is infeasible. This contradicts the assumption that the schedule of the first k intervals is feasible. □

Lemma 2.5 means that our algorithm need not explicitly determine an assignment of intervals to machines.

Our algorithm returns a sequence of interval start times. Given this sequence, we assign intervals to machines using round robin and schedule jobs within intervals using EDF.

Before proceeding, we further specify how EDF breaks ties.

Invariant 2.6. *When EDF removes job j from the priority queue to run at time step t, there may be multiple machines on which j could run. EDF breaks ties by assigning j to the first interval $I \in \mathcal{I}$ that does not yet have a job assigned at time slot t (but contains time slot t).*

Characterization of Pushes

When there are multiple machines, intervals can overlap provide they are on different machines. This leads to more subtlety in how we characterize a push.

Definition 2.7. *A* push *is a move of an interval $I \in \mathcal{I}$ from time t to $t+1$ that may recursively push later intervals in order to maintain the following:*

1. *The ordering of intervals in $\mathcal{I}$ does not change. Thus, any other interval that starts at t and is ordered after I in $\mathcal{I}$ is also pushed.*
2. *The intervals on each processor are nonoverlapping. Thus, any interval that starts at $t+T$ on the same processor as I is also pushed.*

Suppose the push of an interval results in a new set of intervals $\mathcal{I}'$ and let t be the latest start time of any interval in $\mathcal{I}'$. This push is *feasible* for an instance of jobs if by augmenting $\mathcal{I}'$ by zero or more intervals, each starting at time t or later, all jobs can be feasibly scheduled.

We categorize feasible pushes of the last currently-scheduled interval into four cases. Because we only consider pushing the last interval, we do not need to worry about cascades of pushes. There may be jobs in J that cannot be scheduled in any of the intervals.

Definition 2.8. *Each feasible push of the last interval I fits one of four cases. When I is pushed, there may exist some job in I before the push that EDF no longer schedules after the push; denote this job j_{out}. Similarly, there may exist some job not scheduled before the push that EDF schedules after the push; denote this job j_{in}.*

Case 1: *There exists no job j_{out}.*

Case 2: *There exists a job j_{out} but no job j_{in}.*

Case 3: *Job j_{out} is after j_{in} in EDF order.*

Case 4: *Job j_{out} is before j_{in} in EDF order.*

Framework for Optimal Solutions

We extend Lazy-Binning to multiple machines.

Definition 2.9. *A* push algorithm *repeatedly creates a new interval at the end of $\mathcal{I}$ and then pushes that interval. The algorithm*

- *performs all Case-1 and Case-3 pushes,*
- *never performs an infeasible push, and*
- *has the flexibility to choose which Case-2 and Case-4 pushes to perform.*

Once the algorithm stops pushing the interval, if there are remaining unscheduled jobs, the algorithm creates a new interval and begins pushing again.

At intermediate steps of a push algorithm, EDF assigns jobs to intervals only temporarily. EDF may assign some job j to some interval, but when an interval is pushed or a new interval is added to $\mathcal{I}$, re-running EDF may assign j to a different interval.

When $P = 1$, all feasible pushes are Case 1 and Case 3, so Lazy-Binning is a push algorithm. Theorem 2.4 proves a push algorithm is optimal for $P = 1$. Theorem 2.16 shows that an optimal push algorithm always exists for any number of processors.

Algorithm 2 Push Algorithm

```
1:  t ← 0
2:  while not all jobs are scheduled do
3:      Remove all assignments of jobs to intervals
4:      if t is contained in P or more intervals then
5:          t ← next time with ≤ P − 1 intervals
6:      end if
7:      Create a new interval I at t
8:      Schedule jobs in all intervals with EDF
9:      while pushing I is Case 1 or Case 3 do
10:         Push I
11:         t ← t + 1
12:     end while
13:     Perform 0 to T Case-2 or Case-4 pushes on I
14: end while
```

Lemma 2.10. *A push of an interval I starting at t is of type Case 2 or Case 4 if and only if*

1. *before this push there is a $t_f \leq t+T$ such that jobs are scheduled in all calibrated time slots between t and t_f*
2. *each of these jobs is due no later than t_f, and*
3. *the push is feasible.*

The smallest such t_f has the property that $t_f - t \geq 2$ and at least two jobs are due exactly at t_f.

Proof. We first explain why $t_f - t$ must be greater than 1. If $t_f - t = 1$ then there exists a job j that must be scheduled at time t, the first time step of I before the push. Scheduling j later would miss its deadline. Thus after the push, when I starts at $t+1$, EDF does not schedule j in I. By Invariant 2.6, EDF schedules in the earliest possible interval in $\mathcal{I}$, so if j could be feasibly scheduled in an earlier interval it would be scheduled there before the push. Thus, j cannot be scheduled in an interval before I in $\mathcal{I}$. All later intervals must also start after the deadline of j by Definition 2.7, so j cannot be scheduled there either. Thus, if $f = 1$ the push is infeasible.

($\Leftarrow$) If these jobs do exist, there must be a j_{out}, because after we push I forward there are more jobs that must run between t and t_f than there are calibrated time steps between times t and t_f. The push cannot be Case 3 because before the push, any jobs released by time $t + T$ that precede j_{out} in EDF order would have been scheduled before j_{out} and they still will be after the push. If j_{in} is released at $t + T$, its deadline is at least $t + T + 1$. This is after j_{out} because j_{out} has deadline no later than $t_f \leq t + T$.

($\Rightarrow$) We now show that if there is a Case-2 or Case-4 push, such a t_f must exist. Let t_f be the deadline of j_{out}. We must have $t_f \leq t + T$ because if j_{out} could be scheduled at $t + T$, it could be scheduled in I after the push, ruling out Case 2. In Case 4, j_{out} is replaced in the set of scheduled jobs by a job later in EDF order. That would not happen if j_{out} could be feasibly scheduled in the new set of intervals.

After the push, all Y calibrated time slots from t to t_f must contain jobs since j_{out} cannot be scheduled. The configuration after the push is the same as before except that the slot in I at time t is replaced by a slot at time $t + T$. The EDF schedule is the same up to the slot at time t that is removed. This is the last slot in time t since I is the last interval. If there is no job in that slot, then the rest of the schedule can

run as before and there is no j_{out}, which contradicts the type of push. The job j_b that ran at time t in I must compete with newly-released jobs at time $t+1$. Some may have earlier deadlines than j_b so j_b may not recieve the very next slot. If j_b is never scheduled, it is j_{out} and all slots are full through t_f. Otherwise j_b is scheduled at some time $t < t_b \leq t_f$. By the same arguments as above, that slot must have held a job before the push. This job is either j_{out}, delayed by other jobs till past its deadline, or it displaces another till the cascade finally ends with the real j_{out}. All slots are then full until j_{out} fails to meet its deadline at time t_f.

All Y jobs the run between t and t_f in the pushed schedule also ran between t and t_f before the push. All jobs affected by the push (including j_{out}) ran no earlier than time t before the push. Every job in this set runs no earlier after the push (most at the same time except those involved in the cascade). All finish by t_f since only j_{out} was pushed later (to failure). Job j_{out} also ran before t_f before the push, since t_f is job j_{out}'s deadline. Thus, there were $Y+1$ jobs running in the $Y+1$ slots between time t and t_f before the push.

None of the $Y+1$ jobs just described has a deadline later than t_f. Consider the discussion above about how the push affects the schedule. All jobs scheduled between times $t+1$ and t_b, when the displaced job j_b is finally scheduled, had deadlines no larger than job j_b's deadline. This is because j_b was scheduled later by EDF. Job j_c, displaced by j_b, has a deadline no earlier than j_b's deadline and no earlier than the deadline for any job scheduled between t_b and t_c. Thus the displaced jobs have monotonically increasing deadlines and the current displaced job has a deadline no smaller than that of any job schedule since time t. Thus, job j_{out} has deadline no smaller than that of any job that ran between t and t_f.

We now show that at least two jobs are due exactly at the smallest t_f. Consider the smallest t_f. Because $t_f - t > 1$, there is at least one job scheduled at time $t_f - 1$ among the Y jobs described above (in interval I, for example). There is at least one job scheduled before $t_f - 1$ due after $t_f - 1$; otherwise $t_f - 1$ would satisfy the definition for t_f, violating the assumption that this is the smallest t_f. All jobs scheduled at $t_f - 1$ are due at t_f. There must be at least one such job in interval I. Thus, there are at least two jobs due at t_f. □

Corollary 2.11. *Every push algorithm does no more than T Case-2 or Case-4 pushes on any interval.*

Proof. Since there exists a job in I with deadline no later than t_f, by definition of the j_{out} for the push, I cannot be pushed past t_f. Otherwise that job could not be feasibly scheduled. Since $t_f \leq T$ the lemma follows. □

Lemma 2.12. *If intervals are added to a schedule, EDF schedules all jobs no later than without the added intervals.*

Proof. Let S and S' be the schedule before and after the intervals are added. Assume the contrary: there exists a job j that is scheduled later in S' than in S. Let j be the first such job in EDF order and let t be the time when j is scheduled in S. When EDF reaches t in S', there must be some job j' before j in EDF order that has not been scheduled (otherwise EDF would schedule j). But j' must have already been scheduled when EDF reached t in S (otherwise EDF would schedule j' at t in S). Then j' is scheduled later in S' than in S, which contradicts our definition of j. □

Lemma 2.13. *Consider a Case-1 or Case-3 push of an interval I. If we remove any set of jobs J' from J, this will still be a Case-1 or Case-3 push.*

Proof. It is immediate that removing jobs can never result in infeasibility.

We now show that removing jobs cannot cause a push to become Case 2 or Case 4. Assume the contrary, that the push of interval I at time t becomes Case 2 or Case 4 after removing J'. By Lemma 2.10, there must be some time step t_f where each calibrated time slot between t and t_f is running a job that must be scheduled before t_f. In each time slot, removing J' can only cause a job with a later deadline to be scheduled by EDF. Then before J' is removed, all calibrated time slots between t and t_f were already active with jobs with potentially earlier deadlines. Then this was a Case-2 or Case-4 push initially, leading to a contradiction. □

Lemma 2.14. *Performing Case-1 and Case-3 pushes can never increase the number of calibrations. That is, there is always an optimal schedule that performs a Case-1 or Case-3 push whenever one is available during the incremental schedule construction.*

Proof. To obtain a contradiction, assume the contrary: no optimal schedule performs all available Case-1 and Case-3 pushes. Let OPT be the schedule that performs the most Case-1 and Case-3 pushes. We show that we can obtain a schedule that performs one more Case-1 or Case-3 push without increasing the cost, reaching a contradiction.

A Case-1 or Case-3 push is only defined on the last interval placed so far while we are making the schedule, so we refer to the push as Case 1 or Case 3 if the push is Case 1 or Case 3 with subsequent intervals removed. That is, EDF schedules jobs from the beginning after this removal to determine the case of the push.

We show that we can feasibly schedule all jobs in OPT after another Case-1 or Case-3 push on some interval I without increasing the number of intervals. Let t be the start time of the first interval I_1 in OPT that did not have all Case-1 or Case-3 pushes applied. So a push on I_1 is Case 1 or 3. Let I be the last interval started at time t. A push on I must also be Case 1 or 3. If it were Case 2 or 4, then by Lemma 2.10 there must be a time t_f before which all calibrated time slots must contain jobs with deadlines before t_f. But since EDF schedules in time slots in order by interval, all jobs in time slots before t_f in I_1 must also have deadlines before t_f. This means I_1 is Case 2 or 4, which contradicts our definition. Thus pushing I is Case 1 or 3. Furthermore, all subsequent intervals in OPT start at $t+1$ or later, so pushing interval I preserves the ordering of the starting times of the intervals.

Let t be the time when I starts in OPT. For now, assume that all later intervals in OPT start strictly later than t. We consider the case where one starts at t later in this proof.

We set the stage for an exchange argument. Schedule all jobs in OPT using EDF. There is a (possibly empty) set of jobs J' that are (1) scheduled between times t and $t+T-1$ inclusive, and (2) scheduled in an interval after I in OPT. Figuratively, we remove these jobs and set their time slots as inactive. What we really do is peg these jobs to these time slots for our exchange, so that their positions in the schedule are not determined by rerunning EDF.

Perform the push on I in OPT with J' removed; now we have a sequence of intervals OPT$'$. When I was the last interval placed so far, the push was remains a Case-1 or Case-3 push. Now define j_{out} and j_{in} as in Definition 2.8. By Lemma 2.13, job j_{out}, if it exists, is released before j_{in} and has a deadline after j_{in} in EDF order. To summarize, j_{out} and j_{in} are defined, assuming that we run the schedule only allocating intervals up to I.

When we transform OPT into OPT$'$, we push I, and run EDF on OPT$'$ ignoring the jobs in J' and their time slots. Interval I ends at $t+T$. Let j'_{out} be a job scheduled before $t+T$ (in any interval) in OPT but after $t+T$ and not in I in OPT$'$; let j'_{in} be

any job scheduled in an interval after $t+T$ in OPT but before $t+T$ or in I in OPT$'$.

Then j'_{in} is j_{in} and j'_{out} is j_{out} if it exists. This is because when I was the last interval, the push was Case 1 or Case 3. After adding subsequent intervals, none of the jobs scheduled in I are scheduled later than $t+T$, nor will a later job be scheduled in I or earlier than $t+T$.

Now we exchange jobs to show that OPT$'$ is feasible. We schedule the intervals up to the pushed I according to (the new) EDF. We schedule the jobs strictly after the interval I ends according the old EDF, except that in the slot where j'_{in} used to be, we replace that with j'_{out}. We keep the jobs J' exactly where they were before.

Then we have performed an extra Case-1 or Case-3 push of I without increasing the cost of the schedule, contradicting our assumption that we have the OPT with the most Case-3 pushes.

As described, this exchange argument leaves out an important detail. This argument applies when interval I has a gap before the next interval on the same processor. If there is no gap, that is, the two intervals are adjacent, then we need to adjust our argument slightly. A simple adjustment is just to view the adjacent intervals as one bigger interval and then to define the j'_{out} where j'_{in} according to this superinterval. □

Lemma 2.15. *A set of k Case-2 pushes on an interval I can only be a part of an optimal push algorithm if, after the Case-2 pushes are performed, the next push would be a Case-4 push.*

Proof. We show that no job in J can be scheduled in the newly-calibrated time slots after the Case-2 pushes. Let J_I be the set of jobs feasibly scheduled by EDF in the first I intervals, before I is pushed. Let J'_I be the jobs that cannot be feasibly scheduled, so $J'_I = J \setminus J_I$. Since the pushes are Case 2, none of the jobs in J'_I can be scheduled in I after the Case-2 pushes on I. Adding subsequent intervals can only cause jobs to be scheduled earlier by Lemma 2.12, so no job in J_I is scheduled at a later point after we add intervals later than I in $\mathcal{I}$. Thus, no job in either J'_I or J_I can be scheduled in the newly-calibrated time slots.

By Lemma 2.10, after a Case-2 push, any further feasible pushes on the interval are Case 2 or Case 4.

Thus, any newly calibrated time slots due to the push remain inactive, and the pushes cannot decrease the cost of the remaining schedule unless there is a later Case-4 push. □

Theorem 2.16. *There exists a sequence of Case-2 and Case-4 pushes such that the corresponding push algorithm is optimal.*

Proof. This proof generalizes Theorem 2.4. All Case-1 pushes are identical to the single-machine case, and can still always occur without added cost. We show that Case-3 pushes never add to cost in Lemma 2.14. The remainder of the argument proceeds as in Theorem 2.4. □

2-Approximation Algorithm

We now give a 2-approximation algorithm (Algorithm 3). Algorithm 3 is not a push algorithm, though it has structural similarities. Observe that the algorithm also becomes Lazy-Binning if $P = 1$.

Algorithm 3 pushes intervals similarly to a push-algorithm. It never performs an infeasible push and performs all Case-1 and Case-3 pushes of each interval I. Whenever it sees an opportunity for an interval I to have a Case-4 push or a series of Case-2 pushes followed by a Case-4 push, Algorithm 3, instead, creates another interval I' right after I. These extra intervals are ignored for the purpose of determining the case of a push; otherwise the pushed interval may not be the last one scheduled. Like Lazy-Binning, this algorithm is pseudopolynomial as written. If release times and deadlines have exponential values, Algorithm 3 may iterate through an exponential number of time steps. However, an interval need never start more than n time steps before the deadline of any job. Incorporating this restriction makes Algorithm 3 run in polynomial time.

Lemma 2.17. *The set of jobs scheduled in the first i intervals of any push algorithm A can be feasibly scheduled after i rounds of Algorithm 3 (a round refers to an iteration of the while loop in Step 2).*

Proof. By Corollary 2.11, any time when a time slot is calibrated in A's schedule, the corresponding time slot is also calibrated in Algorithm 3's schedule. Thus, for the first i intervals, we can copy the assignments of jobs to machines and times from A directly to time slots calibrated by Algorithm 3, giving a feasible schedule. □

Corollary 2.18. *Algorithm 3 gives a solution with cost no more than twice optimal, i.e. it is a 2-approximation.*

Algorithm 3 Lazy-Binning on Multiple Machines

```
1: t ← 0; I ← ∅
2: while not all jobs are scheduled do
3:     Remove all assignments of jobs to intervals
4:     if P or more intervals overlap with I then
5:         t ← next time with ≤ P − 1 intervals
6:     end if
7:     Create a new interval I at t
8:     while pushing I is Case 1 or Case 3 do
9:         Push I
10:        t ← t + 1
11:    end while
12:    t′ ← t
13:    while pushing I from t′ is Case 2 do
14:        t′ ← t′ + 1
15:    end while
16:    if pushing I is Case 4 then
17:        Create another interval at t + T
18:    end if
19: end while
```

2.6 Online Scheduling with Calibrations

In this section we discuss the online variant of the problem, where jobs are not known ahead of time. This is particularly well-motivated in an industrial setting, where factory owners may want to modify their schedules as new orders come in, or adapt to changes in orders.

Even the ISE project, the original motivation for the problem, may benefit from this generalization. The original calibrations paper states "this testing schedule is determined in advance" [19], but even if some ISE testing requirements are known ahead of time, it seems advantageous to have an algorithm that can adapt to changes.

Online Model

In the online variant of the problem, jobs are not known ahead of time. The algorithm only knows about job j when it has reached time r_j (and has completed the schedule up to time r_j). It cannot go back and change its old decisions. See [105] for further discussion of the online model and a survey of past results. This is one of the simplest and best-motivated ways of examining the problem—oftentimes, any production or testing system using machines that require calibrations are unaware of jobs until they are ready to be scheduled.

Crucially, we also slightly modify the cost function and throughput requirements in the online variant. We assume that each calibration incurs a large cost G. In other words, in this variant we make the large calibration cost explicit. Then our total cost is the total flow time of the jobs, plus the total calibration cost. In other words, if each job j is scheduled at time s_j, then the total cost is

$$G(\#\text{ Calibrations}) + \sum_{j \in J} (s_j - r_j) w_j.$$

In the offline setting, we currently do not allow jobs to have deadlines. Timely processing of jobs is guaranteed by minimizing the total flow time. The parameter G gives a tradeoff between the flow guarantee and the number of required calibrations.

We analyze our algorithms using competitive analysis. An algorithm is said to be *α-competitive* if, for every input I, the algorithm performs at most an α factor worse than the optimal offline algorithm on I. Notably, this kind of analysis differs from worst-case analysis in that it must do well on both hard and easy inputs—if the optimal offline algorithm performs unusually well on an instance, our algorithm must as well. In our analysis we generally fix an input $\mathcal{J}$, and let OPT be the cost of the optimal offline algorithm on $\mathcal{J}$. On occasion, we sometimes also refer to the optimal algorithm itself as OPT for brevity.

2.6.1 Unweighted Single-Machine Algorithm

We give a 3-competitive algorithm. Currently, this algorithm only works for a single machine with unweighted jobs (delaying any job incurs the same cost).

The idea of the algorithm is to delay arriving jobs until their flow is equal to the calibration cost G. However, the algorithm has a maximum number of jobs it can delay, after which it must schedule.

Algorithm 4 Online Unweighted Calibration on One Machine

```
1:  Q ← empty queue of jobs
2:  for each time step t do
3:      if a job j arrives at time t then
4:          Push j onto Q
5:      end if
6:      if t is not calibrated then
7:          f ← total flow of all jobs in Q if scheduled beginning at t + 1
8:          if Q contains ≥ G/T jobs or f ≥ G then
9:              Begin a calibration at t
10:         else
11:             p ← total flow of all jobs scheduled in last calibration
12:             if p < G/2 and a job is released at t then
13:                 Begin a calibration at t
14:             end if
15:         end if
16:     end if
17:     if Q is not empty and t is calibrated then
18:         Pop the job ′ with smallest release time off Q
19:         Schedule j′ at t
20:     end if
21: end for
```

Algorithm 4 is similar to known algorithms, like the optimal solution to the ski rental problem. Both delay until a large penalty cost is reached (G in this case). However, the maximum job requirement appears to differentiate this algorithm from previous results. Without this requirement, the algorithm can perform arbitrarily poorly. Interestingly, although the algorithm appears to schedule somewhat early

in this case (especially if G/T is small), it still performs no worse than 3 times the optimal solution.

We bound the algorithm's performance using a charging argument. Before proving the competitive ratio, we need a structural lemma to show that we never double-charge to an interval. Let J_i be the set of jobs scheduled in interval i by Algorithm 4. We partition J_i into two subsets: J_i^E is the set of jobs scheduled earlier in OPT than in Algorithm 4, and J_i^L is the set of jobs scheduled later.

Lemma 2.19. *Consider an interval i scheduled by Algorithm 4. Let i^{OPT} be the earliest interval in OPT containing a job in J_i. Then if $i' > i$, i^{OPT} contains no jobs in $J_{i'}$.*

Proof. Let j' be the first job in J_{i+1}. If j' is released after $t_i + T$, it (and thus all jobs in i') must be scheduled after $t_i + T$ as well, so can't be scheduled in i^{OPT}. On the other hand, if j' is released before $t_i + T$, since Algorithm 4 schedules in EDF order, there must be T jobs in J_i. Then there are $T + 1$ jobs in $J_i \cup \{j'\}$; thus j' cannot be scheduled in i^{OPT}, and neither can any job in $J_{i'}$. □

We are now ready to prove the competitive ratio.

Theorem 2.20. *Algorithm 4 is 3-competitive.*

Proof. We use a charging argument for each calibration i. We break into two cases: in the first, the calibration was due to total flow of G. In the second, the calibration was due to G/T waiting jobs. Let f_i be the total flow of all jobs in i released before t_i, and let e_i be the total flow of all jobs in i released on or after t_i. If Algorithm 4 calibrated after interval i because i had flow at most $G/2$ (but the normal conditions for calibrating are not satisfied), we call this an *immediate calibration*. If there is an immediate calibration following i, we consider the cost of i and $i + 1$ simultaneously.

Case 1 (Calibrated due to flow G): We split into two subcases. First, we assume that all jobs in i are scheduled later in OPT than in Algorithm 4 (J_i^E is empty). Then we charge to the flow of the jobs in OPT. Algorithm 4 incurs cost $G + f_i + e_i < 2G + e_i$, while OPT incurs cost at least $f_i + e_i \geq G + e_i$, leading to a competitive ratio of 2.

Otherwise, (in the second subcase) there is a job j in J_i scheduled earlier in OPT than in Algorithm 4. We charge to the calibration containing the first job in J_i; this charging is unique by Lemma 2.19. Algorithm 4 incurs a cost of $G + f_i + e_i < 3G$; we charge to a cost of G for a competitive ratio of 3.

Case 2 (Calibrated due to G/T waiting jobs): We split into three subcases.

First, we assume that J_i^E is nonempty. If there is no immediate calibration following i, we can charge to OPT as in Case 1. Specifically, we charge to the first interval of OPT containing a job from J_i; this is only charged to once by Lemma 2.19. Algorithm 4 has cost $G + f_i + e_i < 3G$; we charge to an interval with calibration cost G.

If $f_i + e_i \leq G/2$, If $i+1$ is an immediate calibration, consider whether interval $i+1$ has a job scheduled earlier in OPT (i.e. whether J_{i+1}^E is empty). If interval $i+1$ has a job scheduled earlier in OPT, charge to the interval containing it by Lemma 2.19. Intervals i and $i+1$ have total cost $2G + e_i + f_i + e_{i+1} + f_{i+1} < 6G$; we charge to a calibration cost of $2G$ in OPT. Otherwise, if all jobs in J_{i+1} are scheduled later in OPT, Algorithm 4 has cost $5G/2 + e_{i+1} + f_{i+1}$, and OPT has cost at least $G + e_{i+1} + f_{i+1}$. In all cases, we achieve a competitive ratio of 3.

In the second subcase, we assume that J_i^E is empty (all jobs in J_i are scheduled later in OPT), and all jobs in J_i are delayed to at least $t_i + T$ in OPT. If there is no immediate calibration following i, Algorithm 4 incurs cost at most $2G + f_i + e_i$, while OPT incurs cost at least $G + f_i + e_i$. leading to a competitive ratio of 2. If there is an immediate calibration, since $i+1$ begins before $t_i + 2T$, all jobs in J_{i+1} are scheduled later in OPT (since OPT schedules the jobs in J_{i+1} beginning at $t_i + T$). Thus, the cost of OPT is at least $G + f_i + e_i + f_{i+1} + e_{i+1}$, leading to a competitive ratio of at most 2.

Finally, we assume that all jobs in J_i are scheduled later in OPT, but at least one is scheduled before $t_i + T$. Let i^{OPT} be the interval in OPT containing the first such job. Note that we can charge to i^{OPT}: this interval starts before J_{i+1} starts, so if it contains a job from J_{i+1} we can charge to it by Lemma 2.19. Otherwise, i^{OPT} contains no job from J_{i+1}, and therefore no job from $J_{i'}$ for $i' > i+1$. Since i^{OPT} starts later than any interval i' for $i' < i$ it can only be charged to by i.

First, assume that $i+1$ is an immediate calibration. Then there can be no waiting jobs before $i+1$ begins; otherwise, i^{OPT} contains exactly the T jobs in J_i, and we can improve OPT by scheduling i^{OPT} earlier. Thus, i and $i+1$ have total cost at most $5G/2 + |J_{i+1}|$, as each job in J_{i+1} is scheduled immediately. We charge to cost $G + |J_{i+1}|$ in OPT: G from the calibration cost of i^{OPT}, and $|J_{i+1}|$ since each job in J_{i+1} must incur flow cost at least one in OPT. This gives a competitive ratio of $5/2$.

If $i+1$ is not an immediate calibration and i has total flow at least $G/2$, we still examine calibrations i and $i+1$ simultaneously . These intervals have total cost at most $2G + f_i + e_i + f_{i+1} + e_{i+1}$. Since all jobs in J_i are scheduled later in OPT, we charge to their flow as well as the calibration i^{OPT}, so OPT has cost at least $G+f_i+e_i$. The ratio between these is minimized at when $f_i + e_i$ is minimized, at $f_i + e_i = G/2$. Then Algorithm 4 has cost $5G/2 + f_{i+1} + e_{i+1} < 9G/2$, and OPT has cost at least $3G/2$, giving a competitive ratio of 3.

If $i+1$ is not an immediate calibration and i has flow less than $G/2$, we do consider $i+1$ separately. The first job in $i+1$ must be released at least T time steps after i ends. Thus, we charge i to i^{OPT} as above; i^{OPT} cannot be charged to by $i+1$ because it ends before any job in J_{i+1} is released. □

2.6.2 Weighted Single-Machine Algorithm

When jobs have weights (but must be scheduled on a single machine), we follow a similar algorithm to the unweighted case. The main differences are that the algorithm now calibrates if the set of waiting jobs has total weight G/T, and the algorithm no longer performs the immediate calibrations (when the calibrated interval had flow at most $G/2$).

We assume that all algorithms schedule the heaviest possible job first, breaking ties by release time. Unlike the unweighted case, there is no ordering on the jobs; which job is scheduled next depends on which time slots are calibrated.

This lack of an ordering makes charging much more difficult. Our solution to this is twofold. First, the bounds we achieve have larger constants, due in part to the more difficult charging. Secondly, and more to the point, we show that we can restrict ourselves to algorithms that do schedule in order of release time, losing only a factor of 2 in cost.

Lemma 2.21. *If the optimal solution of a given instance has cost C_{OPT}, there exists a solution with all jobs scheduled in order of release time with cost at most $2C_{OPT}$.*

Proof. We begin with the optimal solution (with C calibrations), and create a new schedule with all jobs in order of release time. This new schedule will schedule all jobs no later than they were in the original instance, so the flow cost will be smaller. Furthermore, the new schedule will have no more than $2C$ calibrations. Thus the total cost of the new schedule is at most $2C_{\text{OPT}}$.

Consider each job in the schedule, in order from latest to earliest release time. We maintain the invariant that after considering a job j, it is scheduled at a time t_j such

Algorithm 5 Online Weighted Calibration on One Machine

```
1: Q ← empty priority queue of jobs
2: for each time step t do
3:     if a job j arrives at time t then
4:         Push j onto Q
5:     end if
6:     if t is not calibrated then
7:         f ← cost of scheduling all jobs in Q at t + 1
8:         if ∑_{j∈Q} w_j ≥ G/T or f ≥ G then
9:             Begin a calibration at t
10:        end if
11:    end if
12:    if Q is not empty and t is calibrated then
13:        Pop the job ′ with smallest weight off Q
14:        Schedule j′ at t
15:    end if
16: end for
```

that $t_j \geq r_j$, and all jobs with $r_{j'} > r_j$ have $t_{j'} > t_j$. At intermediate points while constructing this schedule, several jobs may be scheduled at the same time. However, we also maintain that after considering job j, no two jobs with release times at least r_j is scheduled at the same time.

For each such job j, move j before all jobs j' with $r'_j > r_j$. In other words, move j to the time slot immediately before the job with the next-largest release time is scheduled; if there is a job already scheduled at this time slot, these jobs are scheduled at the same time.

We now show that these invariants are maintained. The job is scheduled before all jobs with later release times by definition. To show $t_j \geq r_j$, we must have all all jobs with $r_{j'} > r_j$ be scheduled at $t_{j'} \geq r_{j'} \geq r_j + 1$; thus, r_j is a time slot before all jobs with later release times are scheduled, so it satisfies the requirement for t_j. Since t_j is the maximum time slot satisfying the requirement, $t_j \geq r_j$. Finally, no job with release time larger than r_j can be scheduled at t_j; thus j is not scheduled at the same time as a job with larger release time. Since the invariant was maintained for

all jobs with larger release time, the invariant is maintained.

Now we show that there are no more than $2C$ calibrations. Each time we add a job, we add one new time slot—this time slot is already calibrated, or is joined to a sequence of newly-filled time slots which touch a calibrated time slot.

Consider a sequence of calibrated time slots in the new schedule. All jobs from this sequence must have come from intervals contained in this sequence, as a job j being pushed back never "skips over" an uncalibrated time slot. As j is pushed back, it may empty some slots, but we assume that they remain calibrated in the new schedule. Thus, all jobs from this calibrated sequence must come from an interval in this sequence. Let p be the number of previously-uncalibrated slots in this sequence. Then the number of extra calibrations necessary is $\lceil p/T \rceil$. Furthermore, the number of calibrations previously in this interval was at least $\lceil p/T \rceil$. Thus the number of calibrations for each sequence is at most doubled. Thus the total number of calibrations is no more than $2C$. □

Now we are ready to prove the performance of our algorithm. We show that it is 6-competitive with the best algorithm that schedules all jobs in order of release time, which implies that it is 12-competitive with the optimal algorithm.

Lemma 2.22. *Algorithm 5 is 12-competitive.*

Proof. Consider an interval i which starts at t_i and contains jobs J_i (note that in this case, J_i is a subsequence of jobs in terms of release time). Let I be a sequence of full intervals beginning with i; all but the last are full. Note that all intervals after I consist of jobs that are released after I ends. Thus, each such sequence consists of a contiguous set of jobs in order of release time. Thus, these jobs must be scheduled by OPT in consecutive intervals. Thus, Algorithm 5 uses $\lceil |I|/T \rceil$ intervals to schedule these jobs, and OPT also must use $\lceil |I|/T \rceil$ to schedule these jobs. For all but the first interval of I, we charge to its pair. If i is the first interval, and i' is its corresponding interval in OPT, then:

1. If i' starts before i, charge to i.

2. If i' starts after i and i was started because the flow time of its jobs is at least G, charge to the flow time of the jobs.

3. If i' starts after i, but before $t_i + T$, and i was started because the sum of the weights was at least G/T, charge to i'.

4. If i' starts after $t_i + T$, and i was started because the sum of the weights was at least G/T, charge to the flow time of the jobs.

First, we show that the flow time of OPT is at least G when we charge to it. Note that all jobs in I must be scheduled after i' begins; in particular, all jobs in i must be scheduled in OPT after i' begins. Thus, if scheduling the jobs in i has flow time G, scheduling them in i' which starts after i incurs flow at least G. Similarly, delaying jobs with weight G/T past $t_i + T$ (when all were released before t_i) incurs flow at least G.

We now bound how many times each interval can be charged to in the above. First, note that no two intervals in the same sequence charge to the same interval (by definition). Let i be the first interval of I, and j be the job with earliest release time in I.

1. Assume that j is scheduled in an interval that starts before i. All jobs in later sequences are released after I ends, so they cannot charge to the same interval. Thus no two sequences can charge their first interval to the same interval in OPT. Since sequences charge to consecutive intervals, this means that only two sequences can charge to the same interval. Thus, any such interval can only be charged to twice. If Algorithm 5 schedules i because there were T waiting jobs, this case must apply.

2. In the second case, i had G/T waiting jobs, and charges to an interval that begins between t_i and $t_i + T$. Any subsequent sequence that charges to this interval starts after this interval begins; thus it can only be charged to once by a later sequence. For any previous sequence, this interval starts after its first interval ends; thus it cannot be charged to by the first interval by a previous sequence. Thus it can only be charged to by the next and previous sequence, or three times in total.

For every interval in the algorithm, we charge to an interval or to a flow cost of G. We can charge to each interval three times. Thus the total competitive ratio is at most 6. Since the optimal algorithm with jobs in order of release time has cost at most 2OPT, Algorithm 5 has cost at most 12OPT. □

2.6.3 Multiple Machines

We give a competitive online algorithm when jobs can be assigned to multiple machines, and jobs do not have weights.

When there are multiple machines, charging becomes almost completely impossible, as small perturbations in the intervals can lead to significant changes in which jobs belong to which interval (if they are scheduled in order of release time).

Instead, we use another method: the primal-dual approach. This allows us to use linear-programming-based techniques to derive a lower bound on the cost of OPT directly, without charging to intervals containing specific jobs. On the other hand, this seems to come at some loss of efficiency: we only show that our algorithm for multiple machines is only 12-competitive.

We briefly review linear programming and how it is used before describing our algorithm and proving the competitive ratio.

Linear Programming: A Brief Overview

Linear programming is an algorithmic paradigm in which a set of constraints must be met while minimizing a cost function. In linear programming, all constraints (and the cost function) must be some linear combination of the variables. Linear programming is one of the most wisely used ideas in all of algorithms, and has further practical applications in operations research. We use LP to refer to both Linear Programming as a concept, as well as a *linear program*, a specific goal and set of linear constraints.

Here, we focus on the *primal-dual method* for analysis. This method itself is far too rich (both in terms of techniques and past work) to fully describe here. See [67] for a full tutorial and a survey of previous results.

We give a short summary of how we use the primal-dual technique which we use to develop the competitive ratio for the multiple-machine algorithm. We begin with an algorithmic problem A (in this case, scheduling with calibrations). The first step is to develop an LP such that any solution to A is also a solution to the LP, with the same cost. Thus, the cost of the optimal solution of A is bounded below by the optimal solution to the LP.

Linear programs have a dual which can be obtained by, essentially, transposing the constraint matrix (see, for example, [45] for more details). The dual LP has a cost function which we want to maximize. Crucially, any solution to this dual LP has a smaller cost than any solution to the primal LP; this is the *weak duality theorem*. Thus, any solution to the dual LP lower bounds the cost of the optimal solution to A.

This leads us to a method for proving the competitive ratio for our algorithm. While our algorithm progresses, we build a solution to the dual LP, trying to maximize its objective function.

In the end, we want that the cost of our algorithm is at most c times the objective function of the dual LP. Since the dual LP lower bounds the cost of an optimal solution for A, the algorithm has cost at most c times OPT.

Algorithm and Analysis

Our algorithm is an extension of Algorithm 5. We wait until there are G/T waiting jobs or the jobs have total flow G to calibrate. Then, we schedule jobs in order of release time, starting with the machine with smallest index first.

With multiple machines, we need to be careful how we define waiting jobs. In particular, once we calibrate, we want the jobs we will schedule in that calibration to no longer count as waiting jobs when deciding if we should calibrate. Thus, when we calibrate, we assign jobs to the intervals immediately. This means that the guarantee that jobs are scheduled in order of release time does not hold across machines for this algorithm.

Algorithm 6 Online Unweighted Calibration Algorithm on Multiple Machines

```
1: Q ← empty priority queue of jobs
2: for each time step t do
3:     for all jobs j arriving at time t do
4:         if there exists a machine m with an uncalibrated time t' ≥ t then
5:             Schedule j on m at t'
6:         else
7:             Push j onto Q
8:         end if
9:     end for
10:    if less than p machines are calibrated at time t then
11:        f ← cost of scheduling all jobs in Q at t + 1
12:        if Q contains at least G/T jobs or f ≥ G then
13:            Begin a calibration at t on the next machine in round robin order
14:            Schedule jobs from Q in this interval in release time order.
15:        end if
16:    end if
17: end for
```

We have four constraints in the linear program given in Figure 1. The first ensures that flow is high until we fully calibrate. The second ensures that the flow can only

Primal (covering):

$$
\begin{array}{lllll}
\text{minimize} & \sum\sum f_{t,j} & +G\sum_t c_t & & \\
& f_{t,j} & +\sum_{t'=r_j-T} c_{t',p} & -a_{j,p} \quad \geq 0 & \forall j, t \geq r_j, p \\
& \sum_{r_j<t}(f_{t,j}-f_{t-1,j}) & +\sum_p \sum_{t'=t-T}^{t} c_{t',p} & \geq 0 & \forall t \\
& & & \sum_p a_{j,p} \quad \geq 1 & \forall j \\
& f_{r_j,j} & & = 1 & \forall j
\end{array}
$$

Figure 1: A linear program for scheduling with calibrations. Every legal schedule is a feasible solution to this linear program.

decrease by 1 (per machine), and only during calibrated time slots. The fourth ensures that each job j is assigned to at least one processor p. The fourth makes sure that every job incurs flow cost of at least 1.

Taking the dual, we obtain the LP given in Figure 2. All variables are required to be nonnegative in the primal. In the dual, we have $x_{t,j} \geq 0$, $y_t \geq 0$, $w_j \geq 0$, and z_j unbounded.

Theorem 2.23. *Algorithm 6 is* 12*-competitive.*

Proof. As mentioned earlier, we use the primal dual technique. We set the variables every time Algorithm 6 calibrates (or, in some cases, only after Algorithm 6 calibrates twice. This is similar to when two intervals are charged simultaneously in the proof of Theorem 2.20). In each case, we show that the increase in the dual objective function (and thus the increase in OPT; see the discussion of Linear Programming above) is at least 1/12 the cost incurred by Algorithm 6. This shows that Algorithm 6 is 12-competitive.

We let $w_j = \max_p \sum_t x_{t,j,p}$. Intuitively, we use $x_{t,j,p}$ to keep track of the flow of job j at time t on processor p. Since Algorithm 6 only assigns flow to one processor, w_j can store the total flow of job j. We set $y_t = G/2T$ for all t, and $z_j = G/2T$ for all j. These variables help contribute to cost when there are a large number of jobs that may not have much flow (i.e. Algorithm 6 calibrates due to T or G/T waiting jobs).

Dual (packing):

$$\begin{array}{lllll}
\text{maximize} & \sum z_j + \sum w_j & & & \\
\sum_p x_{t,j,p} & -y_{t+1} + z_j & & \leq 1 & \forall t, j \text{ with } t = r_j \\
\sum_p x_{t,j,p} & +y_t - y_{t+1} & & \leq 1 & \forall t, j \text{ with } t \neq r_j \\
\sum_{j|r_j \leq t+T} \sum_{t'>t} x_{t',j,p} & +\sum_{t'=t}^{t+T} y_t & & \leq G & \forall t, p \\
\sum_{t>r_j} -x_{t,j,p} & & +w_j & \leq 0 & \forall j, p
\end{array}$$

Figure 2: The dual of the linear program given in Figure 1

We examine each interval i one by one and find the increase in the dual LP objective after the interval is scheduled. Let t_i be the time when interval i is scheduled. Our argument is split into cases based on why the algorithm decided to schedule the jobs.

Case 1: T waiting jobs If we calibrate because we have T jobs (with total flow less than G), then we set $z_j = G/T + 1$ for each of these T jobs. This adds a cost of $G + T$ to the dual, whereas Algorithm 6 has cost at most $2G$.

Case 2: Total flow G In this case, we calibrate because the waiting jobs have total flow at least G. Assume calibration i starts at time t_i. We split into two subcases. First, assume that there is a job which is released before the interval i ends, which is not immediately scheduled in i. Then there are T total jobs scheduled during the interval. We keep $x_{t,j,p} = 0$ for all t, p for the jobs scheduled in the interval, but set $z_j = G/2T$ for all of them. This gives a total dual cost of $G/2$.

Now we can assume that all jobs with $r_j \leq t_i + T$ are scheduled in i. For all $t \leq t_i$ (for all jobs j scheduled in t_i), let $x_{t,j,p} = 1/2$ if job j is waiting at time t (and will be scheduled on processor p), and $x_{t,j,p} = 0$ otherwise. Since these jobs have total flow G, the sum of these $x_{j,t,p}$ is at most $G/2$. Therefore, the third condition is satisfied

for all times t, p. Consider a job j. Its flow consists of two parts: the flow before time t_i, and the flow incurred after t_i while waiting for j to be scheduled. Since jobs are scheduled in order of release time without loss of generality, each job j can only wait for one time step for each waiting job at t_i. However, by the definition of Algorithm 6, this is at most G/T. Thus, for a given j, $\sum_t x_{t,j,p} + G/T$ is at least half the flow incurred by job j. Since the jobs have total flow cost G, we get an added cost of $G/2$ in the dual LP.

Case 3: G/T **waiting jobs** We consider intervals i and $i+1$. These two intervals have cost at most $6G$. We show that the dual increases in cost by at least $G/2$. We set $x_{j,t,p} = 1/2$ for all waiting jobs j, for all times $t_i \leq t \leq t_i + T$. This gives a dual cost of $G/2$. We charge intervals i and $i+1$ to this cost, giving a total cost of $6G$, while the dual cost increases by $G/2$.

□

Chapter 3

Data Structures in External Memory

This section focuses on the external memory model. This model differs from traditional algorithm analysis in that it measures the cost of hard disk accesses, rather than computation time.

In particular, the external-memory, or Disk Access Machine (DAM) model has an internal memory of size M. The input data begins in external memory (which has unbounded size). For a cost of 1, the algorithm can transfer B consecutive elements between internal and external memory. All computation must be performed on data in internal memory; however, this computation is free (only the disk accesses have cost). For more details, see Aggarwal and Vitter's original exposition [3].

The DAM model, and external memory in general, is an extremely rich area, and has become a particularly important area of focus as computers become larger and spend more time accessing their input. Past work has focused on algorithms like matrix multiplication [64, 70] and sorting [3, 33].

This thesis, however, deals only with a simple (but fundamental) property of the DAM model: linear scans are cheap. Scanning an array of size N in consecutive order requires only $O(N/B)$ I/Os in the DAM model. Moreover, rotating-disk hard drives benefit further from linear scans; in practice, linear scans on a rotating disk can take even less time than the DAM model would indicate.

This section gives a packed-memory array (PMA), which is a data structure that keeps data in a closely-packed array, allowing it to be efficiently accessed in the DAM model. This data structure is updating using linear scans as well, so it is I/O efficient both to query and to update.

Our focus is on a new kind of PMA which has a built-in security guarantee: history independence. History independence guarantees that the way we store data does not give away unnecessary information to an observer. We give a history and formal definition of both PMAs and history independence before describing our results.

3.1 Packed-Memory Arrays

A packed-memory array is a linear-sized array that efficiently maintains data, in order, under adversarial inserts.

Definition 3.1. *A packed memory array maintains N elements in an array of size*

$O(N)$ *under the following operations:*[3]

- `insert`(x, y)*: Insert* x *into the data structure given a pointer to the location of* y*, which is already in the data structure.*
- `delete`(x)*: Given a pointer to* x*, remove it from the data structure.*

PMAs are generally analyzed by their worst-case running time. *Worst-case* means that we assume there is an adversary inserting particularly difficult elements into the data structure to cause us to take extra time. Note that since our PMA is randomized, we loosen this requirement slightly to use with high probability analysis.

Many PMAs (in particular the one presented here) have *amortized* performance guarantees. This means that while individual operations may be expensive, the data structures perform well on average. In particular, an amortized bound of $O(\log^2 N)$ means that any k operations on the PMA (starting from an empty state) require $O(k \log^2 N)$ time.

What we refer to here as PMAs have been studied under many different names, such as *sparse tables* [77, 80], *sequential file maintenance* [117–119], *list labeling* [52–55], and *balanced-trees of small height* [6, 84].

PMAs were first studied by Itai et al., who gave a fairly simple data structure achieving $O(\log^2 N)$ amortized time for inserts and deletes [77]. Their data structure is based on a series of thresholds for the density of each subarray. (This corresponds closely to the requirement in our data structure that the balancing point of each range be selected from the candidate set.) When a subarray is out of threshhold, one of the subrarrays that contained it (specifically, the smallest one that *is* within threshold) is rebuilt, spreading all elements evenly.

Itai et al.'s analysis gave a good amortized bound, but some operations could be very expensive. For example, every $O(N/\log N)$ operations may cause the entire data structure to be rebuilt from scratch. This is, of course, undesirable in practice. It would be disastrous for a single insert to cause the data structure to be completely inoperative while the system slowly rebuilds itself from the ground up.

Later, Willard gave a data structure which achieves $O(\log^2 N)$ worst-case amortized time [117–119]. This idea was later explored with a different, and arguably simpler, alternative achieving the same bound [20].

There exist better bounds for special cases. The original PMA achieves $O(\log N)$ amortized rebalance time under random inserts. Bender and Hu gave a PMA variant that achieves $O(\log N)$ amortized rebalance time for random inserts, hammer inserts (i.e. repeated inserts after the same element), and bursty insertion patterns that hammer on random locations [23]. Their data structure retains the $O(\log^2 N)$ worst case amortized bound, while improving it on these special cases.

[3]While the definition permits any constant in the $O(N)$ size bound, since space is valuable we would really like a very small constant here. Most PMAs work with sizes like $2N$ or $8N$.

On the lower bound side, Dietz and Zhang showed an $\Omega(\log^2 N)$ bound in a restricted model, where the data structure can only be modified between operations by evenly spacing elements in a selected subarray [53]. This covered all PMAs known when their paper was published. Furthermore, subarray rebuilds result in cache-efficient updating; in particular, rebuilding a subarray of size k requires k/B I/Os. This results in the $O((\log^2 N)/B)$ amortized update time in the I/O model. Thus, this bound may be regarded as strong evidence that known PMA bounds were tight. However, there are two issues. The first is that Bender et al.'s result (which beats the $O(\log^2 N)$ bound for special cases) does not rebalance in this manner. The second is that the lower bound instance in [53] was merely a sequence of sequential inserts. This seems intuitively to be an easy case. In fact, this intuition is correct in some sense, as Bender et al.'s data structure can achieve $O(\log N)$ time on sequential inserts.

The best possible complexity of a deterministic PMA was recently put to rest by Bulánek et al. [37]. who gave an $\Omega(\log^2 N)$ lower bound. Their lower bound is constructive: essentially, the adversary they construct repeatedly places elements in very dense parts of the PMA. Their paper is fairly technically involved. Most of the difficulty comes from proving that an appropriately dense area exists. Their lower bound is quite strong: for example, they allow the numbers inserted to be integers that must be kept in order (rather than allowing the adversary to repeatedly insert after an arbitrary element, as our upper bound requires). Thus, if two consecutive integers are already in the data structure, they can be safely placed adjacent to one another as no new integer can be placed between them. Bulánek et al. show that even if the integers are bounded above by $O(m)$ (where $m = O(N)$ is the size of the PMA) the $\Omega(\log^2 N)$ bound remains.

Interestingly, the above lower bounds only apply to deterministic data structures. The best-case performance of randomized PMAs remains an open problem. The data structure we present here is randomized, but only for the purposes of history-independence. In fact, due to what we call *randomized rebalances*, our PMA has $\Theta(\log^2 N)$ amortized update time even with very favorable input distributions, while the classic PMA achieves $O(\log N)$ expected amortized update time in some special cases (like random inserts) [23, 77].

3.2 History Independence

History independence is a security guarantee to ensure that the *memory representation* of a data structure does not leak information to an observer.

Consider, for example, a newspaper reporter who has been given important information in a document. She shares this document with the public to back up the story. However, unbeknownst to her, the way the document is stored contains extra information about its edit history. Perhaps the document is stored in an format

which gives some clues as to what parts of the document were added more recently, giving information about when the source obtained their information. In an extreme example, the file format could append recent edits to the document in a simple log, which perhaps even contains information that was intentionally redacted from the file itself.

This kind of information leakage has been surprisingly common in recent history [74, 96]. Thus, it is desirable for data structures to (provably) avoid this leakage if they are used in sensitive applications.

In some cases, achieving this level of security is simply a matter of careful implementation—don't concatenate recent edits in a simple log, for example. However, in some cases it is much less trivial. For example, a balanced tree data structure such as an AVL tree [2] can leak some information about when its contents were inserted. One simple but easy-to-verify case where history is leaked is an AVL tree on only two inserted elements; the more recent element will be the root element.

History-Independent data structures leak no information about the history of operations on the data structure, even to an observer who can see how the data structure is represented in memory. The remainder of this section is dedicated to formally defining history-independence, and briefly discussing how we use it in this thesis.

A *data structure* is a mapping from a set of inputs (and possibly some randomness) to a memory representation. Let Σ be the set of all possible sequences of operations on the data structure. Then a data structure DS of size N which uses r bits of randomness is a mapping $DS : \Sigma \times \{0,1\}^r \to \{0,1\}^N$. The memory representation contains the entirety of the on-disk representation of the data structure: what data is stored, where it is stored in memory, the pointer structure of the data structure, etc.

A *history-independent* (HI)[4] data structure gives the same mapping for all $\sigma \in \Sigma$ that lead to the same set of stored contents S.[5]

Definition 3.2. *Consider a data structure with stored contents S and memory representation M. Then consider two distinct sequences of operations σ_1, σ_2 which both lead to S being stored in the dictionary. Then the data structure is history independent if*

$$\Pr_r(DS(\sigma_1, r) = M) = \Pr_r(DS(\sigma_2, r) = M).$$

This is essentially the same definition given in [96].

[4] We also use HI to refer to history-independence itself.

[5] In the context of a PMA S is just the in-order set of items in the data structure. Other data structures which store other information have slightly different S; however, the definition of history-independence stays otherwise the same.

This notion of history independence is known as *weak history independence* in the literature—weak because it is secure only if the observer can look at the data structure once. *Strong history independence* hides history against an observer who can look at the data structure multiple times. A classic theorem states that a reversible data structure[6] is strongly history independent if and only if it has a canonical representation, up to initial randomness [74]. In other words, the data structure cannot flip further random coins during execution. This turns out to be fatal to obtaining good bounds for storing data in a dynamic array, and thus to a PMA (since a dynamic array is essentially a PMA without the constraint that data must remain in order).

Observation 3.3. *No strongly-history-independent dynamic array can achieve $o(N)$ amortized resize cost per insert or delete with with probability at least $1 - 1/N^{1+\epsilon}$, where N is the maximum size of the array.*

Proof. Consider a strongly-history-independent dynamic array that needs to be strictly greater than 50% full. (The proof generalizes to arbitrary capacity constraints.) For integer k, the adversary chooses a random $\ell \in \{k, k+1, \ldots, 2k\}$. Then the adversary inserts up to ℓ elements into the array, and then alternates between adding and removing an element from the array, so that the array alternates between having ℓ and $\ell + 1$ elements.

Given the capacity constraints on the array, there must be at least two different canonical representations for the arrays of sizes $k, k+1, \ldots, 2k$. Thus, there is a probability of at least $1/k$ that the adversary forces an array resize (with cost $\Omega(N)$) on every insert and delete in the alternation phase.

Observe that k can be arbitrarily large. No matter how long the data structure runs, it cannot avoid an $\Omega(N)$ resize with probability greater than $1 - 1/k$. □

History independence was introduced by Naor and Teague in [96], though past work had already examined ways to hide history from an observer. First Micciancio [92] and later Snyder [108] studied how to hide information in the pointer structure of a tree. (This is a weaker notion than history independence.) History independence was significantly extended in the work of Hartline et al. [74]. Later research includes results for history-independent hashing [27, 95, 96], dictionaries and order-maintenance [27], and data structures for computational geometry [28, 112].

[6] That is, a data structure with contents S that can perform a non-empty series of operations and return to the same set of contents S (formally, the data structure's state transition graph is strongly connected). A PMA is reversible because it allows both inserts and deletes.

History independence has been previously studied in external memory as well. First, Golovin proposed the B-treap [68], a strongly history-independent external-memory B-tree variant based on treaps [9]. Golovin notes that while the B-treap is a unique-representation data structure supporting B-tree operations with low overhead, from a practical point of view, it is complicated and difficult to implement [69]. Golovin thus proposes a strongly history-independent B-skip list [69] as a simpler alternative to the B-treap. Both of these data structures meet classic B tree bounds for all operations (with some additional, though reasonable, assumptions on B).

Retaining balance in data structures is an obstacle to history-independence. For example, as mentioned earlier, an AVL tree is not history independent. However, there are history-independent binary search trees. The *treap* is a history-independent data structure in which each element, when inserted, is promoted randomly through the tree [9]. This gives a randomized balancing scheme independent of the history of the data strucutre. The *skip list* is a similar data structure that also gives strong guaranteed performance bounds [106]. The skip list also promotes elements randomly, but deviates from the classic binary search tree pointer structure, creating a distinct and arguably simpler data structure as a result.

Our goal in this section is to give a randomized balancing scheme for a PMA–which must be much more tightly balanced due to the $O(N)$ space constraint—which retains history-independence.

3.3 History-Independent Packed Memory Array

This subsection gives a history-independent packed memory array (PMA).

Our PMA is randomized and, therefore, our bounds are randomized as well—it is possible, though extremely unlikely, that our data structure will perform much more poorly than the $O(\log^2 N)$ bound on an unlucky execution. Our performance bounds have two parts. We show that our data structure performs well in expectation, so we expect it to have $O(\log^2 N)$ performance on average. But to follow up, we bound how often they can deviate from the expected bound.

In particular, we show that our data structure requires $O(\log^2 N)$ operations and $O((\log^2 N)/B)$ I/Os with high probability. An event occurs *with high probability* if it occurs with probability at least $1 - O(1/N^c)$ for any $c \geq 0$. The parameter c in the probability generally gives a tradeoff with the severity of the event: for example, our HI PMA has cost $O(c \log^2 N)$ with probability $1 - O(1/N^c)$. Thus, while it is possible that the PMA performs slightly worse than expectation, if N is at all large, it is extremely likely that its performance will be very close to the expected bound.

3.3.1 High-Level Structure of HI PMA

Packed-memory arrays and sparse tables in the literature [21, 23, 77, 80, 119] are not history independent; the size of the array, densities of the subarrays, and rebalances depend strongly on the history.

We guarantee history independence for our PMA as follows. First, we ensure the size of the PMA is history independent. We resize using the HI dynamic array allocation strategy described in [74].

Next, we ensure that the N elements in the array of size $N_S = \Theta(N)$ are spread throughout the array according to a distribution (given below) that is independent of past operations.

We maintain this history-independent layout recursively. At the topmost level of recursion, we (implicitly) maintain a set of size $\Theta(N_S/\log N_S)$, which we call the *candidate set*. The candidate set consists of elements that have rank $N/2 \pm \Theta(N_S/\log N_S)$. We pick a random element from the candidate set, which we call the *balance element*. If the balance element has rank r, then we recursively store the first $r-1$ elements in the first half of the array and the remaining $N-r-1$ elements in the second half of the array.

In general, when we are spreading elements out within a subarray A of the PMA, the candidate set has size $\Theta(|A|/\log N_S)$, and as before, the balance element is randomly chosen from this set. The base case is when $|A| = \Theta(\log N_S)$, at which point the elements are spread evenly throughout A.

Thus, how the elements are spread throughout the PMA depends only on the size N_S (which is randomly chosen), the number of elements in the PMA N, and the random choices of all the balance elements.

We maintain the balance elements in each candidate set using a simple generalization of reservoir sampling [113] in which there are deletes, described below. In this particular instance of reservoir sampling, the size of a candidate set at any given level of recursion stays the same, unless the size of the entire PMA changes.

3.3.2 Reservoir Sampling with Deletes

We first review a small tweak on standard reservoir sampling [113], *reservoir sampling with deletes*, which we use to help build the PMA.

Game: We have a dynamic set of elements. The objective is to maintain a uniformly and randomly chosen *leader* of the set, where each element in the set has equal probability of being selected as leader. In other words, we are interested in reservoir sampling with a reservoir of size 1.

Since the set is dynamic, at each step t, an element may be added to or deleted from the set. The adversary is oblivious, which means that the input sequence cannot depend on the particular element chosen as leader.

We can maintain the leader using the following technique. Let n_t denote the number of elements in the set at time t (including any newly-arrived element). Initially, if the set is nonempty, we choose the leader uniformly at random. When a new element y arrives, y becomes the leader with probability $1/n_t$; otherwise the old leader remains. When an element is deleted, there are two cases. If that element was the leader, then we choose a new leader uniformly at random from the remaining elements in the set. If the deleted element was not the leader, the old leader remains.

Lemma 3.4 ([113])**.** *At any time step t, if there are n_t elements in the pool, then each element has a probability $1/n_t$ to be the leader.*

3.4 Detailed Structure of the HI PMA

The size N_S of our history-independent PMA is a random variable that depends on the number of elements N stored in the PMA (similar to dynamic arrays). We select parameter $\hat{N}$ randomly from $\{N, \ldots, 2N-1\}$, and N_S is a function of $\hat{N}$ (as described below).

We view the PMA as a complete binary tree of ranges, where a *range* is a contiguous sequence of array slots. This tree has height $h = \lceil \log \hat{N} - \log\log \hat{N} \rceil$. The root is the entire PMA and has depth 0. The leaves in the binary tree are ranges comprising $\lceil C_L \log \hat{N} \rceil$ slots, where C_L is a constant to be determined later. Thus, the PMA has a total of $N_S = 2^h \lceil C_L \log \hat{N} \rceil \le (2C_L + 1)\hat{N} = \Theta(N)$ slots.

Consider a range R in the binary tree with left child R_1 and right child R_2. Recall from Chapter 3.3.1 that the *balance element* b_R of R is the first element of R_2; all the elements in R of smaller rank than b_R are stored in R_1. The values of b_R for each range/node are stored in a separate tree.

For each non-leaf range R at depth d, define the *candidate set* M_R to be the $\lceil c_1 \hat{N} 2^{-d} / \log \hat{N} \rceil$ middle elements of R. More precisely, if R holds ℓ elements, then we fix the size of M_R and set the first element of M_R to be the $1 + \lceil \ell/2 \rceil - \lceil |M_R|/2 \rceil$th element of R.

Our PMA is parameterized by a constant $0 < c_1 < 1 - 6/\log \hat{N}$.[7] A larger c_1 reduces the amortized update time and increases the space. We require $C_L \ge 1 + c_1 + 6/\log \hat{N}$. The value of C_L and c_1 need not change as $\hat{N}$ changes—values such as $c_1 = 1/2$ and $C_L = 2$ work for sufficiently large $\hat{N}$ (over 4096 in this case).

[7] When $\hat{N} \le 64$, no such c_1 exists. For such small $\hat{N}$, we use a dynamic array instead.

3.5 Dynamically Maintaining Balance Elements

As elements are inserted into or deleted from the PMA, the candidate set M_R for some range R could change. This change might be caused by a newly inserted or deleted element that belongs to M_R or just because insertions at one end of R cause the median element of R to change.

As the candidate set M_R changes, we maintain the invariant that the balance element b_R is selected uniformly and randomly from M_R. (In particular, this means that b_R is selected history-independently.) We use reservoir sampling with deletes as the basis for maintaining this invariant.

Invariant 3.5. *After each operation, for each range R, balance element b_R is uniformly distributed over the candidate set M_R.*

Whenever one element leaves their candidate set, another one joins, since the candidate set size of each range is fixed between rebuilds of the entire PMA. Thus, we describe how to maintain the candidate set when exactly one element is added, and one leaves.

If the balance element is the element leaving the candidate set, we select the new balance element uniformly at random. Otherwise, when a new element enters the candidate set, it has a $1/|M_R|$ chance of becoming the new balance element (as in reservoir sampling).

When the balance element of a range changes, we rebuild the entire range and all of its subranges. Rebuilding a range of $|R|$ slots can be done in $\Theta(|R|)$ time; see Lemma 3.10.

3.6 Detecting Changes to the Candidate Set

In order to determine how inserts and deletes affect the candidate set of a range R, we need to know the rank of the element being inserted or deleted, the current candidate set of R, and the rank of the current balance element of R.

The rank of the element being inserted and deleted is specified as part of the insert or delete operation.

To compute the other information, our PMA maintains an auxiliary data structure containing the number of elements ℓ_R in each range R. During an insert or delete, as we descend the tree of ranges, we keep track of the ranks of the first and last element in each range as follows. Suppose we are at range R whose first element has rank x. If we descend to R_1, then we know the rank of the first element of R_1 is also x. If we descend to R_2, then the rank of its first element is $x + \ell_{R_1}$. Given the rank of the first element of a range and the number of elements in that range, it is easy to compute

its candidate set. Note also that the rank of the balance element of a range R whose first element has rank x is $x + \ell_{R_1}$.

We need to store ℓ_R for each range so that it can be accessed efficiently, both in terms of operations and I/Os. Since the ranges form a complete binary tree, we store the numbers ℓ_R in a binary tree organized in a Van Emde Boas layout (see [21, 104]). We call this auxiliary data structure the *rank tree*.

The Van Emde Boas layout is a deterministic, static, cache-oblivious—and hence history-independent—layout of a complete binary tree. It supports traversing a root-to-leaf path in $O(\log N)$ operations and $O(\log_B N)$ I/Os. Thus, the rank tree is history independent.

Whenever the size of a range changes due to an insert or delete, we update the corresponding entry in the rank tree. Whenever we rebuild a range R in the PMA, we update all entries of the rank tree corresponding to descendants of R. Whenever we rebuild the entire PMA, we rebuild the entire rank tree.

3.7 Correctness and Performance

The main goal of this section is to prove that the PMA performs well; that is,

Theorem 3.6. *There exists a weakly history-independent packed-memory array on N elements which can perform inserts and deletes in $O(\log^2 N)$ amortized element moves with high probability. This PMA requires $O(N)$ space, can perform inserts and deletes in amortized $O\left(\frac{\log^2 N}{B} + \log_B N\right)$ I/Os with high probability, and can perform a range query for k elements in $O(1 + k/B)$ I/Os.*

3.7.1 Balance-Element Structural Lemmas

First, we show correctness—the data structure always finds a slot for any element it needs to store.

Lemma 3.7. *At all times, the size of a range is larger than the number of elements it contains.*

Proof. Consider a range at depth d. This range has $N_S/2^d = 2^{h-d}\lceil C_L \log \hat{N} \rceil \geq C_L \hat{N}/2^d$ slots. We show that the maximum number of elements it can contain is smaller than this number of slots.

The number of elements in the range is at most half of the elements in its parents range, plus half of the size of the parent's candidate set (rounding up). In other

words, if $S(d)$ is the maximum number of elements in a range at depth d, $S(0) \leq \hat{N}$ and

$$\begin{aligned} S(d) &\leq \lceil S(d-1)/2 \rceil + \lceil M_R/2 \rceil \\ &\leq \lceil S(d-1)/2 \rceil + \frac{1}{2}\left\lceil \frac{c_1 \hat{N}}{2^{d-1} \log \hat{N}} \right\rceil + \frac{1}{2} \\ &\leq \frac{S(d-1)}{2} + \frac{c_1 \hat{N}}{2^d \log \hat{N}} + \frac{3}{2}. \end{aligned}$$

By induction,

$$S(d) \leq \frac{\hat{N}}{2^d} + \frac{c_1 d \hat{N}}{2^d \log \hat{N}} + 3.$$

Since $d \leq \log \hat{N}$, and $\hat{N}/2^d \geq \hat{N}/2^h \geq (\log \hat{N})/2$,

$$S(d) \leq \frac{\hat{N}}{2^d}(1 + c_1) + 3 \leq \frac{\hat{N}}{2^d}\left(1 + c_1 + \frac{6}{\log \hat{N}}\right).$$

Since we choose $C_L \geq 1 + c_1 + 6/\log \hat{N}$, we have $S(d) \leq C_L \hat{N}/2^d$. □

As Lemma 3.7 establishes, each leaf range in the PMA (which has $\Theta(\log N)$ slots) never fills up completely. Using a similar argument, it can be shown that each leaf also contains $\Omega(\log N)$ elements if $c_1 < 1 - 6/\log \hat{N}$. Because the elements are spread out evenly in the leaves, this implies there are $O(1)$ gaps between two consecutive elements.

Lemma 3.8. *If $c_1 < 1 - 6/\log \hat{N}$, the leaves are always constant-factor full. There is $O(1)$ space between two consecutive elements in the array.*

Next, we prove weak history independence. As mentioned in Chapter 3.3.1, when we are spreading elements out within a subarray A of the PMA, the balance element is randomly chosen from the candidate set. The elements of A are recursively split between its children according to this balance element. The base case is when $|A| = \Theta(\log \hat{N})$, at which point the elements are spread evenly throughout A.

Thus, how the elements are spread throughout the PMA depends only on $\hat{N}$ (and the related N_S), the number of elements in the PMA N, and the random choices of all the balance elements. This immediately gives weak history-independence, as formalized in the following lemma.

Lemma 3.9. *This PMA is weakly history independent.*

Proof. We show that the memory representation of the PMA is based only on N and some randomness (in particular, the random choices made during balance element selection and the random choice of $\hat{N}$).

Let the PMA contain a set of elements S of size N, with a set of balance elements P. Then P partitions the elements of S into leaf ranges.

Since the elements are evenly spaced in each leaf range, the position of each element within the leaf is determined by the number of elements in that leaf. Since S is partitioned into leaf ranges by P, P determines the position of each element in the PMA. Thus P, $\hat{N}$, and N determine the memory representation of the data structure. By Invariant 3.5, P is selected from a distribution based only on N and $\hat{N}$.

Thus, any two sequences of operations X and Y that insert S into the PMA result in the same distribution on P, and the same distribution on memory representations. □

3.7.2 Proving the Performance Bounds

We begin by bounding the cost of a rebalance. Then we bound the total number of rebalances.

Lemma 3.10. *Rebuilding a range R containing $|R|$ slots takes $O(|R|)$ RAM operations and $O(|R|/B + 1)$ I/Os.*

Proof. The algorithm first recursively chooses the balance elements for all ranges contained in this range R and updates them in the rank tree; this takes $O(|R|)$ time and $O(|R|/B + \log_B |R| + 1) = O(|R|/B + 1)$ I/Os. Then, all elements in R are gathered, and inserted into the appropriate leaf range using a sequence of linear scans. □

Our goal is to bound the cost of our PMA operations. Specifically, we want to show Theorem 3.11.

Theorem 3.11. *Consider k (not necessarily consecutive) operations on a PMA during which its maximum size is N_M, its minimum size is $\Omega(N_M)$, and $k = \Omega(N_M)$. The amortized cost of these operations is $O(\log^2 N_M)$ with high probability with respect to k: in other words, these k operations require $O(k \log^2 N_M)$ total RAM operations with high probability.*

Before proving this theorem, we need some supporting lemmas and definitions.

Definition 3.12. *A rebuild that is not charged to a range R is called a* free rebuild. *Free rebuilds come from two sources: (1) they are rebuilds of an ancestor range (whose cost is charged to the ancestor), or (2) they are triggered by the interstitial operations between the nonconsecutive operations of Theorem 3.11.*

We further categorize non-free rebuilds by their causes. An out-of-bounds rebuild *is a rebuild caused by the pivot leaving the candidate set. A* lottery rebuild *is a rebuild caused by deleting the pivot or by inserting into the candidate set an element that becomes a new pivot.*

Gearing up to Lemma 3.13, we concentrate on the cost of all rebalances at a single depth d. Let M_d be the size of the candidate set at depth d.

We give a lower bound on the probability that two out-of-bounds rebuilds happen in quick succession. This lemma holds regardless of the number of free and lottery rebuilds that happen in between.

Lemma 3.13. *After any rebuild of a range R at depth d, consider a sequence of t operations on R, for any $t \in \{1, \ldots, \lfloor M_d/2 \rfloor\}$, with arbitrary free and lottery rebuilds occurring during these operations. The probability $p(t)$ that no out-of-bounds rebuild happens during these t operations is at least $1 - 2t/M_d$.*

Proof. Let $p_i(t)$ be the probability that no out-of-bounds rebuild happens in the first t time steps, given that exactly i free and lottery rebuilds happen during the first t time steps. We prove the lemma by induction on i.

Define the *guard number* as the number of elements between the pivot and the closest endpoint of the candidate set.

First, the base case: for any t, $p_0(t)$ is at least the probability that the guard numberafter a rebuild is larger than t. Indeed, the balance element cannot be moved closer to an endpoint of the candidate set by more than one element per operation. As the pivot is sampled uniformly after any type of rebuild, we have

$$p_0(t) \geq \Pr(\text{guard number} > t) \geq 1 - 2t/M_d.$$

Now, assume by induction that $p_i(t) \geq 1 - 2t/M_d$ and we want to show $p_{i+1}(t) \geq 1 - t/M_d$. Let t' be the last time step before the $(i+1)$st non-out-of-bounds rebuild.

The following conditions ensure that there are no out-of-bounds rebuilds in the first t operations:

1. there is no out-of-bounds rebuild in the first t' operations, and

2. there is no out-of-bounds rebuild in the subsequent $t - t'$ operations.

These two events are independent since there is a fixed (free or lottery) rebuild between them. The first occurs with probability $p_i(t')$ and the second with probability $p_0(t - t')$. Thus, we have

$$\begin{aligned} p_{i+1}(t) &\geq p_i(t')\, p_0(t - t') \\ &\geq (1 - 2t'/M_d)\,(1 - 2(t - t')/M_d) \geq 1 - 2t/M_d, \end{aligned}$$

and the induction is complete. □

Definition 3.14. *Consider an out-of-bounds rebuild of a range R at depth d.*

We call this rebuild a good out-of-bounds rebuild *if R has only free and lottery rebuilds for the next $M_d/4$ operations.*

By Lemma 3.13, an out-of-bounds rebuild is good with probability at least $1/2$. We are now ready to prove Theorem 3.11.

Proof of Theorem 3.11: Consider the sequence of rebuilds of ranges at a given depth d. We analyze lottery rebuilds and out-of-bounds rebuilds separately. Since variations in $\hat{N}$ slightly change the candidate set size, let M denote the smallest candidate-set size at depth d over the k operations. However, since $N = \Theta(N_M)$ at all times, $M = \Theta(|M_d|)$.

We give a (weak) bound on k/M which helps show that the high-probability bounds hold with respect to k.

In particular,

$$k/M \geq \frac{k \log N_M}{N_M} \geq \frac{k \log k \log N_M}{N_M \log k} = \Omega(\log k),$$

since $k/\log k = \Omega(N_M/\log N_M)$ if $k = \Omega(N_M)$.

Each operation has probability at most $1/M$ of causing a lottery rebuild. Thus, there are k/M lottery rebuilds in expectation. Then using Chernoff bounds, the probability that we have more than $(1 + \delta)k/M$ rebuilds is less than $e^{-\delta k/3M}$. Recall that $k/M = \Omega(\log k)$. Substituting, there are $O(k/M)$ lottery rebuilds with high probability.

Now, we bound the out-of-bounds rebuilds. By the pigeonhole principle, there cannot be more than $k/(M/4) + 2^d = O(k/M)$ good out-of-bounds rebuilds (the second term comes from the number of ranges at depth d).

We bound how many bad out-of-bounds rebuilds can happen before reaching this limit on good out-of-bounds rebuilds. Any out-of-bounds rebuild is good with probability at least $1/2$. Then after $k/M = \Omega(\log k)$ out-of-bounds rebuilds, we

obtain $\Theta(k/M)$ good out-of-bounds rebuilds with high probability, again by Chernoff bounds. As we can only get $O(k/M)$ good rebuilds, we have $O(k/M)$ out-of-bounds rebuilds in total.

Therefore, every k operations, there are $O(k/M)$ out-of-bounds and lottery rebuilds of ranges at depth d with high probability. Each rebuild costs $O(M \log N_M)$ RAM operations by Lemma 3.10 (because that is the number of slots in a range at depth d). Thus, the total rebuild cost for depth d is $O(k \log N_M)$.

Having determined the cost of rebalancing at each depth, we account for the total cost. The amortized rebalance cost, summing over all $h = O(\log N_M)$ levels, is $O(\log^2 N_M)$.

Each resize costs $O(N_M)$ and occurs with probability $O(1/N_M)$ after every insertion. Using Chernoff bounds, there are $O((k \log N_M)/N_M) = \Omega(\log k)$ resizes after k operations with high probability, leading to an additional amortized cost of $O(\log N_M)$.

Finally, each insert or delete requires extra bookkeeping: we must find the appropriate leaf range to insert the element by traversing the rank tree. This traversal takes $O(\log N_M)$ RAM operations, and rebuilding the leaf so that the elements are still evenly spaced takes $O(\log N_M)$ RAM operations.

This gives a total of $O(k \log^2 N_M)$ total RAM operations with high probability.

Using similar analysis, we can also bound the I/O performance. Recall that rebalancing a range of size R takes $O(R/B + 1)$ I/Os, and traversing a tree in the Van Emde Boas layout requires $O(\log_B N_M)$ I/Os. Carrying these terms through the above proof gives the desired bounds. □

To complete the proof of Theorem 3.6, we need to extend this analysis to handle the PMA changing size significantly.

Proof of Theorem 3.6: We partition the $k = \Omega(N)$ operations on the PMA into types based on the size of the PMA. In particular, let $\hat{N}_t$ be the value of $\hat{N}$ during the tth operation. Then operation t is of type 0 if $N \geq \hat{N}_t > N/2$, type 1 if $N/2 \geq \hat{N}_t > N/4$, and type i if $N/2^i \geq \hat{N}_t > N/2^{i+1}$ for $0 \leq i \leq \lceil \log N \rceil$.

We analyze each type of operations as a whole, and bound its total cost, summing to $O(k \log^2 N)$ RAM operations in total. Each type is analyzed using two cases.

First, consider a type i which has at least $\sqrt{N}$ total operations. Then by Theorem 3.11, these operations take amortized $O(\log^2 N)$ RAM operations with high probability.

Second, consider a type i which has less than $\sqrt{N}$ operations. We call the operations of these types *small-type operations*. We show that the total cost of all small-type operations is a lower-order term.

Since the PMA begins as empty, and each operation can only insert one element, there are at least $N/2^{i+1}$ operations of type i. Thus, each small-type operation t has

$\hat{N}_t \leq \sqrt{N}$.

Then overall, a type which has less than $\sqrt{N}$ operations must operate on a PMA of size $\leq \sqrt{N}$. Thus, each type has total cost $O(N)$. Summing over the $O(\log N)$ such types, we get a worst-case total cost of $O(N \log N)$ for small-type operations; amortizing gives a cost of $O(\log N)$ RAM operations.

Thus the total amortized cost is $O(\log^2 N)$ with high probability with respect to N. The I/O cost to rebuild range R is $|R|/B$ by Lemma 3.10; carrying this term through the above analysis, we obtain the desired I/O bounds. □

Chapter 4

High-Performance Computing

Modern CPUs are often able to process data much more quickly than they are able to access it from DRAM, especially while computers become more parallel and more streamlined, while DRAM access speeds stay relatively stagnant. In particular, many of these processes are *memory-bandwidth bound*: they are not able to access data as quickly as they can process it, but increasing the bandwidth of memory alleviates this issue. In the language of the DAM model (see Chapter 3), the performance of memory-bound processes improves as we increase B.

Not all memory-bound processes are memory-bandwidth bound. For example, a process that accesses DRAM pages at random has to perform one I/O per operation regardless of bandwidth. However, many common processes like sorting and sparse matrix multiplication include scans over large amounts of data, which decrease in cost as the bandwidth increases.

Recently, architectural advances have led to a new kind of memory which has the potential to significantly improve the performance of memory-bandwidth-bound processes. This memory has similar latency to DRAM, but higher bandwidth. Thus, we refer to this new memory as High-Bandwidth Memory, or HBM. While its bandwidth is higher, HBM is smaller than classic DRAM. Current HBM implementations are generally around 4-16 GB [81, 87].

High-Bandwidth Memory has been applied to both CPUs [4, 18, 81, 111, 115] and GPUs [4, 115, 120]. In both of these applications, the focus is on highly parallel machines (as without such parallelism, I/Os are generally able to keep up with computation in practice). Our work here focuses on CPUs. We gloss over the parallel aspect of HBM compute nodes for simplicity in our algorithm analysis.[8] However, the practical (and parallel) version of our algorithm is a reasonably straightforward generalization of the ideas presented here. For a full exposition of this parallel generalization, see [18].

Because of this tradeoff between smaller size and larger bandwidth, HBM is not used as a level in the current memory hierarchy. Instead, algorithm designers must decide whether a given page should be stored in large, low-bandwidth DRAM, or the small, high-bandwidth HBM.

This leads to a fundamental question: what data should we store in HBM to take

[8] In particular, we use a very simplified memory hierarchy which ignores the complicated cache structure of these compute nodes.

maximum advantage of the improved bandwidth? Can we use HBM to improve the real-time performance of common memory-bandwidth-bound applications?

In this work, we focus on the theoretical basis for HBM: setting up the theoretical model, and giving an optimal algorithm for sorting. The full paper [18] also includes simulation results which indicate that this algorithm improves wall clock running time in practice.

4.1 Algorithmic High-Bandwidth Memory Model

In this section, we propose an algorithmic model of the HBM, generalizing the DAM (i.e. external-memory model) of Aggarwal and Vitter [3] to include high- and low-bandwidth memory. (This model is described in more depth in Chapter 3.)

HBM model

Both the DRAM and the HBM are independently connected to the cache; see Figure 3. The HBM hardware has an access time that is roughly the same as that of normal DRAM—the difference is that the HBM has limited size but higher bandwidth. We model the higher bandwidth of the HBM by transferring data to and from the DRAM in smaller blocks of size B and to and from the HBM in larger blocks of size ρB, $\rho > 1$.

We parameterize memory as follows: the cache has size Z, the HBM has size $M \gg Z$, and the DRAM is modeled as arbitrarily large. We assume a tall cache; that is, $M > B^2$. This is a common assumption in external-memory analysis, see e.g. [22, 31, 32, 64].

We model the roughly similar access times of the actual hardware by charging each block transfer a cost of 1, regardless of whether it involves a large or small block. Performance is measured in terms of block transfers. Computation is modeled as free because we consider memory-bound computations.

Note that the HBM block size, ρB, in the algorithmic performance model need not be the same as the block-transfer size in the actual hardware. But for the purpose of algorithm design, one should program assuming a block size of ρB, so that the latency contribution for a transfer is dominated by the bandwidth component.

In the following, we say that event E_N on problem size N occurs *with high probability* if $\Pr(E_N) \geq 1 - O(\frac{1}{N^c})$, for some constant c. This is discussed in further detail in Chapter 3.3.

Putting the HBM model in context

The external-memory model of Aggarwal and Vitter [3], assumes a two-level hierarchy comprised of a small, fast memory level and an arbitrarily large second level. Data

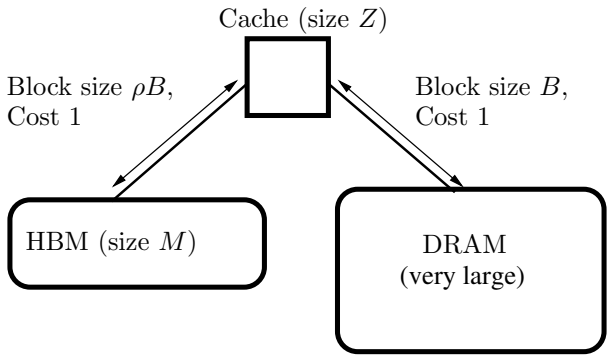

Figure 3: The High Bandwidth Memory model.

is transferred between the two levels in blocks of size B.

Because the HBM model has two different bandwidths but roughly similar latencies, we have extended the external-memory model to have two different block sizes, each with the same transfer cost; see Figure 3.

There do exist other memory-hierarchy models, such as the cache-oblivious model [64, 104], which apply to multi-level memories. However, the different bandwidths but similar latencies means that the cache-oblivious model does not seem to apply. Instead, the HBM model can be viewed as a three-level model, but where two of the levels are in parallel with each other.

Background on external-memory sorting

The following well-known theorems give bounds on the cost of external-memory sorting.

Theorem 4.1 ([3])**.** *Sorting N numbers from DRAM with a cache of size Z and block size L (but no HBM) requires $\Theta\big((N/L)\log_{Z/L}(N/L)\big)$ block transfers from DRAM. This bound is achievable using multi-way merge sort with a branching factor of Z/L.*

Theorem 4.2 ([3])**.** *Sorting N numbers from DRAM with a cache of size Z and block size L (but no HBM) using merge sort takes $\Theta\big((N/L)\lg(N/Z)\big)$ block transfers from DRAM.*

In the analysis of sorting algorithms in the following sections, we will sometimes apply these theorems with $L = B$ and sometimes apply them with $L = \rho B$.

4.2 Sorting with High-Bandwidth Memory

This section gives an optimal randomized algorithm for sorting with one processor with a HBM. This algorithm generalizes multiway merge sort and distribution sort [3], which in turn generalize merge sort and sample sort [62].

We are given an array $A[1, \ldots, N]$ of N elements to sort, where $N \gg M$. For simplicity of exposition we assume that all elements in A are distinct, but this assumption can be removed.

The algorithm works by recursively reducing the size of the input until until each subproblem fits into the HBM, at which point it can be sorted rapidly. In each recursive step, we "bucket" elements so that for any two consecutive buckets (or ranges) R_i and R_{i+1}, every element in R_i is less than every element in R_{i+1}. We then sort each bucket recursively and concatenate the results, thus sorting all elements.

4.2.1 Choosing Bucket Boundaries

To perform the bucketing, we randomly select a sample set X of $m = \Theta(M/B)$ elements from $A[1, \ldots, N]$ and sort them within the HBM. (We assume sampling with replacement, but sampling without replacement also works.)

The exact value of m is chosen by the algorithm designer, but must be small enough to fit in the HBM.

Our implementation uses multiway mergesort from the GNU C++ library to sort within the HBM [39]. Other sorting algorithms could be used, such as quicksort. If ρ is sufficiently large, either sorting algorithm within the HBM leads to an optimal algorithm (see Theorem 4.6); however, the value of ρ based on current hardware probably is not large enough to make quicksort practically competitive with mergesort.

Corollary 4.3. *Sorting x elements that fit in the HBM uses $\Theta\big((x/\rho B) \log_{Z/B} x/B\big)$ block transfers using multi-way merge sort with a branching factor of Z/B, or $\Theta\big((x/\rho B) \lg (x/Z)\big)$ in expectation using quicksort. Both algorithms use $O(x \lg x)$ work.*

4.2.2 Bucketizing

We are given the sample set $X = \{x_1, x_2, \ldots, x_m\}$ $(x_1 < x_2 < \ldots < x_m)$ of sorted elements and input array A. Since $|X| = m$, X fits in the HBM, and A does not.

Our objective is to place each element $A[i] \in A \setminus X$ into bucket R_j if $x_j < A[i] < x_{j+1}$. (We may assume that $x_0 = -\infty$, and $x_{m+1} = \infty$.)

We perform a linear scan through A, ingesting $M - \Theta(m)$ elements into the scratchpad at a time. All of X remains in the HBM for the entire scan.

Once each new group of elements is loaded into the HBM, we sort them (along with X). Their position relative to the elements of X indicate which bucket they are stored in. The resulting bucketed elements get written back to DRAM, while the set X remains in the HBM. We call this process a *bucketizing scan*.

Lemma 4.4. *Each bucketizing scan costs:*

1. $O(N/B)$ *block transfers from DRAM,*
2. $O(N/(\rho B)\log_{Z/(\rho B)} M/(\rho B))$ *block transfers from the HBM, and*
3. $O(N \lg M)$ *operations in the RAM model.*

Proof. We select and sort the set X of sampled points in the HBM (necessary for mergesort), and then bucketize the rest of A. Assume for simplicity that $m \leq M/2$.

Sampling X requires $O(m)$ block transfers and $O(m)$ work, given constant time to choose a random word. Sorting X requires $O(m \lg m)$ work and $O(\frac{m}{\rho B}\log_{Z/B}\frac{m}{B})$ block transfers (via Corollary 4.3).

To bucketize, we read all elements from DRAM to the HBM in blocks of size $M - \Theta(m) \geq M/2$. This requires $O(M/B)$ block transfers from DRAM. The algorithm sorts each of these blocks in $O(M/(\rho B)\log_{Z/(\rho B)} M/(\rho B))$ HBM block transfers and requires $O(M \lg M)$ work to decide their buckets. There are $O(N/M)$ such groups, so this requires $O(N/B + N/(\rho B)\log_{Z/(\rho B)} M/(\rho B))$ total block transfers. We write each bucket to a separate piece of DRAM memory. This does not increase the cost: were the writing done to a contiguous location, it would require $\Theta(M/B)$ block transfers as the reading does. By writing the elements in $O(m) = O(M/B)$ pieces, we may incur up to two extra block transfers per piece: one to start each bucket, and perhaps another to end it. Thus the total number of transfers to write buckets to DRAM (even in seperate pieces) is $\Theta(M/B)$. □

4.2.3 Bounding the Recursion Depth for Sorting

We now conclude our analysis of HBM sorting. Recall that the algorithm successively scans the input, refining it into progressively smaller buckets. Lemma 4.4 from the previous section gave a bound of the complexity of each scan.

In this section, we first show that the random samples are good enough so that with high probability each bucket is small enough to fit into cache after $O(\log_m(N/M))$ bucketizing scans. Then we give a bound on the sorting complexity.

This randomized analysis applies to any external-memory sort. However, to the best of our knowledge, this kind of analysis does not appear in the literature. Previous external memory sorting algorithms were deterministic. However, randomized algorithms are practical.

Lemma 4.5. *Each bucket fits into cache after $O(\log_m(N/M))$ bucketizing scans with high probability.*

Proof. Each newly-created bucket results from a *good* or *bad* split depending on its size. If a split makes a new bucket at least $\sqrt{m}$ factor smaller than the old one we call this a good split; otherwise it is bad.

We show that in a sufficiently large set of $O(\log_m(N/M))$ splits, there are $3\log_m(N/M)$ good splits with high probability. Thus, after $O(\log_m(N/M))$ scans, each bucket is involved in at least $3\log_m(N/M)$ good splits, and fits in HBM.

We first show that the probability of a bad split is exponentially small in $\sqrt{m}$. Let R_i be the bucket being split. For simplicity, assume R_i lists the elements in sorted order, though the bucket itself may not be sorted yet. If a subbucket starting at j came from a bad split, no element in $\{R_i(j), R_i(j+1), \ldots, R_i(j + (|R_i|\sqrt{m}/m))\}$ is sampled to be in X. This happens with probability at most $(1 - \sqrt{m}/m)^m \approx e^{-\sqrt{m}}$.

We next show that of any $(3/2)c\log_m(N/M)$ splits on a root-to-leaf path of the recursion tree, at most $c\log_m(N/M)$ are bad with high probability. Thus there are at least $(c/2)\log_m(N/M)$ good splits; choosing $c \geq 6$ proves the lemma.

By linearity of expectation, the expected number of bad splits is at most $(3/2)ce^{-\sqrt{m}}\log_m(N/M)$. Applying Chernoff bounds [94] with $\delta = (2/3)e^{\sqrt{m}} - 1$, we obtain

$$\Pr\left[\#\text{ bad splits } > c\log_m \frac{N}{M}\right] \leq \left(\frac{e}{2e^{\sqrt{m}}/3}\right)^{c\log_m \frac{N}{M}}.$$

We want to show that this is bounded above by $1/N^\alpha$. Taking the log of both sides of the inequality, this simplifies to

$$c\left((\sqrt{m}-1)\lg e - \lg 3/2\right)\log_m N/M \geq \alpha \lg N.$$

Now there are two cases. First, assume $M \geq \sqrt{N}$. Recall the tall cache assumption, that $M > B^2$. Then $\sqrt{m} = \sqrt{M/B} \geq M^{1/4} \geq N^{1/8}$. Substituting, we obtain

$c((N^{1/8}-1)\lg e - \lg 3/2)\log_m N/M \geq \alpha \lg N$, which is true for sufficiently large N and $c = \alpha$.

On the other hand, let $M < \sqrt{N}$. Note that for $m \geq 2$, $((\sqrt{m}-1)\lg e - \lg 3/2)\,/\lg m \geq 1/2$. Then for $c \geq 4\alpha$,

$$c\left((\sqrt{m}-1)\lg e - 2\right)\log_m \frac{N}{M} \geq \frac{c}{2}\lg\frac{N}{M} \geq \alpha \lg N.$$

This shows that each bucket is sufficiently small with high probability after $O(\log_m N/M)$ bucketizing scans. To show that all must be small enough, take the union bound to obtain a probability no more than $1/N^{\alpha-1}$. Performing the analysis with $\alpha' = \alpha + 1$ gives us the desired bound. □

Theorem 4.6. *There exists a sorting algorithm that performs* $\Theta\left(\frac{N}{B}\log_{M/B}\frac{N}{B} + \frac{N}{\rho B}\log_{Z/\rho B}\frac{N}{B}\right)$ *block transfers with high probability to partition the array into buckets, all of size* $\leq M$*. This bound comprises* $O(\frac{N}{B}\log_{M/B}\frac{N}{B})$ *block transfers from DRAM and* $O(\frac{N}{\rho B}\log_{Z/\rho B}\frac{N}{B})$ *high bandwidth block transfers from the HBM. There is a matching lower bound, so this algorithm is optimal.*

Proof. By Lemma 4.5 there are $O(\log_m N/M) \leq O(1+\log_m N/M) = O(\log_{M/B} N/B)$ bucketizing scans; multiplying this by the block transfers per scan in Lemma 4.4 gives us the desired bounds. Note that once the buckets are of size $\leq M$, we can sort all of them with $O(\frac{N}{\rho B}\log_{Z/\rho B}\frac{M}{\rho B})$ additional block transfers by Corollary 4.3; this is a lower-order term.

To obtain the lower bound we will combine two lower bounds from weaker models. First consider a model where computation can be done in the HBM as well as in cache. Then the standard two-level hierarchy lower bound applies, and sorting requires $\Omega(\frac{N}{B}\log_{M/B}\frac{N}{B})$ block transfers from DRAM (since $M + Z = O(M)$). This bound is the same as the one given in Theorem 4.1, and can be found in [3].

Consider a second model where the DRAM allows high-bandwidth block transfers. Again, the standard two-level hierarchy lower bound applies, and sorting requires $\Omega(\frac{N}{\rho B}\log_{Z/(\rho B)}\frac{N}{\rho B})$ high-bandwidth block transfers.

Both of these bounds must be satisfied simultaneously. Thus, all together we require $\Omega(\frac{N}{B}\log_{M/B}\frac{N}{B} + \frac{N}{\rho B}\log_{Z/\rho B}\frac{N}{\rho B})$ block transfers. Since $(N/\rho B)\log_{Z/\rho B}\rho < N/B$, this simplifies to $\Omega\left(\frac{N}{B}\log_{M/B}\frac{N}{B} + \frac{N}{\rho B}\log_{Z/\rho B}\frac{N}{B}\right)$. □

The following corollary explains how the sorting algorithm performs when quicksort is used within the HBM, and follows by linearity of expectation. The corollary shows that the algorithm remains optimal as long as the bandwidth of the HBM, determined by ρ, is sufficiently large.

Corollary 4.7. *An implementation of sorting using quicksort within the HBM uses* $O(\frac{N}{B}\log_{M/B}\frac{N}{B} + \frac{N}{\rho B}\lg\frac{M}{Z}\log_{M/B}\frac{N}{B}))$ *block transfers in expectation. This is optimal when* $\rho = \Omega(\lg M/Z)$.

References

[1] A. Abouzeid, K. Bajda-Pawlikowski, D. Abadi, A. Silberschatz, and A. Rasin. HadoopDB: an architectural hybrid of MapReduce and DBMS technologies for analytical workloads. *Proceedings of the VLDB Endowment*, 2(1):922–933, 2009.

[2] G. Adelson-Velskii and E. Landis. An algorithm for the organization of information. Technical report, DTIC Document, 1963.

[3] A. Aggarwal and J. S. Vitter. The input/output complexity of sorting and related problems. *Communications of the ACM*, 31(9):1116–1127, Sept. 1988.

[4] `https://www.amd.com/Documents/High-Bandwidth-Memory-HBM.pdf/`.

[5] A. Andersson, T. Hagerup, S. Nilsson, and R. Raman. Sorting in linear time? In *Proceedings of the Twenty-seventh Annual ACM Symposium on Theory of Computing (STOC)*, pages 427–436. ACM, 1995.

[6] A. Andersson and T. W. Lai. Fast updating of well-balanced trees. In *Proceedings of the Second Scandinavian Workshop on Algorithm Theory (SWAT)*, pages 111–121, 1990.

[7] E. Angel, E. Bampis, V. Chau, and V. Zissimopoulos. Minimizing total calibration cost. *arXiv preprint arXiv:1507.02808*, 2015.

[8] K. Anthonisse and J. K. Lenstra. Operational operations research at the mathematical centre. *European Journal of Operational Research*, 15(3):293–296, 1984.

[9] C. R. Aragon and R. G. Seidel. Randomized search trees. In *Proceedings of the Thirtieth Annual Symposium on Foundations of Computer Science (FOCS)*, pages 540–545, 1989.

[10] E. M. Arkin and E. B. Silverberg. Scheduling jobs with fixed start and end times. *Discrete Applied Mathematics*, 18(1):1–8, 1987.

[11] R. Baer. Self-calibrating and/or self-testing camera module, Sept. 30 2005. US Patent App. 11/239,851.

[12] S. K. Bansal, T. Layloff, E. D. Bush, M. Hamilton, E. A. Hankinson, J. S. Landy, S. Lowes, M. M. Nasr, P. A. S. Jean, and V. P. Shah. Qualification of analytical instruments for use in the pharmaceutical industry: A scientific approach. *Aaps Pharmscitech*, 5(1):151–158, 2004.

[13] P. Baptiste. Scheduling unit tasks to minimize the number of idle periods: a polynomial time algorithm for offline dynamic power management. In *Proceedings of the Seventeenth Annual Symposium on Discrete Algorithms (SODA)*, pages 364–367. SIAM, 2006.

[14] P. Baptiste, M. Chrobak, and C. Dürr. Polynomial time algorithms for minimum energy scheduling. In *Proceedings of the Fifteenth Annual European Symposium on Algorithms (ESA)*, pages 136–150. Springer, 2007.

[15] H. P. Barringer. Cost effective calibration intervals using Weibull analysis. In *Annual Quality Congress Proceedings-American Society For Quality Control*, pages 1026–1038, 1995.

[16] A. Barton-Sweeney, D. Lymberopoulos, and A. Savvides. Sensor localization and camera calibration in distributed camera sensor networks. In *Proceedings of the Third International Conference on Broadband Communications (BROADNETS)*, pages 1–10. IEEE, 2006.

[17] Beamex. Traceable and efficient calibrations in the process industry, October 2007. `http://www.iceweb.com.au/Test\&Calibration/BEAMEX\%20Papers/White_Paper_Traceable\%20and\%20efficient\%20calibrations\%20in\%20the\%20process\%20industry.pdf`.

[18] M. A. Bender, J. Berry, S. D. Hammond, K. S. Hemmert, S. McCauley, B. Moore, B. Moseley, C. A. Phillips, D. Resnick, and A. Rodrigues. Two-level main memory co-design: multi-threaded algorithmic primitives, analysis, and simulation. In *Proceedings of the Twenty-Ninth International Parallel and Distributed Processing Symposium (IPDPS)*, pages 835–846. IEEE, 2015.

[19] M. A. Bender, D. P. Bunde, V. J. Leung, S. McCauley, and C. A. Phillips. Efficient scheduling to minimize calibrations. In *Proceedings of the Twenty-fifth Annual ACM Symposium on Parallelism in Algorithms and Architectures (SPAA)*, pages 280–287. ACM, 2013.

[20] M. A. Bender, R. Cole, E. D. Demaine, M. Farach-Colton, and J. Zito. Two simplified algorithms for maintaining order in a list. In *Proceedings of the Tenth Annual European Symposium on Algorithms (ESA)*, pages 152–164. Springer, 2002.

[21] M. A. Bender, E. D. Demaine, and M. Farach-Colton. Cache-oblivious B-trees. *SIAM Journal on Computing*, 35(2):341–358, 2005.

[22] M. A. Bender, R. Ebrahimi, J. T. Fineman, G. Ghasemiesfeh, R. Johnson, and S. McCauley. Cache-adaptive algorithms. In *Proceedings of the Twenty-Fifth Symposium on Discrete Algorithms (SODA)*, pages 116–128, 2014.

[23] M. A. Bender and H. Hu. An adaptive packed-memory array. *ACM Transactions on Database Systems*, 32(4):26, 2007.

[24] J. Benlloch, V. Carrilero, A. González, J. Catret, C. W. Lerche, D. Abellán, F. G. De Quiros, M. Giménez, J. Modia, F. Sánchez, et al. Scanner calibration of a small animal pet camera based on continuous lso crystals and flat panel pspmts. *Nuclear Instruments and Methods in Physics Research Section A: Accelerators, Spectrometers, Detectors and Associated Equipment*, 571(1):26–29, 2007.

[25] R. Bernhardt and S. Albright. *Robot calibration*. Springer Science & Business Media, 1993.

[26] R. Bhatia, J. Chuzhoy, A. Freund, and J. Naor. Algorithmic aspects of bandwidth trading. *Proceedings of the 30th International Colloquium on Automata, Languages and Programming (ICALP)*, pages 193–193, 2003.

[27] G. E. Blelloch and D. Golovin. Strongly history-independent hashing with applications. In *Proceedings of the Forty-Eighth Annual Symposium on Foundations of Computer Science (FOCS)*, pages 272–282, 2007.

[28] G. E. Blelloch, D. Golovin, and V. Vassilevska. Uniquely represented data structures for computational geometry. In *Proceedings of the Eleventh Scandinavian Workshop on Algorithm Theory (SWAT)*, pages 17–28, 2008.

[29] K. Bouzina and H. Emmons. Interval scheduling on identical machines. *Journal of Global Optimization*, 9(3):379–393, 1996.

[30] M. Brehob, S. Wagner, E. Torng, and R. Enbody. Optimal replacement is NP-hard for nonstandard caches. *IEEE Transactions on Computers*, 53(1):73–76, 2004.

[31] G. S. Brodal, E. D. Demaine, J. T. Fineman, J. Iacono, S. Langerman, and J. I. Munro. Cache-oblivious dynamic dictionaries with update/query tradeoffs. In *Proceedings of the Twenty-First Annual ACM-SIAM Symposium on Discrete Algorithms (SODA)*, pages 1448–1456, 2010.

[32] G. S. Brodal and R. Fagerberg. On the limits of cache-obliviousness. In *Proceedings of the Thirty-Fifth Annual ACM Symposium on Theory of Computing (STOC)*, pages 307–315, 2003.

[33] G. S. Brodal, R. Fagerberg, and K. Vinther. Engineering a cache-oblivious sorting algorithm. *Journal of Experimental Algorithmics (JEA)*, 12:2–2, 2008.

[34] A. Brodnik. Computation of the least significant set bit. In *Proceedings of the 2nd Electrotechnical and Computer Science Conference, Portoroz, Slovenia*, volume 90. Citeseer, 1993.

[35] A. Brodnik, P. B. Miltersen, and J. I. Munro. Trans-dichotomous algorithms without multiplicationsome upper and lower bounds. In *Proceedings of the Fifth International Workshop on Algorithms and Data Structures (WADS)*, pages 426–439. Springer, 1997.

[36] P. Brucker and P. Brucker. *Scheduling algorithms*, volume 3. Springer, 2007.

[37] J. Bulánek, M. Koucký, and M. Saks. Tight lower bounds for the online labeling problem. In *Proceedings of the Forty-Fourth Annual ACM Symposium on Theory of Computing (STOC)*, pages 1185–1198, 2012.

[38] C. Burroughs. New integrated stockpile evaluation program to better ensure weapons stockpile safety, security, reliability. `http://www.sandia.gov/LabNews/060331.html`, March 2006. Online; posted March 2006.

[39] P. Carlini, P. Edwards, D. Gregor, B. Kosnik, D. Matani, J. Merrill, M. Mitchell, N. Myers, F. Natter, S. Olsson, S. Rus, J. Singler, A. Tavory, and J. Wakely. The GNU C++ library manual, 2012.

[40] M. Carter and C. Tovey. When is the classroom assignment problem hard? *Operations Research*, pages 28–39, 1992.

[41] L. Chen, N. Megow, and K. Schewior. An O(log m)-competitive algorithm for online machine minimization. In *Proceedings of the Twenty-Seventh Annual Symposium on Discrete Algorithms (SODA)*, pages 155–163. SIAM, 2016.

[42] Z. Chen, T. Jiang, G. Lin, R. Rizzi, J. Wen, D. Xu, and Y. Xu. More reliable protein NMR peak assignment via improved 2-interval scheduling. pages 580–592, 2003.

[43] P.-J. Chuang and N.-F. Tzeng. An efficient submesh allocation strategy for mesh computer systems. In *Proceedings of the Eleventh International Conference on Distributed Computing Systems*, pages 256–263. IEEE, 1991.

[44] J. Chuzhoy, S. Guha, S. Khanna, and J. S. Naor. Machine minimization for scheduling jobs with interval constraints. In *Proceedings of the Forty-Fifth Annual IEEE Symposium on Foundations of Computer Science (FOCS)*, pages 81–90. IEEE, 2004.

[45] V. Chvatal. *Linear programming*. Macmillan, 1983.

[46] M. Cieliebak, T. Erlebach, F. Hennecke, B. Weber, and P. Widmayer. Scheduling with release times and deadlines on a minimum number of machines. In *Exploring New Frontiers of Theoretical Informatics*, pages 209–222. Springer, 2004.

[47] S. A. Cook and R. A. Reckhow. Time-bounded random access machines. In *Proceedings of the Fourth Annual Symposium on Theory of Computing (STOC)*, pages 73–80. ACM, 1972.

[48] G. Dantzig and D. Fulkerson. Minimizing the number of tankers to meet a fixed schedule. *Naval Research Logistics Quarterly*, 1(3):217–222, 1954.

[49] J. Dean and S. Ghemawat. Mapreduce: simplified data processing on large clusters. *Communications of the ACM*, 51(1):107–113, 2008.

[50] M. Dessouky, M. Dessouky, and S. Verma. Flowshop scheduling with identical jobs and uniform parallel machines. *European Journal of Operational Research*, 109(3):620–631, 1998.

[51] M. Dessouky, B. Lageweg, J. Lenstra, and S. Velde. Scheduling identical jobs on uniform parallel machines. *Statistica Neerlandica*, 44(3):115–123, 1990.

[52] P. Dietz and D. Sleator. Two algorithms for maintaining order in a list. In *Proceedings of the Nineteenth Annual Symposium on Theory of Computing (STOC)*, pages 365–372, 1987.

[53] P. Dietz and J. Zhang. Lower bounds for monotonic list labeling. In *Proceedings of the Second Scandinavian Workshop on Algorithm Theory (SWAT)*, pages 173–180, 1990.

[54] P. F. Dietz. Maintaining order in a linked list. In *Proceedings of the Fourteenth Annual Symposium on Theory of Computing (STOC)*, pages 122–127, 1982.

[55] P. F. Dietz, J. I. Seiferas, and J. Zhang. A tight lower bound for on-line monotonic list labeling. In *Proceedings of the Fourth Scandinavian Workshop on Algorithm Theory (SWAT)*, volume 824 of *Lecture Notes in Computer Science*, pages 131–142, 1994.

[56] V. Dondeti and H. Emmons. Fixed job scheduling with two types of processors. *Operations Research*, pages 76–85, 1992.

[57] T. Erlebach and F. Spieksma. Interval selection: Applications, algorithms, and lower bounds. *Journal of Algorithms*, 46(1):27–53, 2003.

[58] R. C. Evans, J. E. Griffith, D. D. Grossman, M. M. Kutcher, and P. M. Will. Method and apparatus for calibrating a robot to compensate for inaccuracy of the robot, Dec. 7 1982. US Patent 4,362,977.

[59] U. Faigle and W. Nawijn. Note on scheduling intervals on-line. *Discrete Applied Mathematics*, 58(1):13–17, 1995.

[60] J. T. Fineman and B. Sheridan. Scheduling non-unit jobs to minimize calibrations. In *Proceedings of the Twenty-Seventy Annual Symposium on Parallelism in Algorithms and Architectures (SPAA)*, pages 161–170. ACM, 2015.

[61] M. Forina, M. Casolino, and C. De la Pezuela Martínez. Multivariate calibration: applications to pharmaceutical analysis. *Journal of Pharmaceutical and Biomedical Analysis*, 18(1):21–33, 1998.

[62] W. D. Frazer and A. C. McKellar. Samplesort: A sampling approach to minimal storage tree sorting. *Journal of the ACM*, 17(3):496–507, 1970.

[63] M. L. Fredman and D. E. Willard. Surpassing the information theoretic bound with fusion trees. *Journal of Computer and System Sciences*, 47(3):424–436, 1993.

[64] M. Frigo, C. E. Leiserson, H. Prokop, and S. Ramachandran. Cache-oblivious algorithms. *ACM Transactions on Algorithms*, 8(1):4, 2012.

[65] V. Gabrel. Scheduling jobs within time windows on identical parallel machines: New model and algorithms. *European Journal of Operational Research*, 83(2):320–329, 1995.

[66] M. R. Garey and D. S. Johnson. Two-processor scheduling with start-times and deadlines. *SIAM Journal on Computing*, 6(3):416–426, 1977.

[67] M. X. Goemans and D. P. Williamson. The primal-dual method for approximation algorithms and its application to network design problems. *Approximation algorithms for NP-hard problems*, pages 144–191, 1997.

[68] D. Golovin. B-treaps: A uniquely represented alternative to B-trees. In *Proceedings of the Thirty-Sixth Annual International Colloquium on Automata, Languages, and Programming (ICALP)*, pages 487–499, 2009.

[69] D. Golovin. The B-skip-list: A simpler uniquely represented alternative to B-trees. *arXiv preprint arXiv:1005.0662*, 2010.

[70] G. H. Golub and C. F. Van Loan. *Matrix computations*, volume 3. JHU Press, 2012.

[71] R. Graham, E. Lawler, J. Lenstra, and A. Kan. Optimization and approximation in deterministic sequencing and scheduling: a survey. *Annals of discrete mathematics*, 5(2):287–326, 1979.

[72] U. Gupta, D. Lee, and J. Leung. An optimal solution for the channel-assignment problem. *IEEE Transactions on Computers*, 100(11):807–810, 1979.

[73] N. Hall and M. Posner. Earliness-tardiness scheduling problems, i: weighted deviation of completion times about a common due date. *Operations Research*, pages 836–846, 1991.

[74] J. D. Hartline, E. S. Hong, A. E. Mohr, W. R. Pentney, and E. C. Rocke. Characterizing history independent data structures. *Algorithmica*, 42(1):57–74, 2005.

[75] A. Hashimoto and J. Stevens. Wire routing by optimizing channel assignment within large apertures. In *Proceedings of the Eighth Design Automation Workshop*, pages 155–169. ACM, 1971.

[76] S. Im, S. Li, B. Moseley, and E. Torng. A dynamic programming framework for non-preemptive scheduling problems on multiple machines. In *Proceedings of the Twenty-Sixth Annual Symposium on Discrete Algorithms (SODA)*, pages 1070–1086. SIAM, 2015.

[77] A. Itai, A. Konheim, and M. Rodeh. A sparse table implementation of priority queues. *Proceedings of the Eighth Annual International Colloquium on Automata, Languages, and Programming (ICALP)*, pages 417–431, 1981.

[78] D. Karger, C. Stein, and J. Wein. Scheduling algorithms. *CRC Handbook of Computer Science*, 1997.

[79] R. Karp. Reducibility among combinatorial problems. *Complexity of Computer Computations*, 1972.

[80] I. Katriel. Implicit data structures based on local reorganizations. Master's thesis, Technion – Israel Inst. of Tech., Haifa, May 2002.

[81] http://www.hpcwire.com/2014/06/24/micron-intel-reveal-memory-slice-knights-landing/.

[82] A. Kolen, J. Lenstra, C. Papadimitriou, and F. Spieksma. Interval scheduling: A survey. *Naval Research Logistics (NRL)*, 54(5):530–543, 2007.

[83] L. Kroon, M. Salomon, and L. Van Wassenhove. Exact and approximation algorithms for the operational fixed interval scheduling problem. *European Journal of Operational Research*, 82(1):190–205, 1995.

[84] T. W. Lai and D. Wood. Updating almost complete trees or one level makes all the difference. In *Proceedings of the Seventh Symposium on Theoretical Aspects of Computer Science (STACS)*, volume 415 of *Lecture Notes in Computer Science*, pages 188–194, 1990.

[85] J. R. Lakin. Establishing calibration intervals, how often should one calibrate? `http://www.inspec-inc.com/home/company/blog/inspec-insights/2014/09/30/establishing-calibration-intervals-how-often-should-one-calibrate`, Sep 2014. Online; posted 30 September 2014.

[86] C. Lee, S. Danusaputro, and C. Lin. Minimizing weighted number of tardy jobs and weighted earliness-tardiness penalties about a common due date. *Computers & operations research*, 18(4):379–389, 1991.

[87] D. U. Lee, K. W. Kim, K. W. Kim, H. Kim, J. Y. Kim, Y. J. Park, J. H. Kim, D. S. Kim, H. B. Park, J. W. Shin, et al. 25.2 A 1.2 V 8Gb 8-channel 128GB/s high-bandwidth memory (HBM) stacked DRAM with effective microbump I/O test methods using 29nm process and TSV. In *2014 IEEE International Solid-State Circuits Conference Digest of Technical Papers (ISSCC)*, pages 432–433. IEEE, 2014.

[88] K.-H. Lin and B.-D. Liu. A gray system modeling approach to the prediction of calibration intervals. *IEEE Transactions on Instrumentation and Measurement*, 54(1):297–304, 2005.

[89] R. Lipton. Online interval scheduling. In *Proceedings of the Fifth Annual Symposium on Discrete Algorithms (SODA)*, pages 302–311. Society for Industrial and Applied Mathematics, 1994.

[90] S. Martello and P. Toth. A heuristic approach to the bus driver scheduling problem. *European Journal of Operational Research*, 24(1):106–117, 1986.

[91] J. Meng, S. McCauley, F. Kaplan, V. J. Leung, and A. K. Coskun. Simulation and optimization of HPC job allocation for jointly reducing communication and cooling costs. *Sustainable Computing: Informatics and Systems Special Issue on Selected Papers from 2013 International Green Computing Conerence (IGCC)*, pages 48–57, June 2015.

[92] D. Micciancio. Oblivious data structures: applications to cryptography. In *Proc. of the 29th Annual ACM Symposium on Theory of Computing (STOC)*, pages 456–464, 1997.

[93] J. D. Moore, J. S. Chase, P. Ranganathan, and R. K. Sharma. Making scheduling" cool": Temperature-aware workload placement in data centers. In *USENIX annual technical conference, General Track*, pages 61–75, 2005.

[94] R. Motwani and P. Raghavan. *Randomized algorithms*. Chapman & Hall/CRC, 2010.

[95] M. Naor, G. Segev, and U. Wieder. History-independent cuckoo hashing. In *Proc. of the 35th International Colloquium on Automata, Languages and Programming (ICALP)*, pages 631–642. Springer, 2008.

[96] M. Naor and V. Teague. Anti-persistence: history independent data structures. In *Proceedings of the Thirty-Third Annual Symposium on Theory of Computing (STOC)*, pages 492–501, 2001.

[97] National Nuclear Safety Administration. Office of test and evaluation. `http://nnsa.energy.gov/aboutus/ourprograms/defenseprograms/stockpilestewardship/testcapabilitiesand-eval`, September 2014. Online; posted 30 September 2014.

[98] H.-N. Nguyen, J. Zhou, and H.-J. Kang. A new full pose measurement method for robot calibration. *Sensors*, 13(7):9132–9147, 2013.

[99] E. Nunzi, G. Panfilo, P. Tavella, P. Carbone, and D. Petri. Stochastic and reactive methods for the determination of optimal calibration intervals. *IEEE Transactions on Instrumentation and Measurement*, 54(4):1565–1569, 2005.

[100] E. Pakbaznia and M. Pedram. Minimizing data center cooling and server power costs. In *Proceedings of the International Symposium on Low Power Electronics and Design*, pages 145–150. ACM, 2009.

[101] H.-M. Park and C.-W. Chung. An efficient mapreduce algorithm for counting triangles in a very large graph. In *Proceedings of the Twenty-Second International Conference on Information & Knowledge Management*, pages 539–548. ACM, 2013.

[102] C. A. Phillips, C. Stein, E. Torng, and J. Wein. Optimal time-critical scheduling via resource augmentation. *Algorithmica*, 32(2):163–200, 2002.

[103] S. Postlethwaite, D. Ford, and D. Morton. Dynamic calibration of cnc machine tools. *International Journal of Machine Tools and Manufacture*, 37(3):287–294, 1997.

[104] H. Prokop. Cache-oblivious algorithms. Master's thesis, Department of Electrical Engineering and Computer Science, Massachusetts Institute of Technology, June 1999.

[105] K. Pruhs, J. Sgall, and E. Torng. Online scheduling. *Handbook of scheduling: algorithms, models, and performance analysis*, pages 15–1, 2004.

[106] W. Pugh. Skip lists: a probabilistic alternative to balanced trees. *Communications of the ACM*, 33(6):668–676, 1990.

[107] S. Seiden. Randomized online interval scheduling. *Operations research letters*, 22(4-5):171–177, 1998.

[108] L. Snyder. On uniquely represented data strauctures. In *Proceedings of the Eighteenth IEEE Symposium on Foundations of Computer Science (FOCS)*, pages 142–146, 1977.

[109] F. Spieksma. On the approximability of an interval scheduling problem. *Journal of Scheduling*, 2(5):215–227, 1999.

[110] V. Subramani, R. Kettimuthu, S. Srinivasan, J. Johnston, and P. Sadayappan. Selective buddy allocation for scheduling parallel jobs on clusters. In *Proceedings of the International Conference on Cluster Computing*, pages 107–116. IEEE, 2002.

[111] http://insidehpc.com/2014/07/cray-wins-174-million-contract-trinity-supercomputer-based-knights-landing.

[112] T. Tzouramanis. History-independence: a fresh look at the case of R-trees. In *Proceedings of the Twenty-Seventy Annual ACM Symposium on Applied Computing (SAC)*, pages 7–12, 2012.

[113] J. S. Vitter. Random sampling with a reservoir. *ACM Transactions on Mathematical Software*, 11(1):37–57, 1985.

[114] J. S. Vitter. External memory algorithms and data structures: Dealing with massive data. *ACM Computing surveys (CsUR)*, 33(2):209–271, 2001.

[115] S. Wasson. `http://techreport.com/review/28294/amd-high-bandwidth-memory-explained`.

[116] T. Wilken, C. Lovis, A. Manescau, T. Steinmetz, L. Pasquini, G. L. Curto, T. W. Hänsch, R. Holzwarth, and T. Udem. High-precision calibration of spectrographs. *Monthly Notices of the Royal Astronomical Society: Letters*, 405(1):L16–L20, 2010.

[117] D. E. Willard. Maintaining dense sequential files in a dynamic environment. In *Proceedings of the Fourteenth Annual ACM Symposium on Theory of Computing (STOC)*, pages 114–121, 1982.

[118] D. E. Willard. Good worst-case algorithms for inserting and deleting records in dense sequential files. In *ACM SIGMOD Record*, volume 15:2, pages 251–260, 1986.

[119] D. E. Willard. A density control algorithm for doing insertions and deletions in a sequentially ordered file in a good worst-case time. *Information and Computation*, 97(2):150–204, 1992.

[120] D. H. Woo, N. H. Seong, D. L. Lewis, and H.-H. S. Lee. An optimized 3d-stacked memory architecture by exploiting excessive, high-density tsv bandwidth. In *HPCA-16 2010 The Sixteenth International Symposium on High-Performance Computer Architecture*, pages 1–12. IEEE, 2010.

[121] A. Wren and J. Rousseau. Bus driver scheduling-an overview. *Computer-Aided Transit Scheduling*, pages 173–187, 1995.

[122] D. W. Wyatt and H. T. Castrup. Managing calibration intervals. *National Conference of Standards Laboratories (NCSL) Annual Workshop and Symposium*, 1991.

[123] M. Yannakakis and F. Gavril. The maximum k-colorable subgraph problem for chordal graphs. *Information Processing Letters*, 24(2):133–137, 1987.

[124] F. Yao, A. Demers, and S. Shenker. A scheduling model for reduced CPU energy. In *Proceedings of the Thirty-Sixth Annual Symposium on Foundations of Computer Science (FOCS)*, pages 374–382. IEEE, 1995.

[125] G. Yu and G. Zhang. Scheduling with a minimum number of machines. *Operations Research Letters*, 37(2):97–101, 2009.

[126] G. Zhang and R. Hocken. Improving the accuracy of angle measurement in machine calibration. *CIRP Annals-Manufacturing Technology*, 35(1):369–372, 1986.

[127] Z. Zhang. A flexible new technique for camera calibration. *Pattern Analysis and Machine Intelligence, IEEE Transactions on*, 22(11):1330–1334, 2000.

[128] Z. Zhang. Method and system for calibrating digital cameras, Aug. 20 2002. US Patent 6,437,823.