Parallel Scheduling with Batching

A Dissertation
submitted to the Faculty of the
Graduate School of Arts and Sciences
of Georgetown University
in partial fulfillment of the requirements for the
degree of
Doctor of Philosophy
in Computer Science

By

Brendan Sheridan, B.S.

Washington, DC
November 17, 2017
Parallel Scheduling with Batching

Brendan Sheridan, B.S.

Dissertation Advisor: Jeremy T. Fineman

Abstract

This work provides and analyzes several parallel scheduling algorithms that utilize 
*batching* by delaying work until it can be executed as part of a larger batch. Batchig 
is often necessary for good performance in problems that have a high start-up cost 
to do any work or reduced cost for a large group of work. These problems are often 
challenging because the delay of work conflicts with standard scheduling constraints 
such as deadlines as well as useful scheduling objectives such as makespan and flow.

We will focus on two particular scheduling problems: dynamically multithreaded 
computations with implicit batching and Integrated Stockpile Evaluation (ISE). The 
former is an online scheduling problem with precedence constraints where the delay 
of data structure operations can be used to reduce total work and increase parallelism with batched operations. ISE is a traditional offline scheduling problem where 
n jobs, each with an arbitrary release time, must be scheduled non-preemptively on 
m machines. It has the extra constraint that machines may only be used if they have 
been recently calibrated and batching can be used to reduce the necessary number 
of calibrations. In both cases, competitive scheduling algorithms must appropriately 
balance the benefit of batching against the cost of delaying work.

Index words: Data structures, work stealing, scheduling, batching, batched data structure, implicit batching, Integrated Stockpile Evaluation, approximation algorithms, calibration, resource augmentation
# Table of Contents

List of Figures ........................................................................................................ vi

## CHAPTER

1 Introduction ........................................................................................................ 1

2 Background and Terminology ........................................................................... 4  
   2.1 Interval Job Scheduling ........................................................................... 4  
   2.2 DAG Scheduling ..................................................................................... 5  
   2.3 Competitive Analysis and Resource Augmentation ............................... 15

3 Minimizing Calibrations for Non-Unit Jobs ................................................... 16  
   3.1 Contributions .......................................................................................... 18  
   3.2 Related Work ......................................................................................... 20  
   3.3 Algorithm Overview .............................................................................. 22  
   3.4 Scheduling Long-Window Jobs ............................................................... 23  
   3.5 Strategy for Short-Window Jobs ............................................................ 42  
   3.6 Conclusions ............................................................................................ 49  
   3.7 Open Problems ...................................................................................... 50

4 Minimizing Calibrations and Flow-Time for Non-Unit Jobs ......................... 52  
   4.1 Contributions .......................................................................................... 54  
   4.2 Related Work ......................................................................................... 54  
   4.3 Single-Machine ISE-flow ....................................................................... 55  
   4.4 Multi-Machine ISE-flow ....................................................................... 79  
   4.5 Conclusions ............................................................................................ 81

5 Implicit Batching for Work-Stealing Schedulers ........................................... 82  
   5.1 Contributions .......................................................................................... 84  
   5.2 Related Work ......................................................................................... 86  
   5.3 Definitions and Analytic Model .............................................................. 90  
   5.4 Implicit Batching in BATCHER ............................................................. 93  
   5.5 The BATCHER Scheduler ..................................................................... 101  
   5.6 Analysis of BATCHER .......................................................................... 108  
   5.7 Experimental Evaluation ...................................................................... 119  
   5.8 Conclusions and Open Problems ........................................................... 122

6 Summary .......................................................................................................... 124
Bibliography .................................................. 126
### List of Figures

<table>
<thead>
<tr>
<th>Figure</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>2.1</td>
<td>Asynchronous shared memory</td>
<td>7</td>
</tr>
<tr>
<td>2.2</td>
<td>Series-parallel precedence DAG</td>
<td>11</td>
</tr>
<tr>
<td>3.1</td>
<td>ISE to TISE transformation</td>
<td>26</td>
</tr>
<tr>
<td>3.2</td>
<td>Calibration rounding</td>
<td>31</td>
</tr>
<tr>
<td>3.3</td>
<td>Fractional job assignments on rounded calibrations</td>
<td>34</td>
</tr>
<tr>
<td>4.1</td>
<td>Rescheduling fractional high-flow jobs</td>
<td>65</td>
</tr>
<tr>
<td>4.2</td>
<td>$\alpha$-Rounding conflict tree</td>
<td>73</td>
</tr>
<tr>
<td>5.1</td>
<td>Parallel counter updates</td>
<td>97</td>
</tr>
<tr>
<td>5.2</td>
<td>A batched-counter implementation</td>
<td>97</td>
</tr>
<tr>
<td>5.3</td>
<td>Scheduler-state transition rules invoked by workers with empty deques</td>
<td>104</td>
</tr>
<tr>
<td>5.4</td>
<td>Pseudo-code for launching a batch</td>
<td>106</td>
</tr>
<tr>
<td>5.5</td>
<td>BATCHER performance</td>
<td>122</td>
</tr>
</tbody>
</table>
We consider three parallel scheduling algorithms, selected because they are each well motivated, theoretically interesting, and encourage some form of *batching*. Generally speaking, batching refers to the organization of work into logical groups such that each group is executed together. However, what it means to execute work together is problem specific. In addition, each problem contains work released during the schedule, either by explicit release times or by online discovery. Batching therefore often requires delaying work until the entire batch is ready. These delays conflict with other time-sensitive scheduling constraints and objectives. The problems considered are all inherently $\mathcal{NP}$-hard, but the presented algorithms use clever partitioning of work and resource augmentation to balance the benefits of batching against other scheduling concerns, achieving provably good polynomial-time approximation bounds.

Chapter 3 studies the *integrated stockpile evaluation (ISE)* problem, designed for a maintenance program with the same name. ISE is a traditional offline multi-machine job scheduling problem with release times and deadlines. It is distinguished by a *calibration* requirement. A machine cannot be used unless it is calibrated and calibrations last a fixed amount of time. Calibrations are assumed to be very expensive and the goal is therefore to minimize the total number of calibrations. Batching for this problem involves grouping jobs together into a calibration. Naturally, large batches tend to reduce the necessary number of calibrations. However, delaying a job may conflict with its deadline.
ISE has been previously studied, but only for the unit-length jobs. This work, first published in SPAA 2015 by Fineman and Sheridan [34], studies the more general variable length case. It proves that ISE is a generalization of a classic interval scheduling problem where the goal is to minimize the number of machines and presents a good polynomial-time approximation algorithm building on this observation. More specifically, given a black-box interval scheduling algorithm that finds an $s$-speed $\alpha m$-machine solution to the interval scheduling problem, the proposed ISE algorithm finds an $O(\alpha)$-approximation for number of calibrations using $O(\alpha m)$ machines with $s$-speed augmentation. Machine minimization is a very well studied problem so it would be difficult to improve on this bound.

Chapter 4 considers another motivating application and accompanying objective for the ISE problem. Manufacturing settings often have similar calibration requirements and costs, but without hard deadlines for jobs. Rather, it is preferable to balance the cost of calibrations against the cost of delaying job completions. Specifically, we consider an objective that takes the linear combination of total flow-time (the sum of time between release and completion for each job) and total number of calibrations. As in Chapter 3, batching involves grouping jobs together into a calibration. However, the benefits of batching must be weighed directly against the increased flow caused by delaying jobs.

We again consider the unsolved variable length job case. For the single machine variant this work offers an $O(1)$-approximation algorithm using $O(1)$-machines and $O(1)$-speed augmentation. This matches the asymptotic bounds for non-preemptive interval schedule to minimize total flow time so it would be difficult to improve upon asymptotically.

In Chapter 5, we shift to the online problem of scheduling a dynamically multi-threaded computation on shared-memory identical machines. More specifically, we
focus on multi-threaded computations utilizing implicit batching directives for parallel
data-structures. This problem is well motivated because it is common but often chal-
lenging to maintain parallel data-structures in these programs. Typically, one employs
either a concurrent data-structure, which allows asynchronous access from multiple
threads, or a parallel batched data-structure, which processes data-structure opera-
tions in large batches and relies on the encompassing program to organize batches and
hide latency. Both of these approaches have potential downsides in this context. Con-
current data-structures often offer weak theoretical makespan guarantees and parallel
batched data-structures can negate many of the benefits of dynamic multi-threading
by forcing the programmer to anticipate runtime scheduling.

This work combines the best of both worlds by employing a novel *implicit batching*
scheduler. It adapts the standard work-stealing strategy by incorporating a language
construct to distinguish data-structure operations. This allows the scheduler to switch
between the main program and batches of data-structure operations as necessary. The
initial work, detailing the language construct, augmented scheduler, and associated
bounds was published in SPAA 2014 [1]. It proved that the expected makespan of the
implicit batching scheduler is $O((T_1 + W(n) + ns(n))/P + s(n)T_\infty)$ where $T_1$ is the
work of the program, $T_\infty$ is the span, $P$ is the number of processors, $W(n)$ is the total
work of all $n$ data-structure operations, and $s(n)$ is the span of a size-$P$ batch. Given
a high cost data structure like a search tree and appropriately large $n$, it reduces to
$O((T_1 + n \log n)/P + T_\infty \log n)$. This bound roughly matches the linear speedup guarantee
of a work-stealing scheduler without abandoning dynamic multi-threading principles
to manage the data-structure.
The majority of related work discussion is reserved for inclusion in the appropriate chapter. This chapter reviews general knowledge and establishes the terminology used throughout the rest of the dissertation.

Scheduling theory is, broadly speaking, the study of assigning scarce resources to tasks given some objective function. In parallel scheduling, the primary resource is typically “machines”. A\textit{ machine} should be thought of as the main resource dictating parallelism. That is, a task must be assigned to a machine to make progress and only one task can make progress on a machine at any one time. Depending on context, a machine may represent a CPU in a computer, a node in a cluster, a specialized tool, etc. Models often include additional constraints to further restrict valid assignments of tasks to machines as necessary.

“Task” and “job” are commonly interchanged across the various fields of parallel scheduling, referring simply to some notion of work. However, for ease of discussion, we use them to refer to two logically distinct contexts in this dissertation. In particular, we use “task” in the context of directed acyclic graph (DAG) scheduling and “job” in the context of more traditional interval scheduling.

2.1 \textbf{Interval Job Scheduling}

Interval job scheduling is the traditional algorithmic problem of assigning jobs to run on a machine at specified times. We primarily focus on the multi-machine case
where the scheduler is given some set of machines and may assign a job to run on any machine in the set. We define jobs conventionally.

**Definition 1.** A *job* $j$ is some abstract unit of work with release time $r_j$ and processing time $p_j$. It is valid to assign a job at some time $t$ if and only if $r_j \leq t$. A job assigned at time $t$ occupies the machine until time $t + p_j$.

Less formally: the release time is the absolute time after which a job is “ready” and can be assigned by the schedule to run on a machine, the deadline is the absolute time by which the job must be completed, and the processing time is the time interval required to complete a job after it begins running on a machine.

Variations of interval scheduling, such as the ISE problem we consider in Chapter 3, introduce additional constraints and objectives on the resulting schedule to reflect the intended real-world usage.

### 2.2 DAG Scheduling

Directed acyclic graph (DAG) scheduling is a problem that commonly emerges out of parallel program scheduling.

Basic operations, assumed to take unit time, are represented as nodes in the graph. Directed edges represent precedence constraints imposed by data flow and explicit parallel control statements. Scheduling the program entails a graph traversal problem where each node must be visited and assigned to some executing machine.

**Definition 2.** Given a DAG, $G = (V, E)$, we define tasks relative to nodes in the DAG. Intuitively, a *task*, $\tau_r$, should include all the self-contained work starting at node $r$. More formally, a task $\tau_r$ is the largest subgraph of $G$ s.t. all paths from nodes $u \notin \tau_r$ to nodes $v \in \tau_r$ pass through $r$. As discussed in Section 2.2.2, we will only be
considering series-parallel DAGs. Therefore, all our programs begin as a single task which diverges into various sub-tasks.

We can also allow for a dynamic program execution, i.e. one where the control flow is determined by values that vary at runtime, by assuming that the graph traversal is done online. That is, we assume that children of a node are only revealed after the node has been executed. This is consistent with branching determined by runtime variables.

Parallel algorithms are often evaluated with two important features of the resulting DAGs, work and span [28].

Definition 3. The work of a DAG, $T_1$, is simply the total number of operations represented, i.e. the number of nodes in the graph. It is equivalent to the time required to execute the program serially.

Definition 4. The span of a DAG, $T_\infty$, is the length of the longest path existing in the DAG. It is equivalent to the time required to execute the program with an infinite number of machines.

In order to more precisely define the ordering constraints represented by edges in the graph, it is helpful to review our motivating parallel programming model. Specifically, we focus on fork-join and loop-parallel programming designed for an asynchronous parallel random access memory (PRAM) model [38].

2.2.1 Asynchronous Shared-Memory Parallel Computing

In keeping with current commodity hardware, we focus on the use-case of multi-core processors. For programming purposes, these processors can be accurately thought of as integrated circuits housing two or more independent processing cores. As with
older single-core processors, each core has a small memory cache, but must read and write data to an external main-memory.

To reduce data-movement and increase efficiency, commodity hardware overwhelmingly employs **shared-memory** across the independent processing cores. That is, every core uses the same memory space as diagrammed in Figure 2.1 and each core may issue concurrent read or write instructions to the same memory address.

The correct way to model this shared-memory theoretically, particularly resolving concurrent instructions to the same address, is the topic of much debate. However, this work adopts an asynchronous PRAM model [38] that is commonly used in the context of dynamically multi-threaded programming platforms and other work-depth focused models [6, 14, 15, 16, 17, 56]. In the asynchronous PRAM model, each processor has its own local program, running in private memory, and executes asynchronously. A processor may issue one instruction per time-step: a global read/write, a local opera-
tion, or a synchronization step. Synchronization, described in action in Section 2.2.2, can be used to logically prevent race-conditions.

Using synchronization, we consider only deterministic algorithms. That is, if a task writes to a shared memory address, no other task may read or write to that address unless it is a descendent. In this way, we avoid most of the practical issues that arise from modelling concurrent write instructions. They are handled internally and efficiently by a scheduler with special primitives. For a justification of these primitives as well as an example implementation, consult the Cilk-5 implementation details [37].

We allow algorithms to concurrently read the same locations, but concurrent read instructions do not typically cause notable contention on commodity hardware. For simplicity, we will not explicitly analyze the cost of synchronization. It does not dominate in any of the discussed algorithms under the aforementioned asynchronous PRAM assumptions.

2.2.2 DYNAMICALLY MULTI-THREADED PARALLEL PROGRAMS

In dynamic multithreading, the programmer is only responsible for identifying concurrency, typically using a programming language construct. The runtime scheduler is responsible for spawning workers in the form of operating system threads, work assignment, data-movement, and coordination.

In keeping with popular platforms such as Cilk [37], Thread Building Blocks [67], and X10 [23], we further limit our consideration to loop parallelism and fork-join parallelism programming constructs. Consequently, the computation DAGs we will be considering are all series-parallel. Consider a DAG composed only of a source and a sink connected by a single edge; this is the series-parallel base case. A DAG is series-parallel if it can be built recursively from two series-parallel DAGs, $G_1$ and $G_2$, by (1) adding an edge from the sink of $G_1$ to the source of $G_2$ — series — or (2)
merging the source of $G_1$ with the source of $G_2$ as well as the sink of $G_1$ with the sink of $G_2$ — parallel. This property often simplifies analysis, but does not necessarily restrict generality. Arora et al. [6] show that there is no asymptotic difference in the runtime complexity of their work-stealing scheduler in the general case even for arbitrary computation DAGs.

Loop parallelism is typically specified with a keyword preceding the normal loop. For example, in Algorithm 2.1 there is a parallel keyword inserted before the standard for loop. The **parallel** keyword instructs the scheduler that the bodies of all $n$ iterations can logically be run in parallel. There is an implicit synchronization after all iterations have been computed, ensuring that no instructions following the loop are executed until all iterations are complete.

```
Algorithm 2.1 Simple example of loop parallelism.
parallel for $i \leftarrow 1$ to $n$ do
  something
```

Fork-join parallelism is specified with keywords identifying potential parallelism as well as a keyword to indicate required local synchronization. Algorithm 2.2 gives a naive parallelization of MergeSort (we will later see that the sequential Merge limits parallelism) often used to introduce recursive fork-join parallelism using the convention of **spawn** and **sync**. This is consistent with the standard terminology of Cormen et al. [28]. We will use these conventions throughout.

The **spawn** keyword instructs the scheduler that the following statement or function call may logically be run in parallel with the parent function. The **sync** keyword instructs the scheduler that all parallel execution within the current scope must complete before the function can proceed any further. It functions as a barrier but not an interrupt.
Returning to our MergeSort example, at each level of the recursion, the MergeSort calls operate on disjoint halves of the array so we can modify a traditional merge sort algorithm by simply adding a spawn prefix to one of the calls. This allows the scheduler to run both calls in parallel. However, the resulting two array halves must be sorted before we merge them so a synchronization is necessary before the Merge call.

**Algorithm 2.2** Simple example of MergeSort using recursive fork-join parallelism [28]

**Function** MergeSort(A[, index start, index end])

if end − start < 2 then
    return;

index middle = ⌊(start + end)/2⌋

spawn MergeSort(A, start, middle)
    MergeSort(A, middle + 1, end)

sync

Merge(A, start, middle, end)

return A

**Function** Merge(A[, index start, index middle, index end])

nleft = middle − start + 1
nright = end − middle
L[nleft + 1], R[nright + 1]

for i = 1 to nleft do
    L[i] = A[start + i − 1]

for i = 1 to nright do
    R[i] = A[middle + i]

L[nleft + 1] = ∞
R[nright + 1] = ∞
i = 1

for k = start to end do
    if L[i] ≤ R[j] then
        A[k] = L[i ++]
    else
        A[k] = R[j ++]

return A

If we assume for simplicity that N is a power of 2, the DAG resulting from Algorithm 2.2 is given in Figure 2.2.2. The parallel calls to MergeSort recursively branch
out in tree fashion over a depth of $\log_2(N)$ until there are $N$ children size 1. Subsequently, all the children merge back together in the same pattern with synchronizations followed by calls to Merge until the entire array is sorted.

Because Algorithm 2.2 has not introduced any additional computation, its work is simply the runtime complexity of traditional merge sort. Using the master theorem, $T_1(n) = 2T_1(n/2) + O(n) \implies T_1(n) = O(n \log n)$. Due to the simple
recursive nature of the parallelism, we can similarly analyze the span by noting that we need only consider the most expensive sub-problem (consistent with the idea that we can utilize an infinite number of machines to run the sub-problems in parallel). The result is dropping the coefficient from the recurrence relation, i.e. $T_\infty(n) = T_\infty(n/2) + O(n) \implies T_\infty(n) = O(n)$. Alternatively, it should be easy to see from Figure 2.2.2 that the longest path through the DAG is the depth of the initial recursion branching, $\log_2(n)$, plus the sum of a sequential merge at each level of the merging phase, $\sum_{i=1}^{\log_2 n} 2^i$, and $\log_2(n) + \sum_{i=1}^{\log_2 n} 2^i = O(n)$.

Given the work, $T_1(n) = O(n \log(n))$, and the span, $T_\infty(n) = O(n)$, we can also talk about the **parallelism** of the algorithm. In this case, $T_1(n)/T_\infty(n) = O(\log(n))$ so we say the parallelism is $O(\log(n))$. Ideally, we want parallelism to be large because it indicates how much speed-up you can obtain by running the algorithm on an appropriate number of machines. The parallelism in this case is relatively small owing to the serial bottleneck in the Merge function. Less naive merge sort algorithms increase the parallelism by parallelizing the Merge function. For example, Cormen et al. [28] use binary search on the sorted sub-arrays to determine the necessary indices for a recursive merge.

### 2.2.3 Greedy Schedulers

Work and span are useful metrics for gauging the effectiveness of an algorithm in the extreme single and infinity machine cases, but to model runtime performance in the more practical $P$-machine case (where $P$ is some constant > 1 decided at runtime) we must better define how the DAG will actually be scheduled at runtime, i.e. the **execution schedule**. We are primarily interested in the makespan, $T_p$, of the execution schedule. In addition, because our goal is dynamic multi-threading, we focus on the online variant of the execution scheduling problem. That is, we assume that
the sub-tasks of any particular task are unknown until the preceding computations have been scheduled.

Based on this model, we can immediately identify some lower bounds for an optimal makespan, $T_{\text{opt}}$. Specifically, we can say $T_{\text{opt}} > T_\infty$ because the schedule must respect the precedence constraints in the DAG. We can also say that $T_{\text{opt}} > T_1/P$ because our model does not include any features that would allow super-linear speedup.

Drawing on results from Brent [19] and Graham [41], we can also establish a fairly tight upper-bound. Specifically, Brent showed that there always exists a schedule $T_p \leq T_1/P + T_\infty$. As the sum of the aforementioned lower-bounds, this upper-bound is tight to within a factor of 2. It is also possible to achieve this bound using a greedy strategy. A greedy schedule is any schedule in which a machine is never idle while there is work available.

It is relatively trivial to prove that any centralized greedy schedule achieves the upper bound. This is typically called the Greedy-Scheduling Theorem. [6, 17]

**Theorem 2.1.** Given a computation DAG with $T_1$ work and $T_\infty$ span, as well as a greedy execution schedule on $P$ machines, the makespan of the execution schedule is $T_p \leq T_1/P + T_\infty$.

**Proof.** Consider two buckets, a work bucket and an idle bucket. For each time-step in the execution schedule, assume that each machine increments the work bucket (if it is executing an instruction) or the idle bucket (if it is idle). At the end of the computation, the work bucket must be exactly $T_1$ because that is the number of instructions in the program. The idle bucket must be $\leq T_\infty P$ because every time-step with one or more idle machines reduces the maximum path length in the unexecuted DAG by one (any critical path through the DAG must have a node executed at every
time-step unless all machines are busy by definition of a greedy schedule). Combined, the bucket total is $\leq T_1 + T_\infty P$ and every time-step contributed $P$ so the makespan must be $\leq T_1/P + T_\infty$. 

\[ \square \]

2.2.4 Work-Stealing Schedulers

Greedy schedulers offer very useful features. However, the asynchronous shared-memory computing model does not facilitate centralized scheduling without significant overhead. **Work-stealing scheduling algorithms** [17, 42] are one way to extend greedy scheduling strategies into a distributed context.

In standard work-stealing algorithms like the Cilk scheduler, each machine maintains a personal doubly ended queue (deque) of tasks that are ready to be executed. At each step, each machine places any newly revealed sub-tasks (from the previous step) onto the *bottom* of its deque and tries to pop a task from the *bottom* of its deque to begin work. If a machine’s deque is empty, that machine instead tries to **steal** work from another machine. The stealing machine is called a **thief**. It chooses another machine at random, the **victim**, and tries to pop a task from the *top* of that machine’s deque. Data structure and synchronization details vary based on implementation, but this core concept has proven extremely versatile and efficient for solving the online decentralized execution scheduling problem.

Efficient work-stealing schedulers (e.g., [17]) can execute a program in $T_1/P + O(T_\infty)$ (expected) time on $P$ machines. This running time is asymptotically optimal and guarantees linear speedup to programs with sufficient parallelism.
2.3 Competitive Analysis and Resource Augmentation

Competitive analysis is a simple method originally designed by Sleator and Tarjan [71] to gauge the usefulness of online algorithms. In many cases, it is impossible for an online algorithm to exactly match the quality of an optimal offline solution. However, it is helpful to determine the maximum ratio by which the two might differ on any given input, i.e. the competitive ratio. More formally given a problem with optimal solution $OPT$, we can say that an algorithm $A$ is $c$-competitive if, for any given input $I$,

$$A(I) \leq c \cdot OPT(I)$$

Similarly, we will be using approximation bounds in offline contexts to analyze the quality of polynomial-time solutions to $\mathcal{NP}$-hard problems. In many cases, competitive bounds and approximation bounds are overly pessimistic. In these cases, it is often helpful to combine them with resource augmentation.

Resource augmentation was first introduced by Kalyanasundaram and Pruhs [50]. It assumes that the algorithm is given more or faster machines than the optimal solution against which it is compared. This method was originally introduced as a technique for producing online algorithms, but has become commonplace in recent offline scheduling literature.

More precisely, define resource augmentation as follows. Let $OPT(I)$ denote the best possible value of the objective function over feasible $m$-machine schedules of instance $I$. Adopting terminology introduced by Phillips et al. [63], we say that an algorithm is a $w$-machine $s$-speed $\rho$-approximation algorithm if it always achieves a feasible schedule with value at most $\rho OPT(I)$ given $wm$ machines each operating $s$ times faster.
Chapter 3

Minimizing Calibrations for Non-Unit Jobs

The first selected problem focuses on an interesting generalization of the interval scheduling problem that requires batching of jobs into calibrated time segments. The Integrated Stockpile Evaluation (ISE) problem is an offline multi-machine scheduling problem, recently introduced by Bender et al. [13]. What distinguishes the ISE problem from more standard interval scheduling problems is that a machine is unusable unless it undergoes a calibration, and jobs may only be scheduled on a machine if that machine has been calibrated recently. Specifically, if a calibration is performed on a machine at time $t$, then the machine remains usable or calibrated for the interval $[t, t+T)$, for fixed $T$. Each machine may be calibrated multiple times, but the machine must remain idle between its calibrated intervals. Calibrations are instantaneous, so it is feasible to calibrate the machine at times $0$, $T$, $2T$, and so on. Although calibrations have no temporal cost, they are considered the expensive feature of a solution.

More precisely, the ISE problem (denoted $P|r_j, d_j|\#\text{calibrations}$ in standard scheduling notation [40]) is defined as follows [13]. The input consists of a set $J$ of $n$ jobs, an integer number $m$ of identical machines, and an integer $T \geq 2$ specifying a calibration length. Each job $j$ has a processing time $p_j \leq T$, a release time $r_j$, and a deadline $d_j \geq r_j + p_j$. A schedule is feasible if it schedules jobs nonpreemptively

\footnote{Nonpreemptive means that when job $j$ is scheduled on a machine, it must be scheduled for $p_j$ consecutive timesteps.}
on machines such that 1) every job completes before its deadline, and 2) every job is
scheduled without preemption completely within a single calibrated interval. The goal
is to find a feasible schedule that minimizes the number of calibrations performed.

The ISE problem formalizes scheduling issues that arise as part of a nuclear-
weapons-testing program with the same name at Sandia National Laboratories [21].
The high-level goal is to perform tests on a set of nuclear weapons to verify their
integrity, with the constraint that the testing devices be frequently re-calibrated to
guarantee accurate results. See [13, 21] for more details. Clearly, this is an inter-
esting scheduling problem with practical applications, but it is included in this work
because of a limitation in the original formulation and associated solution. Specifi-
cally, Bender et al. [13] only offer a solution for unit-length processing times. That is,
for all jobs $j$, $p_j = 1$. This limitation is obviously undesirable for practical applications
featuring varying job types that require varying time to complete. Moreover, there
is no straightforward way to generalize the unit-length solution for use with variable
length processing times.

This chapter solves the ISE problem for non-unit processing times, which Bender
et al. [13] leave as an open problem. It is not hard to see that testing whether a feasible
schedule exists is NP-hard by a reduction from Partition. (Use $m = 2$ machines, and
assign all jobs the same release time $r_j = 0$ and deadline $d_j = T$.) Thus ignoring the
goal of minimizing calibrations, obtaining a polynomial-time algorithm that finds any
feasible schedule necessitates resource augmentation.

To understand how good a solution we can expect for the ISE problem, observe
that the ISE problem extends the classic machine-minimization (MM) problem: given
a set of jobs with release times, deadlines, and processing times, find the minimum
number of machines necessary to schedule all jobs by their deadlines. Specifically,
given an instance to MM, construct an ISE instance by setting $T = \max_j \{d_j\} -$
min_j \{r_j\}. A \(w\)-machine \(s\)-speed solution to the ISE problem (with any approximation quality) would yield an \(s\)-speed \(w\)-approximation to the MM problem. Thus if the best \(s\)-speed approximation algorithm to the MM problem uses \(\alpha\) times the optimum number of machines, then the ISE problem requires \(\alpha\)-machine augmentation when limited to \(s\)-speed augmentation. Similarly, because the number of machines and the number of calibrations are identical in this construction, the best \(s\)-speed \(\alpha\)-machine approximation we can expect for the ISE problem has an approximation ratio \(\rho \geq \alpha\).

To understand the solution quality possible without relaxing the unit job-length constraint, consider that to ensure feasibility a non-unit ISE instance would require a pre-processing step where all of the job lengths are made uniform. An adversary can provide an instance with one job length \(T\) and the remaining jobs unit-length, forcing the pre-processing step to round all job lengths to \(T\). A unit-ISE algorithm obviously cannot hope to do better than an \(O(T)\)-approximation in this case, or \(O(\Delta)\) where \(\Delta\) is the maximize ratio of job sizes.

### 3.1 Contributions

The main contribution of this chapter is an algorithm that uses any MM algorithm as a black box, with only a constant factor overhead in terms of machine augmentation. Specifically, given an \(s\)-speed \(\alpha\)-approximation algorithm for the MM problem, the algorithm results in an \(O(\alpha)\)-machine \(s\)-speed \(O(\alpha)\)-approximation to the ISE problem. As explained above, this is the best one can expect to within constant factors.

To understand this result concretely, let us consider the current best approximation algorithms for the MM problem. Chuzhoy et al. [26] give an \(O(\text{OPT})\)-approximation for the MM problem, meaning that the solution uses \(O(\text{OPT}^2)\)
machines. If $\text{OPT} = O(1)$, then their algorithm is a 1-speed $O(1)$-approximation algorithm. More generally, combining their solution with Raghavan and Thompson’s previous best $O(\log n / \log \log n)$-approximation \cite{65} yields an $O(\sqrt{\log n / \log \log n})$-approximation for the MM problem for arbitrary $\text{OPT}$. Both of these results use no speed augmentation and apply to the general case.

In more recent work, Bansal et al. \cite{7} study the case that $\text{OPT} = 1$, giving an $O(1)$-speed 1-approximation (i.e., a one-machine solution) for this special case. Unfortunately, it is not clear how to generalize this result past 1 machine. Im et al. \cite{48} give an algorithm guaranteeing either a $(1 + \epsilon)$-speed 2-approximation or a $(2 + \epsilon)$-speed 1-approximation for the MM problem for any $\text{OPT}$. But unlike the preceding results, their algorithm runs in quasi-polynomial time, i.e., $O(n^{O(\log^c n)})$ for some $c > 0$, not polynomial time.

Combining the algorithm with the above MM algorithms, yields the following concrete results:

- A polynomial-time $O(\sqrt{\log n / \log \log n})$-machine 1-speed algorithm giving an $O(\sqrt{\log n / \log \log n})$ approximation for the ISE problem. That is, given an ISE instance that is feasible using $m$ machines and $C$ calibrations, the algorithm produces a solution using $O(C \sqrt{\log n / \log \log n})$ calibrations on $O(m \sqrt{\log n / \log \log n})$ machines.
- Whenever the input instance is feasible on $O(1)$ machines, it yields a polynomial-time $O(1)$-machine 1-speed $O(1)$-approximation for the ISE problem. More generally, if the instance is feasible on $m$ machines, it yields a 1-speed $O(m)$-machine $O(m)$-approximation, which is better than the first bound if $m = o(\sqrt{\log n / \log \log n})$.
- A quasi-polynomial-time $O(1)$-machine $O(1)$-speed $O(1)$-approximation for the ISE problem.
3.2 Related Work

Bender et al. [13] give the first algorithms for the ISE problem. They study the restricted case that for all jobs $j$, $p_j = 1$, denoted $(P|r_j,d_j,p_j = 1|\#\text{calibrations})$. Even in this special case of unit processing times, the problem is nontrivial. Bender et al. [13] give two greedy scheduling algorithms. Their first algorithm guarantees an optimal schedule (i.e., minimizing the number of calibrations) whenever a 1-machine schedule is feasible. Their second algorithm gives a 2-approximation for the multi-machine case.

Angel et al. [5] study several variations of ISE on a single machine using primarily dynamic programming. For the single machine non-unit case with pre-emption $(1|r_j,d_j,pmtn|\#\text{calibrations})$, they claim an optimal quadratic-time solution. For the non-unit case having variable length calibrations with variable cost $(1|r_j,d_j,pmtn,\{\ell_1,\ldots,\ell_K\}\text{cost(calibrations)})$, they show the problem is $\mathcal{NP}$-Hard. For the unit case with a calibration delay (that is, the machine is not usable until a fixed number of time-steps after the calibration starts $- 1|r_j,d_j,pmtn,\lambda + \{\ell_1,\ldots,\ell_K\}\text{cost(calibrations)})$, they give an optimal polynomial-time algorithm.

Most recently, McCauley [57] and Chau et al. [24] revisited the unit processing time ISE problem in an online setting. That is, jobs are not known ahead of time. Instead, they are revealed to the scheduler at their release time, $r_j$. Strict deadlines are not feasible in an online model so they modified the objective to include a linear combination of calibrations and weighted total flow-time $- G(\#\text{ calibrations}) + \sum_{j \in J}(s_j + p_j - r_j)w_j$, where $w_j$ is the weight of job $j$, $s_j$ is the time it is started, $p_j$ is the processing time ($p_j = 1$ for the unit case), and $G$ is some fixed cost for each calibration. Like the original work, they use greedy strategies designed to delay calibrations until enough jobs are ready. For the unweighted (i.e. $w_j = 1$ for all $j$) single-machine case,
they give a 3-competitive algorithm. They give a 12-competitive algorithms for both
the weighted single-machine case and the unweighted multi-machine case.

Beyond its practical applications, ISE is an interesting interval scheduling vari-
ation because it is often optimal to delay the scheduling of a job. This property is
unusual in more standard metrics like machine minimization and throughput maxi-
mization. However, it is certainly not unique.

Interval scheduling for power minimization is a popular [4] and ostensibly similar
problem when the goal can be reduced to minimizing idle periods for a continuous
interval schedule. Like ISE, this makes starting work on a machine expensive, which
tends to discourage scheduling a job as early as possible and reward job clustering.
It should be noted however that since calibrations last a discrete amount of time, the
problems are subtly different. Baptiste et al. [9, 10] give an $O(n^5)$-time dynamic pro-
gramming based algorithm for finding an optimal solution on a single processor with
preemption, reducing to $O(n^4)$-time in the unit-job case. Demaine et al. [30] extend
their work to a multi-processor environment, yielding a polynomial-time optimal algo-

Chang et al. [22] propose a model for minimizing active processor time which is
deceptively similar to ISE. Instead of calibrations of length $T$, they consider timesteps
of depth $B$. That is, up to $B$ jobs can be scheduled in the same timestep at no
additional cost and the goal is to minimize the active number of time-steps. However,
they consider only preemptive scheduling and do not offer an approximation for the
$B > 2$, NP-complete version.

Mäcker et al. [55] propose a cloud scheduling model that similarly encourages
delays and grouping. They assume that starting work on a new machine carries
upfront delay and cost. Consequently their problem formulation similarly reduces
to machine minimization in certain causes. However, their solution focuses on minimizing tardiness instead of allowing machine augmentation.

3.3 Algorithm Overview

This section gives a top-level algorithm for the ISE problem. The high-level algorithm is simple: partition the input jobs $J$ into two subsets $J_{\text{long}}$ and $J_{\text{short}}$, each containing the jobs with “long” and “short” windows respectively; then schedule those jobs independently, on disjoint machines, using the specialized algorithms described in Sections 3.4 and 3.5. The partitioning itself is trivial, and this process at most doubles the number of calibrations and machines beyond either of the algorithms.

More precisely, we define long and short jobs as follows. Note that the definition of long and short is based on the job’s window (release time and deadline), not its processing time.

**Definition 5.** We say that job $j$ is **long** or a **long-window job** if $d_j - r_j \geq 2T$. We say that a job $j$ is **short** or a **short-window job** if $d_j - r_j < 2T$.

The algorithm for long-window jobs uses an integer-program relaxation followed by a greedy rounding procedure (Section 3.4). Assuming all jobs have long windows, the algorithm yields an $O(1)$-machine 1-speed $O(1)$-approximation to the ISE problem. The algorithm can also be modified to trade more speed for fewer machines, giving a 1-machine $O(1)$-speed $O(1)$-approximation. These bounds are better than those stated in Section 3.1 with regard to the approximation ratio on number of calibrations and the amount of machine-augmentation employed. The higher overheads arise in scheduling short-window jobs (Section 3.5), where an MM algorithm is applied as a black box. The short-window algorithm also increases the number of machines and
the approximation ratio by a constant factor. Unfortunately, for small-window jobs
(as for the MM problem in general), it is not clear how to trade speed for machines.

Combining all of these results yields the main theorem, whose proof follows directly
from Theorems 3.12 and 3.20.

**Theorem 3.1.** Suppose there is an $s$-speed $O(\alpha)$-approximation algorithm $A$ for the
MM problem. Then using $A$ as a black box, our algorithm is an $O(\alpha)$-machine $s$-
speed $O(\alpha)$-approximation for the ISE problem. Moreover, the running time of our
algorithm is a polynomial in the length of the input (i.e., polynomial in $n$ and the
precision of other numbers), multiplied by the running time of $A$.

### 3.4 Scheduling Long-Window Jobs

This section focuses on the special case of ISE where all jobs have long windows. For
this special case, the algorithm yields a 1-speed $O(1)$-machine $O(1)$-approximation.
This solution can be transformed into an $O(1)$-speed 1-machine $O(1)$-approximation.
Intuitively, long jobs are easier to cope with than short jobs because they have more
options on where to be scheduled, but capturing this intuition is not trivial.

This section builds on one one key insight: introducing an extra restriction to the
long-window ISE problem makes it easier to solve, without significantly compromising
the quality of solution. Specifically, we will focus on the **trimmed ISE (TISE)**
problem. The TISE problem is exactly the same as the ISE problem, except that
there is one additional restriction on the schedule: a job may be scheduled only inside
a calibration that falls completely within the job’s window. Said differently, consider
a calibration starting at time $t$, spanning the time interval $[t, t+T)$. Job $j$, with
window $[r_j, d_j)$, may only be scheduled in this calibration if $r_j \leq t \leq d_j - T$. We will
refer to this extra restriction as the **TISE restriction** or **TISE constraint**. Note
that the TISE constraint is specific to long jobs because jobs with windows shorter than $T$ are infeasible in the TISE problem.

The main advantage of the TISE problem is that whenever a job is scheduled within a particular calibration, it would be feasible to schedule the job anywhere within that calibration, which allows flexibility in the schedule. Moreover, because of this flexibility, given an assignment of jobs to calibrations one can infer a schedule. We thus need only focus on 1) finding a schedule of calibrations, and 2) finding an assignment of jobs to those calibrations.

The bulk of this section gives an algorithm for the TISE problem, which includes several steps. First, the algorithm constructs a linear-programming (LP) relaxation of the TISE problem. The LP allows for both fractional calibrations and fractional job assignments. Then, it performs a greedy rounding step that yields an integer calibration schedule such that a fractional assignment of jobs to calibrations remains feasible. This rounding step increases the number of calibrations and machines by a constant factor. Note that fractional job assignments correspond to a preemptive schedule, whereas integer job assignments correspond to a nonpreemptive schedule. Finally, it converts the preemptive schedule to a nonpreemptive schedule, using a constant factor extra machines and calibrations, through a variant of earliest deadline first (EDF) scheduling. Earliest deadline first does not generally work for nonpreemptive scheduling with arbitrary release times, deadlines, and processing times, but the TISE restriction makes this possible.

3.4.1 A TISE solution is good enough

Because the TISE problem is more restricted, any valid TISE schedule is also a valid ISE schedule. The question is what happens to the quality of the solution. The following lemma argues that the TISE solution is as good, to within constant
factors. Specifically, the optimal TISE solution uses at most three times the number of machines and calibrations as the optimal ISE solution. It thus suffices to solve a TISE problem on \( m' = 3m \) machines.

The proof of the following lemma leverages the definition of long-window jobs. In particular, in order for the presented construction to apply, the threshold for being long must be at least \( 2T \). This proof is the reason for the choice of constant in Definition 5. (Making the threshold larger is okay, but that would weaken the bounds for short-window jobs.)

**Lemma 3.2.** Consider any long-window ISE instance. Suppose that there exists a feasible ISE schedule using \( m \) machines and \( C \) calibrations. Then there exists a feasible TISE schedule using at most \( m' = 3m \) machines and \( 3C \) calibrations.

**Proof.** The proof is by construction, which is illustrated by Figure 3.1. Consider any machine \( i \) in the ISE schedule. The TISE schedule uses three machines, denoted \( i' \), \( i^+ \), and \( i^- \). For a calibration starting at time \( t \) on machine \( i \) in the ISE schedule, create three calibrations in the TISE schedule: a calibration on machine \( i' \) at time \( t \), a calibration on machine \( i^+ \) at time \( t + T \), and a calibration on machine \( i^- \) at time \( t - T \). Because the calibrations on machine \( i^+ \) and \( i^- \) are \( T \)-step translations of the calibrations on \( i \), the calibrations themselves are feasible.

The next step is to transform the schedule of jobs. Consider each job \( j \) in the ISE schedule. Let \( x_j \) be the job’s start time in the ISE schedule. Let \( i \) be the machine on which the job is scheduled. Let \( t_j \) be the start time of the calibration containing the job, i.e., such that \( x_j \in [t_j, t_j + T) \). If \( r_j \leq t_j \leq d_j - T \), i.e., the job is already feasibly scheduled with regards to the TISE restriction, then schedule the job at time \( x_j \) on machine \( i' \). If \( r_j > t_j \), then delay the job, scheduling it at time \( x_j + T \) on machine \( i^+ \). If \( d_j < t_j + T \), then advance the job, scheduling it at time \( x_j - T \) on machine \( i^- \).
Figure 3.1: ISE to TISE transformation. Example of the transformation from a feasible ISE schedule on one machine to a feasible TISE schedule on 3 machines with $3 \times$ the calibrations, as described in proof of Lemma 3.2. (A) indicates the job windows for the relevant long-window jobs, with the endpoints of line $j$ corresponding to $r_j$ and $d_j$. (B) shows the original feasible schedule on machine $i$, and (C) shows the constructed TISE schedule given by the proof, where the buckets are calibrations and the shaded rectangles are jobs—the width of rectangle $j$ is its processing time $p_j$. Jobs 1 and 5 are moved to the advanced calibrations on $i^-$ because their deadlines fall within the original calibration. Similarly, job 7 is moved to the delayed calibrations on $i^+$ because its release time falls within the original calibration.

It is necessary to show that delaying or advancing a job enforces the TISE restriction. Consider a delayed job $j$, i.e., one with $r_j > t_j$ in the ISE schedule. Because the job is long, we have $d_j \geq r_j + 2T > t_j + 2T$. Thus the calibration $[t_j + T, t_j + 2T)$ is contained fully within $j$'s window. Moreover, the job must complete by time $t_j + T$. 

26
in the ISE schedule, so it must also complete by time \( t_j + 2T \) in the TISE schedule. A similar argument applies to advanced jobs.

Finally, because the ISE schedule is a nonpreemptive schedule such that no two jobs run at the same time on the same machine, and each of the machines \( i', i^+, \) and \( i^- \) only receives a subset of jobs from the ISE schedule on \( i \) (all translated by \( 0, +T, \) or \( -T \) timesteps), the TISE schedule is also a valid nonpreemptive schedule. \( \square \)

### 3.4.2 Polynomially many calibration points suffice

When constructing scheduling linear programs (LPs), it is common practice to use variables indexed by each timestep and then argue after the fact that the LP can be transformed to one where the number of variables is polynomial in \( n \). This approach, however, may require that time be discrete, whereas the TISE problem statement does not require that release times, processing times, or deadlines be integers. We thus determine what the important times are up front before constructing the LP.

The following lemma states that there are only \( n^2 \) times that matter.

**Lemma 3.3.** There exists an optimal solution to the TISE problem such that the following holds for every calibration on every machine \( i \). If a calibration is made at time \( t \) on machine \( i \), then either \( t \) is equal to the release time of a job, or the calibration immediately follows the preceding calibration on that machine (i.e., there is a calibration at time \( t - T \) on machine \( i \)).

**Proof.** Consider an optimal schedule for the TISE problem. It’s possible to iteratively transform it to one that obeys the lemma as follows. Consider the calibrations on each machine in increasing order of time. If the \( k \)th calibration does not obey the lemma, then decrease its start time (and the corresponding start time of any jobs therein) until the calibration’s start time hits a release time or the end of the \((k - 1)\)th calibration,
whichever comes first. Because the calibration is not moved past any release times, all jobs in the calibration can be advanced without sacrificing feasibility.

Because an optimal solution does not use any empty calibrations, the lemma implies that there are at most $n^2$ possible calibrations on each machine. Specifically, there may be a calibration at any release time. There may also be calibrations packed immediately after this one, but there can only be $n$ such calibrations.

Define $\mathcal{T} = \{r_j + kT | j \in J, k \in \{0, 1, 2, \ldots, n\}\}$ as the set of potential calibration points.

3.4.3 A linear program for the TISE problem

The goal of the following LP is to determine a schedule of calibrations on machines and an assignment of jobs to those calibration points. (Recall that for the TISE problem, a full schedule can be inferred by an assignment of jobs to calibrations because the jobs can, by definition, be scheduled in any order.) An integer solution to the LP
corresponds to a feasible TISE schedule. The algorithm will start from a fractional solution and round it.

Before the LP is formally stated, consider two simplifying ideas. It should be clear that both of the simplifications can only improve the value of the optimal solution because feasible TISE schedules can be trivially transformed into LP solutions.\(^2\) First, the LP ignores the mapping of calibrations to machines, instead only requiring that at most \(m'\) calibrations overlap at any time. Second, the LP groups calibrations by time, ignoring how jobs are partitioned across same-time calibrations.

Leveraging these simplifications, the LP has two types of variables. The variable \(C_t\) denotes the number of calibrations made at time \(t\). The variable \(X_{jt}\) indicates whether (or how much of) job \(j\) is assigned to the calibrations at time \(t\). In both cases, following from Lemma 3.3, it uses the restriction that \(t \in T\) be one of the potential calibration points.

This results in the following LP relaxation of the TISE problem:

The first constraint guarantees that there are not more than \(m'\) calibrations at any timestep. The second constraint says that each job can only be assigned to each calibration once (or more accurately, the fraction of a job assigned to a calibration point cannot exceed the fraction of calibrations performed at that point). The third constraint enforces that the total work assigned to a calibration point (i.e., the fraction of jobs times their processing time) be at most the total processing power of the calibration point (i.e., the number of calibrations times \(T\)). The fourth constraint requires that every job be scheduled completely. The fifth constraint enforces the TISE restriction, that jobs only be assigned to calibrations that are contained in their

\(^2\)In fact, one can argue that, for the fractional solution, the value of the optimal solution is unchanged. For an integer solution, however, it may not be feasible to produce a TISE schedule from the integer solution.
Algorithm 3.2 LP-formulation for minimizing calibrations

\[
\text{minimize } \sum_{t \in \mathcal{T}} C_t \\
\text{subject to } \sum_{t' \in \mathcal{T}, t-T<t' \leq t} C_{t'} \leq m' \quad \forall t \in \mathcal{T} \tag{1}
\]
\[
X_{jt} \leq C_t \quad \forall j \in J_{\text{long}}, t \in \mathcal{T} \tag{2}
\]
\[
\sum_j X_{jt} p_j \leq C_t T \quad \forall t \in \mathcal{T} \tag{3}
\]
\[
\sum_{t \in \mathcal{T}} X_{jt} = 1 \quad \forall j \in J_{\text{long}} \tag{4}
\]
\[
X_{jt} = 0 \quad \forall j \in J, t \in \mathcal{T} \text{ s.t. } t < r_j \text{ or } t+T > d_j \tag{5}
\]
\[
X_{jt}, C_t \geq 0 \quad \forall j \in J_{\text{long}}, t \in \mathcal{T} \tag{6}
\]

windows. Finally, the last constraint is a non-negativity constraint on job assignments and calibrations.

3.4.4 The TISE algorithm

As noted previously, the TISE algorithm has three steps. The first is solving the LP relaxation for \( m' = 3m \) machines. The LP solution could have both fractional calibrations \( C_t \) and fractional job assignments \( X_{jt} \) at this point. Second, a simple greedy-rounding algorithm, given by Algorithm 3.1, is applied to produce an integer calibration schedule on \( 3m' \) machines. The rounding algorithm scans calibrations \( C_t \) in order of time, keeping a running total. Whenever the total reaches the next multiple of \( 1/2 \), the algorithm creates 1 new calibration at that time. Figure 3.2 shows an example of this process. The resulting calibrations are assigned to machines in round-robin fashion. Let \( \mathcal{C} \) to denote the schedule of calibrations produced by the
rounding step. The third and final step is to assign jobs to calibrations and it is given by Algorithm 3.3. It begins by double the calibration schedule $C$ using twice as many machines (for $6m'$ in total). Then it can scan the calibrations in non-decreasing time order, and assign jobs using earliest-deadline-first scheduling. More precisely, it chooses a job with earliest deadline from those unscheduled jobs obeying the TISE constraint, with ties broken arbitrarily. If there is still room in the calibration, it schedules the job. Otherwise, it finishes the calibration and moves on to the next one.

Note that the rounding step discards any of the job assignments $X_{jt}$, so it should not be obvious that the algorithm schedules all jobs. The following proofs will show
Algorithm 3.3 Assign jobs $J_{\text{long}}$ given calibration schedule $C$

mirror the calibration schedule $C$ on twice as many machines
let $C'$ be the resulting calibration schedule

foreach calibration in $C'$ in nondecreasing order of time do
    let $t$ be the start time of the calibration
    $used = 0$;  // work in calibration
    let $J' = \{ j \in J_{\text{long}} | j$ unscheduled and $r_j \leq t \leq d_j - T \}$

    let $j \in J'$ be a job with earliest deadline
    while $j \neq \text{null}$ and $p_j + used \leq T$ do
        schedule job $j$ at time $t + used$ in the calibration
        $used = used + p_j$
        remove $j$ from $J'$
    let $j \in J'$ be a job with earliest deadline

that a fractional assignment of jobs to calibrations is still feasible after the rounding step. Intuitively, this assignment shows that a preemptive schedule is possible on those calibrations. The final EDF step transforms the preemptive schedule into a nonpreemptive one. This transformation does not work in general with EDF scheduling, so it should not be obvious that it works here.

3.4.5 Correctness and performance analysis

We begin by focusing on Algorithm 3.1. Showing correctness requires two conditions. First, that the rounded calibrations are valid (i.e., calibrations do not overlap on a machine). Second, that the calibration schedule has a feasible, fractional job assignment. Given those, we can conclude that the rounding step has only a constant-factor overhead.
Algorithm 3.4: Augmented calibration-rounding procedure used only for the proof of Lemma 3.5 and Corollary 3.6

```plaintext
carryover = 0 ; // carried calibration fraction
set y_j = 0 for all j ; // carried job fractions
C = ∅ ; // new calibration schedule
 foreach t ∈ T in increasing order do
    while carryover + C_t ≥ 1/2 do
        create a calibration at time t in C
        frac = \frac{1/2 - carryover}{C_t} ;
        carryover = carryover + frac \cdot C_t
    foreach job j do
        y_j = y_j + frac \cdot X_{jt}
        X_{jt} = X_{jt} - frac \cdot X_{jt}
        if r_j ≤ t ≤ d_j - T then
            schedule 2y_j fraction of job j in calibration
            reset y_j = 0
        carryover = 0
    C_t = C_t - frac \cdot C_t
    carryover = carryover + C_t
    foreach job j do
        y_j = y_j + X_{jt}
```

Lemma 3.4. The rounding process (Algorithm 3.1) produces a valid schedule C of calibrations on 3m' machines, where no two calibrations on the same machine overlap.

Proof. Due to constraint (1), slightly restated, the LP solution guarantees that for all t, we have \( \sum_{t' \in T, t \leq t' < t+T} C_{t'} \leq m' \). That is, there are at most m (total fractional) calibrations started in any time-T period. The rounding process delays at most a 1/2 calibration at a time. It follows that the number of integer calibrations output in the range \([t, t+T]\) can be at most \( 2 \cdot (1/2 + \sum_{t' \in T, t \leq t' < t+T} C_{t'}) \leq 2(1/2 + m') = 2m' + 1 \leq 3m' \).
Because we have at most $3m'$ integer calibrations started within a size-$T$ window, the oldest calibration must end before the $(3m'+1)$th calibration begins. Thus, round-robin scheduling suffices.
The following is a proof that the rounded calibration schedule permits a fractional assignment of all jobs. It proceeds constructively by using an augmented algorithm, Algorithm 3.4, that maps job assignments while creating rounded calibrations. Because this is an existential proof, the modified algorithm is only used for the proof.

The idea of Algorithm 3.4 is similar to Algorithm 3.1. Namely, continue to accrue fractional calibrations carryover until reaching a total of exactly $\frac{1}{2}$ a calibration, then create a new full calibration. Logically, the fractional calibrations (and all their jobs) are simply delayed. The augmentation is that, when accruing the fractional calibrations, also record the fractions $y_j$ of jobs being delayed, and write them into the full calibration whenever they obey the TISE constraint. Figure 3.3 gives an example of this process. Because some of the job can be delayed past a TISE-feasible calibration, it overschedules a $2y_j$ fraction of the job.

**Lemma 3.5.** At any point when executing Algorithm 3.4, we have $y_j \leq \text{carryover}$. That is, the unscheduled fraction of job $j$ being delayed is at most the unscheduled fraction of calibration being delayed. Moreover, the fraction of jobs fits within the fractional calibration, i.e., $\sum_j y_j p_j \leq \text{carryover} \cdot T$.

**Proof.** The proof is by induction over iterations. From the pseudocode, $y_j$ only increases by $\text{frac} \cdot X_{jt}$ after carryover increases by $\text{frac} \cdot C_t$. Similarly, at the end of the loop, $y_j$ only increases by $X_{jt}$ after carryover increases by $C_t$. From LP constraint (2), $C_t \geq X_{jt}$, so carryover increases by at least as much $y_j$. Using a similar argument with LP constraint (3), we also have $C_t T \geq \sum_j X_{jt} p_j$, so the increases to $y_j p_j$ cannot exceed the increases to carryover $\cdot T$.

**Corollary 3.6.** For every job $j$, the fractional assignments of job $j$ to calibrations produced by Algorithm 3.4 sum to at least 1. Moreover, the total work assigned to any calibration does not exceed $T$. 


Proof. Consider a time $t$ when deciding whether to schedule a $2y_j$ fraction of job $j$, i.e., determining whether the current calibration obeys the TISE constraint for the job. If $X_{jt'} > 0$ for any $t' \geq t$, then the current calibration is feasible, because the LP only assigns jobs to feasible calibrations. Thus, the only time the calibration can be infeasible for job $j$ is the last time $y_j$ is reset. By Lemma 3.5, $y_j \leq \text{carryover} \leq 1/2$ at this point. Thus, at least the first half of the job is scheduled. The extra factor of 2 when scheduling the job means that the job is (at least) fully scheduled. The bound on total work follows from Lemma 3.5 even if the $2y_j$ fraction of the job is always scheduled in the calibration. \hfill $\square$

**Lemma 3.7.** Assuming the TISE instance is feasible on $m'$ machines, the calibration schedule $C$ resulting from Algorithm 3.1 is a feasible set of calibrations such that all jobs can be (fractionally) assigned to calibrations without violating the TISE constraint. Moreover, $C$ uses at most $3m'$ machines and $2C^*$ calibrations, where $C^*$ is the minimum possible number of calibrations on $m'$ machines.

Proof. The correctness of the calibrations and feasibility of the fractional assignment follow from Lemma 3.4 and Corollary 3.6. Moreover, $C$ also uses $3m'$ machines by construction.

To get the bound on calibrations, observe that any feasible TISE schedule is feasible for the LP. Thus, the optimal solution to the LP has value at most $C^*$. The rounding process doubles the number of calibrations (creating a full calibration for every $1/2$ calibration produced by the LP), yielding a total of at most $2C^*$ calibrations. \hfill $\square$

We next turn our attention to the earliest-deadline-first (EDF) variant (Algorithm 3.3) for scheduling the jobs within the calibrations. Recall that an assignment of jobs to calibrations $C$ can be interpreted as a preemptive schedule. First,
Lemma 3.8 shows that whenever a preemptive schedule is possible on \( C \), a preemptive version of EDF is feasible. Perhaps unsurprisingly, this proof is similar to the preemptive optimality of EDF when considering the classic problem without calibrations. Lemma 3.9 shows that for a TISE instance, EDF can be transformed into a nonpreemptive schedule by doubling the number of machines.

For the purposes of the proof only, the **fractional EDF** algorithm is defined as follows (similar to Algorithm 3.3). Consider the calibrations in nondecreasing order of start time. For the current calibration, let \( J' \) be the set of unscheduled fractional jobs whose windows contain the calibration (i.e., obeying the TISE constraint). Let \( j \in J' \) be the job with the earliest deadline, with ties broken by job number. Assign as much of job \( j \) as possible to the calibration. When \( J' \) is empty or the calibration is full, continue to the next calibration.

**Lemma 3.8.** Consider a valid integer calibration schedule \( C \). Suppose that a fractional TISE assignment of jobs to calibrations is feasible. Then the fractional EDF strategy produces a feasible fractional TISE job assignment to calibration schedule \( C \).

**Proof.** Suppose for the sake of contradiction that the EDF schedule is not feasible. Let \( S \) be a feasible TISE schedule that shares the longest possible prefix with the EDF schedule. Consider the earliest point at which \( S \) and EDF differ, and let \( k \) be the rank of the calibration (in sorted order) during which that difference occurs. Then \( S \) schedules some job \( j \) during calibration \( k \), whereas EDF schedules a different job \( j' \). Because this is the first point of difference, \( S \) must schedule \( j' \) to some later calibration \( k' \). Swap (as much as possible) of \( j \) with \( j' \). If this swap is feasible, this contradicts the assumption that there is a longest matching prefix.

This swap must also be feasible. Because \( j \) is not the EDF job, we know \( d_j \geq d_{j'} \). Moreover, \( j' \) is the EDF job, so it must have release time before the start of calibration.
and hence also before calibration $k'$. It follows that calibration $k'$ is feasible for job $j$ with respect to the TISE constraint.

**Lemma 3.9.** Suppose that EDF produces a valid fractional TISE assignment on calibration schedule $C$. Then there is a feasible integer schedule using twice as many machines and calibrations, specifically by duplicating $C$ across another set of machines.

**Proof.** Let $S$ be the fractional EDF schedule on $C$. Consider each calibration in turn. If the last job assigned to that calibration is fractional, instead assign the full job to the corresponding mirrored calibration. Remove any other fractional pieces of the job.

Because at most one new fractional job can be created at the end of each calibration, this process resolves all fractional assignments.

The following lemma states that Algorithm 3.3 is at least as good as the mapping in the preceding lemma. In some sense, this lemma is not necessary—the algorithm of Lemma 3.9 could be used in place of Algorithm 3.3. However, Algorithm 3.3 is more natural and increases the analytical complexity only slightly.

**Lemma 3.10.** After the $k$th calibration, all jobs completed by the fractional EDF transformation of Lemma 3.9 are also completed by Algorithm 3.3.

**Proof.** The proof is by induction on $k$. For each job chosen by fractional EDF, either it has the earliest deadline and hence our algorithm would also choose it, or the job has already been executed.

Finally, Lemma 3.11 and Theorem 3.12 conclude by giving bounds on the full TISE algorithm and the ISE algorithm respectively.
Lemma 3.11. Assuming the TISE instance is feasible on \(m'\) machines using at most \(C^*\) calibrations, our TISE algorithm produces a feasible schedule on \(6m'\) machine using at most \(4C^*\) calibrations.

Proof. From Lemma 3.7, the LP and calibration-rounding steps produce a feasible calibration schedule using at most \(2C^*\) calibrations on \(3m'\) machines. Lemma 3.10 states that our algorithm is at least as good as the fractional-EDF-to-integer transformation of Lemma 3.9. Combining this fact with Lemma 3.8, we conclude that our algorithm produces a feasible (integer) schedule using twice the number of machines and calibrations, i.e., \(6m'\) machines and \(4C^*\) calibrations.

The following theorem states that using our TISE algorithm to solve an ISE instance gives an \(O(1)\) approximation using \(O(1)\) machine augmentation and no speed augmentation.

Theorem 3.12. Consider any feasible long-job ISE instance on \(m\) machines. Let \(C^*\) denote the minimum possible number of calibrations to feasibly solve the problem on \(m\) machines. Then running our TISE algorithm on that instance produces a feasible TISE schedule using at most \(18m\) machines and \(12C^*\) calibrations. Moreover, a TISE schedule is also a valid ISE schedule.

Proof. This follows directly from Combining the factor of 3 from Lemma 3.2 with Lemma 3.11.

3.4.6 Trading Speed Augmentation for Machine Augmentation

Thus far it has been shown how to construct an \(O(1)\)-machine \(1\)-speed \(O(1)\)-approximation for the ISE problem (producing a more restricted TISE solution). This section concludes by showing how to transform this TISE solution into a
1-machine $O(1)$-speed $O(1)$-approximation. The fact that we are working with long jobs and a TISE solution is pivotal here. It is not clear how to make this sort of transformation in general.

Suppose we have a TISE schedule on $cm$ machines, for integer $c$. (Following from Theorem 3.12, set $c = 18$.) The algorithm is given below:

Group the machines arbitrarily into groups of $c$ machines that will all map to one target speed-2$c$ machine. First, construct the calibration schedule on the target machine as follows. Start at time $t = 0$. Repeat the following steps. If any calibration on the source machine includes timestep $t$, calibrate the target machine at time $t$, advance to time $t = t + T$, and repeat. Otherwise, increase $t$ to the next earliest calibration on any of the source machines. This calibration schedule guarantees that every calibrated timestep on any of the source machines is also a calibrated timestep on the target machine.

Next, consider each calibration interval $[t, t+T)$ on the target machine in any order. For any calibration $\ell$ on a source machine $i \in \{0, 1, \ldots, c - 1\}$ that fully contains the first half of the target calibration, i.e., $[t, t + T/2)$, assign $\ell$ to the size-$T/(2c)$ time interval $[t + iT/2c, t + (i + 1)T/2c)$. Keep the jobs in the same order within the interval; just scale the processing times by a factor of $1/2c$. Perform a similar process for each source calibration fully containing the second half, $[t+T/2, T)$, of the target calibration.

**Lemma 3.13.** Given a TISE schedule on $cm$ speed-1 machines with $C$ calibrations, the above algorithm produces an ISE schedule on $m$ speed-2$c$ machines with at most $C$ calibrations.

**Proof.** Consider a single group of $c$ source machines. At most one source machine is mapped to each size-$T/(2c)$ subinterval of a target calibration. Moreover, because
calibrations on each machine do not overlap, at most one calibration on each machine can be mapped there.

These mappings are feasible because a) the target interval is fully contained in the source calibration, and b) we start from a TISE instance, meaning that the jobs can be run at any time within their calibration.

To see that every source calibration is mapped somewhere, suppose that a source calibration only overlaps part of the end of $[t, t+T)$. Then there is another calibration on the target machine at time $[t+T, t+2T)$. The source calibration must either overlap half of $[t, t+T)$ or half of $[t+T, T+2T)$. A similar argument applies on the front end of target calibrations. Thus, every source calibration is mapped.

Finally, to count the number of calibrations, consider the calibration process. A calibration only occurs on the target machines if 1) there is a calibration on some source machine at the same time, or 2) there is a calibration on some source machine between the previous calibration and the current one. Thus, it’s possible to charge all target calibrations against source calibrations.

Combining this speed transformation with Theorem 3.12 yields the following theorem, meaning a 1-machine $O(1)$-speed $O(1)$-approximation.

**Theorem 3.14.** Consider any feasible long-job ISE instance on $m$ machines. Let $C^*$ denote the minimum possible number of calibrations to feasibly solve the problem on $m$ machines. Then running our TISE algorithm followed by the machine-to-speed transformation produces a feasible ISE schedule using at most $m$ machines, each at $36$ speed, with at most $12C^*$ calibrations.
3.5 Strategy for Short-Window Jobs

This section presents an ISE algorithm for the special case that all jobs have short windows. The algorithm exploits similarities between the ISE problem and the classic machine minimization (MM) problem, applying any MM algorithm as a black box while asymptotically preserving its approximation guarantees.

The main idea stems from the following simplified case. Suppose that all jobs fall within a single size-$T$ time interval, i.e., $\max_j \{d_j\} - \min_j \{r_j\} \leq T$. Then an optimal ISE solution uses either 0 or 1 calibrations per machine. Thus, minimizing the number of machines and minimizing the number of calibrations are equivalent, and applying the MM algorithm as a black box yields a good solution to the ISE instance. With some manipulation, this relationship can be generalized and used to construct solutions for a short-window ISE input. Note that applying the MM algorithm globally does not work, because there may be long periods of time where it is optimal to use fewer machines (and hence fewer calibrations).

Throughout this section, to avoid confusion about where constants are being introduced let $\gamma T$ denote the maximum window length of a short job, i.e., $\gamma = 2$ according to Definition 5.

3.5.1 The Algorithm

The rough strategy of the algorithm is as follows. Partition time into size-$2\gamma T$ intervals. For each interval, consider the subset of jobs whose windows are contained inside the interval. Apply an MM algorithm to the interval. Transform the MM schedule to an ISE schedule by adding appropriate calibrations, but executing all jobs at the same time as before. The final schedule is the union of the schedules for each interval.
Algorithm 3.5 Partitioning short jobs into length- \(2\gamma T\) intervals

Let \(J_{\text{short}}\) be the set of all short-window jobs

Allocate disjoint sets of machines \(M_1\) and \(M_2\)

\[
t \leftarrow 0
\]

\[
\textbf{while } t \leq \max_j \{d_j\} \textbf{ do}
\]

\[
\text{Let } J' \subseteq J_{\text{short}} \text{ be the jobs nested in } [t, t + 2\gamma T),
\]

\[
\text{i.e., with } t \leq r_j < d_j \leq t + 2\gamma T.
\]

\[
\text{Schedule } J'\text{ on machines } M_1 \text{ using Algorithm 3.6}
\]

\[
t \leftarrow t + 2\gamma T
\]

Remove from \(J_{\text{short}}\) any jobs scheduled above

\[
t \leftarrow \gamma T
\]

\[
\textbf{while } t \leq \max_j \{d_j\} \textbf{ do}
\]

\[
\text{Let } J' \subseteq J_{\text{short}} \text{ be the jobs nested in } [t, t + 2\gamma T),
\]

\[
\text{i.e., with } t \leq r_j < d_j \leq t + 2\gamma T.
\]

\[
\text{Schedule } J'\text{ on machines } M_2 \text{ using Algorithm 3.6}
\]

\[
t \leftarrow t + 2\gamma T
\]

The simple partitioning strategy does not quite work because there may be arbitrarily many jobs whose windows span the boundaries separating intervals. Fortunately, there is a trivial fix for this issue: partition time again but at an offset of \(\gamma T\), and schedule any remaining jobs on a new set of machines as before. This revised partitioning step is given as pseudocode in Algorithm 3.5. For clarity, the partitioning pseudocode as presented is linear in the length of the schedule, but it is straightforward to transform the code to be polynomial in the number of jobs.

For each of the intervals \([t, t + 2\gamma T)\) produced by the partitioning step, produce a subschedule for the jobs \(J'\) in that interval as follows (see Algorithm 3.6). First construct an MM schedule \(S\) for jobs \(J'\) using \(w\) machines. Note that the MM schedule \(S\) specifies a start time \(x_j\) for each job as well as a machine on which to run that
Algorithm 3.6 Scheduling each length-2γT interval

Let $t$ be the start time of the interval
Let $J'$ be the set of jobs assigned to this interval

Run an MM algorithm on $J'$ to produce schedule $S$
Let $w$ be the number of machines used by $S$

Use $3w$ machines for $S'$

for $i = 1$ to $w$ do
  calibrate machine $i$ at $t + kT$ for $k \in \{0, 1, 2, \ldots, 2\gamma - 1\}$

for each job $j \in J'$ do
  Let $m_j$ be the machine to which $j$ is assigned in $S$
  Let $x_j$ be the start time of $j$ in $S$
  if $j$ is not a crossing job then
    assign $j$ to machine $m_j$ at time $x_j$ in $S'$
  else
    if $j$ is a $k$-th crossing job for even $k$ then
      calibrate machine $w + m_j$ at time $x_j$ in $S'$
      assign $j$ to machines $w + m_j$ at time $x_j$ in $S'$
    else
      calibrate machine $2w + m_j$ at time $x_j$ in $S'$
      assign $j$ to machine $2w + m_j$ at time $x_j$ in $S'$

job, but an ISE schedule $S'$ must also specify a schedule of calibrations on each machine. Moreover, $S'$ must ensure that each job’s execution fall fully within a single calibration. Transforming the MM schedule $S$ into ISE schedule $S'$ is done in several steps. First calibrate each of the $w$ machines $2\gamma$ times. Then, map $S$ to $S'$ by preserving the times at which the jobs are executed. The remaining question is how to assign jobs to machines. There are two cases. If a job is fully contained in a calibration, it is assigned to the same machine in $S'$ as in $S$. The more challenging case arises when
a job crosses calibration boundaries, an issue that is accommodated by introducing more machines.\textsuperscript{3} Call a job \( j \) a \textbf{\textit{k-th crossing job}} if the start time \( x_j \) of the job falls in the \( k \)-th calibration, i.e., \( t + kT \leq x_j < t + (k+1)T \), but the completion time of the job falls in a different calibration, i.e., \( x_j + p_j > t + (k+1)T \). For each machine used by \( S \), introduce a new machine to handle crossing jobs for odd \( k \), and another machine to handle crossing jobs for even \( k \). For each crossing job, create a new calibration dedicated specifically to the job. It remains to prove that all of the jobs in \( S' \) fall fully within a calibration, and no calibrations on a single machine overlap, so the schedule \( S' \) is a feasible schedule.

3.5.2 \textbf{Correctness and performance analysis}

To show that the algorithm produces a valid schedule, Lemma 3.15 begins by proving that the subschedules produced for each interval are feasible. Lemma 3.16 then proves that the main algorithm combines these interval schedules without introducing any conflicts.

\textbf{Lemma 3.15.} \textit{Consider any set of short jobs \( J' \) with windows nested inside a time interval \([t, t + 2\gamma T]\). Algorithm 3.6 produces a valid ISE schedule for these jobs.}

\textit{Proof.} A feasible MM schedule has two main properties. 1) Every job is scheduled nonpreemptively within its window, i.e., starting no earlier than its release time and finishing no later than its deadline. 2) Jobs on the same machine cannot have overlapping execution periods. An ISE schedule adds two additional restrictions, namely: 3) Every job’s execution must be contained fully within a calibration on the machine.

\textsuperscript{3}If a calibration is allowed to be performed before the previous calibration ends, then no extra machines are necessary, just extra calibrations. We focus here on the more difficult version of the ISE problem, where calibrations cannot be invoked less than \( T \) timesteps of each other.

45
to which it is assigned, and 4) for each machine, the calibrations on that machine must be nonoverlapping.

Algorithm 3.6 starts with a feasible MM schedule, preserving all execution times, so property (1) holds trivially. Moreover, for each machine in the MM schedule, the jobs assigned to that machine are spread across three machines in the ISE schedule, so (2) also holds trivially.

Next, we consider properties (3) and (4). For the first $w$ machines, where $w$ is the number of machines used by the MM schedule, property (4) holds by construction—calibrations are performed exactly every $T$ timesteps. Moreover, noncrossing jobs satisfy property (3) on those machines by definition. For each crossing job, Algorithm 3.6 creates a new calibration, thereby satisfying property (3). It’s sufficient to argue that those calibrations do not overlap each other, and hence the schedule also observes property (4). Consider two crossing jobs assigned to the same machine in the ISE schedule. Because they are assigned to the same machine, they must have the same crossing parity (even or odd). Moreover, they must start from the same machine in the MM schedule, so those crossing numbers must differ by at least 2. Thus, the scheduled start times for the jobs must differ by at least $T$, meaning that the calibrations do not overlap.

\[\square\]

**Lemma 3.16.** The short-window algorithm (combining Algorithms 3.5 and 3.6) produces a valid schedule for an ISE instance of short-window jobs.

**Proof.** This proof first argues that all jobs are assigned to some interval in the partitioning step, and hence all jobs are part of some interval schedule. It then shows that the interval schedules do not interfere, and hence a final schedule can be formed by taking the union across intervals. Combining these two facts implies that the overall schedule is feasible.
Consider a particular job $j$ in the partitioning step. If $j$ is scheduled during the first loop of Algorithm 3.5, we are done. Suppose instead that $j$ is not scheduled during the first loop. Then $j$’s window crosses a multiple of $2\gamma T$, say $2k\gamma T$ for integer $k$. Because $j$ is short, its window has length at most $\gamma T$, and hence $r_j \geq 2k\gamma T - \gamma T$ and $d_j \leq 2k\gamma T + \gamma T$. Thus, $j$’s window is contained completely in the interval $[(2k - 1)\gamma T, (2k + 1)\gamma T)$, and $j$ is assigned to an interval in the second loop.

It remains to argue that the interval schedule in Algorithm 3.6 only creates calibrations nested inside the interval $[t, 2\gamma T)$. The calibrations for noncrossing jobs fall inside $[t, 2\gamma T)$ by construction as long as $\gamma$ is an integer. Crossing jobs, on the other hand, must have start times within the range $[t, (2\gamma - 1)T)$, so those calibrations are also inside the interval. Thus, taking the union of interval schedules is feasible as long as the intervals themselves are disjoint (which is true for the first or second loop of Algorithm 3.5).

The following lemmas analyze the performance of the short-window algorithm. This analysis has a few components. First, it argues that for each interval instance, the MM solution serves as a lower bound for the number of calibrations for the ISE problem. Then it extends the lower bound across a set of disjoint intervals as produced by each pass of the partitioning phase. Finally, it concludes by arguing that the algorithm only loses a constant factor beyond the MM algorithm applied.

**Lemma 3.17.** Consider any set of short jobs $J'$ with windows nested inside a time interval $[t, t + 2\gamma T)$. Let $w^*$ be the value of the optimal solution to the MM problem on $J'$, i.e., the minimum number of machines. Then the ISE problem requires at least $w^*$ calibrations and machines.

**Proof.** The lemma follows from the fact that all feasible solutions require at least $w^*$ machines, and each machine must be calibrated at least once to schedule any jobs. □
Lemma 3.18. For fixed offset time $\tau$, suppose $J_i$ is a set of short jobs with windows nested inside time interval $[\tau + 2i\gamma T, \tau + 2(i+1)\gamma T)$. Let $w_i^*$ be the value of the optimal solution to the MM problem on $J_i$.

Then any feasible solution to the ISE problem on the $r+1$ disjoint intervals $\cup_{i=0}^r J_i$ requires at least $\max_i w_i^*$ machines. Moreover, an optimal solution to the ISE problem on $\cup_{i=0}^r J_i$ requires at least $\sum_{i=0}^r w_i^*/2$ calibrations.

Proof. From Lemma 3.17, each subset requires $w_i^*$ machines. Adding more jobs only increases the number of machines. Thus $\max_i w_i^*$ is a lower bound.

Consider every other interval $J_0, J_2, J_4, \ldots$ or $J_1, J_3, J_5, \ldots$. Intervals $J_i$ and $J_{i+2}$ are separated by much more than $T$ timesteps. So no calibration used for $J_i$ can also be used for $J_{i+2}$. Thus, following from Lemma 3.17, $w_0^* + w_2^* + w_4^* + \cdots$ is a lower bound for the minimum possible number of calibrations. Similarly, $w_1^* + w_3^* + w_5^* + \cdots$ is also a lower bound. Taking the maximum of the two, we have a lower bound of $\sum_{i=0}^r w_i^*/2$ calibrations.

Lemma 3.19. Consider any set of short jobs $J'$ with windows nested inside a time interval $[t, t + 2\gamma T)$. Let $w$ be the number of machines found by the black-box MM algorithm. Then the ISE solution on $J'$ performs at most $4\gamma w$ calibrations on $3w$ machines.

Proof. Consider a single machine in the MM solution. The algorithm constructs three machines for the ISE schedule. The first of these machines gets calibrated every $T$ timesteps, for $2\gamma$ calibrations. Each of the crossing jobs is assigned to one of the other two machines, with one calibration per crossing job. Since there can be at most $2\gamma - 1$ crossing jobs, there are at most $4\gamma - 1$ calibrations arising from this machine. Multiplying by the number $w$ of machines completes the proof.
Finally, this section concludes that for constant $\gamma$, the algorithm asymptotically preserves the approximation guarantees of the MM algorithm applied.

**Theorem 3.20.** Consider any short-job instance $J_{\text{short}}$. Let $w^*$ denote the minimum possible number of machines among feasible ISE schedules. Let $C^*$ be the minimum possible number of calibrations among feasible ISE schedule.

Suppose that we have a black-box $\alpha$-approximation algorithm to the MM problem. Then the ISE algorithm produces a feasible ISE schedule on at most $6\alpha w^* = O(\alpha w^*)$ machines using at most $16\gamma \alpha C^* = O(\alpha C^*)$ calibrations.

**Proof.** Consider a sequence of disjoint intervals as defined in Lemma 3.18. From Lemma 3.18, we have $C^* \geq \sum_i w_i^*/2$ and $w^* \geq \max_i w_i^*$, where $w_i^*$ is the optimal number of machines for the $i$th interval. For the $i$th interval, the MM algorithm finds a solution using at most $\alpha w_i^*$ machines. Thus applying Lemma 3.19, Algorithm 3.6 makes at most $(4\gamma)\alpha w_i^*$ calibrations on $3w_i^*$ machines. Summing across all $i$, the ISE algorithm uses at most $4\gamma \alpha \sum_i w_i^* \leq 8\gamma \alpha C^*$ calibrations on $3 \max_i \alpha w_i^* \leq 3\alpha w^*$ machines.

A factor of 2 is lost in both bounds because Algorithm 3.5 runs on two sets of disjoint intervals using disjoint machines.

3.6 Conclusions

This chapter showed how to reduce the non-unit ISE problem to the MM problem, producing approximation guarantees that are almost as good as those for the MM problem. To within constant factors, this is the best possible. In the case that all jobs have long windows, the algorithm is asymptotically optimal.

The best general approximation to the MM problem is an $O(\sqrt{\log n / \log \log n})$ approximation so this work makes little effort to minimize the relevant constants. It
is likely that some of the constants in the reduction could be reduced. That said, partitioning jobs into long and short jobs inherently has an overhead of at least 2 in terms of both machines and calibrations. It would be nice to achieve constants that look like \((1 + \epsilon)\), but that would require some new ideas.

This solution also serves to highlight the importance of batching for this problem. The generalization of variable length jobs makes greedy strategies much more difficult. Instead, the provided algorithm uses an LP-rounding strategy to more deliberately balance the benefits of batching against the scheduling constraints of the input. Notably, this includes an explicit partition between long-window jobs that can be moved to significantly improve batching without conflicting with their deadline and short-window jobs that cannot. This deliberate consideration of batching benefits and constraints appears necessary to achieve good bounds for the problem.

3.7 Open Problems

The first open problem is a holdover from Bender et al. [13]. It seems likely that testing the feasibility of a unit-length processing time multi-machine ISE problem is \(\mathcal{NP}\)-Hard (as was shown in the non-unit case with reduction from Partition). However, finding an appropriate reduction has been more difficult due to the homogeneity of processing times (allowing greedy selection of ready jobs based on deadline).

The second open problem is a further generalization of the problem. It would be interesting to investigate the effects of allowing a variable length calibration time based on the machine, \(T_m\). It is not immediately clear whether this generalization can be solved with the same strategy or if it would require a similarly drastic change coupled with model augmentation.
In addition, one potentially interesting exercise would be to test how well an LP-relaxation strategy performs on unit-length job problem instances. It is very unlikely that it will improve upon the known 2-approximation, but a variable-length solution that matches the bounds of greedy solutions in the unit-length case would be a desirable generalization.
Chapter 4

Minimizing Calibrations and Flow-Time for Non-Unit Jobs

The next problem continues our focus on variations of the original ISE formulation. As in Chapter 3, we will consider a well-motivated use-case that requires batching jobs into calibrations. Batching similarly conflicts with other scheduling concerns, in this case flow-time. The provided solution carefully balances the benefits of batching against the cost of delaying jobs.

More specifically, this chapter will focus on offline single machine ISE instances without hard job deadlines. A problem such as ISE that captures equipment and instrumentation calibration is, of course, well motivated outside of stockpile evaluation. Many manufacturing sectors such as pharmaceuticals [8], electronics [73], and utilities [3] require extremely precise robotics and measurement equipment that must be calibrated periodically to maintain accuracy. As in ISE, these calibration intervals are often reduced to a fixed time-length based on a variety of reliability, cost, and regulatory factors [11, 76]. However, unlike stockpile evaluation, manufacturing may not entail strict deadlines but rather a cost for delaying jobs that should be weighed against the cost of calibrating the necessary equipment.

To capture this use-case, Chau et al. [24] introduced the problem of minimizing total flow time with calibrations, or more specifically, the linear combination of total flow time with the number of calibrations. This problem will be referred to as ISE-flow. This chapter tackles the offline single-machine variant with variable job lengths (denoted $1|r_j|H \cdot \#\text{calibrations} + \sum F_j$ in standard scheduling notation where $H$ is a
non-negative constant). The input consists of a set $\hat{J}$ of $n$ jobs and an integer $T \geq 2$ specifying the **calibration length**. Each job $j$ has a processing time $p_j \leq T$, and a release time $r_j$. A schedule is feasible if it schedules jobs nonpreemptively on a single machine such that every job is scheduled without preemption completely within a calibrated time period.\(^1\) The goal is to find a feasible schedule that minimizes the objective function,

$$H \cdot \#\text{calibration} + \sum_{j \in \hat{J}} F_j$$

where $F_j$ is the flow of job $j$, defined as its completion time $S_j$ minus its release time $r_j$.

As in Chapter 3, we are focusing on the non-unit version of a problem that has been solved for the unit case. It is trivial to see that this problem reduces to the classic problem of minimizing total flow when $H = 0$. Minimizing total weighted flow is $\mathcal{NP}$-Hard and does not lend itself readily to approximation algorithms (Kellerer et al. [51] proved that it cannot be approximated to within a factor of $n^{1/2-\epsilon}$ for $\epsilon > 0$ unless $\mathcal{P} = \mathcal{NP}$). Therefore this chapter will again focus on resource augmenting solutions. There are known $O(1)$-approximation and $O(1)$-speed $O(1)$-approximation algorithms for unit ISE-flow and minimizing total flow respectively, so the goal is a constant approximation algorithm with constant augmentation.

To understand the solution quality possible without relaxing the unit job-length of the problem model, recall that we face the same pre-processing dilemma as in Chapter 3. A unit ISE-flow algorithm cannot do better than a $O(T)$-approximation given an arbitrary non-unit input, or $O(\Delta)$ where $\Delta$ is the maximum job size ratio.\(^1\) This relaxes the requirement from Chapter 3 that every job be scheduled within a single interval by allowing jobs to cross calibration boundaries provided the machine is calibrated throughout the time-frame. This choice was made for algorithm simplicity and readability. It’s possible to adhere to the stricter requirement using a constant-factor more augmentation.

\(^1\)This relaxes the requirement from Chapter 3 that every job be scheduled within a single interval by allowing jobs to cross calibration boundaries provided the machine is calibrated throughout the time-frame. This choice was made for algorithm simplicity and readability. It’s possible to adhere to the stricter requirement using a constant-factor more augmentation.
Thus, any result better than an $O(T)$-approximation marks an improvement over known results.

4.1 Contributions

The primary result of this chapter is an algorithm for single-machine non-unit ISE-flow. It is proven to be a 72-speed 17-approximation polynomial time algorithm but requires one additional machine. This constant approximation with constant augmentation marks a strict asymptotic improvement over known results.

For the sake of completeness, this chapter also provides an algorithm for multi-machine non-unit ISE-flow using a simple job-size partitioning and the unit ISE-flow algorithm from McCauley [57]. The result is a 4-speed $\lceil \lg n \rceil$-machine $12\lceil \lg n \rceil$-approximation polynomial time algorithm. This is a good bound relative to best known flow minimization results. It roughly matches the $O(\log n)$-machine $(1+o(1))$-approximation bound from Phillips et al. [63].

4.2 Related Work

Non-preemptive scheduling to minimize total flow time without the ISE constraint is a classic problem with relatively tight bounds for both single and multiple machine variants. Without release times, the problem is trivially solved with a greedy shortest processing time strategy generally attributed to Smith [72]. Lenstra et al. [53] show that the introduction of release times makes the problem $\mathcal{NP}$-complete. The rest of this discussion assumes arbitrary release times and will therefore focus on approximation algorithms.
The single machine version of the problem has surprisingly poor approximation bounds without resource augmentation. Kellerer et al. [51] give an $O(\sqrt{n})$-approximation algorithm by systematically eliminating preemptions from an optimal SRPT-based schedule. They prove that this bound is essentially tight because the approximation is $\Omega(n^{1/2-\epsilon})$ unless $P = NP$.

This similarly extends to the multi-machine version. Leonardi and Raz [54] give a $O(\sqrt{n/m \log \frac{n}{m}})$-approximation algorithm and prove a $\Omega(n^{1/3-\epsilon})$ approximability lower bound. The poor approximability of this problem makes it a good candidate for resource augmentation.

This chapter draws heavily on a single machine $O(1)$-speed $O(1)$-approximation result from Bansal et al. [7]. They use a clever LP objective function that allows them to maintain the flow bound after a laminar-based $\alpha$-rounding step. Their result also extends to weighted total flow-time and other min-sum based objectives. It is currently unclear if this technique can be extended to the multiple machine setting.

The best known bound for multiple machines with augmentation comes from Phillips et al. [63]. They prove an online $O(\log n)$-machine $(1 + o(1))$-approximation algorithm using a laminar-based partitioning of job sizes among machines. Clairvoyance has not been shown to improve these bounds.

For more discussion of work related to the ISE problem, refer back to Section 3.2.

4.3 Single-Machine ISE-flow

This section covers the main contribution of the chapter, a provably good algorithm for solving single-machine ISE-flow. More specifically, it proves the following theorem via construction.
Theorem 4.1. There exists a 72-speed \((16+\epsilon)\)-approximation deterministic polynomial-time algorithm on 2 machines for \(1|r_j|H(#\text{Calibrations}) + \sum F_j\).

4.3.1 Algorithm Overview

This section gives the top-level overview of the algorithm and analysis designed to prove Theorem 4.1.

The proof follows the general strategy (specifically steps 1-2 and 5-7) of the algorithm given by Bansal et al. [7] for \(1|r_j|H(#\text{Calibrations}) + \sum F_j\), with additions that guarantee an efficient calibration schedule. However, it is difficult to succinctly show that these additions do not break the original proof structure. In particular, introducing calibrations seems to necessitate several additional partitioning and rounding steps with non-obvious guarantees at each major stage. Therefore, a complete proof is provided. The major steps are as follows:

1. Round down job sizes in the original instance, \(\hat{J}\), to form a geometric series \(\hat{J}\) with constant base \(\beta\), using a speed factor of \(2\beta\). This guarantees that every job belongs to some size class \(\beta^i\) and can be scheduled at an integer multiple of its size.

2. Give a linear programming formulation that produces a fractional solution to \(J, LP = (x, C)\), and show that the objective lower-bounds \(OPT(J)\).

3. Round the calibration solution from the LP into an integer schedule, \(C^H\), using 3 machines and 3 \(OPT\) calibrations.

4. Partition and convert the fractional job schedule \(x\) into two disjoint schedules \(x^H\) and \(x^L\) that are feasible with an integer calibration schedule.
(a) Partition jobs into **high-flow** and **low-flow** based on their LP-defined (high flow jobs have $F' \leq T$).

(b) High flow jobs can be assigned on $C^H$ by sacrificing a 2-factor in the overall flow.

(c) Low flow jobs require an augmented version of $C^H$, $C^L$ using an additional 6 machines and 6 $OPT$ calibrations, as well as a 1.5-factor increase in flow.

5. Apply $\alpha$-rounding separately to both partitions, converting them into single-machine pseudo-schedules $x^{H'}$, $C^{H'}$ and $x^{L'}$, $C^{L'}$.

6. Resolve conflicts in the pseudo-schedules using another $m \left(2 + \frac{1}{\beta-1}\right)$-factor increase in speed, where $m$ is the number of machines used before conversion into the pseudo-schedule. This produces feasible integer schedules for each partition at the cost of a 2-factor increase in flow.

7. Finally, argue that the algorithm can be de-randomized with another small increase in flow.

Perhaps surprisingly, the key intuition behind the partitioning in step 4 closely resembles that of the long/short-window partition in Chapter 3. In Chapter 3 the algorithm was forced to carefully handle short-window jobs because their deadlines potentially conflicted with a greedy rounding of the LP solution. Similarly, this algorithm handles low-flow jobs carefully because moving them significantly could drastically increase their flow. It is unclear if this is a coincidence or if this sort of partition is somehow inherent to a job-centric LP-rounding for non-unit ISE-based problems.
4.3.2 Input Preparation

The algorithm starts by rounding down job sizes in the original instance, \( \hat{J} \) such that they form a geometric series. This allows us to focus on aligned schedules, drastically simplifying the later rounding process. An **aligned schedule** is one in which jobs may only start on an integer multiple of their processing time. Assume \( \beta \in \mathbb{Z} > 1 \).

The algorithm rounds all the processing times in \( J \) to form a geometric series, i.e.

\[
p_j = \frac{1}{2} \beta^{\lfloor \log_\beta \hat{p}_j \rfloor} \quad \forall j \in J
\]

Note that since \( p_j \geq \hat{p}_j/(2\beta) \), it is possible to convert a solution for \( J \) back into a solution for \( \hat{J} \) at the end using a \( 2\beta \)-factor speedup.

**Lemma 4.2.** (Lemma 2 in [7]) There exists an aligned schedule for \( J \) that lower-bounds the optimal solution for \( \hat{J} \).

**Proof.** Given an optimal total flow schedule \( OPT(\hat{J}) \), construct an aligned schedule for \( J \) as follows: for each job start time \( S_j \in OPT(\hat{J}) \), schedule \( j \) at the next aligned time-step, \( S_j = S_j + (p_j - (S_j \mod p_j)) \). We know that, since \( p_j \leq \hat{p}_j/2 \), \( S_j \) falls within \([S_j, S_j + \hat{p}_j/2]\). We also know, based on the rounding, that \( p_j \leq \hat{p}_j/2 \) so \( j \) in this schedule completes at least as early as \( \hat{j} \) in \( OPT(\hat{J}) \). \( S_j \leq S_j < S_j + p_j \leq S_j + \hat{p}_j \) so it does not conflict with the calibration schedule from \( OPT(\hat{J}) \), making it a valid aligned schedule. All jobs finish earlier so the total flow is a lower bound on \( OPT(\hat{J}) \).

Assume WLOG that the time axis for \( J \) is scaled by a factor of 2. This lets us denote job sizes as integer power of \( \beta \), \( p_j \in \{\beta^0, \beta, \beta^2, \ldots\} \). Let \( G_i \) denote **class-\( i \) jobs**, all jobs \( \{j \in J | p_j = \beta^i\} \).
4.3.3 Linear Programming Formulation

This section gives a Linear Programming formulation for a fractional version of the problem. Note that the job assignments are significantly more precise than the LP in Chapter 3, denoting a specific starting timestep in addition to the containing calibration.

**Algorithm 4.1** LP-formulation for total flow-time minimization

\[
\forall j \in J : \\
F'_j = \frac{1}{2} \left( \sum_t (t + p_j - r_j)x_{jt} + p_j + \sum_{k: G_k^{-1} > G_j^{-1}} \sum_{t \in [r_j - p_k + 1, r_j]} (t + p_k - r_j)x_{kt} \right)
\]

minimize \( H \sum_t C_t + \sum_{j \in J} F'_j \)

subject to

\[
\sum_t x_{jt} C_t = 1 \quad \forall j \in J \tag{1}
\]

\[
\sum_{t-T<\tau\leq t} C_\tau \leq 1 \quad \forall t \in T, \tau \in T \tag{2}
\]

\[
x_{jt} C_t \leq C_t \quad \forall j \in J, t \in T, C_t \in C \tag{3}
\]

\[
x_{jt} C_t = 0 \quad \forall C_t > t, C_t < t + p_j - T, t < r_j \tag{4}
\]

\[
\sum_j \sum_{\tau: t-\tau \leq t} x_{jt C_t} \leq C_t \quad \forall t \in T, C_t \in C \tag{5}
\]

\[
x_{jt}, F_j, C_t \geq 0 \quad \forall j \in J, t \in T \tag{6}
\]

Also note that the objective of Algorithm 4.1 uses the novel two-term representation of flow devised by Bansal et al. [7]. The first term is a standard fractional interpretation of flow. The second term, the **penalty term**, is used after the \( \alpha \)-rounding process to charge the potentially increased flow of a smaller job against the corresponding flow of a larger job that caused its delay.

Otherwise, it is a standard time-indexed formulation of the problem where \( C_t \) represents a calibration at time \( t \) and \( x_{jt} C_t \) represents whether job \( j \) is scheduled at
time $t$ in a calibration scheduled at time $C_t$: constraint one requires that every job is fully scheduled, constraint two requires that the calibration at any time step cannot exceed 1, constraint three requires that the fraction of the job assigned to a calibration cannot exceed the size of the calibration, constraint four constrains job assignments to valid calibrations, constraint five requires that the total job volume active at any time within a calibration cannot exceed the size of that calibration, and constraint six ensures non-negativity.

It is not immediately obvious why this two-term representation of flow is useful, but for now it suffices to show that each of these terms independently lower bound the flow of $OPT(J)$ so their average must also lower bound $OPT(J)$.

**Lemma 4.3.** (Lemma 3 in [7]) The two terms of $F'_j$, $(\sum_t (t + p_j - r_j)x_{jt})$ and $(p_j + \sum_{k:G_k^{-1}>G_j^{-1}} \sum_{t \in [r_j-p_k+1,r_j]} (t + p_k - r_j)x_{kt})$ each independently lower bound the traditional flow-time of a job $j$ in $OPT(J)$.

**Proof.** The first term is clearly equivalent to flow-time of a job in an integer schedule. The second term can be proven with the observation that, for any integer schedule, if a job $k$ is running at time $r_j$, then job $j$ cannot start earlier than the finish time of $k$ and its total flow cannot be less than $p_j$ plus the difference of $k$’s finish time and $r_j$.

Next, we note that the combined objective still adequately lower-bounds the desired optimal schedule.

**Lemma 4.4.** $\min \left( \sum_t C_t + \sum_{j \in J} F'_j \right) \leq OPT(J)$

**Proof.** Any feasible calibration schedule for $OPT(J)$ is still a valid schedule in the LP formulation with the same cost. The two terms of the flow representation each lower-bound the actual flow of an integer solution to $OPT(J)$ per Lemma 4.3 so
their average similarly admits any solution to $OPT(J)$ with equal or lesser cost. The minimized value of the combined terms must therefore lower-bound $OPT(J)$. \qed

Finally, it remains to show that the LP solution can be converted into a canonical one with no loss in performance. This is not important for correctness. However, it will allow us to prove several properties that will prove useful later. Let $LP = (x, C)$ be the fractional solution to Algorithm 4.1. We define a fractional schedule to be \textbf{canonical} if for any two jobs, $i, j \in J$ s.t. $p_i = p_j$ and $r_i < r_j$ (with ties in release time broken by job number), job $i$ completes before job $j$ is started. Let $X(t, j) = \sum_{s \leq t} x_{js}$ denote the fraction of job $j$ complete at time $t$. Let $X_i(t) = \sum_{j \in G_i} \sum_{s \leq t} x_{js}$.

\textbf{Lemma 4.5.} (Lemma 5 in [7]) Given an optimal solution, $LP$, it is possible to construct a canonical schedule that is still optimal in polynomial time.

\textit{Proof.} For each non-canonical pair, $r_i < r_j$, replace in chronological order all non-zero instances of $x_{jt}$ with $x_{it} = x_{it} + x_{jt}$ until $i$ is fully complete. Replace the remaining instances of $x_{it}$ with $x_{jt} = x_{it} + x_{jt}$. The resulting flow-time must be equal to or less than the original flow-time because $j$ completes at the same time as the second to complete in the original schedule; $i$ completes no later than the first job to complete in the original schedule because it runs without interruption. \qed

A canonical schedule gives us a total ordering within each class $G_i$. Specifically, we denote $j \preceq_i k \iff p_i = p_j \land r_j < r_k$.

\textbf{Lemma 4.6.} In a canonical schedule $LP$, for any two jobs $j \preceq_i k$, $0 < X(t, k) < 1 \implies x_{jt} = 0$.

\textit{Proof.} This follows directly from the definition of a canonical schedule. Since $j$ precedes $k$ in the class, $j$ must complete before $k$ is started. \qed
4.3.4 Rounding the Calibration Schedule

This section constructs an integer calibration schedule using a simple volume-based rounding strategy and three machines. This schedule will serve as the basis of the schedules created in both Sections 4.3.5 and 4.3.5. The procedure is given in Algorithm 4.2.

Algorithm 4.2 Procedure for rounding the fractional calibration schedule \( C \) onto three machines

\[
\text{carryover} = 0 \; ; \\
CH = \emptyset \; ; \\
i = 0 \; ; \\
\begin{array}{l}
\text{foreach } t \in T \text{ in increasing order do} \\
\text{carryover} = \text{carryover} + C_t \\
\text{if } \text{carryover} \geq 1/2 \text{ then} \\
\text{while } \text{carryover} > 0 \text{ do} \\
\quad \text{add a calibration at time } t \text{ on machine } i \text{ to } CH \\
\quad \text{carryover} = \text{carryover} - 1/2 \\
\quad i = i + 1 \mod 3
\end{array}
\]

For now, it is sufficient to show that the resulting schedule is valid. We will see in Section 4.3.5 that it accommodates a feasible constant-approximate high-flow fractional job schedule. Section 4.3.5 will prove that additional augmentation admits a feasible and constant-approximate low-flow fractional job schedule.

**Lemma 4.7.** Algorithm 4.2 produces a valid schedule of calibrations on 3 machines, where no two calibrations on the same machine overlap.

**Proof.** The LP-formulation guarantees that, for all \( t \in LP \), we have \( \sum_{\tau \in T, t \leq \tau < t+T} C_\tau \leq 1 \). The rounding process delays strictly less than a 1/2 calibration at a time. It follows that the number of integer calibrations output in the range \([t, t+T)\) can be at most \( 2 \cdot \left( 1/2 - \epsilon + \sum_{\tau \in T, t \leq \tau < t+T} C_\tau \right) \leq 3 \). Because we have at most 3 integer
calibrations started within a size-T window, it suffices to round-robin calibrations in
time-order between the three machines.

4.3.5 Partitioning and Converting the LP job schedule

This section shows that the calibration schedule from Section 4.3.4, $C^H$, can be used
to generate a feasible constant-approximation schedule using the fractional job assign-
ments from $x$. For this step, it is easiest to use a job-partitioning strategy based on
the flow of each job in $LP$.

To see why, consider jobs with very large flow in the $LP$. With large flow jobs there
is very little guidance from the LP on where the job should be placed. The algorithm
can potentially move assignments without significantly impacting the overall flow of
the job. Moreover, a large flow job may have assignments spread across uncalibrated
regions, requiring that we move some or all assignments. In this case, the algorithm
uses a strategy of delaying the fractional job assignment in tandem with the original
fractional calibration.

In contrast, low flow jobs are essentially immobile because moving them signifi-
cantly would inflate their flow. However, they also guarantee a nearby calibration in
$C^H$. In this case the algorithm uses a strategy of keeping valid assignments in place
and augmenting the calibration schedule to accommodate.

These strategies potentially conflict, so it’s necessary to solve the partitions sep-
arately. More formally, the algorithm partitions and schedules jobs in $LP$ using the
threshold $T$. Let set $H$ contain all jobs $j$ s.t. $F'_j > T$. Let set $L$ contain all jobs $j$ s.t.
$F'_j \leq T$. 

63
Rounding for High Flow Jobs

This section is devoted to creating a feasible fractional job schedule for high-flow jobs on the calibration schedule from Section 4.2.

Algorithm 4.3 gives the formal strategy for rescheduling jobs from \( H \). It delays each fractional assignment corresponding to the delay of its original calibration. If there is enough remaining volume, it also doubles the assignment by a factor of two. This ensures that flow for each job in \( H \) is increased by no more than a constant factor. Intuitively, it is treating the original fractional calibrations and job assignments from \( C \) as “slices”, then simply “stacking” those slices inside the associated rounded calibration in \( C^H \). It keeps job assignments fixed within each slice, creating a one-to-one mapping. Figure 4.1 gives a graphical example of this intuition. We will see later that doubling the volume keeps flow within a constant factor of optimal.

Algorithm 4.3 Procedure for scheduling fractional job assignments from \( H \) onto calibration schedule \( C^H \)

\[
\text{foreach } j \in H \text{ do} \\
\quad \text{foreach } t \in T \text{ in increasing order do} \\
\quad \quad \text{foreach } x_{jt} \text{ do} \\
\quad \quad \quad \text{cal}_{\text{offset}} = \min_\tau \{ \text{s.t. } C^H_\tau \geq 1 \text{ and } \tau \geq t \} - C_t(x_{jt}) \\
\quad \quad \quad \text{offset} = \text{cal}_{\text{offset}} + x_{jt} - C_t(x_{jt}) \\
\quad \quad \quad \text{set } x_{j, \text{offset}, \text{cal}_{\text{offset}}}^H = \min \{ 1 - X(j, \text{offset} - 1), \ 2x_{jt}C_t \}
\]

Lemma 4.8 begins by showing correctness of the generated fractional schedule.

**Lemma 4.8.** Algorithm 4.3 produces a valid schedule, \( x^H \), of high-flow jobs onto the calibration schedule \( C^H \).

**Proof.** This step shows the feasibility of a job schedule with fractional volumes but fixed job lengths onto a fixed integer calibration schedule. The analysis must therefore prove three properties: every job is fully scheduled; every job fits inside the bounds of
Figure 4.1: Rescheduling fractional high-flow jobs. Example of rescheduling fractional high-flow jobs as defined by Algorithm 4.3. The before image contains the schedule resulting from the LP. Buckets represent calibrations and shaded rectangles represent job assignments with processing time indicated by width. Height represents the size of the fractional assignments. The after image shows the same schedule after Algorithm 4.3 has been applied. Algorithm 4.2 schedules full calibrations after the second and fourth fractional calibration. The dotted line marks the intuitive distinction between slices. The position of jobs inside calibrations are kept the same, resulting in a stacking of slices. Due to doubling assignment size, jobs 4 and 5 are split between the two calibrations, but this is inconsequential because the job schedule is still fractional and the calibrations must be aligned when this occurs.

its release time and assigned calibration; and the total job volume at every timestep is less than the calibration volume.

The first property simply follows from the algorithm structure. It steps through every job and every timestep, doubling the original job assignments until the job is
finished so the job must be fully scheduled. Every job is fully scheduled in $x$ so it follows that every job is fully scheduled in $x^H$.

The second property follows from the aforementioned strategy of stacking calibration slices. Note that $\text{offset} \geq 0$. Since job assignments in $x^H_j$ fall on or after the corresponding job assignment in $x_j$, they clearly come after the job’s release time due to LP constraint (4). Similarly $\text{cal \_ offset} \leq \text{offset} \leq \text{cal \_ offset} + T - p_j$ because $x^H_j$’s position inside the calibration corresponds to a valid job assignment in $x_j$.

The third property also follows from the one-to-one mapping in the slice stacking strategy. From constraint (5), we have that the job volume at every timestep in the slice is less than the size of the calibration. The size of calibrations is doubled in the construction of $C^H$ whereas job volumes are, at most, doubled so the property must hold for $(x^H, C^H)$.

It is worth noting that this step technically breaks the aligned schedule property by delaying jobs relative to timesteps in $C$. However, this property can be easily recovered.

**Lemma 4.9.** $x^H$ can be converted into an aligned schedule with no additional augmentation.

**Proof.** This follows the logic from Lemma 4.2. □

In addition to being feasible, $x^H$ must maintain the flow properties of LP to within a constant factor. It is important to maintain the penalty term during this intermediate representation because the analysis will need to charge against it when resolving the $\alpha$-rounding pseudo-schedule.

**Lemma 4.10.** The LP defined flow time of each job $j \in H$ is no more than $2F^*_{j,2}$.

---

2This is the only step that prevents the algorithm from handling weighted flow (moving weighted assignments can increase the penalty term by much more than $T$). However, it is not immediately clear how to address this issue.
Proof. It suffices to show that the fractional flow term has increased by no more than $T$ and that the penalty term has increased by no more than $3T$. The job is guaranteed to have $F_j' > T$ in $x$ because it was partitioned into $H$ and the two terms are averaged so this yields the desired bound.

To show that the fractional flow term has increased by no more than a 2-factor, we consider the maximum job volume that can be delayed at any one time. Let $\tau$ be the time $t$ in the original schedule $x$ that job $j$ is half scheduled, i.e. $X(j,t) = 1/2$. Algorithm 4.2 guarantees that, at any time $t$, carryover $< 1/2$. Furthermore, the LP guarantees that sum of job assignments from a single job $j$ to a calibration cannot exceed the size of the calibration, and Algorithm 4.3 guarantees that job assignments from $x$ are scheduled in the $C^H$ calibration immediately following their original calibration in $C$ (the rounded calibration containing their slice). It follows that the maximum sum of job assignments from $x$ being delayed at any time $t$ in Algorithm 4.3 is $< 1/2$. This ensure that jobs assignments $x_j, t \leq \tau$ are scheduled no later than $\tau + T$ in $x^H$. Algorithm 4.3 also doubles the size of these assignment until job completion so we are guaranteed that $j$ is fully scheduled in $x^H$ by time $\tau + T$. Therefore the fractional flow of $j$ is increased by no more than $T$ in $x^H$.

To bound the increase in the penalty term, it suffices to examine the worst case. Recall that the non-static portion of the penalty term is determined by $(t + p_k - r_j)x_{kt}$ where $x_k$ is a job assignment (from a larger class) whose execution overlaps with the release time of $j$. $C^H$ allows as many as 3 calibrations at any time $r_j$. Thus the obvious worst case is three jobs with $x_k^H = 1$ and $p_k = T$ all started at $r_j$. The contribution from these jobs to the penalty term would be $3T$.

At this point, we have a feasible constant approximate fractional schedule for high-flow jobs, $x^H$, on the calibration schedule $C^H$. \hfill \Box
Rounding for Low Flow Jobs

This section generates a feasible and constant approximate fractional schedule for low-flow jobs in two steps. First, it generates a new calibration schedule exclusively for low-flow jobs, $C^L$, by augmenting $C^H$. Then, it generates a job schedule on $C^L$ by simply increasing valid job assignments from $x$ by a 2-factor until complete.

Algorithm 4.4 gives the calibration augmentation strategy for generating $C^L$ as well as the simply rounding strategy for scheduling low-flow jobs. It uses a calibration schedule cloned from $C^H$ on three new machines. Then, it uses three more machines to schedule another calibration immediately preceding each of the previous calibrations. This ensures that the machines are adequately calibrated for a rounding of the “early”, $t \leq r_j + t - p_j$, fractional job assignments from each job in $L$.

Algorithm 4.4 Procedure for scheduling fractional job assignments from $L$ onto an augmented mirror of the calibration schedule from $C^H$ using three additional machines

Let $C^L = clone(C^H)$;  // new set of calibrations for low-flow jobs

For each calibration, $C_t$, in $C^H$ do

- add a calibration at time $\min\{0, t - T\}$ to $C^L$

Set $x^L_{jt} = \min\{1 - X(j,t - 1), 2 \times \sum_{C_t(x_{jtC_t})}\} \quad \forall t, j \in L$

First we will see that the new calibration schedule $C^L$ is valid.

Lemma 4.11. Algorithm 4.4 produces a valid calibration schedule, $C^L$ on six machines.

Proof. Calibrations with $t \neq 0$ must all be valid since they simply offset the original schedule by $-T$ and it is possible to place them on the three additional machines. These cannot conflict with calibrations at $t = 0$ because Algorithm 4.2 guarantees that the total number of calibrations in the time range $[0, T]$ in $C^H$ is at most 3.

The next step is to prove that the job schedule is feasible by showing three key properties of the calibration schedule combine with the new low-flow job assignment.
values: that there is sufficient early job volume for each job in $x$, that these timesteps are all calibrated, and that the calibration volume in $C^L$ is sufficient for the job volume in $x^L$ at every timestep.

Lemma 4.12. $X(j, r_j + T) > \frac{1}{2}$  \( \forall j \in L \). That is, jobs in $L$ must be at least half-complete before $T$ time-steps after their release.

Proof. By definition of a low-flow job, we are guaranteed that, for all $j \in L$, $\sum_t (t + p_j - r_j) x_{jt} \leq T$. Therefore, more than half the job must be finished before time $r_j + T$.

Lemma 4.13. There is a valid active calibration for every $x^L_{jt} \neq 0$ s.t. $t \leq r_j + T$.

Proof. From Lemma 4.12, more than half the job must be scheduled between $[r_j, r_j + T]$. Since more than half a job is scheduled in the range $[r_j, r_j + T]$, the structure of Algorithm 4.2 guarantees that there is a rounded calibration at some time step in that range. In addition Algorithm 4.4 guarantees another corresponding calibration offset by $-T$. Therefore, there must be active calibrations over the entire range $[r_j, r_j + T]$.

Lemma 4.14. $\sum_j \sum_{\tau: \tau \in (t-p_j, t]} x^L_{j\tau C_t} \leq C^L_t \forall t \in T, C_t \in C$. That is, every timestep is sufficiently calibrated.

Proof. Lemma 4.13 ensures that there is valid calibration for every $x^L_{jt} \neq 0$ so it only remains to address timesteps, $t$, for which $\sum_j \sum_{\tau: \tau \in (t-p_j, t]} x^L_{j\tau C_t} > 1$. Algorithm 4.4 increases assignments from $x$ by at most a 2-factor so this can only be true when $\sum_j \sum_{\tau: \tau \in (t-p_j, t]} x^L_{j\tau C_t} > 1/2$ and the sum from $x^L$ is $\leq 2$. Algorithm 4.2 guarantees $C^H = 2$ (and consequently $C^L = 2$) for every timestep where this is true.

Next, we see that the LP-defined fractional flow properties have been maintained to within a constant-factor. The fractional flow of jobs in $L$ is guaranteed to be less
than or equal to $x$ while the penalty term is guaranteed to be within a 2-factor of $x$. The LP flow definition averages these two so the combination is guaranteed to be within a 1.5-factor.

**Lemma 4.15.** The fractional flow of each job $j \in L$ is $\leq \sum_t (t + p_j - r_j) x_{jt}$.

**Proof.** As a result of the progress guarantee from Lemma 4.12 combined with the volume doubling from Algorithm 4.4, we are guaranteed that the each job in $L$ is completed in $x^L$ by the time half of its volume is completed in $x$. Their fractional flow is therefore no more than their fractional flow in $x$. $\square$

**Lemma 4.16.**

$$\sum_{k: G_k^{-1} > G_j^{-1}} \sum_{t \in [r_j - p_k + 1, r_j]} (t + p_k - r_j) x_{kt} \leq 2 \left( \sum_{k: G_k^{-1} > G_j^{-1}} \sum_{t \in [r_j - p_k + 1, r_j]} (t + p_k - r_j) x_{kt} \right)$$

**Proof.** This follows directly from the observation that Algorithm 4.4 increases any value $x_{kt}$ by at most a 2-factor. $\square$

### 4.3.6 Generating a Single-Machine Randomized Pseudo-Schedule

This section shows how to construct a **pseudo-schedule** for each partition, that is, a complete scheduling of all jobs that is not yet valid because it may result in multiple jobs running at the same time on a single machine. The procedure given in Algorithm 4.5 constructs these schedules by applying Bansal et al.’s $\alpha$-rounding strategy, adjusted for multiple machines, to each partition independently. Let $x^m$ be the partition being scheduled and $m$ be the number of machines.

The resulting pseudo-schedule has the following properties. It may not be immediately obvious that these combined properties are useful. However, they will be leveraged in Section 4.3.7 to generate and prove a feasible randomized integer schedule.
Algorithm 4.5 Procedure for generating a psuedo-schedule from $x^m$

let $\alpha = \text{rand}(0, 1)$

for each class $G_i$ do
  for $h = 0$; unscheduled jobs in $G_i$; $h = h + 1$ do
    let $t_h = \min_t \mid X_i(t) > h + \alpha \land G_i^* > 0$
    let $j = \min_j \mid t_h > r_j$ ; // based on total ordering
    schedule $j$ at time $t_h$

Lemma 4.17. Each job $j \in J$ is scheduled in $x^\alpha$ exactly once.

Proof. This follows directly from Algorithm 4.5.

Lemma 4.18. Each job $j \in x^\alpha$ is scheduled on a calibrated time segment.

Proof. Combining Lemma 4.8 and Lemma 4.13, we know that every fractional job assignment from $x^m$ is scheduled on a calibrated time-segment. Algorithm 4.5 only schedules a new job from any particular class when the total calibrated volume for that class passes a threshold. This can only happen on a timestep with a job assignment from that class in $x^m$. Therefore the job scheduled in $x^\alpha$ must also be on a calibrated time segment.

Lemma 4.19. (Lemma 6.2 in [7]) No more than $m$ jobs from class $G_i$ are scheduled in the same $\beta^i$-interval.

Proof. The total class $G_i$ job volume scheduled on a single machine during a $\beta^i$-interval in $LP$ is at most $\beta^i$. Algorithm 4.5 schedules a new job from class $G_i$ only after the cumulative value of $X_i(t)$ has increased by 1, requiring a scheduled job volume of at least $\beta^i$, which can’t occur more than $m - 1$ times during a single $\beta^i$-interval.
Lemma 4.20. (Lemma 6.3 in [7]) Consider any \( \beta^i \)-interval for a non-empty class \( G_i \), the total size of all the jobs scheduled to that interval from classes \([G_0, G_i]\) is at most \( m(\beta^i + \frac{\beta^{i+1}-1}{\beta - 1}) < m\beta^i(2 + \frac{1}{\beta - 1}) \).

Proof. From Lemma 4.19, the total volume of new jobs scheduled during \( \beta^i \) cannot exceed \( m\beta^i \). However, there are at most \( m \) previously scheduled jobs from each class \([G_0, G_i]\) overlapping with the interval, giving a total volume of at most \( m(\beta^i + \sum_{j=0 \rightarrow i} \beta^k) = m(\frac{\beta^{i+1}-1}{\beta - 1} + \beta^i) < m\beta^i(2 + \frac{1}{\beta - 1}) \). \( \square \)

Lemma 4.21. (Lemma 7 in [7]) For job \( j \in J \), the expected flow time of \( j \) in the pseudo-schedule is \( \sum_t (t + p_j - r_j)x_{jt}^m \).

Proof. Since \( \alpha \in [0, 1) \) is chosen uniformly at random, \( j \)'s chance of being schedule at any time-step, \( t \) is directly proportional to \( x_{jt}^m \). Therefore, its expected flow time is \( \sum_t (t + p_j - r_j)x_{jt}^m \). \( \square \)

4.3.7 Converting the Pseudo-Schedule into a Feasible Solution

The previous step clearly creates conflicts in an actual schedule since multiple jobs may be scheduled at the same time. This section converts the pseudo-schedule into a feasible randomized integer schedule with speed augmentation. The main strategy is to represent conflicts in the \( \alpha \)-rounding schedule using \( \beta \)-ary trees where each new level contains a smaller job class. Each node contains up to \( m \) jobs corresponding to the \( \leq m \) jobs from each class guaranteed by Lemma 4.19. It is then possible to systematically resolve conflicts using tree traversals to decide ordering.

We call a job maximal if, in the \( \alpha \)-rounding schedule, it does not conflict with any jobs from a larger class. Note that a maximal job is not necessarily unique since we have as many as \( m \) jobs from a class for every aligned interval. The algorithm constructs a forest of conflict trees with a root for every set of maximal jobs. The
procedure for generating a tree from a maximal job set is given in Algorithm 4.6. However, the procedure flows naturally from the above intuition. Figure 4.2 gives a graphical example of a conflict tree.

Based on this forest, Algorithm 4.7 reschedules conflicting jobs. Let job $j$ be the maximal root of the tree scheduled on the interval $I = [\tau, \tau + \beta_i)$. Another partition is needed here to distinguish between early descendants, those released before time $\tau$ (i.e. $j_k$ s.t. $r_k < \tau$), and late descendants, those released inside the interval $I$ (i.e. $j_l$ s.t. $\tau \leq j_l$).

Early jobs may be scheduled anywhere in the interval and the term $(\tau + p_j - r_k)x_{j_k}^m$ does not impact its LP-defined flow. In contrast, late jobs are restricted by their release time and their flow is impacted by the term $(\tau + p_j - r_l)x_{j_l}^m$.
Algorithm 4.6 Procedure for generating a tree from a set of maximal jobs

input maximals
node root = maximals
let start = start(maximals[0])
let interval_step = size(maximals[0]) / \beta

build_children (root, start, interval_step)

if interval_step < 1 then
  return

for i = 0; i < \beta; i + + do

  node child = \{size interval_step jobs scheduled at start + i \cdot interval_step\}

  build_children(child, start + i \cdot interval_step, interval_step / \beta)

  root.children[i] = child

If a node contains more than one job in a partition, the algorithm schedules the competing jobs from that node in arbitrary order.

We start by seeing that \( x^I \) is a valid schedule of job assigned to the interval, within the interval bounds. Following from the structure of the algorithm, it is sufficient to show that every job remains within its \( \alpha \)-rounding bounds and that the two partitions do not conflict in the center of the interval.

Lemma 4.22. (Lemma 8.1 in [7]) No two jobs in \( x^I \) overlap.

Proof. Lemma 4.20 guarantees that the original job volume in \( T_I \) < \( m\beta^i \left( 2 + \frac{1}{\beta-1} \right) \). The interval is \( \beta^i \) long and the algorithm sped up all processing times by a factor of \( m \left( 2 + \frac{1}{\beta-1} \right) \). Thus, jobs scheduled consecutively from either side of the interval cannot meet in the center causing an overlap without violating the total job volume guarantee.

Lemma 4.23. (Lemma 8.2 in [7]) Every early job \( j \) starts no earlier than \( \tau \) and no later than its pseudo-schedule completion time, \( X^p_{jt} + p_j \).
Algorithm 4.7 Procedure for generating a schedule from a conflict tree

input tree $T_I$ ; // Input conflict tree

**foreach** $j \in T_I$ **do**

\[ p_j = \frac{p_j}{m(2 + \frac{1}{m-1})} ; \] // shrink all job sizes by the speedup factor

let early = \emptyset ; // set of early jobs
let late = \emptyset ; // set of late jobs

PARTITION($T_I$, early, late, start($I$))

let $index = 0$ ; // time index
let $x_{I_jt}^I = 0 \forall j, t \in T_I$ ; // new job schedule

// schedule early jobs consecutively in post-order, // left-justified
**foreach** $j \in \text{POSTORDER}(early)$ **do**

schedule $x_{j, index}^I = 1$

\[ index = index + p_j \]

let $index = \text{end}(T_I) - 1$

// schedule early jobs consecutively in pre-order, // right-justified

**foreach** $j \in \text{PREORDER}(late)$ **do**

\[ index = index - p_j \]

schedule $x_{j, index}^I$

output $x^I$
Proof. By algorithm 4.7, the first job starts at $\tau$ and execution order up to and including $j$ is determined by POSTORDER($j$). It includes as many as $m$ sub-trees. Using the logic from Lemma 4.20, the job volume in each sub-tree, before speedup, is less than $p_j \left( 2 + \frac{1}{\beta-1} \right)$. Thus, the total job volume, up to and including $j$, is $< mp_j \left( 2 + \frac{1}{\beta-1} \right)$. Early jobs are scheduled consecutively and all jobs are sped up by $m \left( 2 + \frac{1}{\beta-1} \right)$ so $j$ must finish by $\tau + p_j \leq X_{jt}^p + p_j$. \hfill $\Box$

**Lemma 4.24.** (Lemma 8.3 in [7]) Every late job $j$ starts no earlier than $X_{jt}^p$ and no finishes no later than $\tau + \beta^i$.

Proof. Symmetric with Lemma 4.23, the first job from PREORDER starts at $\tau + \beta^i - p_j$ and all subsequent jobs are scheduled consecutively in decreasing time order. The total job volume from PREORDER($j$) is $< mp_j \left( 2 + \frac{1}{\beta-1} \right)$. Thus, after speedup, $j$ must start no earlier than $X_{jt}^p$. \hfill $\Box$

Next, we see that merging these conflict schedules produces a feasible and complete overall schedule.

**Lemma 4.25.** Merging all conflict schedules, $X^I$, yields a feasible and complete schedule $X^f$.

Proof. Every job belongs to a conflict tree and every conflict tree is scheduled on its original interval. The feasibility of each interval schedule is shown by the previous lemmas and conflict trees do not conflict with each other by definition of a maximal node. The schedule resulting from merging all conflict schedules must therefore be complete and feasible. \hfill $\Box$

Finally, we see that the resulting overall schedule is still approximate to within a constant factor.
Lemma 4.26. (Lemma 9 in [7]) The expected total flow-time of $X^f$ is at most $2 \sum_{j \in J} F_j' \leq 2 \cdot F(OPT(J))$.

Proof. From Lemma 4.3 and Lemma 4.23, we can easily say that early jobs contribute less flow than their counterparts in $x^m$. We will therefore focus on bounding the contribution from late jobs.

Consider a job $j$. Let $F_j^p$ be its flow time in the psuedo-schedule and $F_j^f$ be its flow time in the final schedule. Recall that the psuedo-schedule, and consequently the final schedule, were both generated based on the randomly chosen $\alpha$ so we can consider these both random variables and say that,

$$E\left[F_j^f\right] = E\left[F_j^p\right] + E\left[F_j^f - F_j^p\right]$$

Denote the event that $j$ is a late job with $E_j$. We can say that,

$$E\left[F_j^f\right] = E\left[F_j^p\right] + Pr\left[E_j\right] \cdot E\left[F_j^f - F_j^p \mid E_j\right] + Pr\left[\neg E_j\right] \cdot E\left[F_j^f - F_j^p \mid \neg E_j\right]$$

Furthermore, it is possible to substitute the first term with the bound from Lemma 4.21 and drop the third term in favor of an inequality. More specifically, we know from Lemma 4.23 that the $\left[F_j^f - F_j^p \mid \neg E_j\right] \leq 0$. These changes yield the inequality,

$$E\left[F_j^f\right] \leq \sum_t (t + p_j - r_j) x_{jt}^m + Pr\left[E_j\right] \cdot E\left[F_j^f - F_j^p \mid E_j\right]$$

We can break this inequality down further by considering the interval on which a late job $j$ is scheduled. Let $k$ be the maximal node in $j$ with start time $\tau$. Because
is late, we can say \( r_j \geq \tau \) and that \( x^f_j \) is in the range \([r_j, \tau + p_k] \). Thus, \( \Pr \left[ E_j \right] \leq \sum_{\tau: \tau \in [r_j - p_k + 1, r_j]} x_{k\tau}^m \). Therefore \( j \)'s expected flow time is,

\[
\mathbb{E} \left[ F^f_j \right] \leq \sum_{\tau: \tau \in [r_j - p_k + 1, r_j]} (\tau + p_k - r_j)x_{k\tau}^m
\]

This is at most twice the contribution of job \( k \) to the flow time of \( j \) in \( LP \), corresponding to the late penalty term,

\[
\sum_{k: G^{-1}_k > G^{-1}_j} \sum_{t \in [r_j - p_k + 1, r_j]} (t + p_k - r_j)x_{kt}^m
\]

The contribution of early jobs to \( F^f \) does not increase relative to \( F' \) and the contribution of late jobs increases by at most a factor of 2. Therefore \( F^f \leq 2F' \)

\[\square\]

**Lemma 4.27.** Applying Algorithms 4.5, 4.6, and 4.7 to \((x^H, c^H)\) and \((x^L, C^L)\) produces a valid randomized 2-machine 16-approximation schedule using 72-speedup.

**Proof.** Recall that, during the full process to generate a schedule for high-flow jobs, the algorithms used a speedup of \( 2\beta \) to generate an aligned schedule, then 3 machines and 3 OPT calibrations to generate a rounded calibration schedule. Fractional job rounding onto the calibration schedule cost a 2-factor increase in flow. Finally, alpha-rounding and conflict resolution cost a \( 3 \left( 2 + \frac{1}{\beta - 1} \right) \)-factor increase in speed and a 2-factor increase in flow, but collapsed the schedule back to a single machine. Combined, the cost is \( 6\beta \left( 2 + \frac{1}{\beta - 1} \right) \)-speed, 3 OPT-calibrations, and 4 OPT-flow.

During the full process to generate a schedule for low-flow jobs, the algorithms used a speedup of \( 2\beta \) to generate an aligned schedule, then 6 machines and 6 OPT calibrations to generate a rounded calibration schedule. Fractional job rounding onto
the calibration schedule caused a 1.5-factor increase in the flow. Subsequently, alpha-rounding and conflict resolution cost a $6 \left(2 + \frac{1}{\beta - 1}\right)$-factor increase in speed and a 2-factor increase in flow, again collapsing the schedule onto a single machine. This gives a combined cost of $12\beta \left(2 + \frac{1}{\beta - 1}\right)$-speed, 6 OPT-calibrations, and 3 OPT-flow. Noting that $\beta = 2$ is optimal and combining these two schedules yields a 2-machine 72-speed 16-approximation schedule. \hfill $\square$

4.3.8 Derandomizing the schedule in polynomial time

**Lemma 4.28.** (Section 2.6 in [7]) $LP^f$ can be derandomized in polynomial time with a $(1 + o(1))$-factor increase in flow.

**Proof.** Varying $\alpha$ over $[0, 1]$ produces $O(n)$ changes to class-$i$ jobs in any given $\beta^i$-interval. Therefore there are only $O(n \times m)$ relevant values of $\alpha$ to consider where $m$ is the number of variables in $LP$. Since there are polynomial relevant value of $\alpha$ it is possible to find a feasible value via brute force. \hfill $\square$

Combined with Lemma 4.3.7, this proves the stated 2-machine 72-speed 17-approximation schedule.

4.4 Multi-Machine ISE-flow

This section provides a brief algorithm and proof for multi-machine non-unit ISE-flow by reduction to multiple instances of unit ISE-flow and application of the multi-machine unit ISE-flow solution from McCauley [57]. Specifically, it proves the following theorem.

**Theorem 4.29.** There exists a 4-speed $\lceil \log n \rceil$-machine $12 \lceil \log n \rceil$-approximation polynomial-time algorithm for $m | r_j | H(\#Calibrations) + \sum F_j$. 

79
The algorithm is as follows. First, round job sizes and restrict them to an aligned schedule as in Section 4.3.2, letting \( \beta = 2 \). Furthermore, delay all release times until the next aligned timestep. Second, partition each job class onto a separate set of \( m \) machines. Finally, solve each partition independently using the 12-approximation multi-machine unit-length algorithm from McCauley [57] and union the results.

**Proof.** We have from Lemma 4.2 that a rounded aligned schedule lower bounds the optimal solution. Delaying release times until the next available timestep imposes no further restrictions because the job may not be scheduled until that timestep per the definition of an aligned schedule. As before, the cost for this process is a speed factor of 4.

Per the rounding procedure, the following partition produces at most \( \lceil \lg n \rceil \) partitions using at most \( m \lceil \lg n \rceil \) machines. Each partition contains only jobs with uniform size. An ISE-flow instance with uniform job sizes, an alignment constraint, and aligned release times is functionally equivalent to a unit ISE-flow instance. Therefore, it’s possible solve each partition independently by applying the 12-approximation multi-machine unit-length algorithm from McCauley [57]. Each partition solution is a 12-approximation so their union gives us a \( 12 \lceil \lg n \rceil \)-approximation for the entire instance. The resulting solution is a 4-speed \( 12 \lceil \lg n \rceil \)-approximation polynomial-time algorithm on \( m \lceil \lg n \rceil \) machines.

This algorithm is a relatively simple adaptation of McCauley’s unit-length result and the bound is not very exciting, but it seems difficult to improve upon given known multi-machine flow minimization bounds.
4.5 Conclusions

This chapter offers an $O(1)$-machine $O(1)$-speed $O(1)$-approximation polynomial-time algorithm for single machine non-unit ISE-flow. This matches the best known solution for the single-machine interval scheduling problem to reduce total-flow time that it generalizes and should be difficult to improve upon asymptotically.

Similar to Chapter 3, this algorithm requires a deliberate partition between high flow jobs that can be moved to improve batching with little penalty and low flow jobs which cannot. This partitioning seems to be somewhat inherent to non-unit versions of the ISE problem. The algorithm serves as another example of this dissertation’s overall observation that deliberate balancing of batching against other scheduling concerns can be useful in obtaining good theoretical bounds.

For completeness, the chapter also offers an $O(\log n)$-machine $O(1)$-speed $O(\log n)$-approximation polynomial-time algorithm for multi-machine non-unit ISE-flow. This is the first known algorithm for the problem.
We next focus on the online problem of scheduling a dynamically multi-threaded computation utilizing implicit batching directives for parallel data-structures. This chapter uses a standard application of batching so the benefits are more obvious. However, due to inclusion in a larger computation, it is still important to appropriately balance batched operation in the data-structure against potential impact on the overall makespan.

The work in this chapter was particularly motivated by the problem of *series-parallel maintenance*. That is, determining whether two memory accesses in a series-parallel program are logically parallel, a central concept in race-detection. There are no known concurrent data-structures that would support an asymptotically optimal race-detector and such a construct is likely impossible. Conversely, there is no natural way to incorporate an appropriate parallel batched data-structure into a fork-join programming model. Instead, this work added an additional feature to the standard Cilk programming model to better support the use-case. A BATCHER style construct eventually facilitated the creation of an asymptotically optimal race-detector for Cilk, developed by Utterback et al. [74].

A common approach when using data structures within parallel programs is to employ *concurrent data structures* — data structures that can cope with multiple simultaneous accesses. Not only is it challenging to design and analyze concurrent data structures, but the existing performance theorems do not often imply linear speedup.
for the enclosing program. The straightforward way to analyze a program that uses a concurrent data structure is to assume the worst-case latency for each access. For a limited set of concurrent data structures (see Section 5.2), the worst-case latency is low enough that this approach is effective. In more general cases, the worst-case latency is often linear in the number of processors in the system (or worse), e.g., for Braginsky and Petrank’s lock-free B+-tree [18].\(^1\) If \(n\) data-structure accesses each keep a processor busy for \(\Omega(P)\) timesteps, then the running time on \(P\) processors is at least \(\Omega(nP/P) = \Omega(n)\). An \(\Omega(n)\) bound means that accesses are essentially sequential — there is no significant speedup guarantee when running on \(P\) processors.

Concurrent data structures are in some sense overkill for use within a single parallel program because they are designed to cope with difficult access patterns. Since the data-structure accesses belong to the same enclosing program, they can, in principle, coordinate with each other. A key idea behind this work is to leverage a runtime scheduler and to handle this coordination.

The primary goals are to 1) describe a good scheduler for executing a broad class of parallel programs that make parallel accesses to data structures, and 2) provide a corresponding general performance theorem for this scheduler. The performance theorem exhibits an attractive modularity: the data structure may be analyzed in isolation of the program that uses it, and the parallel program may be analyzed without considering the specific implementation of the data structure. This modularity makes the theorem easy to apply while still achieving provably good speedup, e.g., for \(n\) parallel accesses to a search tree with large enough \(n\), this scheduling the-

\(^1\)The worst case occurs when all processors concurrently insert contiguous keys. The progress bound proven [18] is worse, since the data structure is designed to cope with processor failures. But even assuming no failures and simplifying the data structure, an \(\Omega(P)\) worst-case bound still occurs when \(P\) processes attempt a compare-and-swap on the same node in the tree.
orem proves a completion time of $\Theta(n \log n / P)$, which is asymptotically optimal and has linear speedup. There are no known comparable aggregate bounds for concurrent search trees.

5.1 Contributions

This work, published in SPAA 2014 [1], shows how to achieve a modular analytic abstraction whereby the data structure $D$ and the enclosing program $C$ may be analyzed separately, then combined through a strong performance theorem guaranteeing good parallel running time. This result applies to the broad class of dynamically multi-threaded computations. The runtime theorem is achieved through the novel technique of implicit batching coupled with an efficient runtime scheduler.

5.1.1 Implicit batching

This chapter focuses on the novel technique of *implicit batching*, which achieves benefits of both concurrent and batched data structures. In implicit batching, the programmer provides two components: (1) a parallel program $C$ containing parallel accesses to an abstract data type $D$, and (2) a batched data structure implementing the data structure $D$. The scheduler dynamically and transparently organizes the program’s parallel accesses to the data structure into batches, with at most one batch executing at a time.

5.1.2 Scheduler and performance theorem

Implicit batching poses its own challenges for performance analysis. For a parallel program using implicit batching, which sequence of batches should be analyzed? What overhead does the scheduler incur when creating batches? In general, the performance
of a parallel program using implicit batching depends on the particular runtime scheduler used to execute the program.

To yield a performance theorem, we will analyze BATCHER, a work-stealing scheduler designed for implicit batching. Given a dynamically multithreaded program C that makes parallel accesses to an ADT D, and a batched implementation of D, it yields the following bound.

**Theorem 5.1.** Consider a dynamically multithreaded program C having $T_1$ work and $T_\infty$ span. Let $n$ be the total number of data-structure operations (accesses to ADT D), and $m$ be the maximum number of data-structure operations along any sequential dependency chain. For the given implementation of D, let $W(n)$ be the worst-case total work for $n$ data-structure operations grouped arbitrarily into batches, and let $s(n)$ be the worst-case span of a parallel size-$P$ batched operation.\(^2\) Then the expected running time of this program on $P$ processors using BATCHER\(^3\) is at most

$$O\left(\frac{T_1}{P} + \frac{W(n) + ns(n)}{P} + ms(n)\right).$$

An important feature about this bound is that $T_1$ and $T_\infty$ are the work and span of the core program C, independent of the data structure implementation. Similarly, $n$ and $m$ count the data-structure calls in the program, and depend only on the program C, not the data structure implementation. Moreover, $s(n)$ and $W(n)$ are performance measures of the batched implementation D of the data structure, independent of the enclosing program. Thus the bound is essentially adding the program’s cost to the data structure’s cost. This bound also applies when the analysis of the batched data structure is amortized, through a more general definition of $s(n)$. For many parallel

\(^2\)Only binary forking is employed, so $s(n) \geq \log P$ implicitly.

\(^3\)Just as in standard work-stealing results, the theoretical bounds assume that the only synchronization of the input algorithm occurs through “syncs” or “joins”; the algorithm or data structure code itself does not use explicit synchronization primitives, e.g., locks or compare-and-swaps.
batched data structures, this performance theorem implies nearly linear speedup. This scaling was experimentally verified using a modified Cilk-5 runtime [37] to implement a skip-list data structure [64].

5.2 Related Work

Implicit batching closely resembles flat combining [43], where each concurrent access to a data-structure queues up in a list of operation records, and this list of records (i.e., a batch) is later executed sequentially. Implicit batching may be viewed as a generalization of flat combining in that it allows parallel implementations of batched operations, instead of only a sequential one allowed by flat combining. Due to sequential batches, flat combining does not guarantee provable speedup guarantees. However, flat combining has been shown to be more efficient in practice than some of the best concurrent data structures under certain loads. Viewing flat combining as a specific implementation of implicit batching already shows the practical effectiveness of implicit batching — this work focuses on obtaining a provably good runtime theorem.

5.2.1 Runtime scheduling

This work focuses on parallel programs expressed through dynamic multithreading as described in Section 2.2.2. Recall that a parallel program (without parallel data structure accesses) having $T_1$ work — the running time on 1 processor, and $T_\infty$ span — the length of the critical path, can be executed in $O(T_1/P + T_\infty)$ (expected) time on $P$ processors (or workers) using a work-stealing scheduler.

This work generalizes the above result by describing a scheduler and corresponding performance theorem for dynamic multithreaded programs that make parallel accesses to a data structure. This performance theorem implies that parallel programs (with
sufficient parallelism) using efficient data-structure implementations can execute with linear speedup, even if the program’s work is dominated by data-structure accesses.

One notable feature of BATCHER and its analysis is the use of two distinct deques (per worker) to switch between different types of work. Similar techniques have been used in several work-stealing augmentations since BATCHER’s original publication. Latency-hiding work-stealing, allowing a user-level thread to suspend on high-latency operations without blocking the underlying work, was accomplished using a dynamic number of deques to hold suspended operations [59]. WSP-Order [74] used a closely related two-deque technique to solve the series-parallel maintenance problem in asymptotically optimal expected time. PORRidge [75] used a similar technique to provide processors-oblivious record-and-replay functionality for data-race free Cilk programs.

5.2.2 Concurrent data structures

Concurrent data structures themselves are widely studied [46]. Most theoretical work on concurrent data structures focuses on correctness and forward-progress guarantees like linearizability [47], lock freedom [45], or wait freedom [44]. While wait-free structures often include a worst-case performance bound, the bound may not be satisfying when applied in the context of an enclosing algorithm. For example, a universal wait-free construction of Chuong et al. [25] has a worst-case cost that includes a factor of $P$, the number of processors, which implies serializing all data structure operations. Experimental studies of various concurrent B-tree data structures alone span over 30 years of research [12, 14, 18, 49]. These results typically fall short of bounds on running time, with Bender et al. [14] being one exception assuming uniformly random accesses.
BATCHER most closely resembles various software combining techniques, designed primarily to reduce concurrency overhead in concurrent data structures. In some combining techniques [33, 43, 60], each processor inserts a request in a shared queue and a single processor sequentially executes all outstanding requests later. These works provide empirical efficiency, but there is no known theory bounding the running time of an algorithm using these combiners. BATCHER improves upon these techniques by operating on the “request queue” in parallel and by providing runtime theory. Other software-combining techniques include (static) combining trees [39] or (dynamic) combining funnels [70] which apply directly to data structures with combinable operations like lock objects, counters, or stacks. These do have a provably $O(\lg P)$ overhead, but do not address more general structures.

Several related mechanisms designed for dynamic multithreading have a grounding in theory. Reducers [36] in Cilk can be used to eliminate contention on some shared global variables, but are not designed to replace a generic concurrent data structure, since they create local views on each processor rather than maintain a single global view. It is also unclear how to analyze reducers that include highly variable amortized costs. Helper locks [2] provide a mechanism that allows blocked workers to help complete the critical section that is blocking them and is not specifically designed for data structures. Conceptually, one can use this mechanism to execute batches; however, directly applying the analysis of [2] leads to worse completion time bounds compared to using BATCHER.

5.2.3 Batched data structures

One way to programmatically coordinate data-structure accesses is to use only \textbf{batched data structures}, where the program invokes an entire set of data-structure operations synchronously, and the batched data structure performs these operations
collectively in parallel. The main advantages of batched data structures are: (1) only one batch is active at a time, obviating the need for complicated concurrency control within the data structure, (2) parallelism may be used to accelerate individual batches, and (3) they are relatively easy to analyze; if a program generates a sequence of batches, it is possible to simply add the running time of this sequence of batches to the running time of the program. For example, (batched) parallel priority queues \cite{20,29,31,68} have been utilized to prove efficient running time on parallel algorithms such as shortest paths and minimum spanning tree \cite{20,31,61}.

Several batched search trees exist, including 2-3 trees \cite{62}, weight-balanced B-trees \cite{32}, and red-black trees \cite{35}. Moreover, some of these data structures \cite{32,35} exhibit good practical performance.

Getting provably good performance by replacing a concurrent data structure with a batched data structure can require drastic code restructuring, however, since the parallel program must explicitly group accesses into batches. In some cases, such a restructuring may not even be possible. For example, an on-the-fly race detector \cite{15,58,66} updates a series-parallel-maintenance data structure on forks and joins while executing an input program. In this application, the data structure must be updated before the program flow continues past the calling point, so it seems impossible to reorganize the operations into batches by restructuring the algorithm.

Using implicit batching gives the benefits of batched data structures without restructuring the enclosing program \(C\). The scheduler is responsible for grouping any concurrent accesses to the abstract data type \(D\) into batches and invoking the appropriate implementation of a batched operation. The data structure’s batched operation may be implemented using dynamic multithreading. The data-structure’s implementation need not cope with concurrency since at most one batch is executing at any time, and hence locks or atomic operations may be omitted. The scheduler handles
all synchronization and communication between the parallel program $C$ and the data structure $D$.

5.3 Definitions and Analytic Model

Recall that a programmer provides two inputs to the BATCHER scheduler: (1) A parallel program $C$ that makes parallel accesses to an abstract data type $D$, and (2) a batched data structure that implements $D$ and need only support one batch at a time. This section defines how the parallel program and the batched data structure are modeled.

5.3.1 Execution dag model

In the absence of data-structure operations, the execution of a dynamically multi-threaded computation can be modeled as a directed acyclic graph (dag) that unfolds dynamically (see [28, Ch. 27]). In this execution dag, each node represents a unit-time sequential subcomputation, and each edge represents control-flow dependencies between nodes. A node that corresponds to a “fork” has two\(^4\) outgoing edges, and a node corresponding to a “join” has two or more incoming edges.

A scheduler is responsible for choosing which nodes to execute on each processor during each timestep. The scheduler may only execute **ready nodes** — those unexecuted nodes whose predecessors have all been executed. The convenient feature about the computation dag is that it models the control-flow constraints within the program without capturing the specific choices made by the scheduler. The dag unfolds dynamically — only the immediate successors of executed nodes are revealed to the scheduler. This unfolding can also be nondeterministic. Hence the scheduler must make online

\(^4\)In general, forks may have arbitrary out-degree, but this work pessimistically assumes binary forking.
decisions. All of the analyses are with respect to the *a posteriori* dag. The two key features of a dag are its *work*, which is the number of (unit-length) nodes in the dag, and its *span*, which is the length of the longest directed path through the dag.

### 5.3.2 Extending the dag model to implicit batching

We first model the batched data structure that implements \( \mathcal{D} \). An implementation of a batched operation is itself a parallel (sub)computation that may include forks and joins. We thus model the execution of each batch \( A \) by its own *batch dag* \( G_A \). We use the terms *batch work*, denoted by \( w_A \), and *batch span*, denoted by \( s_A \), to refer to the work and span of the batch dag, respectively.

To analyze a batched data structure as a whole, we consider worst-case sequences of arbitrary batches, such that the total number of data structure operations across all batches is \( n \), and each batch contains at most \( P \) data structure operations. We define the *data-structure work*, denoted by \( W_P(n) \), to be the maximum total work of any such sequence of batches. We also define the *data-structure span*, denoted by \( s_P(n) \), to be the worst-case span of any batch dag \( A \) in any such sequence subject to the restrictions that \( w_A / s_A = O(P) \), meaning that the batch has limited parallelism. In the case when the data structure’s analysis is not amortized, the data-structure span may be stated more concisely as the worst-case span of any batch dag that represents a size-\( P \) batch, since all batches of the same size have the same span. For data structures with amortized analysis, however, batches with the same number of operations may have different spans — therefore, the batch span is defined in terms of the parallelism of the batch dag rather than the number of operations in the batch. Since \( P \) (the number of workers) is static throughout this paper, we use \( W(n) \) and \( s(n) \) as shorthands for \( W_P(n) \) and \( s_P(n) \). Note that whereas data-structure work corresponds to the total work of all batches that cumulatively contain \( n \) operations,
Thus far we have not considered a program that makes accesses to the data structure, we have only considered the data structure implementation. It should thus be clear that \( W(n) \) and \( s(n) \) are metrics of the data structure implementation itself.

This work models the enclosing program \( C \), which makes parallel calls to a data structure (these operations will be implicitly batched), as another kind of dag, called the core dag \( G \). A core dag is just like a standard execution dag, except that it includes two kinds of nodes. Each data-structure operation (that is to be implicitly batched) is represented by a special data-structure node. All other non-data-structure nodes in the dag are called core nodes. Whereas all core nodes by definition take unit-time to execute on a processor, the data-structure nodes represent blocking calls that may take longer to complete. Our metrics for the core dag, however, avoid this issue — we define the core work, which we generally denote by \( T_1 \), to be the number of nodes in the core dag, and the core span, denoted by \( T_\infty \), to be the longest path through the core dag in terms of number of nodes. We also generally use \( n \) to refer to the number of data-structure nodes in the core dag, and \( m \) to denote the maximum number of data-structure nodes falling along any directed path through \( G \). Although the core dag includes data-structure nodes whose “execution times” are not defined, the metrics \( T_1, T_\infty, n, \) and \( m \) are functions of only the core program, not the implementation of batched operations.

5.3.3 No extra dependencies between data-structure nodes

It may be surprising that modeling the core dag and batch dags separately as described throughout this section can be sufficient for the analysis of any scheduler. A priori, one might expect execution-dependent “happens-before” relationships across all data-structure calls, particularly since the scheduler must group operations into
batches. Moreover, one might be surprised that the “length” of data-structure nodes is not modeled anywhere. Nevertheless, Section 5.6 proves that this simple model is sufficient for the BATCHER scheduler, which is a key contribution of the chapter.

5.4 Implicit Batching in BATCHER

This section overviews implicit batching in the context of the BATCHER scheduler with respect to the core and batch dags defined in Section 5.3. The specific algorithms employed by the scheduler itself are deferred to Section 5.5. This section also gives a simple example of a program using an implicitly batched data structure to provide concrete examples of applying the performance theorem.

5.4.1 Programming Interface

BATCHER provides distinct interfaces to the algorithm programmer, who writes a program $C$ that makes parallel accesses to ADT $D$; and the data-structure programmer, who provides the batched implementation of ADT $D$. The runtime system stitches together these interfaces and does the scheduling. Figures 5.1 and 5.2 (discussed later in this section) show a simple example program making $n$ parallel increments to a shared counter using this interface style.

To perform a data-structure operation, the program $C$ makes a call into the runtime system, denoted by Batchify here. As far as the algorithm programmer is concerned, Batchify (corresponding to a data-structure node in the core dag) resembles a normal procedure call to access a concurrent data structure, and the control flow blocks at this point until the operation completes.

A BATCHER data structure, on the other hand, must provide an implementation of a parallel batched operation, which we denote by Bop. Since Bop is a batched...
implementation, it takes as input a set (i.e., an array) of operations to the ADT $D$ to perform. Note that BOP is itself a dynamically multithreaded function that can use spawn/sync or parallel loops to generate parallelism. A single invocation of the Bop function corresponds to a single batch dag.

5.4.2 Batching

At a high level, calls to BATCHIFY correspond to data-structure nodes and BATCHER is responsible for implicitly batching these data structure operations and then executing these batches by calling BOP. When a worker $p$ encounters a data-structure node $u$ (i.e., $p$ executes a call to BATCHIFY), $p$ alerts the scheduler to the operation by creating an operation record $op$ for that operation and placing it in a particular memory location reserved for this processor. Eventually, $op$ will be part of some batch $A$ and the scheduler will call BOP on $A$. Unlike core nodes, however, the data-structure node can logically block for longer than one time step and $u$’s successor(s) in the dag do not become ready until after this call to BOP returns, that is, the operation corresponding to $u$ is actually performed on the data structure as part of a batch. Thus from the perspective of the core program, a data-structure node $u$ has the same semantics as a blocking access to a concurrent data structure.

Inherent to implicit batching is the idea that the scheduler invokes only one batch at a time. Hence the data-structure implementation need not cope with concurrency, simplifying the data-structure design. The following invariant states this property for BATCHER.

**Invariant 1.** *At any time during a BATCHER execution, at most one batch is executing.*
There are many other choices that go into a scheduler for implicit batching. For BATCHER, we made specific choices guided by the goal of proving a performance theorem. Three of the main questions are what basic type of scheduler to use, how large batches should be, and when and how batches are launched. As far as the low-level details are concerned, we chose in favor of simplicity where possible. BATCHER is a distributed work-stealing scheduler. BATCHER also restricts batch sizes, as stated by the following invariant; this size cap ameliorates application of the main theorem as it simplifies the analysis of any specific data structure.

**Invariant 2.** *In a BATCHER execution, batches contain at most $P$ data-structure nodes.*

Finally, whenever an operation record is created and no batch is currently in progress, BATCHER immediately launches a new batch; it does not wait for a certain number of operations to accrue; this decision is important for the theoretical analysis. Therefore, batches can contain as few as one operation. Launching a batch includes some (parallel) setup to gather all operation outstanding operation records, executing the provided (parallel) batched operation $Bop$ thereby inducing a batch dag, and some (parallel) cleanup after completing. Since the setup/cleanup overhead is scheduler dependent, we account for the overhead separately, and the batch dag comprises only the steps of $Bop$.

### 5.4.3 Intuition behind the analysis

The analysis of BATCHER (Section 5.6) relies on specific features of the scheduling algorithm (Section 5.5). Nevertheless, we have already exposed one significant difficulty: since batches launch as soon as possible, some batches may contain just a single data-structure node. If this were true for every batch, then all operations would be
sequentialized according to Invariant 1, and it would seem impossible to show good speedup. In addition, the batch setup and cleanup overhead is the same, regardless of batch size; therefore, having many small batches may incur significant overhead.

Fortunately, small batches fall into two cases, both being good. (1) Many data structure nodes accrue while a small batch is executing. These will be part of the next batch, meaning that the next batch will be large and make progress toward the batch work $W(n)$. (2) Not many data structure nodes are accruing. Then the core dag is not blocked on too many data-structure nodes, and progress is being made on the core work $T_1$. In both cases, the setup and cleanup overhead of the small batch can be amortized either against the work done in the next batch or the work done in the core dag.

5.4.4 Examples and applying the performance bound

To understand the BATCHER performance bound (Theorem 5.1), let us turn to some specific examples. We are not developing new batched data structures here — the point is only to illustrate the power of batched data structures, and to see how to apply the bound.

As a simple example, consider a core program that makes $n$ completely parallel increments to a shared counter, as given by Figure 5.1. This example is for illustration only, and is not intended to be very deep. This program has $\Theta(n)$ core work and $\Theta(\lg n)$ core span (with binary forking). The shared counter is an abstract data type that supports a single operation \texttt{Increment}, which atomically adds a value (possibly negative) to the counter and returns its current value.
parallel_for \( i = 1 \) to \( n \)
\[ \text{do } B[i] = \text{INCREMENT}(A[i]) \]

**Figure 5.1: Parallel counter updates.** A parallel loop that performs \( n \) parallel updates to a shared counter. Here, \( A[1..n] \) is an array of values by which to increment (or decrement if negative) the counter, and \( B[1..n] \) holds any return values from the `INCREMENT`.

```c
3 struct OpRecord { int value; int result; }
```

```c
4 \text{INCREMENT}(\text{int } x)
5 \text{OpRecord } op
6 \text{op.value } = x
7 \text{return op.result}
```

```c
\text{Bop(\text{OpRecord } D[1..size])}
8 \text{let } v \text{ be the value of the counter}
9 \text{D[1]'s value field } = v + \text{D[1]'s value}
10 \text{perform parallel-prefix-sums on value fields of } D[1..size],
11 \text{storing sums into result fields of } D[1..size]
\text{// now } D[i]'s \text{ result } = \sum_{k=1}^{i} D[k]'s \text{ value}
12 \text{set the counter to } D[\text{size}]'s \text{ result}
```

**Figure 5.2: A batched-counter implementation.** As we shall see in Section 5.5, line 6 logically blocks, but the processor does not spin-wait. The Bop is called by the scheduler automatically.
Concurrent counter

A trivial concurrent counter uses atomic primitives like fetch-and-add to \textsc{increment}. If the primitive is mutually exclusive (which is true for fetch-and-add on current hardware), then $n$ \textsc{increments} take $\Omega(n)$ time. The total running time of the program is thus $\Omega(n)$ regardless of the number of processors.

One could instead use a provably efficient concurrent counter, e.g., by using the more complicated combining funnels [69, 70]. Doing so would indeed yield a good overall running time, but these techniques are not applicable to more general data structures. As we shall see next, the implicitly batched counter achieves good asymptotic speedup with a trivial implementation.

Batched counter

Figure 5.2 shows a sample batched counter. Here, when the core program makes an \textsc{increment} call, it creates an operation record which is handed-off to the scheduler. The scheduler later runs the batch increment \textsc{bop} on a set of increments. The main subroutine of the batched operation is "parallel prefix sums", which \textit{in parallel} computes $\sum_{k=1}^{i} D[k]$ for every $i$. It is easy to prove that returning $\sum_{k=1}^{i} D[k]$ yields linearizable [47] counter operations. Prefix sums is a commonly used and powerful primitive in parallel algorithms, and hence we consider this 4-line implementation of \textsc{bop} to be trivial. Adaptations of Ladner and Fischer’s approach to prefix sums [52] to the fork-join model have $O(x)$ work and $(\lg x)$ span for $x$ elements.

To analyze the execution of this program using \textsc{batcher}, we need only bound $W(n)$, the total work of arbitrarily batching $n$ operations, and $s(n)$, the span of a batched operation that processes $P$ operation records (performs $P$ increment operations). Since the work of prefix sums is linear, we have $W(n) = \Theta(n)$. Since a size-$P$
batch has $O(\lg P)$ span (dominated by prefix sums), we have $s(n) = O(\lg P)$. We thus get the bound $O\left(\frac{T_1 + n\lg P}{P} + m \lg P + T_\infty\right)$ for performing $n$ INCREMENTS, with at most $m$ along any path. The core dag of Figure 5.1 has $T_1 = O(n)$, $T_\infty = O(\lg n)$, $m = 1$, so we have a running time of $O\left(\frac{n\lg P}{P} + \lg n\right)$ for $n > P$. This nearly linear speedup is much better than for the trivial counter.

**APPLYING BATCHER TO A SEARCH TREE**

There exists an efficient batched 2-3 tree [62] in the PRAM model, and it is not too hard to adapt this algorithm to dynamic multithreading. The main challenge in a search tree is when all inserts occur in the same node of the tree, e.g., when inserting $P$ identical keys. The main idea of this batched search tree is to first sort the new elements, then insert the middle element and recurse on each half of the remaining elements. This process allows for each of the new keys to be separated by existing keys without concurrency control. It is not obvious how to leverage the same idea in a concurrent search tree.

See [62] for details of the batched search tree. Suffice it to say that a size-$x$ batch is dominated by two steps: 1) a parallel search for the location of each key in the tree, having $O(x \lg n)$ work and $O(\lg n + \lg x)$ span, and 2) a parallel sort of the $x$ keys, having $O(x \lg x)$ work. The data-structure span is thus $s(n) = O(\lg n + \text{sort}(P))$, where $\text{sort}(P) = O(\lg P \lg \lg P)$ [27] is the span of a parallel sort on $P$ elements in the dynamic-multithreading model. The data-structure work $W(n)$ is maximized for $n/P$ batches of size $x = P$, yielding $W(n) = O(n \lg n)$ data-structure work. Applying Theorem 5.1, we get a running time of $O\left(\frac{T_1 + W(n) + s(n)}{P} + ms(n) + T_\infty\right) = O\left(\frac{T_1 + n\lg P}{P} + m \lg n + m \lg P \lg \lg P + T_\infty\right)$. For large enough $n$ (specifically, $n = \Omega(P^{\lg \lg P})$, this reduces to $O\left(\frac{T_1 + n\lg n}{P} + m \lg n + T_\infty\right)$, which is asymptotically
optimal in the comparison model and provides linear speedup for programs with sufficient parallelism. For instance, a program obtained by substituting the increment operation with an insert in Figure 5.1 would yield the running time of $O(n \lg n/P)$, implying linear speedup, even though the program only performs data structure accesses.

**Amortized LIFO stack**

We now briefly describe an example, namely a LIFO stack, which has amortized performance bounds. The data structure is an array that supports two operations: a **Push** that inserts an element at the end of the array, and a **Pop** that removes and returns the last element. Such an array can be implemented using a standard table doubling [28] technique, whereby the underlying table is rebuilt (in parallel) whenever it becomes too full or too empty. To **Push** a batch of $x$ elements into an $n$-element array, check if $n + x$ elements fit in the current array. If so, in parallel simply insert the $i$th batch element into the $(n+i)$th slot of the array. If not, first resize the array by allocating new space and copying all existing elements in parallel. **Pops** can be simultaneously supported by breaking the batch into a **Push** phase followed by a **Pop** phase.

To analyze this data structure, the (amortized) work of a size-$x$ batch is $\Theta(x)$, yielding $W(n) = \Theta(n)$ (worst case). The work of any individual batch, however, can be as high as $\Theta(n)$ when a table doubling occurs. More importantly, any batch $A$ that has $w_A$ batch work has batch span $s_A = O(\lg w_A)$. Hence any batch $A$ that performs $w_A \geq P^2$ work has parallelism $w_A/s_A = \Omega(P^2/\lg P)$. We thus conclude that the data-structure span is $s(n) = O(\lg(P^2)) = O(\lg P)$. Plugging these bounds into Theorem 5.1, we get a total running time of $O(\frac{T_1 + n\lg P}{P} + m\lg P + T_\infty)$.
This section presents the high-level design of the BATCHER scheduler, a variant of a distributed work-stealing scheduler. We use $P$ to refer to the number of workers, or threads/cores, given to the scheduler. Since BATCHER is a distributed scheduler, there is no centralized scheduler thread and the operation of the scheduler can be described in terms of state-transition rules followed by each of the $P$ workers. First, we describe the internal state that BATCHER maintains in order to implicitly batched data-structure operations and to coordinate between executing the core and batch dags. We then describe how batches are launched and how load-balancing is done using work-stealing.

5.5.1 BATCHER state

The BATCHER scheduler maintains three categories of shared state: (1) collections for tracking the implicitly batched data-structure operations, (2) status flags for synchronizing the scheduler, and (3) deques for each worker tracking execution-dag nodes (see Section 5.3) and used by work stealing. With the exception of one global flag, most of this state is distributed across workers, with each worker only managing specific updates according to the provided rules that define the scheduler.

To track active data-structure nodes, BATCHER maintains two arrays. When a worker encounters a data-structure node (executes a call to \texttt{Batchify(op)}), instead of accessing the data-structure directly, an operation record $op$ is created and placed in the \textit{pending array} and the data-structure node is suspended. BATCHER guarantees that each worker has at most one suspended node / pending operation at any time; therefore this pending array may be maintained as a size-$P$ array, with a dedicated slot for each of the $P$ workers. BATCHER also maintains a \textit{working set}, which is
a densely packed array of all the operation records being processed as part of the
currently executing batch.

To synchronize batch executions, BATCHER maintains a single global active-
batch flag. In addition, each worker \( p \) has a local work-status flag (denoted
\( Status[p] \)), which describes the status of \( p \)'s current data-structure node. BATCHER
guarantees that at any instant, each worker has at most one data-structure node \( u \)
that it is trying to execute. For concreteness, think in terms of the following four
states for worker status \( Status[p] \):

- **pending**, if \( p \) has an operation record \( op \) for a suspended data structure node
  \( u \) in the pending array.
- **executing**, if \( p \) has an operation record \( op \) for a suspended data structure node
  \( u \) in the working set, i.e., a batch containing \( u \) is currently executing.
- **done**, if the batch \( A \) containing \( u \) has completed its computation, but \( p \) has not
  yet resumed the suspended node \( u \).
- **free**, if \( p \) has no suspended data-structure node.

If \( Status[p] \) is pending, executing, or done, we say \( p \) is **trapped** on operation \( u \).
Otherwise, we say \( p \) is **free**.

Finally, BATCHER maintains two deques of ready nodes on each worker: a **core
deque** for ready nodes from the core dag, and a **batch deque** for ready nodes from
a batch dag. In particular, the deques in BATCHER obey the following invariant:

**Invariant 3.** Ready nodes belonging to the core dag \( G \) are always placed on some
worker’s core deque, whereas ready nodes that belong to some batch \( A \)’s batch dag \( G_A \)
are always placed on some worker’s batch deque.
Associated with these deques, each worker $p$ also has an **assigned node** — the node that $p$ is currently executing. At any instant, the assigned node of $p$ may conceptually be associated with either the core deque or the batch deque, depending on the type of node being executed by that worker. Some workers may be executing core nodes while others are executing batch nodes.

5.5.2 Background: traditional work stealing

In a traditional work-stealing scheduler [17], each of $P$ workers maintains a core deque of ready nodes, and at any time, a worker $p$ has at most one assigned node $u$ that the worker is currently executing. When $u$ finishes, it may enable at most 2 nodes. If 1 or 2 node(s) are enabled, $p$ assigns one to itself and places the other (if any) at the bottom of its deque. If none are enabled, then $p$ removes the node at the bottom of its deque and assigns it to itself. If $p$’s deque is empty, then $p$ becomes a **thief**, randomly picks a **victim** worker and steals from the top of the victim’s deque. If the victim’s deque is not empty, then the steal attempt **succeeds**, otherwise it **fails**.

5.5.3 BATCHER algorithm

BATCHER uses a variant of work stealing, with some augmentations to support implicit batching. Free workers and trapped workers behave quite differently. Initially all workers are free, and all ready nodes belong to the core dag, and BATCHER behaves similarly to traditional work stealing. As data-structure nodes are encountered, however, the situation changes. The scheduling rules are outlined in Figure 5.3 and described below.

Free workers behave closest to traditional work stealing. A free worker is allowed to execute any node (core or batch), but it only steals if both of its deques are empty. Specifically, if either deque is non-empty, the worker executes a node off the non-empty
When \( p \) is free and both deques are empty:
steal from random victim, using alternating-steal policy

When a data-structure node \( u \) is assigned to (free) worker \( p \):
insert operation record into \( \text{pending}[p] \)
\( \text{Status}[p] = \text{pending} \)
suspend \( u \)
\(/\!/ p \) is now trapped

When \( p \) is trapped and its batch deque is empty:
\( \text{If } \text{Status}[p] = \text{done} \)
\( \text{Then } \text{Status}[p] = \text{free} \)
resume executing the core deque from suspended data-structure node \( u \)
\( /\!/ p \) is now free
\( \text{Else If } \) global batch flag = 0 and compare-and-swap(global batch flag, 0, 1)
\( \text{Then } \) run \textsc{LaunchBatch}
\( \text{Else } \) steal from random victim’s batch deque

\textbf{Figure 5.3:} Scheduler-state transition rules invoked by workers with empty deques. When the appropriate deque is not empty, the worker removes the bottom node from the deque and executes it.
deque, and any newly enabled nodes are placed on the same deque. BATCHER thus maintains the following invariant:

**Invariant 4.** *Workers that have free status in BATCHER can have at most one of their deques non-empty, i.e., they have nodes either on the batch deque or core deque, but not both.*

If both deques are empty, however, the free worker performs a steal attempt according to an **alternating-steal policy**: each worker’s $k$th steal attempt (successful or not) is from a random victim’s core deque if $k$ is even, and from a random victim’s batch deque if $k$ is odd. The alternating-steal policy is important to achieve the performance bound in Section 5.6.

When a (free) worker $p$ executes a data-structure node $u$, $p$ first inserts the corresponding operation record $op$ in its dedicated slot in the pending array, and then it changes its own status to **pending**. At this point, the $p$ becomes trapped on $u$, and according to Invariant 4, it has an initially empty batch deque.

Unlike free workers, which are allowed to execute both core and batch work, trapped workers are only allowed to execute nodes from a batch deque. If a trapped worker $p$ has a non-empty batch deque, it simply selects a node off the batch deque as in traditional work stealing. If it has an empty batch deque, however, it performs the following step. First, it checks whether its data-structure node $u$ has finished, i.e., if $Status[p] = \text{done}$. If so, it changes its own status to **free** and resumes from the suspended data-structure node on the core deque. Otherwise, it checks the global batch status flag and tries to set it using an atomic operation if no batch is executing. If successful in setting the flag, $p$ “launches” a batch. If it is unsuccessful (someone else successfully set the flag and launched a batch) or if a batch is already executing (status flag was already 1) it simply tries to steal from a random victim’s deque. BATCHER
5.5.4 Launching a batch

Launching a batch corresponds to injecting the parallel task \textsc{LaunchBatch} (see Figure 5.4), i.e., by inserting the root of the subdag induced by this code on a worker’s batch deque. This process has five steps. First, the pending array is processed in parallel, changing the status of all \texttt{pending} workers to \texttt{executing}, thereby acknowledging the operation record. Second, the \texttt{executing} records are packed together in the working-set array, which can be performed in parallel using a parallel prefix sums computation. Third, the actual batched operation (BOP) is executed on the records in working set. Fourth, the pending array is again processed in parallel, changing the

\begin{verbatim}
LaunchBatch()
1  parallel_for i = 1 to P
    do if Status[i] = pending then Status[i] = executing
2  compact all executing op records, moving them from pending array to working set
   // using parallel prefix sums subroutine
3  execute BOP (the actual parallel batch) on records in working set
4  parallel_for i = 1 to P
    do if Status[i] = executing then Status[i] = done
       remove done op records from the working set.
5  reset global batch-status flag to 0
\end{verbatim}

Figure 5.4: Pseudo-code for launching a batch. This method executes as an ordinary task in a dynamic multithreaded computation, i.e., it may run using any number of workers between 1 to $P$ workers, depending on how work-stealing occurs.

guarantees that if no batch is executing, then all workers have status either \texttt{pending}, \texttt{done} or \texttt{free}; therefore, only pending workers can succeed in launching a new batch.

\begin{verbatim}
 Launching a batch
\end{verbatim}
status of all executing workers to done. Finally, the batch-status flag is reset to 0. In practice, several of these steps can be merged, but we are not concerned about these low-level optimizations in this paper.

As mentioned above, launching a batch incurs some overhead, such as updating status fields and compacting the pending array into working-set, beyond the execution of the batched operation $Bop$ itself. We refer to this overhead as the **batch-setup overhead**. Note that this set-up procedure is itself a dynamic multithreaded program with $\Theta(P)$ work and $\Theta(\lg P)$ span, primarily due to the cost of the `parallel_for` and parallel prefix sums computations over $P$ elements. This set-up work is performed in exactly the same way as the batched operation $Bop$ is performed — that is, the nodes of this procedure are placed on batch deques and are executed in parallel (via work-stealing) by workers working on these deques. Note, however, that for the purposes of the dag metrics, the overhead is *not* counted as part of the batch work, batch span, or data-structure span defined in Section 5.3; this omission is by design since the overhead is a function of the scheduler, not the input program. This fact is one of the challenges in proving that BATCHER has a good running time, and it is exacerbated by the fact that the overhead is as high for a batch containing 1 operation as it is for a batch containing $P$ operations. Nevertheless, we shall show (Section 5.6) that BATCHER is provably efficient.

**Not trapped long**

The following lemma shows that a worker is not trapped for very long by a particular operation (at most 2 batches).

**Lemma 5.2.** *Once the operation record for a data-structure node $u$ is put into the pending array, at most two batches execute before the node completes.*
Proof. Consider an operation \( u \) whose status changes at time \( t \) to pending. Any batch finishing before time \( t \) does not delay \( u \). Any batch \( A \) launched after time \( t \) observes \( u \) in the pending array, and incorporates it in \( A \) and completes it; this accounts for one batch execution. According to Invariant 1, there can only be one batch that is launched before \( t \) and finishes after \( t \), which accounts for the second batch. \( \square \)

Correctness of state changes

Each worker is responsible for changing its own state from done to free and free to pending. There is thus no risk of any races on these state changes. The changes from pending to executing and executing to done may be performed by an arbitrary worker, but these changes occur as part of the parallel computation LaunchBatch. Since LaunchBatch is itself race free and only one LaunchBatch occurs at a time (protected by the global batch-status flag), these transitions are also safe.

5.6 Analysis of BATCHER

We now analyze the performance of BATCHER. This section will start by providing some definitions and the statement of the completion time bounds. It will then use a potential function argument to prove these bounds.

5.6.1 Definitions and theorem statements

We will analyze the running time using the computation dag \( G \) and the set of batch dags that represent batches generated due to implicit batching performed by BATCHER. We will analyze the running time using an arbitrary parameter \( \tau \), which will be later related to the data-structure span \( s(n) \). We define a few different types of batches. A batch \( A \) is \( \tau \)-wide if its batch work is more than \( P\tau \). A batch is \( \tau \)-long if
its batch span is more than \( \tau \). These definitions only count the work and span within the batched operations themselves, not the batch-setup overhead due to BATCHER. Since \( \tau \) is implied, we often drop it and call batches wide or long. Finally, a batch is \textbf{popular} if it processes more than \( P/4 \) operation records; that is, it contains more than \( P/4 \) data structure nodes. A batch is \textbf{big} if it is either long, wide or popular, or if it occurs immediately before or after a long, wide or popular batch. All other batches are \textbf{small}.

The above definitions are with respect to individual batches that arise during an execution. We next define a property of the data structure itself, analogous to data-structure work (Section 5.3).

**Definition 6.** Consider any sequence of parallel batched operations and a real value \( \tau \). The \textbf{\( \tau \)-trimmed span of the sequence} of batches is the sum of the spans of the long batches in the sequence. The \textbf{\( \tau \)-trimmed span of a data structure}, denoted by \( S_\tau(n) \), is the worst-case \( \tau \)-trimmed span for \( n \) data-structure nodes grouped arbitrarily into batches.

The following is the main theorem for this chapter, a bound on the total running time of a BATCHER computation, which is proven at the end of this section. The restriction that \( \tau \geq \lg P \) arises from binary forking.

**Theorem 5.3.** Consider a computation with \( T_1 \) core work, \( T_\infty \) core span, and \( n \) data-structure nodes with at most \( m \) falling along any path through the dag. For any \( \tau \geq \lg P \), let \( S_\tau(n) \) and \( W(n) \) denote the worst-case \( \tau \)-trimmed span and total work of the data structure, respectively. Then BATCHER executes the program in \( O\left(\frac{T_1 + W(n) + mn}{P} + T_\infty + S_\tau(n) + m\tau\right) \) expected time on \( P \) processors.

This theorem holds for any \( \tau \geq \lg P \); however, it does not provide intuition about which \( \tau \) is best. There is a tradeoff: increasing \( \tau \) increases \( n\tau \) and \( m\tau \), but decreasing
τ increases $S_\tau(n)$ since more batches become long. As we shall see at the end of this section, setting $\tau = s(n)$, the data-structure span (defined in Section 5.3), yields Theorem 5.1 as a corollary since other terms dominate $S_\tau(n)$.

5.6.2 Intuition behind the analysis

As with previous work-stealing analyses, our analysis separately bounds the total number of processor steps devoted to various activities; in our case, these activities are core work, data-structure work, stealing (and failed steal attempts), and the batch-setup overhead. We then divide this total by $P$, since each processor performs one processor step per timestep, to get the completion time.

It is relatively straightforward to see that the number of processor steps devoted to core work is $T_1$ and the number of time steps devoted to data structure work is $W(n)$. The difficulty is in bounding the number of steal attempts and the batch set up overhead. To bound the number of steal attempts, we adopt a potential function argument similar to Arora et al.’s work-stealing analysis [6], henceforth referred to as ABP. In the ABP analysis, each ready node is assigned a potential that decreases geometrically with its distance from the start of the dag. For traditional work stealing, one can prove that most of the potential is in the ready nodes at the top of the deques, as these are the ones that occur earliest in the dag. Therefore, $\Theta(P)$ random steal attempts suffice to process all of these nodes on top of the deques, causing the potential to decrease significantly. Therefore, one can prove that $O(PT_\infty)$ steal attempts are sufficient to reduce the potential to 0 in expectation.

The ABP analysis does not directly apply to bounding the number of steal attempts by BATCHER for the following reason. When a data structure node $u$ becomes ready and is assigned to worker $p$, $p$ places the corresponding operation record in the pending array and $u$ remains assigned (control flow does not go past
until results from $u$ are available. But $u$ may contain most of the potential of the entire computation (particularly if $p$’s deque is empty; in this case $u$ has all of $p$’s potential). Since $u$ cannot be stolen, steals are no longer effective in reducing the potential of the computation until the batch containing $u$ completes. To cope with this difficulty, we apply different progress arguments to big batches and small batches.

**Bounding steal attempts during big batches**

For big batches, we apply the ABP potential function to each batch’s computation dag. Nodes in a batch dag are never “suspended” in the way data-structure nodes are, so the ABP argument applies nearly directly. We charge this case against the $\tau$-trimmed span or the data-structure work. (As a technical detail, it remains to show that $P$ steal attempts overall equate to $\Omega(P)$ steal attempts from batch deques in order to complete the argument.)

**Bounding other steal attempts**

Unfortunately, small batches do not contribute to the $\tau$-trimmed span, so the above approach does not apply.\(^5\) Instead, we apply extra machinery to bound these steal attempts. The intuition is that if many steal attempts actually occur during a small batch, then the batch should complete quickly (i.e., within $O(\tau)$ timesteps). On the other hand, if few steal attempts occur then the workers are being productive anyway, since they are doing useful work (either core work or data-structure work) instead of stealing. To apply this intuition more formally within the ABP framework, we

---

\(^5\)Adding even $P$ steal attempts for each of potentially $n$ small batches would result in $\Omega(nP)$ steal attempts or $\Omega(n)$ running time, i.e., no parallelism.
augment each data-structure node to comprise a chain of $\tau$ “dummy nodes,” which captures these cases by appropriate potential decreases in the augmented dag.

### 5.6.3 Setup: dag augmentation and potential function

We create an augmented computation dag, the $\tau$-execution dag $G(\tau)$, by adding a length $\Theta(\tau)$ chain of **dummy nodes** before each data-structure node in the computation dag. The work of this dag is $W_{G(\tau)} = T_1 + O(n\tau)$ and span is $S_{G(\tau)} = T_\infty + O(m\tau)$.

For the purpose of the analysis, we suppose the scheduler executes the augmented dag instead of the original dag. The scheduler operates with one corresponding difference: when a worker encounters a data-structure node, this node remains assigned to the worker, but $\Theta(\tau)$ nodes of the dummy-node chain are placed at the bottom of its core deque. If a worker $p$ steals from another worker $p'$’s core deque and a dummy node is on the top of that deque, then $p$ steals and immediately processes the dummy node. This steal is considered a successful steal attempt. When a worker returns from a batch, all the dummy nodes on the bottom of its deque disappear. Note that dummy nodes are only for accounting. Operationally, this runtime system is identical to the one described in Section 5.5, except the analysis now just counts some unsuccessful steals as successful steals. More precisely, whenever a dummy node is stolen from a victim’s deque, the corresponding steal in the real execution is unsuccessful because the victim’s deque was empty.

We now define the potentials using this augmented dag. Each node in $G$ has **depth** $d(u)$ and **weight** $w(u) = S_G - d(u)$. Similarly, for a node $u$ in the batch dag $G_A$, $d(u)$ is its depth in that dag, and its weight is $w(u) = s_A - d(u)$. The weights are always positive.

**Definition 7.** The potential $\Phi_u$ of a node $u$ is $3^{2w(u)} - 1$ if $u$ is assigned, and $3^{2w(u)}$ if $u$ is ready.
The core potential of the computation is the sum of potentials of all (ready or assigned) nodes \( u \in G \). The batch potential is the sum of the potentials of all \( u \in G_A \) where \( A \) is the currently active batch (if one exists). The following structural lemmas follow in a straightforward manner from the arguments used throughout the ABP paper [6], so we state them without proof here. \(^6\)

**Lemma 5.4.** The initial core potential is \( 3^{SG} \) and it never increases during the computation.

**Lemma 5.5.** Let \( \Phi(t) \) denote the potential of the core dag at time \( t \). If no trapped worker’s deque is empty, then after \( 2P \) subsequent steal attempts from core deques the core potential is at most \( \Phi(t)/4 \) with probability at least \( 1/4 \).

**Lemma 5.6.** Suppose a computation (core or batch) has span \( S \), and that every “round” decreases its potential by a constant factor with at least a constant probability. Then the computation completes after \( O(S) \) rounds in expectation, and the total number of rounds is \( O(S + \lg(1/\epsilon)) \) with probability at least \( 1 - \epsilon \).

The following two lemmas extend Lemmas 5.4 and 5.5 to batch potentials. The proofs of these lemmas can also be derived from ABP proofs in a similar manner.

**Lemma 5.7.** The batch potential \( \Phi_A \) increases from 0 to \( 3^{2SA} \) when \( A \) becomes ready, and never increases thereafter.

**Lemma 5.8.** Let \( \Phi_A(t) \) be the potential of batch \( A \) at time \( t \). After \( 2P \) subsequent steal attempts from batch deques, the potential of \( A \) is at most \( \Phi_A(t)/4 \) with probability at least \( 1/4 \).

\(^6\)ABP does not explicitly capture these three lemmas as claims in their paper — some of their proof is captured by “Lemma 8” and “Theorem 9” of [6], but the rest falls to interproof discussion within the paper.
Since different arguments are required for big and small batches, we partition steal attempts into three categories. A **big-batch steal attempt** is any steal attempt that occurs on a timestep during which a big batch is executing. A **trapped steal attempt** is a steal attempt made by a trapped worker (a worker whose status is not **free**)) on a timestep when no big batch is active. A **free steal attempt** is a steal attempt by a free worker (a worker whose status is **free**) on a timestep when no big batch is active. We can now bound the different types of steal attempts and the batch-setup overhead.

### 5.6.4 Big-batch steal attempts

The big-batch steal attempts are bounded by the following lemma. The proof of this lemma is the most straightforward of the three cases.

**Lemma 5.9.** The expected number of big-batch steal attempts is $O(n\tau + PS_\tau(n) + W(n))$.

**Proof.** We first prove that if $L$ is the set of big batches, the expected number of big batch steal attempts is $O(P \sum_{A \in L} s_A)$.

Consider a particular big batch $A$. When the first round starts, the potential of the batch is $3^{2s_A}$ (Lemma 5.7). Divide the steal attempts that occur while the batch is executing into rounds of $4P$ steal attempts, except for the last round, which may have fewer. While $A$ is executing, at least half the steal attempts are from batch deques, since all the trapped steals are from batch deques, and half the free steals are from batch deques by the alternating steal policy. Therefore, in every round, at least $2P$ steal attempts are from batch deques. Applying Lemma 5.8, the potential of the batch decreases by a constant factor with probability $1/4$ during each round.
Therefore, applying Lemma 5.6, we can conclude that there are expected $O(s_A)$ rounds while $A$ is active.

We use linearity of expectation to add over all big batches. We first add over long, wide and popular batches. The total span of long batches is $S_{\tau}(n)$ by definition. There are at most $W(n)/P\tau$ wide batches, and at most $n/P$ popular batches. If they are not also long, they have span less than $\tau$. We triple the number to account for batches before and after long, wide or popular batches. Therefore, we can see that there are the number of rounds during big batches are $O(S_{\tau}(n) + n\tau/P + W(n)/P)$. Since each round has at most $4P$ steal attempts, we get the desired bound.

\[\square\]

5.6.5 Free steal attempts

Here, each “round” consists of $4P$ consecutive free steal attempts (during which no big batch is active). Recall that when a worker becomes trapped, it places $\Theta(\tau)$ dummy nodes on the bottom of its core deque. We say that a round is bad if, at the beginning of the round, some trapped worker’s core deque is empty (does not have any core nodes or dummy nodes). Otherwise, a round is good. Note that bad rounds only occur while some batch is executing; otherwise no worker is trapped. We bound good and bad rounds separately.

Good rounds do not have the problem of too much potential being concentrated in a suspended data-structure node of a trapped worker. During a good round, there is more potential in the dummy nodes than the suspended data-structure node itself, and steal attempts reduce potential.

**Lemma 5.10.** The number of good rounds is $O(S_G)$ in expectation and $O(S_G + \lg(1/\epsilon))$ with probability at least $1 - \epsilon$. Therefore, the number of free steal attempts in good rounds is $O(PS_G)$ in expectation and $O(PS_G + P\lg(1/\epsilon))$ with probability at least $1 - \epsilon$. 

115
Proof. During a good round, there are $4P$ total steal free steal attempts, and thus by the alternating steal policy, half of these ($2P$) are from core deques. Since no trapped worker’s deque is empty when the round begins, we can apply Lemma 5.5 to conclude that each round decreases the core potential by a constant factor with a constant probability; being interrupted by a big batch only decreases the potential further. We can then apply Lemma 5.6 to conclude that there are $O(S_G)$ rounds, showing the requisite bound; multiplying by $P$ gives the bound on the number of free steal attempts during good rounds.

We can now bound the number of bad rounds using the following intuition. The number of bad rounds is small since small batches have small spans; chances are most small batches finish before any trapped worker runs out of dummy nodes.

**Lemma 5.11.** The total number of free steal attempts during bad rounds is $O(n\tau)$ in expectation.

*Proof.* A worker $p$ places $\Theta(\tau) = b\tau$ dummy nodes, for constant $b$, on its core deque when it becomes trapped. There is a bad round if its core deque is stolen from at least $b\tau$ times before $p$ becomes free again. There are two cases:

**Case 1:** worker $p$ is trapped for $k\tau$ rounds, for some constant $k$; applying a Chernoff bounds, during $k\tau$ rounds, each core deque is stolen from $< k_1\tau + k_2\log P$ times with probability $> (1 - 1/P^2)$ for appropriate settings of constants $k_1$ and $k_2$. If $\tau \geq \log P$ and $b = k_1 + k_2$, then $p$’s deque runs out of dummy nodes with probability $< 1/P^2$. Since there can be at most $k\tau$ bad rounds, we get the expected number of bad rounds $O(\tau/P)$.

**Case 2:** worker $p$ is trapped for more than $k\tau$ rounds, for constant $k$. From Lemma 5.2, we know that $p$ is trapped for at most 2 batches, say $A_1$ and $A_2$. Therefore, at least one of $A_1$ and $A_2$, say $A_i$, must be active for more than $k\tau/2$ rounds. We first
bound the number of rounds during which $A_i$ can be active, with high probability. If $A_i$ is active throughout a round, then there are at least $2P$ steal attempts from batch deques during the round $r$ (since half the free steal attempts hit batch deques) and Lemma 5.8 applies. If a batch starts or ends during $r$, its potential decreases by a constant factor trivially. We can then apply Lemma 5.6 to show that with probability at least $1 - \epsilon$ the batch $A_i$ is active for $O(s_{A_i} + \lg(1/\epsilon)) = O(\tau + \lg(1/\epsilon))$ rounds, since $A_i$ is not long. ($p$ is waiting for the small batch $A_2$; therefore, the preceding batch $A_1$ is also not long.) We know that $O(\tau + \lg 1/\epsilon) < k_1\tau + k_2\lg 1/\epsilon$ for some constants $k_1$ and $k_2$; we set $\epsilon = 1/P^2$ and $k/2 = k_1 + 2k_2$. The probability that $A_i$ is active for $k\tau/2$ rounds is at most $1/P^2$. There can be at most $P\tau$ bad rounds for $A_i$, since each round takes at least one timestep, and a small batch has at most $P\tau$ work. Therefore, the expected number is at most $O(\tau/P)$.

Adding over the $n$ batches that can trap a worker, and over $P$ workers, gives us $O(n\tau)$ in total.

**Corollary 5.12.** Ignoring the batch-setup overhead, the expected number of steps taken by free processors when no big batch is active is $O(T_1 + W(n) + n\tau + P\mathcal{S}_G)$.

**Proof.** A free worker is either working (at most $T_1 + W(n)$ steps) or stealing (bounded by Lemmas 5.10 and 5.11).

### 5.6.6 Trapped steal attempts and batch-setup overhead

We next analyze the steal attempts by trapped workers during small batches. The key idea is as follows. Recall that a worker is trapped by a batch $A$ only if it has a pending data structure node whose operation record is being processed by $A$ or will be processed by the succeeding batch $A'$ (see Lemma 5.2). If more than $P/2$ workers
are trapped on a $A$, then either $A$ or $A'$ must be popular, in which case $A$ is called big. Therefore, at most $P/2$ workers can be trapped by a small batch.

**Lemma 5.13.** The expected number of processor steps taken due to batch-setup overhead and trapped steal attempts is $O(T_1 + W(n) + n\tau + PS_G + PS_\tau(n))$.

**Proof.** The batch-setup overhead is $O(P)$ per batch. After it launches, each batch executes for at least 1 timestep and only one batch executes at a time. For big batches, during this one timestep, the workers perform $P$ steps of either work (bounded by $T_1 + W(n)$) or big batch steals (bounded by Lemma 5.9). We can amortize the batch-setup overhead against these $P$ steps. For small batches, at least $P/2$ processors are free and again they perform either work or free steals (bounded by Corollary 5.12), and we can amortize the batch-setup overhead against this quantity. Adding these gives us the bound on batch-setup overhead.

Even if we pessimistically assume that trapped workers do nothing but steal during small batches, since at least half the workers are free, we can amortize these steals against the steps taken by free workers which either work or steal or perform batch setup steps. □

### 5.6.7 Overall running time

We can now bound the overall running time. We combine the bounds from Lemmas 5.9, 5.10 and 5.11, and substitute $S_G = T_\infty + m\tau$ and divide by $P$ (since there are $P$ workers performing these steps) to prove Theorem 5.3.

**Proof of Theorem 5.3.** From Lemmas 5.9, 5.10, 5.11 and 5.13, we know that the expected number of big-batch steal attempts is $O(n\tau + PS_\tau(n) + W(n))$, free steal attempts is $O(PT_\infty + Pm\tau + n\tau)$, and trapped steal attempts is $O(T_1 + W(n) + n\tau + W(n))$. Therefore, the overall running time is $O(\tau + P(mT_\infty + m\tau + \tau) + \frac{PT_\infty}{P} + \frac{Pm\tau}{P} + \frac{n\tau}{P} + \frac{W(n)}{P})$. Since $P$ is a constant, the overall running time is $O(n\tau + \tau)$. □
$PS_G + PS_r(n))$. The total batch-setup overhead is $O(T_1 + W(n) + n\tau + PS_G + PS_r(n))$.

Adding the total work and dividing by $P$ gives the result.

We can now set an appropriate value for $\tau$ to get the bound on BATCHER performance. This corollary is equivalent to Theorem 5.1.

**Corollary 5.14.** BATCHER executes the program described in Theorem 5.3 in expected time $O\left(\frac{T_1 + W(n) + n\tau}{P} + ms(n) + T_\infty\right)$.

**Proof.** We get this bound by setting $\tau$ to be equal to the data structure span $s(n)$. Recall that long batches are defined as batches with batch span longer than $\tau$, and $\tau$-trimmed span $S_\tau(n)$ is defined as the sum of the spans of all long batches. Recall, also, from the definition of the data-structure span $s(n)$ is defined as follows: For any sequence of batches comprising a total of $n$ data structure nodes, such that no batch contains more than $P$ data structure nodes, $s(n)$ is the worst case span of any batch individual $A$ that also has parallelism limited by $w_A/s_A = O(P)$.

Since the program has a total of $n$ data-structure nodes, and BATCHER only generates batches with at most $P$ data structure nodes, the only batches with $s_A > s(n)$ are those where $w_A/s_A = \Omega(P)$. Now, say $L$ is the set of long batches. For all $A \in L$, we have $w_A = \Omega(PS_A)$, since all other batches have span smaller than $s(n)$, hence also smaller than $\tau$, since $s(n) = \tau$. That is, the long batches are all batches with large parallelism. Therefore, $W(n) \geq \sum_{A \in L} w_A = \sum_{A \in L} \Omega(PS_A)$. Since $S_\tau(n) = \sum_{A \in L} s_A$, we conclude that $W(n) = \Omega(PS_\tau(n))$, or $W(n)/P = \Omega(S_\tau(n))$. The bound follows from Theorem 5.3 as $W(n)/P$ dominates $S_\tau(n)$.

### 5.7 Experimental Evaluation

We implemented a prototype of BATCHER within the Cilk-5 [37] runtime system. Our preliminary evaluation, presented here, is based around a skip-list data structure.
Note that the primary contribution of this paper is the theory; these experiments are meant to be only proof of concept, not a comprehensive study. Nevertheless, the results indicate that implicit batching is a promising direction, at least for expensive data structures and large-enough batches. For the particular experiment here, BATCHER’s performance on 1 processor is comparable to that of a sequential skip list, and hence the overhead is not prohibitive. In addition, BATCHER provides speedup when running on multiple processors.

We conducted experiments on a 2-socket machine with 8 cores per socket running Ubuntu 12.04. The processor was Intel Xeon E5-2687W. The machine has 64GB of RAM and 20MB of L3 cache per socket. For our experiments, we pinned the threads to a single socket on this machine.

5.7.1 BATCHER and skip-list implementations

We implemented the BATCHER scheduler by modifying the Cilk-5 runtime system, essentially as described in Section 5.5. The main difference between the theoretical and the practical design is within the LAUNCHBATCH operation (Figure 5.4). Because we are running on only 8 cores, we used a sequential implementation for the status changes status changes (lines 1 and 4) and the compaction (line 2).

Our batch insert (BOP) into the skip list has three steps. 1) build a new skip from a set of records, 2) perform searches for these nodes in the main skip list, and 3) splice the new list into the main list. Since the new list is small (batch size), we perform steps 1 and 3 sequentially, whereas the searches into the large main list in step 2 are performed in parallel. The core program is simply a parallel-for loop that inserts into the skip list in each iteration (e.g., as in Figure 5.1). Note that this is a bad case for BATCHER since all of the work happens within the data structure, and hence the overheads are tested.
5.7.2 Experimental scaling

In all our experiments, we first initialize the skip list with an initial size. We then timed the insertion of 100,000 additional elements into this skip list. To simulate bigger batches without the NUMA effects of going to multiple sockets, each BATCHIFY call creates 100 insertion records. We compared BATCHER with a sequential implementation where all 100,000 elements are inserted sequentially (without concurrency control).

Figure 5.5 shows the throughput of BATCHER and a sequential skip list with initialSize 20,000, 100,000, 1 million, 10 million and 100 million (e.g. BAT20000 shows BATCHER’s with initial size 20,000). For initial size 20,000 and 100,000, SEQ performs better than BAT on a single processor. This is because inserts into small skip lists are so cheap that BATCHER’s overheads begin to dominate. However, even on these small skip lists, BATCHER provides speedup, and outperforms the sequential skip list on multiple processors. For larger skip lists, the inserts get expensive enough that they dominate BATCHER’s overhead, and BATCHER performs comparably with the sequential list even on 1 processor.

More interestingly, BATCHER’s speedup increases as the skip list gets larger. At size 100 million, BATCHER provides a speedup of about $3\times$ on 6 processors, and $3.33\times$ on 8 workers.

Flat combining

We view flat combining [43] as a special case of implicit batching where batches execute sequentially. They show that flat combining significantly outperforms a good concurrent skip list, at least for certain workloads, validating the idea of implicit batching. On our experiments, flat combining and BATCHER perform similarly on one processor. However, the performance of flat combining decreases with increasing
Figure 5.5: BATCHER performance. Throughput of BATCHER and sequential skip list insertion for various initial sizes of skip lists (higher is better).

5.8 Conclusions and Open Problems

The BATCHER scheduler is provably efficient, and preliminary experiments indicate that it could provide speedup in practice, especially when data structure operations are expensive enough to amortize the overheads. This chapter is an excellent example of the potential benefits associated with deliberate balancing of batching in both the scheduling algorithm and its associated analysis.
There are several open problems remaining. It is unclear if it is possible to remove or reduce the $O(\lg P)$ overhead by using a more clever communication mechanism. It is also unclear what data structures are easily and efficiently expressible by this batch mechanism and whether or not BATCHER improves the performance of real-world parallel programs. Finally, although BATCHER is designed with work-stealing in mind, note that it may also be applicable to pthreaded programs that use data structures. A pthreaded program could run as normal, with data-structure calls replaced by BATCHER calls allowing work-stealing to operate over the data structure batches while static threading operates over the main program.
Chapter 6

Summary

This dissertation provided several parallel scheduling problems that make provably good and deliberate use of batching. In each case, the problem was well-motivated and theoretically interesting. It is potentially noteworthy that all of the presented algorithms require a careful partitioning of easily batched work, both in the analysis and the algorithm itself. However, without impossibility results or a wider survey of problems, it is difficult to make a strong claim.

Chapter 3 focused on a recent variant of interval scheduling problems, ISE, involving minimizing machine calibrations by batching jobs into calibrations. This work generalized the original problem to allow non-unit processing times for each job, consistent with real-world scenarios where there are a mix of job types being performed on the same machines. The main contribution was a polynomial-time approximation algorithm using a combination of resource augmentation, LP rounding techniques, and a reduction to a machine minimization problem. The LP rounding strategy required a partition between work that could be easily moved to improve batching and work that could not due to deadlines. By reducing to machine minimization using only constant approximation and augmentation factors and by showing that ISE is a generalization of machine minimization, the algorithm was proven to have a good bound. It offers asymptotic improvement over adaptations of known solutions whenever the maximum job size ratio is not extremely small, consistent with the original motivation.
Chapter 4 covered another ISE problem with a flow-time objective instead of strict deadline constraints. The main contribution was a polynomial-time constant approximation algorithm for the single machine case using constant-factor resource augmentation, LP rounding, and a reduction that could be solved with known flow-time minimization algorithms. The LP rounding step again required a partition between work that could be easily moved and work that could not; this time due to maintaining the flow bounds of the LP. This algorithm was shown to have an asymptotically optimal bound and improves upon existing algorithms for any non-constant maximum job size ratio.

Chapter 5 studied the scheduling of dynamically multi-threaded computations with implicit batching directives and focused on the use-case of a program maintaining a parallel data-structure. The main contribution was an implicit batching work-stealing scheduler, BATCHER. BATCHER includes associated language constructs for the implicit batching directives and a rigorous analysis of the performance, both of which allow for modular separation of the data-structure itself. The scheduler effectively balances the benefits and cost of batching data-structure work by keeping it logically separate and periodically switching between batch operations and the rest of the computation.
Bibliography


