Solving the Restricted Assignment Problem to Schedule Multi-Get Requests in Key-Value Stores

***Abstract*—Modern distributed key-value stores, such as Apache Cassandra, enhance performance through *multi-get requests*, minimizing network round-trips between the client and the database. However, partitioning these requests for appropriate storage server distribution is non-trivial and may result in imbalances. This study addresses this optimization challenge as the Restricted Assignment problem on Intervals (RAI). We propose an efficient $(2 - 1/m)$-approximation algorithm, where m is the number of machines. Then, we generalize the problem to the Restricted Assignment problem on Circular Intervals (RACI), matching key-value store implementations, and we present an optimal $O(n \log n)$ algorithm for RACI with fixed machines and unitary jobs. Additionally, we obtain a $(4 - 2/m)$-approximation for arbitrary jobs and introduce new heuristics, whose solutions give a median ratio to the optimal of 1.031. Finally, we show that optimizing multi-get requests individually also leads to global improvements, increasing achieved throughput by 27%–34% in realistic cases compared to state-of-the-art strategy.**

***Index Terms*—Key-Value Stores, Multi-Get, Scheduling, Restricted Assignment, Intervals, Approximation**

I. Introduction

Many theoretical scheduling problems capture the essence of practical challenges in modern distributed systems. Among those, NoSQL databases such as distributed key-value stores, which spread data over several servers and map items to unique keys, became central components in the architecture of online cloud applications, thanks to their excellent performance and capability to scale linearly with the dataset [5], [12]. They are often subject to high throughput, and must therefore be able to serve requests with low latency to meet user expectations. Hence, the proper scheduling of these requests is of paramount importance, and has a direct effect on the overall observed performance of the system [17].

The API of modern distributed key-value stores offer various operations to interact with the dataset, among which single reads and writes are the most common. However, as most web-services often need to retrieve several data items to perform their own calculations, some APIs provide a special type of operations called *multi-get* requests, which permit to retrieve several items from a given key set in a single round-trip [16]. When executing such a multi-get request, the key-value store needs to partition the requested key set into several sub-operations at the destination of storage servers, and it should carefully balance the keys between these sub-operations in order to respond as quickly as possible.

To ensure accessibility of data in case of node failure, which happens regularly in large clusters, the dataset is usually replicated on several servers. This makes each single key accessible at different *replica* servers, and unlocks the possibility to execute the corresponding read operation on any of these replicas. In this paper, we show how the partitioning and scheduling of a multi-get request may be seen as the so-called Restricted Assignment problem, whose objective is to schedule jobs to machines in such a way that the makespan (i.e., maximum completion time) is minimized, with the additional constraint that a given job can be processed only by a particular subset of machines. Unfortunately, this problem is strongly $\mathbf{NP}$-hard, meaning that partitioning multi-get requests in an optimal manner clearly cannot be done in reasonable time in the general case. On the positive side, the actual variant of the Restricted Assignment problem that applies to multi-get request partitioning is slightly easier than the general problem: the data replication strategy often consists in duplicating items in such a way that the replica sets are *contiguous* intervals of machines. This enables us to develop low-cost, guaranteed algorithms for multi-get request partitioning, giving good results in practice without dominating the actual service time.

Contributions. We start from prior work, done by Lin et al. [14], on the Restricted Assignment problem on Intervals (RAI). The authors provide an efficient algorithm in the special case of unitary jobs, although we find their analysis to be incorrect. We take their results further by generalizing the proposed algorithm, providing a corrected proof and a corrected version of its complexity analysis. In the case of unitary jobs, this enables us to derive an optimal algorithm, called Estimated Least Flexible Job (ELFJ), which runs in time $O(m^2 + n \log n + mn)$, where m is the number of machines and n is the number of jobs. We also prove that ELFJ (with a small adaptation) is a $(2 - 1/m)$-approximation algorithm when jobs have arbitrary processing times.

Moreover, we further generalize the RAI problem by introducing the notion of *circular* intervals, i.e., intervals that may begin at the end of the list of machines and loop back to the start, which match the actual replication strategy of distributed key-value stores. We call this generalization the Restricted Assignment problem on Circular Intervals (RACI). In the case of K different processing times, we develop a general framework that provides an optimal algorithm for the RACI problem, running in time $O(n^K f(n))$, assuming that one knows an optimal algorithm running in time $O(f(n))$ for the corresponding RAI problem. We illustrate how this framework applies to existing solutions by giving an optimal algorithm for the RACI problem with K job types, running in time $O(mn^{3K} \log \sum p_j)$. Finally, we revisit the case with unitary jobs by adapting ELFJ for the RACI version, and we show that this generalization does not affect the time complexity of the algorithm, i.e., it also runs in time $O(m^2 + n \log n + mn)$.

We present the model in Section II, where we formally define the Restricted Assignment problem and its variants, and where we explain the motivation of this work more exhaustively. Next, we review related work in Section III. In Section IV, we present the approximation algorithm ELFJ for the RAI problem, and we introduce the general RACI problem in Section V. Then, we build on ELFJ to derive the approximation algorithm DELFJ for the RACI problem in Section VI. Finally, we present our new heuristics and evaluate their performance in Section VII, before concluding the paper in Section VIII.

II. Applicative Context & Formal Model

In this section, we introduce the partitioning of multi-get requests in distributed key-value stores (Section II-A), and we give the formal definition of the corresponding Restricted Assignment problem (Section II-B).

A. Partitioning Multi-Get Requests in Key-Value Stores

Key-value stores are low-latency databases where each data item is associated with a unique key [12], [5]. In these systems, a read operation consists in retrieving the value that corresponds to a given key, whereas a write operation consists in adding a new association between a value and a key. As it is too large to fit on a single server, the overall dataset is split into several data partitions, and each partition is stored on a different server. Moreover, in order to guarantee accessibility of data in case of node failure, each partition is replicated on different physical servers. Although the replication strategy differs from one system to another, a common and practical way consists in arranging the servers on a virtual ring, and replicating the partition of each server i on its two successors $i+1$ and $i+2$ (modulo the number of servers m). In other words, servers are virtually ordered, and each key/value couple is stored on an interval of three different consecutive servers.

In contrast with *single* read operations, *multi-get* requests involve several keys. Such aggregated operations are useful, for instance, to reduce the number of network round-trips between a web-service and the database, as a single end-to-end request often requires to retrieve several data items before responding to the client [16], [9]. In a multi-get request, the requested keys (which constitute the *key set* of the request) may be located in different data partitions, which are physically stored on different servers. Thus, the contacted server must split the multi-get request into several sub-requests, each sub-request being redirected towards the appropriate storage server. In other words, the key set must be partitioned into several subsets (one per sub-request), and each subset must include keys that are located on the same server. These subsets are called the *opsets* of the multi-get request. Choosing these opsets is a crucial step, because the key-value store cannot respond before gathering all requested data items (i.e., executing all sub-requests). As we do not want a few very

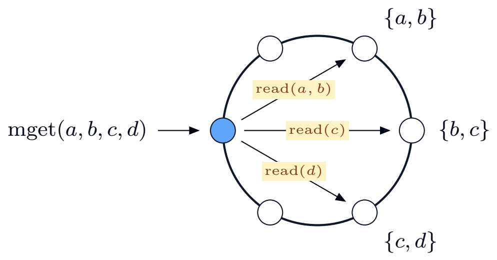


Fig. 1: Example of a multi-get request. The keyset $\{a, b, c, d\}$ is partitioned into three opsets ($\{a, b\}$, $\{c\}$ and $\{d\}$), which are sent to the appropriate servers.

fast sub-requests, and one that is very slow, the opsets must be well-balanced to guarantee a good response time for the overall multi-get request. Figure 1 gives an example of execution of a multi-get request in a distributed key-value store.

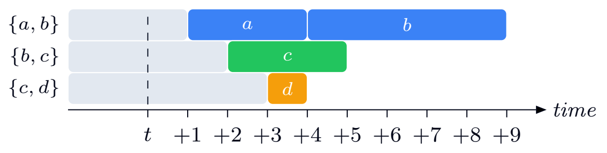
Partitioning the key set of a multi-get request in opsets may be seen as a scheduling problem, where servers correspond to machines, and each single read operation for a given key correspond to a job, whose processing time is the time required to retrieve the data item from the store. Each job may be processed only by a subset of machines, which correspond to the physical servers on which the requested key is located. Then, partitioning the jobs on machines in the context of this scheduling problem is equivalent to choosing the opsets of the multi-get request, and minimizing the maximum completion time of jobs is equivalent to minimizing imbalance between these opsets. In Figure 2, we show how a partitioning of a multi-get request may be suboptimal.

As the number of jobs and machines are relatively small in this context, a simple solution to this problem could consist in an integer programming formulation. However, this approach is not scalable, as solving such model is a costly operation that would be usable for a single multi-get request, but not for a continuous stream to treat in real-time (especially in key-value stores, which are usually dimensioned to handle extreme throughput). We need instead a polynomial, guaranteed (even if not optimal), and ideally greedy algorithm, to ensure that the time taken to partition a multi-get request does not dominate the time required to execute the request itself. In this paper, we propose to work from the fact that the formulated scheduling problem corresponds to the well-known Restricted Assignment problem, which we describe in the following section.

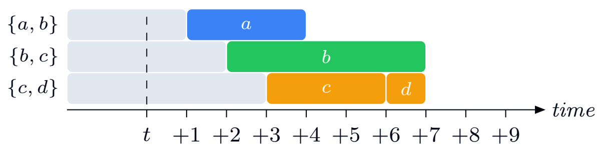
B. The Restricted Assignment Problem

In the problem of scheduling jobs on unrelated machines (also known as the $R\,||\,C_{\max}$ problem in Graham's classification), we are given a set of n jobs $J = \{1, \cdots, n\}$ and a set of m machines $M = \{1, \cdots, m\}$, where each job $j \in J$ has a processing time $p_{ij} > 0$ on machine $i \in M$. The objective is to schedule (non-preemptively) the jobs on machines so as to minimize the makespan, that is to say, the maximum completion time of the jobs.

The Restricted Assignment (RA) problem is a special case of $R\,||\,C_{\max}$, where each job $j \in J$ can be processed only on



(a) Non-optimal partitioning. The request completes at time $t+9$.



(b) Optimal partitioning. The request completes at time $t+7$.

Fig. 2: Two possible partitions of a multi-get request released at time t. The gray areas represent the current load of each machine, while the sets written on the left represent the values stored on each machine. In the first case, the opsets are $\{a,b\}$, $\{c\}$ and $\{d\}$, which causes imbalance. In the second case, the opsets are $\{a\}$, $\{b\}$ and $\{c,d\}$, which is the best possible choice for the current multi-get request.

a subset of machines $\mathcal{M}_j \subseteq M$, which we call the *processing set* of j. In this setting, the job j has processing time p_j on machine i if and only if $i \in \mathcal{M}_j$, and $+\infty$ otherwise. The RA problem is sometimes noted $P \,|\, \mathcal{M}_j \,|\, C_{\max}$.

The $R \,||\, C_{\max}$ problem, and more specifically the RA problem, are well-known **NP**-hard problems in the strong sense, and it has even been proved that no algorithm can approximate an optimal solution within a factor better than $3/2$ unless $\mathbf{P} = \mathbf{NP}$ [13]. Hence, specific cases of the RA problem have also been the subject of extensive research. One possible manner to reduce the complexity of the problem is to bring structure in the processing sets of jobs. For example, a common sub-problem consists in solving the RA problem on *nested*[1] processing sets, that is to say, one of the following properties holds for any two jobs $j, j' \in J$: $\mathcal{M}_j \subseteq \mathcal{M}_{j'}$, $\mathcal{M}_{j'} \subseteq \mathcal{M}_j$, or $\mathcal{M}_j \cap \mathcal{M}_{j'} = \emptyset$. An even more specific case is when the processing sets are *inclusive*, i.e., for any two jobs $j, j' \in J$, either $\mathcal{M}_j \subseteq \mathcal{M}_{j'}$, or $\mathcal{M}_{j'} \subseteq \mathcal{M}_j$.

In this paper, we focus on *interval* processing sets, which also constitute a particular case of the RA problem, but more general than the nested and inclusive cases. Here, the machines can be rearranged such that the processing sets of jobs consist in contiguous intervals of machines. More formally, let us note $\langle a,b \rangle$ the interval[2] ranging from machine a (inclusive) to machine b (inclusive, $a \leq b$), and $I_{\langle a,b \rangle} = \{a, a+1, \cdots, b\}$. In the Restricted Assignment problem on Intervals (RAI), for all jobs $j \in J$, we define $\mathcal{M}_j = I_{\langle a_j, b_j \rangle}$, where a_j and b_j are respectively the lower and upper bounds of the interval of machines on which the job j can be assigned.

As a generalization of the classical makespan problem $P \,||\, C_{\max}$, the RAI problem, noted $P \,|\, \mathcal{M}_j(\mathit{interval}) \,|\, C_{\max}$ in Graham's classification, remains **NP**-hard in the strong sense. In the following, we show that formal guarantees can still be obtained, especially on slightly simpler variants such as the case with unitary jobs, which may happen in homogeneous workloads where all requested data items have similar sizes. Another example is the case with K types of jobs, which corresponds to a discrete categorization of read operations, for example by considering *small* and *large* data items.

[1]This special case is also known as *laminar* processing sets.

[2]We will extend the interval definition later, thus we do not use the common notations of integer intervals.

III. Related Work

Key-value stores have been the subject of extensive research since the nominal publication on the Dynamo system [5]. Their wide adoption in the industry, partly due to their excellent performance, availability and scalability properties, has led to the development of numerous optimization techniques, in particular to mitigate the well-known tail latency problem [4]. One such technique consists in batching single read operations into so-called multi-get requests, in order to reduce the natural variability that arises with large numbers of requests, and increase the network efficiency by reducing the number of round-trips. However, as explained in the previous section, the service time of a multi-get request is equal to its slowest read operation, which can be largely improved by carefully balancing the load.

Reda et al. [16] proposed Rein, a scheduler that is able to identify the bottleneck of a given multi-get request and that assigns different priorities to the contained operations to improve response time. Compared to the default First-Come First-Served policy, the priority-based scheduler reduces the median latency by $1.5\times$ and the 99th percentile of latency by $1.9\times$. Under heterogeneous workloads, in which the dataset is categorized into small/large items and multi-get requests consist of a varying number of operations, Jaiman et al. [9] proposed TailX, a scheduler that is able to perform better than Rein by taking into account an estimation of the actual service time of read operations. Their evaluation shows a 75% improvement on the median latency and a 70% improvement on tail latency compared to Rein.

On the theoretical side, the Restricted Assignment (RA) problem has received a significant attention, as it captures the essence of many practical problems, among which, as demonstrated in this paper, the partitioning of multi-get requests. It is a subcase of the more general unrelated scheduling problem $R \,||\, C_{\max}$, for which a famous 2-approximation algorithm, based on linear programming, has been proposed by Lenstra et al. [13]. The authors also considered the RA problem and proved that no polynomial algorithm may give an approximation better than $3/2$, unless $\mathbf{P} = \mathbf{NP}$. A Quasi-Polynomial Time Approximation Scheme (QPTAS) has recently been derived for the RA problem, which approximates an optimal solution within a factor $11/6 + \varepsilon$ in time $O((n+m)^{O(1/\varepsilon \log(n+m))})$ [11].

Various subcases of the RA problem have also been considered in the literature. For instance, Ebenlendr et al. [6] studied the Graph Balancing problem, which corresponds to the RA problem where each job may be processed by at most two different machines. They show the $3/2$-hardness of the problem, and provide a $7/4$-approximation algorithm. List-scheduling is a famous $(2-1/m)$-approximation for the problem $P\,||\,C_{\max}$, and it has also been proved to give the same guarantee for the nested case of RA, at the condition that jobs are initially sorted by non-decreasing size of their processing set [8]. The authors also derived an efficient $3/2$-approximation for the inclusive case, running in time $O(nm\log m)$. On the negative side, Maack et al. proved that no Polynomial Time Approximation Scheme (PTAS) exists for the interval problem RAI unless $\mathbf{P}=\mathbf{NP}$ [15]. Interestingly, there is a PTAS when all intervals are overlapping without any strict inclusion, i.e., for any two jobs j, j', $\mathcal{M}_j \not\subset \mathcal{M}_{j'}$ and $\mathcal{M}_{j'} \not\subset \mathcal{M}_j$ [18]. There is also a PTAS for the nested case and other variants [7].

Some authors also studied the RA problem with restricted processing times. Jansen et al. proved that even when considering only two possible processing times, there is no algorithm giving an approximation better than $4/3$ [10]. However, in the specific case where $p_j \in \{1,2\}$, there is a $3/2$-approximation algorithm [8]. By approaching the RA problem as a matching problem, Biró et al. derive a $(2-1/2^k)$-approximation when $p_j \in \left\{1,2,\cdots,2^k\right\}$ for all jobs [2]. They also give an optimal algorithm when the same constraint is applied on the nested case. When jobs are unitary, the RA problem becomes simpler, and it is possible to find optimal schedules in time $O(n^3\log n)$ by coupling a binary search procedure to a network flow formulation of the problem [14].

IV. An Algorithm for the Restricted Assignment Problem on Regular Intervals

We simplify the practical partitioning problem for now by considering only regular intervals of machines, i.e., we focus on the standard RAI problem $P\,|\,\mathcal{M}_j(interval)\,|\,C_{\max}$, for which Lin et al. [14] have proposed a polynomial algorithm when jobs are unitary. They argue that their algorithm runs in time $O(m(m+n))$, although we found their analysis to be slightly incorrect. We also noticed an error in their proof of optimality. In this section, we give a correct version of their proof, and we generalize their approach to derive the following results:

(i) an optimal algorithm for the RAI problem with unitary jobs, which runs in time $O(m^2+n\log n+mn)$ (Theorem 1), and
(ii) a tight $(2-1/m)$-approximation algorithm for the RAI problem with arbitrary jobs, which also runs in time $O(m^2+n\log n+mn)$ (Theorem 2).

Let us introduce Algorithm 1, called Estimated Least Flexible Job (ELFJ), which generalizes Lin et al.'s algorithm. ELFJ takes a time λ as parameter and builds a schedule that is guaranteed to finish before this time. In other words, λ denotes an upper bound on the optimal makespan, i.e., the better the quality of the bound, the closer ELFJ gets to an optimal schedule. The algorithm performs two steps. First, it sorts the jobs in non-decreasing order of interval upper bound b_j (in time $O(n\log n)$). Second, it greedily assigns jobs on machines (in time $O(mn)$), and returns an assignment vector μ, where μ_j denotes the machine on which job j is assigned. In the following, we explain how to choose λ to get various guarantees on the quality of the schedule.

Let us introduce some notations and definitions. For any interval of machines $\langle\alpha,\beta\rangle$, where $1\le\alpha\le\beta\le m$, we define $K_{\langle\alpha,\beta\rangle}$ as the set of jobs whose processing set is included in $I_{\langle\alpha,\beta\rangle}$, i.e., $K_{\langle\alpha,\beta\rangle}=\left\{j\in J \text{ s.t. } \mathcal{M}_j\subseteq I_{\langle\alpha,\beta\rangle}\right\}$. We denote the total processing time of jobs in $K_{\langle\alpha,\beta\rangle}$ by $w_{\langle\alpha,\beta\rangle}$, i.e., $w_{\langle\alpha,\beta\rangle}=\sum_{j\in K_{\langle\alpha,\beta\rangle}} p_j$. Let $\tilde{w}_{\langle\alpha,\beta\rangle}$ represent the minimum average work that *any* schedule must perform on machines $\alpha,\cdots,\beta$, i.e.,

$$\tilde{w}_{\langle\alpha,\beta\rangle}=\frac{w_{\langle\alpha,\beta\rangle}}{\beta-\alpha+1},$$

and let $\tilde{w}_{\max}$ be the maximum value of $\tilde{w}_{\langle\alpha,\beta\rangle}$ over all intervals ($\tilde{w}_{\max}=\max_{1\le\alpha\le\beta\le m}\left\{\tilde{w}_{\langle\alpha,\beta\rangle}\right\}$). From these definitions, we can easily derive a lower bound on the optimal makespan $C^{\text{OPT}}_{\max}$ for a given instance $\mathcal{I}$ of the RAI problem, as shown by Lin et al. in their original work [14].

Lemma 1. *The optimal makespan is bounded by $\tilde{w}_{\max}$, i.e., $C^{\text{OPT}}_{\max}\ge\tilde{w}_{\max}$. If all processing times are integers, then we have $C^{\text{OPT}}_{\max}\ge\lceil\tilde{w}_{\max}\rceil$.*

Proof: Let $C^{\text{OPT}}_{\langle\alpha,\beta\rangle}$ be the maximum completion time of machines $\alpha,\cdots,\beta$ in an optimal schedule. We clearly have $C^{\text{OPT}}_{\langle\alpha,\beta\rangle}\ge\tilde{w}_{\langle\alpha,\beta\rangle}$ for any interval $\langle\alpha,\beta\rangle$, because all jobs in the set $K_{\langle\alpha,\beta\rangle}$ must be done between machines α and β, and in the best case, the jobs are perfectly balanced on the $\beta-\alpha+1$ machines of the interval $\langle\alpha,\beta\rangle$.

Let $\langle a,b\rangle$ be the interval of machines such that $\tilde{w}_{\max}=\tilde{w}_{\langle a,b\rangle}$. We have $C^{\text{OPT}}_{\langle a,b\rangle}\ge\tilde{w}_{\langle a,b\rangle}$. As $C^{\text{OPT}}_{\max}\ge C^{\text{OPT}}_{\langle\alpha,\beta\rangle}$ for any α,β, we necessarily have $C^{\text{OPT}}_{\max}\ge C^{\text{OPT}}_{\langle a,b\rangle}$, i.e., $C^{\text{OPT}}_{\max}\ge\tilde{w}_{\max}$.

If all processing times are integers in the considered instance, we have $C^{\text{OPT}}_{\langle\alpha,\beta\rangle}\ge\lceil\tilde{w}_{\langle\alpha,\beta\rangle}\rceil$ for any interval $\langle\alpha,\beta\rangle$, because jobs are not divisible. Thus, in an analogous manner, we necessarily have $C^{\text{OPT}}_{\max}\ge\lceil\tilde{w}_{\max}\rceil$. ■

Algorithm 1 Estimated Least Flexible Job (ELFJ)

```
Input: jobs $J$, machines $M$ and makespan $\lambda$
Output: an assignment $\mu$
1: sort jobs in non-decreasing order of $b_j$
2: for all machines $i \in M$ do
3:     $\delta_i \leftarrow 0$
4:     for all unassigned jobs $j \in J$ such that $i \in I_{\langle a_j, b_j\rangle}$ do
5:         if $\delta_i + p_j \le \lambda$ then
6:             $\mu_j \leftarrow i$
7:             $\delta_i \leftarrow \delta_i + p_j$
8: return $\mu$
```

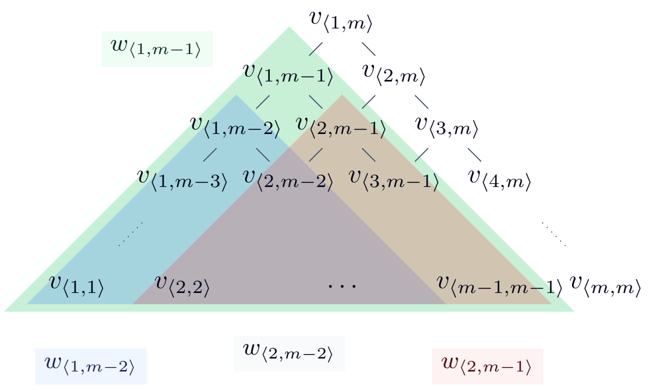

$w_{\langle 1,m-1\rangle}$
$v_{\langle 1,m\rangle}$
$v_{\langle 1,m-1\rangle}$
$v_{\langle 2,m\rangle}$
$v_{\langle 1,m-2\rangle}$
$v_{\langle 2,m-1\rangle}$
$v_{\langle 3,m\rangle}$
$v_{\langle 1,m-3\rangle}$
$v_{\langle 2,m-2\rangle}$
$v_{\langle 3,m-1\rangle}$
$v_{\langle 4,m\rangle}$
$v_{\langle 1,1\rangle}$
$v_{\langle 2,2\rangle}$
$\cdots$
$v_{\langle m-1,m-1\rangle}$
$v_{\langle m,m\rangle}$
$w_{\langle 1,m-2\rangle}$
$w_{\langle 2,m-2\rangle}$
$w_{\langle 2,m-1\rangle}$

Fig. 3: Interval hierarchy represented as a lattice graph. Each node represents an interval $\langle x,y\rangle$ and is labeled with the value $v_{\langle x,y\rangle}$. Nodes are organized by level, where nodes on level h represent intervals of size h, e.g., if $m=3$, the node on level m is the interval of size 3, nodes on level $m-1$ are intervals of size 2, and nodes on level $m-2$ are intervals of size 1.

A. *Computing $\tilde{w}_{\max}$ for Arbitrary Jobs*

The idea of Lin et al. is to use $\lceil \tilde{w}_{\max} \rceil$ as the value of the parameter λ in ELFJ to get an optimal schedule when jobs are unitary. Suppose for a moment that all processing times are unitary, i.e., for all intervals $\langle \alpha,\beta\rangle$, $w_{\langle \alpha,\beta\rangle} = \left|K_{\langle \alpha,\beta\rangle}\right|$ (thus $\tilde{w}_{\max} = \max_{1\le\alpha\le\beta\le 1}\left\{\left|K_{\langle \alpha,\beta\rangle}\right|/(\beta-\alpha+1)\right\}$). In the original paper, the authors propose the following procedure to compute each $\left|K_{\langle \alpha,\beta\rangle}\right|$. First, for all machines i, construct the sets $A_i = \{j \in J \text{ s.t. } i \le a_j\}$ and $B_i = \{j \in J \text{ s.t. } b_j \le i\}$ in time $O(mn)$. Then, for all intervals $\langle \alpha,\beta\rangle$, compute $\left|K_{\langle \alpha,\beta\rangle}\right| = |A_\alpha \cap B_\beta|$.

We argue that counting the number of common elements in two sets is clearly not a constant-time operation in the general case. Hence, as there are $O(m^2)$ possible intervals, the time complexity of this procedure is at least $O(c_n \cdot m^2)$, where c_n is the time complexity of counting common elements in two sets of size $O(n)$. If we recall that the original algorithm performs a sorting operation (in time $O(n\log n)$) and the assignment of jobs to machines (in time $O(mn)$), we conclude that the total complexity $O(m^2+mn)$ given by Lin et al. for their algorithm is underestimated, and we argue that their approach gives in fact an algorithm with time complexity $O(c_n \cdot m^2 + n\log n + mn)$. Last but not least, their method is not suitable to the case where processing times are arbitrary.

We provide a new procedure to compute $\tilde{w}_{\max}$ in time $O(m^2+n)$ for any instance of the RAI problem with arbitrary processing times. We notice that the set of intervals in a list of m machines can be represented by a graph, where nodes correspond to intervals. For all intervals $\langle \alpha,\beta\rangle$ such that $\alpha<\beta$, the node $\langle \alpha,\beta\rangle$ is the parent of two children nodes $\langle \alpha,\beta-1\rangle$ and $\langle \alpha+1,\beta\rangle$ (see Figure 3). Let $J_{\langle \alpha,\beta\rangle}$ be the set of jobs whose processing set is exactly $I_{\langle \alpha,\beta\rangle}$, i.e., $J_{\langle \alpha,\beta\rangle} = \left\{j \in J \text{ s.t. } \mathcal{M}_j = I_{\langle \alpha,\beta\rangle}\right\}$, and let $v_{\langle \alpha,\beta\rangle}$ be their total processing time. We have a recursive relation between the values $w_{\langle \alpha,\beta\rangle}$: for a given interval $\langle \alpha,\beta\rangle$ that has two children intervals, the work $K_{\langle \alpha,\beta\rangle}$ includes the work $J_{\langle \alpha,\beta\rangle}$, the work $K_{\langle \alpha,\beta-1\rangle}$, and the work $K_{\langle \alpha+1,\beta\rangle}$, minus the work $K_{\langle \alpha+1,\beta-1\rangle}$, as it is included both in $K_{\langle \alpha,\beta-1\rangle}$ and $K_{\langle \alpha+1,\beta\rangle}$. Then, for any α,β, we have

$$w_{\langle \alpha,\beta\rangle} = v_{\langle \alpha,\beta\rangle} + w_{\langle \alpha,\beta-1\rangle} + w_{\langle \alpha+1,\beta\rangle} - w_{\langle \alpha+1,\beta-1\rangle},$$

with the convention $w_{\langle \alpha,\beta\rangle} = 0$ if $\alpha > \beta$. Values $v_{\langle \alpha,\beta\rangle}$ can be pre-computed in time $O(n)$ by scanning jobs, and the computation of values $w_{\langle \alpha,\beta\rangle}$ is done in time $O(m^2)$. Thus, $\tilde{w}_{\max}$ can be found in time $O(m^2+n)$ and space $O(m^2)$, as shown in Algorithm 2.

B. *Optimality of ELFJ for Unitary Jobs*

We now prove that, if $\lambda = \lceil \tilde{w}_{\max} \rceil$, then ELFJ is optimal for unitary jobs (i.e., the problem $P\,|\,\mathcal{M}_j(\mathit{interval}), p_j = 1\,|\,C_{\max}$). The principle of the proof comes from Lin et al. [14], although we found their demonstration to be incorrect. We know from Lemma 1 that the optimal makespan $C^{\mathrm{OPT}}_{\max}$ is at least $\lceil \tilde{w}_{\max} \rceil$ (as jobs are unitary, all processing times are integers), which is the value of λ here. We seek to prove that ELFJ gives a schedule whose makespan is at most λ.

By contradiction, Lin et al. assume there exists a unitary job j that could not be assigned by ELFJ on any machine before time λ, which means that all machines between a_j and b_j must be full. Then we consider the machine with the smallest index $\alpha \le a_j$ such that all machines between α and b_j are full. Let $\beta = b_j$. Now the goal is to prove that all jobs assigned by ELFJ on machines $\alpha, \alpha+1, \cdots, \beta$ come from the set $K_{\langle \alpha,\beta\rangle}$, that is, the processing set of any job assigned between machines α and β is included in $I_{\langle \alpha,\beta\rangle}$. Proving this property leads to the conclusion $\lambda < \tilde{w}_{\langle \alpha,\beta\rangle}$, which is a contradiction because $\lambda = \lceil \tilde{w}_{\max} \rceil$.

To do so, Lin et al. argue that any job j' assigned on a machine between α and β must have $a_{j'} \ge \alpha$, otherwise j' would have been put on $\alpha-1$ (which is not full), and $b_{j'} \le \beta$, because jobs have been assigned by non-decreasing order of b_j. This last justification is an error, as highlighted by the following counterexample: suppose that $\alpha = a_j - 1$, and there are λ jobs with interval $\alpha, \alpha+1, \cdots, \beta+1$ (call these jobs the *filling* jobs). The job j must be done in the interval $\alpha+1, \alpha+2, \cdots, \beta$. Then, the filling jobs will be assigned on machine

Algorithm 2 Computing $\tilde{w}_{\max}$

```
 1: $v_{\langle \alpha,\beta\rangle} \leftarrow 0$ for all $0 \le \alpha \le \beta \le m$
 2: for all jobs $j \in J$ do
 3:     $v_{\langle a_j,b_j\rangle} \leftarrow v_{\langle a_j,b_j\rangle} + p_j$
 4: $\tilde{w}_{\max} \leftarrow 0$
 5: for all $l$ from 0 to $m-1$ do
 6:     for all $a$ from 1 to $m-l$ do
 7:         $b \leftarrow a+l$
 8:         $w_{\langle a,b\rangle} \leftarrow v_{\langle a,b\rangle} + w_{\langle a,b-1\rangle} + w_{\langle a+1,b\rangle} - w_{\langle a+1,b-1\rangle}$
 9:         $\tilde{w}_{\langle a,b\rangle} \leftarrow w_{\langle a,b\rangle}/(b-a+1)$
10:         if $\tilde{w}_{\langle a,b\rangle} > \tilde{w}_{\max}$ then
11:             $\tilde{w}_{\max} \leftarrow \tilde{w}_{\langle a,b\rangle}$
```

α by ELFJ, even if we have $b_{j'} = \beta + 1 > \beta$ for all filling jobs j', because they are the only jobs that are feasible on α. Therefore, we cannot conclude that all jobs assigned on machines $\alpha, \alpha + 1, \cdots, \beta$ come from $K_{\langle \alpha, \beta \rangle}$.

We present here a constructive proof that also consists in exhibiting a contradiction by finding a machine $\alpha \leq a_j$ such that all jobs assigned between α and β come from $K_{\langle \alpha, \beta \rangle}$. However, α is more carefully chosen in this new version. We start from the interval $\langle a_j, b_j \rangle$, and we extend this interval step by step until the appropriate condition is met.

Theorem 1. *Let $\lambda = \lceil \tilde{w}_{\max} \rceil$. Then ELFJ (Algorithm 1) is optimal for $P \,|\, \mathcal{M}_j(interval), p_j = 1 \,|\, C_{\max}$, and the full procedure runs in time $O(m^2 + n \log n + mn)$.*

Proof: The beginning of the proof is similar to the one of Lin et al. By contradiction, suppose that ELFJ does not give a feasible schedule with makespan at most λ. Let j_0 be one of the non-assigned jobs. Then, as all jobs are unitary and λ is an integer, all machines in $\mathcal{M}_{j_0}$ must finish at least at time λ. Let $\beta = b_{j_0}$, and let $\gamma \leq a_{j_0}$ be the smallest machine index such that all machines between γ and β complete at time λ. This means that the machine $\gamma - 1$ completes before time λ if $\gamma > 1$.

Now our goal is to find a machine α between γ and a_{j_0} such that all jobs assigned on machines $\alpha, \alpha + 1, \cdots, \beta$ come from the set $K_{\langle \alpha, \beta \rangle}$. The process here is constructive. For the first step, let j be a job assigned on a machine between a_{j_0} and β. Then, we have $b_j \leq \beta$, otherwise j_0 would have been scheduled instead of j. Now there are two cases: either we have $a_j \geq a_{j_0}$ for all j assigned between a_{j_0} and β, or $a_j < a_{j_0}$ for at least one job j assigned between a_{j_0} and β.

If the first case holds, then we set $\alpha = a_{j_0}$, and we are done: all jobs assigned between α and β have a processing set included in $I_{\langle \alpha, \beta \rangle}$. If the second case holds, let us choose such j with the smallest a_j (then $a_j < a_{j_0}$), and let us call this job j_1. Now we proceed to the next step. If j_1 has been assigned between a_{j_0} and β, it means that $b_j \leq b_{j_1} \leq \beta$ for all jobs j assigned on machines $a_{j_1}, a_{j_1} + 1, \cdots, a_{j_0} - 1$, otherwise we would have scheduled j_1 instead. Moreover, we have two cases again: either we have $a_j \geq a_{j_1}$ for all j assigned between a_{j_1} and $a_{j_0} - 1$, or $a_j < a_{j_1}$ for at least one job j assigned between a_{j_1} and $a_{j_0} - 1$.

In the first case, we set $\alpha = a_{j_1}$, and we are done. Otherwise, we choose j with the smallest a_j, we call this job j_2, and we proceed to the next step by repeating the same reasoning.

To conclude, note that we have $a_j \geq \gamma$ for all jobs j assigned on a machine whose index is greater than or equal to γ, otherwise j would have been put on machine $\gamma - 1$, as it completes before time λ. By applying the described process iteratively, we inevitably reach a step k where there cannot exist a job j such that $a_j < a_{j_k}$, and we set $\alpha = a_{j_k}$.

Therefore, there exist $\alpha \leq \beta$ such that

(i) $j_0 \in K_{\langle \alpha, \beta \rangle}$,
(ii) machines $\alpha, \alpha + 1, \cdots, \beta$ complete at time λ, and
(iii) all jobs assigned on machines $\alpha, \alpha + 1, \cdots, \beta$ belong to $K_{\langle \alpha, \beta \rangle}$.

Then we have

$$w_{\langle \alpha, \beta \rangle} \geq (\beta - \alpha + 1)\lambda + 1 > (\beta - \alpha + 1)\lambda,$$

i.e., $\lambda < \tilde{w}_{\langle \alpha, \beta \rangle}$, which is a contradiction. Hence, ELFJ gives a schedule feasible in λ time units, which means that $C_{\max}^{\text{OPT}} \leq \lambda$. By Lemma 1, we also know that $C_{\max}^{\text{OPT}} \geq \lambda$. We conclude that $C_{\max}^{\text{OPT}} = \lambda$, thus ELFJ is optimal. Moreover, as demonstrated earlier, the computation of λ is done in time $O(m^2 + n)$, and ELFJ runs in time $O(n \log n + mn)$, which gives a total time complexity of $O(m^2 + n \log n + mn)$. ■

In this proof, we avoid the error from Lin et al. by making sure that $b_j \leq \beta$ for all jobs j assigned between either a_{j_0} and β, or between a_{j_1} and β, or between a_{j_2} and β, etc. In the previous counterexample, we would have stopped at $\alpha = a_j$ and $\beta = b_j$.

C. An Approximation for Arbitrary Jobs

As shown by Lin et al., ELFJ is optimal for unitary jobs if $\lambda = \lceil \tilde{w}_{\max} \rceil$. The same principle may well be applied to arbitrary jobs, but does not produce an optimal schedule. However, we show here that, subject to a small adaptation on the value of λ, ELFJ also constitutes an approximation algorithm for this more general case. In the following, we note $p_{\max}$ the maximum processing time among all jobs.

Theorem 2. *Let $\lambda = \tilde{w}_{\max} + \left(1 - \frac{1}{m}\right) p_{\max}$. Then, ELFJ (Algorithm 1) is a tight $(2-1/m)$-approximation algorithm for RAI, and the full procedure runs in time $O(m^2+n \log n+mn)$.*

Proof: Suppose by contradiction that ELFJ does not give a feasible schedule with makespan at most λ. Let j_0 be the first non-assigned job. Then all machines in $\mathcal{M}_{j_0}$ must finish after time $\lambda - p_{j_0}$, otherwise we would have assigned j_0. Let $\beta = b_{j_0}$, and let $\gamma \leq a_{j_0}$ be the smallest machine index such that all machines between γ and β complete after time $\lambda - p_{\max}$. This means that the machine $\gamma - 1$ completes at or before time $\lambda - p_{\max}$ if $\gamma > 1$. Hence, we have $a_j \geq \gamma$ for all jobs j assigned on a machine whose index is greater than γ, otherwise j would have been assigned on $\gamma - 1$ by ELFJ, as $p_j \leq p_{\max}$ (by definition of $p_{\max}$).

Now let $S(a, b, t)$ be the set of jobs assigned by ELFJ between machines a and b, and scheduled to start at or before time t ($S(a, b, t) = \emptyset$ if $a > b$). We can see this set $S(a, b, t)$ as a work area, whose minimal shape is the rectangle delimited by a, b and t. Our goal is to prove that there exists a machine α between γ and a_{j_0} such that all jobs in the set $S(\alpha, a_{j_0} - 1, \lambda - p_{\max}) \cup S(a_{j_0}, \beta, \lambda - p_{j_0})$, whose minimal work area can be represented by two adjacent rectangles, come from the set $K_{\langle \alpha, \beta \rangle}$, which includes all jobs whose processing set is in the interval $\langle \alpha, \beta \rangle$. The Figure 4 highlights the work areas of interest.

To do so, we adapt the constructive process that we used in the previous proof. Let us prove that there exists a non-empty

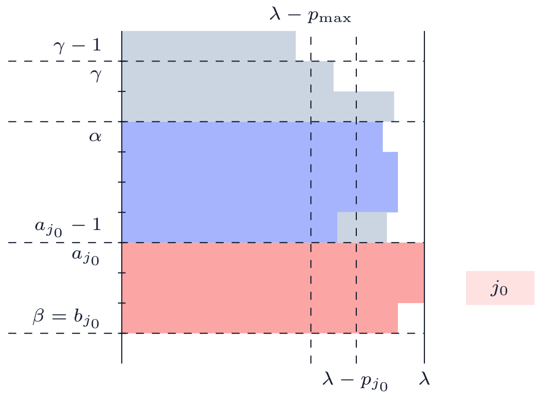


Fig. 4: Work areas between machines α and β. The blue area is $S(\alpha, a_{j_0}-1, \lambda - p_{\max})$. The red area is $S(a_{j_0}, \beta, \lambda - p_{j_0})$. Gray areas are the other jobs. We seek to prove that the blue and red areas are made of jobs included in $K_{\langle \alpha,\beta \rangle}$.

set of machines $u_1 > u_2 > \cdots > u_x$ between a_{j_0} and γ such that

- for all u_k, $b_j \leq \beta$ for all jobs $j \in S(u_k, a_{j_0}-1, \lambda - p_{\max}) \cup S(a_{j_0}, \beta, \lambda - p_{j_0})$,
- for all $u_{k<x}$, there exists $j \in S(u_k, a_{j_0}-1, \lambda - p_{\max}) \cup S(a_{j_0}, \beta, \lambda - p_{j_0})$ such that $\gamma \leq a_j < u_k$, and
- $a_j \geq u_x$ for all $j \in S(u_x, a_{j_0}-1, \lambda - p_{\max}) \cup S(a_{j_0}, \beta, \lambda - p_{j_0})$.

Base case ($u_1 = a_{j_0}$). Let $j \in S(a_{j_0}, \beta, \lambda - p_{j_0})$ be a job assigned between a_{j_0} and β, and starting at or before $\lambda - p_{j_0}$. We have $b_j \leq b_{j_0} = \beta$, otherwise the job j_0 could have been scheduled instead of job j. Then, either $a_j \geq a_{j_0}$ for all $j \in S(a_{j_0}, \beta, \lambda - p_{j_0})$, or there is $j \in S(a_{j_0}, \beta, \lambda - p_{j_0})$ such that $\gamma \leq a_j < a_{j_0}$. In the first case, we set $x = 1$ and we are done. In the second case, we proceed to the next step.

Induction step. Suppose that $b_j \leq \beta$ for all $j \in S(u_k, a_{j_0}-1, \lambda - p_{\max}) \cup S(a_{j_0}, \beta, \lambda - p_{j_0})$. Moreover, suppose there exists $j_1 \in S(u_k, a_{j_0}-1, \lambda - p_{\max}) \cup S(a_{j_0}, \beta, \lambda - p_{j_0})$ such that $\gamma \leq a_{j_1} < u_k$ at step k. Let us choose j_1 such that a_{j_1} is minimal, and let $u_{k+1} = a_{j_1}$.

Now let $j_2 \in S(u_{k+1}, u_k - 1, \lambda - p_{\max})$ be any job assigned between machines u_{k+1} and $u_k - 1$. We have $b_{j_2} \leq b_{j_1}$, otherwise the job j_1 would have been scheduled instead of job j_2. Hence, $b_{j_2} \leq \beta$ (by induction hypothesis), thus $b_j \leq \beta$ for all $j \in S(u_{k+1}, a_{j_0}-1, \lambda - p_{\max}) \cup S(a_{j_0}, \beta, \lambda - p_{j_0})$, because $S(u_{k+1}, u_k - 1, \lambda - p_{\max}) \cup S(u_k, a_{j_0}-1, \lambda - p_{\max}) = S(u_{k+1}, a_{j_0}-1, \lambda - p_{\max})$.

Then, either $a_{j_2} \geq u_{k+1}$ for all $j_2 \in S(u_{k+1}, u_k - 1, \lambda - p_{\max})$, or there exists $j_2 \in S(u_{k+1}, u_k - 1, \lambda - p_{\max})$ such that $\gamma \leq a_{j_2} < u_{k+1}$. In the first case, we conclude that $a_j \geq u_{k+1}$ for all $j \in S(u_{k+1}, a_{j_0}-1, \lambda - p_{\max}) \cup S(a_{j_0}, \beta, \lambda - p_{j_0})$, because we have chosen j_1 in a way that a_{j_1} is minimal (thus $a_j \geq a_{j_1} = u_{k+1}$ for all $j \in S(u_k, a_{j_0}-1, \lambda - p_{\max}) \cup S(a_{j_0}, \beta, \lambda - p_{j_0})$). Hence, we set $x = k+1$ and we stop there. In the second case, we proceed to the next step.

Therefore, we proved that we can find a machine u_x such that $a_j \geq u_x$ and $b_j \leq \beta$ for all $j \in S(u_x, a_{j_0}-1, \lambda - p_{\max}) \cup S(a_{j_0}, \beta, \lambda - p_{j_0})$. In other words, all jobs in the set $S(u_x, a_{j_0}-1, \lambda - p_{\max}) \cup S(a_{j_0}, \beta, \lambda - p_{j_0})$ come from the set $K_{\langle u_x, \beta \rangle}$.

Recall that all machines $a_{j_0}, a_{j_0}+1, \cdots, \beta$ finish after time $\lambda - p_{j_0}$, and by construction, all machines $u_x, u_x+1, \cdots, a_{j_0}-1$ finish after time $\lambda - p_{\max}$, because $u_x \geq \gamma$. Thus, we set $\alpha = u_x$, and we have

$$w_{\langle \alpha,\beta \rangle} > (\beta - \alpha + 1)(\lambda - p_{\max}) + (\beta - a_{j_0} + 1)(\lambda - p_{j_0} - (\lambda - p_{\max})) + p_{j_0},$$

which gives the following inequality:

$$\lambda < \tilde{w}_{\langle \alpha,\beta \rangle} - \frac{p_{j_0}}{\beta - \alpha + 1} - \frac{(\beta - a_{j_0} + 1)(p_{\max} - p_{j_0})}{\beta - \alpha + 1} + p_{\max}.$$

As $\beta - a_{j_0} + 1 \geq 1$ and $\beta - \alpha + 1 \leq m$, we have $\frac{\beta - a_{j_0}+1}{\beta - \alpha + 1} \geq \frac{\beta - a_{j_0}+1}{m} \geq \frac{1}{m}$, and as $p_{\max} - p_{j_0} \geq 0$,

$$\lambda < \tilde{w}_{\langle \alpha,\beta \rangle} - \frac{p_{j_0}}{\beta - \alpha + 1} - \frac{1}{m}(p_{\max} - p_{j_0}) + p_{\max},$$

thus,

$$\lambda < \tilde{w}_{\langle \alpha,\beta \rangle} + \left(1 - \frac{1}{m}\right) p_{\max} + \left(\frac{1}{m} - \frac{1}{\beta - \alpha + 1}\right) p_{j_0}.$$

Finally, we have $\frac{1}{\beta - \alpha + 1} \geq \frac{1}{m}$, i.e., $\frac{1}{m} - \frac{1}{\beta - \alpha + 1} \leq 0$, and $p_{j_0} \geq 0$. Therefore,

$$\lambda < \tilde{w}_{\langle \alpha,\beta \rangle} + \left(1 - \frac{1}{m}\right) p_{\max},$$

which is a contradiction.

Hence, ELFJ gives a schedule that is feasible in time λ, i.e., $C_{\max} \leq \lambda$. By Lemma 1, we have $C^{\text{OPT}}_{\max} \geq \tilde{w}_{\max}$, and obviously, $C^{\text{OPT}}_{\max} \geq p_{\max}$, so $\lambda \leq (2 - 1/m) \cdot C^{\text{OPT}}_{\max}$. We conclude that $C_{\max} \leq (2 - 1/m) \cdot C^{\text{OPT}}_{\max}$.

Note that this approximation ratio is tight. Consider an instance with one large job of size L (where L is an integer), and $L(m-1)$ unitary jobs. All jobs are feasible on all machines. Thus, we have $\lambda = (2 - 1/m)L$, which means that ELFJ will keep scheduling jobs until time $\lambda' = \lfloor (2 - 1/m)L \rfloor$. As L is an integer, there must exist positive integers c, d such that $L = cm + d$, with $d < m$. Therefore, $\lambda' = \lfloor (2-1/m)(cm+d) \rfloor = \lfloor 2(cm+d) - c + d/m \rfloor = 2(cm+d) - c$, as $d/m < 1$. The optimal solution has a makespan of L. Hence, the approximation ratio is

$$\lambda'/L = 2 - \frac{c}{cm+d} = 2 - \frac{1}{m + d/c},$$

which tends to $2 - 1/m$ as $c \to +\infty$. ■

V. A General Framework for Circular Intervals

In this section, we present a generalization of the RAI problem to so-called circular intervals, which match the usual replication strategy of key-value stores.

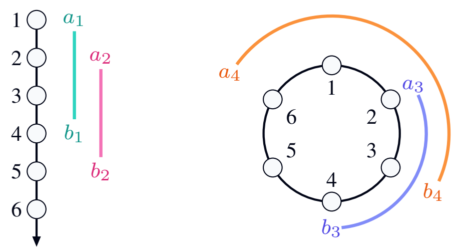


Fig. 5: Comparison between instances of the standard RAI problem (on the left) and its generalization to circular intervals (on the right). In this example, we have $a_1 = 1, b_1 = 4, a_2 = 2, b_2 = 5, a_3 = 2, b_3 = 4, a_4 = 6,$ and $b_4 = 3$. Thus, $\langle a_4, b_4\rangle$ is a circular interval, whereas the other intervals are regular.

A. Introducing Circular Intervals

In the standard RAI problem, machines are linearly arranged, that is to say, they are numbered from 1 to m and virtually placed on a line. As we have seen in the introduction, distributed key-value stores often organize machines in a virtual ring, where the machines able to answer a query for a particular key are consecutively arranged in this ring. We generalize here the notion of interval to take into account this setting. In addition to *regular* intervals $\langle a, b\rangle$ (with $a \le b$), we introduce *circular* intervals such that $a > b$. In this case, the corresponding set $I_{\langle a,b\rangle}$ includes machines $a, a+1, \cdots, m$ and machines $1, 2, \cdots, b$, i.e., we have

$$I_{\langle a,b\rangle} = \begin{cases} \{a, a+1, \cdots, b\} & \text{if } a \le b, \\ \{1, 2, \cdots, b\} \cup \{a, a+1, \cdots, m\} & \text{otherwise.} \end{cases}$$

Note that we clearly cannot always rearrange machines to transform an instance with circular intervals to an instance without circular intervals. Consider the instance with 3 machines and 3 jobs with processing sets $\mathcal{M}_1 = \{1, 2\}$, $\mathcal{M}_2 = \{2, 3\}$ and $\mathcal{M}_3 = \{3, 1\}$: any permutation of the machines will exhibit exactly one circular interval. Figure 5 illustrates the generalization of the RAI problem to circular intervals. By extension, we call this generalized problem the Restricted Assignment problem on Circular Intervals (RACI).

Definition 1. *The interval* $\langle a_g, b_g\rangle$ precedes *the interval* $\langle a_h, b_h\rangle$ *if and only if* $a_g \le a_h$ *and* $b_g \le b_h$. *In this case, we note* $\langle a_g, b_g\rangle \preceq \langle a_h, b_h\rangle$.

For a given instance, let Z^* be the set of circular intervals that are associated to at least one job ($Z^* = \{\langle a_j, b_j\rangle \text{ s.t. } j \in J \text{ and } a_j > b_j\}$). In this section, we restrict ourselves to instances where the previously-defined relation $\preceq$ is a total order on Z^*. In other words, for any $g, h \in Z^*$, we cannot have $I_g \subset I_h$ or $I_h \subset I_g$. This constitutes a particular case of RACI, but it is still a more general case than RAI. Moreover, we assume that there are K types of jobs, and each job of type k has processing time $p(k)$.

B. An Optimal Procedure for K Job Types

We introduce in this section a general procedure that solves the RACI problem for the described restricted instances,

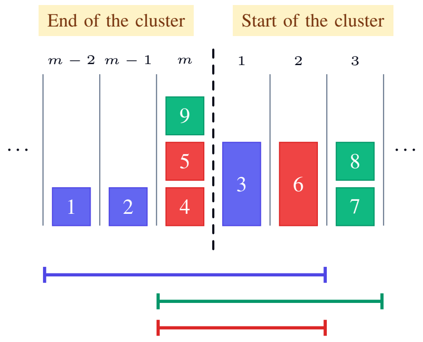


Fig. 6: Example of circular jobs in a schedule. Colors denote jobs with common processing intervals. Jobs 1, 2, 4, 5, 9 are left jobs, whereas jobs 3, 6, 7, 8 are right jobs. Moreover, there are two types of jobs. Jobs 1, 2, 4, 5, 7, 8, 9 are of type 1 (with $p(1) = 1$), whereas jobs 3 and 6 are of type 2 (with $p(2) = 2$). Thus, in this example, $G_1 = \{1, 2, 4, 5, 9\}$, $D_1 = \{7, 8\}$, $G_2 = \emptyset$, and $D_2 = \{3, 6\}$.

assuming that one already knows an optimal algorithm $\mathcal{A}$ for the standard RAI problem with K job types.

Theorem 3. *Let $\mathcal{A}$ be an optimal algorithm for the RAI problem with K job types running in time $O(f(n))$. Then there exists a procedure that solves the corresponding RACI problem on totally ordered circular intervals in time $O(n^K f(n))$.*

We begin with a few definitions. Then we present the procedure, before proving our result.

Preliminaries. Let J^* be the subset of jobs whose processing set is a circular interval, i.e., $J^* = \{j \in J \text{ s.t. } a_j > b_j\}$, and we note $n^* = |J^*|$. We call J^* the *circular* jobs (by extension, the jobs $J \setminus J^*$ are called *regular* jobs). We also partition J^* into K subsets $J_1^*, \cdots, J_K^*$, such that all jobs in J_k^* are of type k, and we note $n_k^* = |J_k^*|$.

Moreover, in a given schedule, we say that a circular job j assigned between a_j (inclusive) and the last machine m (inclusive) is a *left* job. Equivalently, a circular job j assigned between the first machine 1 (inclusive) and b_j (inclusive) is a *right* job. This means that a schedule π implicitly defines a partition of each set J_k^* into two subsets G_k and D_k, where G_k contains γ_k left jobs, and D_k contains δ_k right jobs. Figure 6 shows an example of such schedule.

We present here the intuition on how to compute an optimal schedule by considering all possible partitions of jobs with circular processing sets into left and right jobs. For the moment, we simplify the problem by considering only one type of jobs. A schedule defines a partition of jobs J^* into left and right jobs, which means that we need to find *how many* jobs in J^* should be assigned to the left or to the right. Thus, assume that we know that r jobs of J^* must be assigned to the right in an optimal schedule. Intuitively, the r circular jobs with rightmost intervals should be put on the right, and the remaining jobs of J^* should be put on the left. For example, consider only the small jobs in the instance of Figure 6. If we suppose that $r = 5$ (arbitrarily), then we guess that the

2 red jobs and the 3 green jobs should be put on the right (i.e., between machines 1 and 3), and the 2 blue jobs should be put on the left (i.e., between machines $m-2$ and m), as the red and green intervals are more on the right than the blue interval. We introduce below the notion of *right-sorted* schedules that captures this intuition, and we will prove later that there always exists at least one optimal schedule that has this property.

Definition 2. *A schedule π is* right-sorted *if and only if for all types k, the property $\langle a_j, b_j\rangle \preceq \langle a_{j'}, b_{j'}\rangle$ holds for any jobs $j \in G_k$ and $j' \in D_k$.*

We denote the set of all possible schedules for a given instance $\mathcal{I}$ by $\Pi(\mathcal{I})$ ($\mathcal{I}$ is omitted when it is clear from the context). Let $\mathbf{R}^K$ be the set of all vectors $\mathbf{r} = (r_1, \cdots, r_K)$ such that r_k is an integer and $0 \leq r_k \leq n_k^*$ for all k ($n*_k$ is the number of circular jobs of type K). For a given vector $\mathbf{r} \in \mathbf{R}^K$, we call $\Pi_\mathbf{r}$ the subset of schedules Π that put exactly r_k jobs of type k on the right, and $n_k^* - r_k$ jobs of type k on the left. Recall that $C_{\max}^{\mathrm{OPT}}$ denotes the optimal makespan among all schedules Π. We define analogously $C_\mathbf{r}^{\mathrm{BEST}}$ as the *best possible makespan* among schedules $\Pi_\mathbf{r}$. Note that the subsets $\Pi_\mathbf{r}$ define a partition of Π, and thus

$$C_{\max}^{\mathrm{OPT}} = \min_{\mathbf{r}\in\mathbf{R}^K} \left\{C_\mathbf{r}^{\mathrm{BEST}}\right\}. \tag{1}$$

Optimal procedure. We introduce a polynomial procedure $\phi_\mathbf{r}$ that transforms any instance $\mathcal{I}$ of the (totally ordered) RACI problem into another instance $\mathcal{I}' = \phi_\mathbf{r}(\mathcal{I})$ that does not include any circular interval (i.e., $\mathcal{I}'$ is an instance of RAI). We prove later the following two statements:

(i) Applying an optimal algorithm $\mathcal{A}$ to $\mathcal{I}'$ produces a valid solution for $\mathcal{I}$.
(ii) The makespan of this solution is at most $C_\mathbf{r}^{\mathrm{BEST}}$.

Given these statements and Equation (1), we can find an optimal solution for $\mathcal{I}$ by performing an exhaustive search of the best vector $\mathbf{r} \in \mathbf{R}^K$. For a given instance $\mathcal{I}$, the function $\phi_\mathbf{r}$ works as follows:

1) Sort jobs J^* by non-increasing order of b_j, and sort jobs with identical b_j by non-increasing order of a_j. Note that this corresponds to sorting jobs by non-increasing order of $\preceq$. As $\preceq$ is a total order on Z^*, all jobs are comparable.
2) For each type k, set $a_j = 1$ for the r_k first jobs of J_k^*, and $b_j = m$ for the $n_k^* - r_k$ other jobs, effectively removing circular intervals.

Let $\Pi_\mathbf{r}^{\preceq}$ be the subset of schedules $\Pi_\mathbf{r}$ that are right-sorted. The proof of the two statements of interest is structured as follows. As $\mathcal{A}$ is optimal for the standard RAI problem, we know that it finds one of the best schedules among $\Pi(\phi_\mathbf{r}(\mathcal{I}))$. Hence, we will prove two lemmas. On the one hand, we show in Lemma 2 that the set $\Pi(\phi_\mathbf{r}(\mathcal{I}))$ is exactly the same as the set of right-sorted schedules $\Pi_\mathbf{r}^{\preceq}(\mathcal{I})$ for the initial instance. On the other hand, we show in Lemma 3 that there always exists a right-sorted schedule that has the best possible makespan.

Lemma 2. *For any $\mathbf{r} \in \mathbf{R}^K$, we have $\Pi(\phi_\mathbf{r}(\mathcal{I})) = \Pi_\mathbf{r}^{\preceq}(\mathcal{I})$, i.e., the schedules produced from the transformed instance are the same as the right-sorted ones from the initial instance.*

Proof: Let $\mathbf{r}$ be an arbitrary vector of $\mathbf{R}^K$. First we show that $\Pi(\phi_\mathbf{r}(\mathcal{I})) \subseteq \Pi_\mathbf{r}^{\preceq}(\mathcal{I})$. Let $\pi \in \Pi(\phi_\mathbf{r}(\mathcal{I}))$. By definition of $\phi_\mathbf{r}$, for all types k, there are $n_k^* - r_k$ jobs in π that were circular jobs in the initial instance $\mathcal{I}$ and that are on the left (similarly, there are r_k jobs in π that were circular and that are on the right). Moreover, the circular jobs have been sorted in $\phi_\mathbf{r}$, which means that for all k, we have $\langle a_j, b_j\rangle \preceq \langle a_{j'}, b_{j'}\rangle$ for any $j \in G_k$ and $j' \in D_k$ in π. In other words, π is right-sorted, and thus belongs to $\Pi_\mathbf{r}^{\preceq}$.

Now we show that $\Pi_\mathbf{r}^{\preceq}(\mathcal{I}) \subseteq \Pi(\phi_\mathbf{r}(\mathcal{I}))$. By definition of $\Pi_\mathbf{r}^{\preceq}$, in any schedule $\pi \in \Pi_\mathbf{r}^{\preceq}(\mathcal{I})$, for all types k, we have $\langle a_j, b_j\rangle \preceq \langle a_{j'}, b_{j'}\rangle$ for any $j \in G_k$ and $j' \in D_k$. Moreover, there are exactly $n_k^* - r_k$ jobs in G_k and r_k jobs in D_k. Thus, π is clearly a valid solution for $\phi_\mathbf{r}(\mathcal{I})$ and belongs to $\Pi(\phi_\mathbf{r}(\mathcal{I}))$. ■

Lemma 3. *For any $\mathbf{r} \in \mathbf{R}^K$, there exists a right-sorted schedule $\pi \in \Pi_\mathbf{r}^{\preceq}$ that has the best possible makespan $C_\mathbf{r}^{\mathrm{BEST}}$.*

Proof: Let $\mathbf{r}$ be an arbitrary vector of $\mathbf{R}^K$. Let $\pi \in \Pi_\mathbf{r}$ be a schedule that has the best possible makespan $C_\mathbf{r}^{\mathrm{BEST}}$. If π is right-sorted, we are done. Otherwise, there necessarily exists a type k such that two jobs $j \in G_k$ and $j' \in D_k$, scheduled in π, are not sorted according to $\preceq$, i.e., we have $\langle a_j, b_j\rangle \npreceq \langle a_{j'}, b_{j'}\rangle$. In other words, either $a_j > a_{j'}$, or $b_j > b_{j'}$. We know that $\preceq$ is a total order on Z^*, which means that if $a_j > a_{j'}$, then we necessarily have $b_j \geq b_{j'}$. In a similar way, if $b_j > b_{j'}$, then we necessarily have $a_j \geq a_{j'}$. This means that even if j is a left job and j' is a right job in π, there is "more room" to put j on the right side and j' on the left side of their respective interval. Moreover, as j and j' have the same type k, they have identical processing times. Hence, we can clearly swap j and j' in π without changing the makespan of π.

By repeatedly swapping non-sorted jobs of the same type, we reach another schedule π' that has the same makespan than π, that also belongs to $\Pi_\mathbf{r}$, and that is right-sorted. ■

Now we are able to conclude.

Proof of Theorem 3: By hypothesis, we know that $\mathcal{A}$ finds a schedule with the smallest makespan among $\Pi(\phi_\mathbf{r}(\mathcal{I}))$. By Lemma 3, we also know that there exists at least one schedule in $\Pi_\mathbf{r}^{\preceq}(\mathcal{I})$ that has the best possible makespan $C_\mathbf{r}^{\mathrm{BEST}}$. Therefore, we deduce by Lemma 2 that:

- the solution given by $\mathcal{A}$, when applied to $\phi_\mathbf{r}(\mathcal{I})$, belongs to $\Pi_\mathbf{r}^{\preceq}(\mathcal{I})$, which means that it also belongs to $\Pi_\mathbf{r}(\mathcal{I})$, i.e., it is a valid solution for $\mathcal{I}$ (Statement (i)), and
- the solution given by $\mathcal{A}$, when applied to $\phi_\mathbf{r}(\mathcal{I})$, has makespan $C_\mathbf{r}^{\mathrm{BEST}}$ (Statement (ii)).

It follows that, for any instance $\mathcal{I}$, we can find the best possible schedule among $\Pi_\mathbf{r}(\mathcal{I})$ for any vector $\mathbf{r} \in \mathbf{R}^K$. Moreover, for all vectors $\mathbf{r} \in \mathbf{R}^K$, we have $r_k \leq n_k^* \leq n$ for all k. Thus, the number of possible vectors $\mathbf{r}$ is bounded by

$O(n^K)$, i.e., we can find an optimal schedule for any instance $\mathcal{I}$ by searching over all possible vectors in time $O(n^K f(n))$, assuming that we know an algorithm $\mathcal{A}$ that runs in time $O(f(n))$ when applied to $\phi_{\mathbf{r}}(\mathcal{I})$. This concludes the proof of Theorem 3. ■

We now study two special cases where this procedure can be applied: the adaptation of an existing dynamic programming algorithm for K job types and the ELFJ algorithm presented above. In the latter case, we are able to largely reduce the complexity compared to Theorem 3, as we achieve for ELFJ on circular intervals the same complexity as ELFJ on regular intervals.

C. A Dynamic Program for K Job Types

We illustrate how our framework can be successfully applied to derive a polynomial algorithm for the RACI problem on intervals of equal length and K job types. Wang et al. [18] showed how to solve the corresponding problem on regular intervals with a dynamic program. For completeness, we recall their solution in the following.

Let n_k be the number of jobs of type k ($1 \leq k \leq K$), and let us sort jobs by non-decreasing value of b_j. Suppose that λ is a value that represents a hard deadline for all jobs. Define $F_i(s_1, s_2, \cdots, s_K) = 1$ if and only if it is feasible, for all types k, to schedule s_k jobs of type k on machines $1, 2, \cdots, i$ such that the makespan is at most λ, and $F_i(s_1, s_2, \cdots, s_K) = 0$ otherwise.

Let $F_0(0, \cdots, 0) = 1$, and $F_i(s_1, s_2, \cdots, s_K) = 1$ if and only if there exist $s'_1 \leq s_1, s'_2 \leq s_2, \cdots, s'_K \leq s_K$ such that:

(i) $F_{i-1}(s'_1, s'_2, \cdots, s'_K) = 1$,
(ii) for each k, the next $s_k - s'_k$ jobs of type k in the sorted set contain the machine i in their processing set, and
(iii) $\sum_{k=1}^{K} (s_k - s'_k) \cdot p(k) \leq \lambda$.

Then we have $F_m(n_1, n_2, \cdots, n_K) = 1$ if and only if there exists a schedule feasible in time λ. For a given value of λ, an array with all values of F can be computed in time $O(mn^{2K})$. Finally, the optimal value of λ can be found by performing a binary search. Thus the overall complexity of the algorithm is $O(mn^{2K} \log \sum p_j)$.

By using our framework, adapting this approach to the RACI problem is straightforward. Let $\mathcal{A}$ be the dynamic program described above. By Theorem 3, we know that we can find an optimal schedule for any instance in time $O(n^K f(n))$, where $f(n)$ is the complexity of $\mathcal{A}$. Therefore, the time complexity of the derived algorithm is $O(mn^{3K} \log \sum p_j)$.

D. Revisiting the Unitary Job Case

We proved in Theorem 1 that ELFJ is an optimal algorithm for the standard RAI problem on unitary jobs, which runs in time $f(n) = O(m^2 + n \log n + mn)$. Recall that ELFJ consists in 3 distinct steps:

1) computing the optimal makespan, in time $O(m^2 + n)$,
2) sorting the jobs, in time $O(n \log n)$,
3) performing the actual job assignment, in time $O(mn)$.

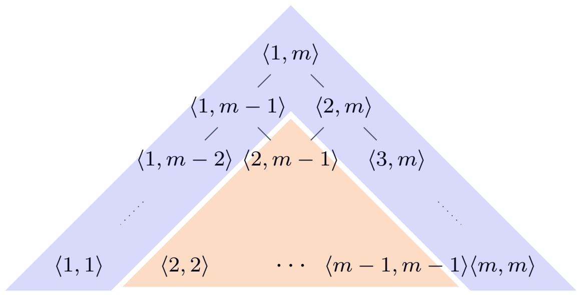


Fig. 7: Outside and inside intervals in the lattice graph representation. Outside intervals (blue area) are of the form $\langle 1, x\rangle$ or $\langle x, m\rangle$, with $1 \leq x \leq m$, and inside intervals (red area) are of the form $\langle x, y\rangle$, with $1 < x \leq y < m$.

By applying our framework around ELFJ, and because we have only one type of jobs in this specific case, we know from Theorem 3 that we can solve the generalized problem on totally ordered circular intervals in time $O(nf(n)) = O(m^2 n + n^2 \log n + mn^2)$. We now show how to improve the complexity of this solution, as stated in the following theorem.

Theorem 4. *The totally ordered RACI problem with unitary jobs can be solved in time $O(m^2 + n \log n + mn)$.*

Proof: The basic idea is to extract and reorganize some internal computation steps from the exhaustive search procedure to avoid doing any redundant work. We first observe that the only step of ELFJ that actually depends on knowing an optimal number r of right jobs (in the set of circular jobs) is the computation of the optimal makespan. Once we know the best values of r and λ, the sorting and job assignment steps are straightforward. Thus, we know that we can easily refine the complexity to $O(n(m^2 + n) + n \log n + mn) = O(m^2 n + n^2)$.

To reduce further the complexity, we notice that we do not really need to recompute the matrix w from the beginning (in time $O(m^2 + n)$) for each possible value of r in order to find the minimum makespan. We remark that there are two kinds of regular intervals: the ones that may result from cutting a circular interval $\langle a, b\rangle$ in two sub-intervals $\langle a, m\rangle$ and $\langle 1, b\rangle$, which we call the *outside* intervals, and the ones that cannot, which we call *inside* intervals. In other words, outside intervals are all regular intervals of the form $\langle 1, x\rangle$ or $\langle x, m\rangle$, with $1 \leq x \leq m$, and inside intervals are all regular intervals of the form $\langle x, y\rangle$ with $1 < x \leq y < m$. When representing the regular interval hierarchy as a lattice graph, the outside intervals are in fact all the nodes on the sides of the lattice, and the inside intervals are the others, as shown in Figure 7.

Recall that $w_{\langle \alpha, \beta\rangle}$ represents the total work of all regular jobs whose interval is included in $\langle \alpha, \beta\rangle$. When we update our guess on the optimal number of right jobs, we transform the instance by shrinking the intervals of circular jobs: if a job j is a right job, we keep the right part of the interval, i.e., the sub-interval $\langle 1, b_j\rangle$, and if it is a left job, we keep the left part of the interval, i.e., the sub-interval $\langle a_j, m\rangle$. This means that the only values of w that may change when we transform the instance are the ones that are associated to the outside intervals. All other values remain unchanged, no matter how we partition the circular jobs.

Hence, we can decompose the computation of $\tilde{w}_{\max}$ in two steps. First, compute the value

$$\tilde{w}_{\max}^{inside} = \max_{1<\alpha\leq\beta<m} \left\{\tilde{w}_{\langle\alpha,\beta\rangle}\right\},$$

which represents the maximum value of $\tilde{w}$ among all inside intervals. This value does not depend on r, and can be computed only once. Second, compute the value

$$\tilde{w}_{\max}^{outside} = \max_{1\leq x\leq m} \left\{\max\left(\tilde{w}_{\langle 1,x\rangle}, \tilde{w}_{\langle x,m\rangle}\right)\right\},$$

which represents the maximum value of $\tilde{w}$ among all outside intervals. We clearly have

$$\tilde{w}_{\max} = \max\left(\tilde{w}_{\max}^{inside}, \tilde{w}_{\max}^{outside}\right).$$

In other words, each time we update our guess on r, we only need to recompute the value of $\tilde{w}_{\max}^{outside}$, which can be done in time $O(m)$ as there are exactly $2m-1$ outside intervals. Thus, we can search for the minimum value of $\max\left(\tilde{w}_{\max}^{inside}, \tilde{w}_{\max}^{outside}\right)$ by pre-computing $\tilde{w}_{\max}^{inside}$, and then trying each possible value of r by updating only $\tilde{w}_{\max}^{outside}$.

The only remaining question is how we know which values to update in the matrix w when we make a new guess on r. We avoid recomputing the values that are associated to inside intervals. We can also avoid recomputing the value $w_{\langle 1,m\rangle}$, as it is always exactly equal to n, and we have $\tilde{w}_{\langle 1,m\rangle} = n/m$. Recall that the set of circular jobs is sorted by decreasing order of $\preceq$ (by definition of $\phi_{\mathbf{r}}$), and for a given number r, we know that the first r jobs in the sorted set are right jobs. We set $r=0$ and we pre-compute w, $\tilde{w}_{\max}^{inside}$ and $\tilde{w}_{\max}$. Then we loop over the sorted set of circular jobs by adding them progressively on the right side, i.e., for each job $j \in J^*$, we add 1 to $w_{\langle 1,\beta\rangle}$ for all $b_j \leq \beta < m$ and we subtract 1 to $w_{\langle\alpha,m\rangle}$ for all $1 < \alpha \leq a_j$. The full procedure is given in Algorithm 3.

We conclude that the RACI problem with totally ordered circular intervals and unitary jobs can be solved in time $O(m^2 + n\log n + mn)$, or $O(n \log n)$ if we assume that m is fixed. ■

VI. An Approximation for the Restricted Assignment Problem on Circular Intervals

In this section, we introduce an approximation algorithm to assign jobs on circular intervals, based on the following intuition: under certain conditions, it is possible to split the problem into two sub-problems, such that each of them consider only regular intervals. On each of these sub-problems, we can use the $(2-1/m)$-approximation algorithm presented in Section IV-C to get a guaranteed solution.

We consider jobs whose processing set is a circular interval, that is $J^* = \{j \in J \text{ s.t. } a_j > b_j\}$, and we define the smallest "left" index of these intervals, namely $z_{left} = \min_{j\in J^*}\{a_j\}$, as well as their largest "right" index $z_{right} = \max_{j\in J^*}\{b_j\}$. We assume in this section that the "leftmost" circular interval

Algorithm 3 Computing r and $\tilde{w}_{\max}$

```
1:  sort circular jobs by decreasing order of $\preceq$
2:  transform circular jobs as left jobs, then compute $w$,
    $\tilde{w}_{\max}^{inside}$ and $\tilde{w}_{\max}$
3:  $r_{cur} \leftarrow 0$
4:  for all circular jobs $j \in J^*$ do
5:      $\tilde{w}_{\max}^{outside} \leftarrow \tilde{w}_{\langle 1,m\rangle}$
6:      for all $\beta$ from $b_j$ to $m-1$ do
7:          $w_{\langle 1,\beta\rangle} \leftarrow w_{\langle 1,\beta\rangle} + 1$
8:          $\tilde{w}_{\langle 1,\beta\rangle} \leftarrow \frac{w_{\langle 1,\beta\rangle}}{\beta}$
9:          if $\tilde{w}_{\langle 1,\beta\rangle} > \tilde{w}_{\max}^{outside}$ then
10:             $\tilde{w}_{\max}^{outside} \leftarrow \tilde{w}_{\langle 1,\beta\rangle}$
11:     for all $\alpha$ from 2 to $a_j$ do
12:         $w_{\langle\alpha,m\rangle} \leftarrow w_{\langle\alpha,m\rangle} - 1$
13:         $\tilde{w}_{\langle\alpha,m\rangle} \leftarrow \frac{w_{\langle\alpha,m\rangle}}{m-\alpha+1}$
14:         if $\tilde{w}_{\langle\alpha,m\rangle} > \tilde{w}_{\max}^{outside}$ then
15:             $\tilde{w}_{\max}^{outside} \leftarrow \tilde{w}_{\langle\alpha,m\rangle}$
16:     $r_{cur} \leftarrow r_{cur} + 1$
17:     $\tilde{w}_{cur} \leftarrow \max\left(\tilde{w}_{\max}^{inside}, \tilde{w}_{\max}^{outside}\right)$
18:     if $\tilde{w}_{cur} < \tilde{w}_{\max}$ then
19:         $r \leftarrow r_{cur}$
20:         $\tilde{w}_{\max} \leftarrow \tilde{w}_{cur}$
```

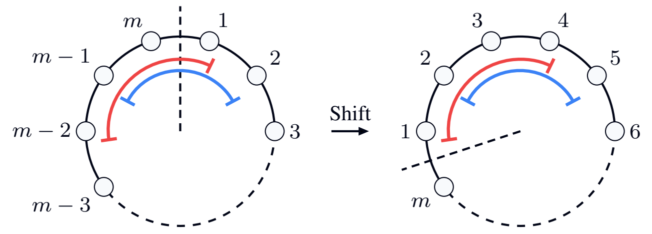


Fig. 8: Shifting the circular intervals "to the left" to transform them into regular intervals. In this example, there are two circular intervals (red and blue). Moreover, $z_{left} = m-2$ and $z_{right} = 2$. The shifted machine corresponding to the machine with index i has index $i - z_{left} + 1$ if $i \geq z_{left}$, $i - z_{left} + 1 + m$ otherwise.

does not intersect the "rightmost" circular interval, that is, $z_{left} > z_{right}$. This assumption holds in particular for intervals of size k if and only if $m \geq 2(k-1)$, as $z_{left} \geq m-(k-1)+1$ and $z_{right} \leq k-1$, i.e., $z_{left} - z_{right} \geq m - 2k + 3$.

The proposed algorithm, named Double ELFJ (DELFJ) and presented in Algorithm 5, works as follows: jobs with regular intervals are first allocated on the servers using the $(2-1/m)$-approximation algorithm ELFJ presented in Section IV-C. To allocate the remaining jobs (from J^*), we use the same algorithm. However, ELFJ only handles regular intervals. Hence, we first shift all intervals so that the leftmost circular intervals start on machine 1 before applying ELFJ (see Figure 8), and we shift back the allocation in the end. Thanks to our assumption on z_{left} and z_{right}, we know that shifting the initially circular intervals will result in all these intervals becoming regular. As the two categories of jobs are

Algorithm 4 Shifting functions

```
1: function ShiftLeft($(x, y)$)
2:     $z \leftarrow x - y$
3:     if $z < 1$ then
4:         $z \leftarrow z + m$
5:     return $z$
6: function ShiftRight($(x, y)$)
7:     $z \leftarrow x + y$
8:     if $z > m$ then
9:         $z \leftarrow z - m$
10:    return $z$
```

Algorithm 5 Double ELFJ (DELFJ)

```
Input: jobs $J$ and machines $M$
Output: an assignment $\mu$
1: $J^* \leftarrow \{j \in J$ s.t. $a_j > b_j\}$
2: $\mu \leftarrow$ apply ELFJ on jobs $J \setminus J^*$
3: $z_{left} \leftarrow \min_{j \in J^*} \{a_j\}$
4: for all jobs $j \in J^*$ do
5:     $a_j \leftarrow$ ShiftLeft$(a_j, z_{left} - 1)$
6:     $b_j \leftarrow$ ShiftLeft$(b_j, z_{left} - 1)$
7: $\mu^* \leftarrow$ apply ELFJ on jobs $J^*$
8: for all jobs $j \in J^*$ do
9:     $\mu_j \leftarrow$ ShiftRight$(\mu^*_j, z_{left} - 1)$
10: return $\mu$
```

allocated separately using a $(2 - 1/m)$-approximation, we obtain a $(4 - 2/m)$-approximation algorithm, as stated in the following theorem.

Theorem 5. Double ELFJ *(Algorithm 5) is a tight* $(4 - 2/m)$*-approximation algorithm provided that* $z_{left} > z_{right}$.

Proof: Let $\mathcal{I}$ be the initial instance and $\mathcal{I}_1$ be the same instance $\mathcal{I}$ restricted to jobs with non-circular intervals, i.e, with jobs in $J \setminus J^*$. ELFJ applied to $\mathcal{I}_1$ is a $2 - 1/m$ approximation, hence $\mathrm{ELFJ}(\mathcal{I}_1) \leq (2 - 1/m) \cdot \mathrm{OPT}(\mathcal{I}_1)$.

We consider the instance $\mathcal{I}_2$ made of remaining jobs, i.e, with jobs from J^*, but where all intervals have been shifted so that the leftmost intervals starts at machine 1. More formally, a circular interval $\langle \alpha, \beta \rangle$ is transformed into $\langle \alpha - z_{left} + 1, \beta - z_{left} + m + 1 \rangle$ by applying the ShiftLeft function. We assumed that $z_{left} > z_{right}$, hence we have

$$\beta - z_{left} + m + 1 \leq z_{right} - z_{left} + m + 1 < m + 1$$

which proves that all transformed intervals are non-circular. When applying ELFJ on $\mathcal{I}_2$, we also have $\mathrm{DELFJ}(\mathcal{I}) \leq \mathrm{ELFJ}(\mathcal{I}_1) + \mathrm{ELFJ}(\mathcal{I}_2)$. Both $\mathcal{I}_1$ and $\mathcal{I}_2$ are subsets of $\mathcal{I}$ (with possible machine renumbering), hence
$\mathrm{OPT}(\mathcal{I}_1) \leq \mathrm{OPT}(\mathcal{I})$ and $\mathrm{OPT}(\mathcal{I}_2) \leq \mathrm{OPT}(\mathcal{I})$. Thus, $\mathrm{DELFJ}(\mathcal{I}) \leq 2(2 - 1/m) \cdot \mathrm{OPT}(\mathcal{I})$.

Note that this approximation ratio is tight. Let ℓ be an arbitrary integer such that $\ell \geq 2$. Consider an instance with $m = 3 + 2\ell$ and the following intervals of machines: $I_{\langle 3,m \rangle}$ (non-circular), $I_{\langle 3,1 \rangle}$ (circular), $I_{\langle 2,2 \rangle}$ (non-circular) and $I_{\langle m,1 \rangle}$ (circular). One large job of size L (where L is an integer) and $L(\ell - 1)$ unitary jobs must be done on $I_{\langle 3,m \rangle}$. The exact same set of jobs must be done on $I_{\langle 3,1 \rangle}$. Moreover, one job of size L must be done on $I_{\langle 2,2 \rangle}$, and 2 jobs of size L must be done on $I_{\langle m,1 \rangle}$. Note that, for this kind of instance, an optimal solution would have a makespan of L.

In the first round (i.e., assignment of non-circular jobs only), we necessarily have $\tilde{w}_{\max} = L$, because a large job must be done on $I_{\langle 2,2 \rangle}$, which consists of one machine. Thus, the estimated makespan is $\lambda_1 = (2 - 1/m)L$, and Double ELFJ will keep assigning jobs on $I_{\langle 3,m \rangle}$ until time $\lambda'_1 = \lfloor (2 - 1/m)L \rfloor$. In particular, the machine 3 is necessarily busy until time λ'_1.

In the second round (i.e., assignment of circular jobs only), we also necessarily have $\tilde{w}_{\max} = L$, because 2 large jobs must be done on $I_{\langle m,1 \rangle}$, which consists of 2 machines. Thus, the estimated makespan is $\lambda_2 = (2 - 1/m)L$, and Double ELFJ will keep assigning jobs on $I_{\langle 3,1 \rangle}$ until time $\lambda'_2 = \lfloor (2 - 1/m)L \rfloor$. As the second round is completely oblivious to the first round, the machine 3, which was already busy until time λ'_1, will be filled until time $\lambda'_1 + \lambda'_2$.

To summarize, the approximation ratio for this instance is $2\lfloor (2 - 1/m)L \rfloor / L$. From the end of Theorem 2, we know that $\lfloor (2 - 1/m)L \rfloor / L$ tends to $2 - 1/m$ as $L \to +\infty$, thus, $2\lfloor (2 - 1/m)L \rfloor / L$ tends to $4 - 2/m$ as $L \to +\infty$, which concludes the proof.

■

VII. Experimental Evaluation

We now derive a new heuristic from our guaranteed algorithm DELFJ to partition multi-get requests, and we perform a series of experiments to evaluate its practical performance.

A. Introducing the DSLFJ Heuristic

Let us begin with the description of the new heuristic that we introduce in this section, called Double Searched LFJ (DSLFJ). It is based on the same principle as the guaranteed algorithm DELFJ of the previous section, i.e., it assigns jobs in two rounds. Recall that DELFJ uses ELFJ as a sub-algorithm to assign jobs in each round, with the estimated makespan $\lambda = \tilde{w}_{\max} + p_{\max}$. As a consequence, it suffers from the fact that it keeps putting jobs on the same machine until it reaches λ, which differs from the optimal by a factor of $2 - 1/m$.

The heuristic DSLFJ also performs two rounds (i.e., it assigns regular jobs in the first round, and circular jobs in the second), but it uses a different sub-algorithm called Searched LFJ (Algorithm 6). This variant no longer computes an approximated objective value, but instead progressively searches for a feasible makespan by successively applying ELFJ, starting from $\lceil \tilde{w}_{\max} \rceil$ (which is a lower bound on the optimal makespan). The searching procedure directly depends on a function UpdateMakespan, which defines how the makespan λ grows through iterations: a slow progression will yield a better final objective, but the worst-case time complexity will necessarily be higher. Note that, as we proved in Theorem 2 that ELFJ necessarily finds a feasible solution

Algorithm 6 SEARCHED LFJ (SLFJ)

```
Input: jobs $J$ and machines $M$
Output: an assignment $\mu$
1: compute $\tilde{w}_{\max}$
2: $\delta \leftarrow 0$
3: repeat
4:     $\mu \leftarrow$ apply ELFJ on jobs $J$ with $\lambda = \lceil \tilde{w}_{\max} \rceil + \delta$
5:     $\delta \leftarrow$ UPDATEMAKESPAN($\delta$)
6: until all jobs $J$ are assigned in $\mu$
7: return $\mu$
```

if $\lambda = \lceil \tilde{w}_{\max} \rceil + p_{\max}$, there can be at most $p_{\max}$ iterations in SLFJ. In practice, it is possible that a feasible solution exists for a given makespan lower than $\lceil \tilde{w}_{\max} \rceil + p_{\max}$, but that SLFJ fails to find it. To increase the chances of finding a feasible solution as soon as possible, we perform two additional improvements over DELFJ.

First, we choose the start of the ring more carefully. Until now, we considered that jobs whose interval intersects the first machine of the cluster were the circular jobs J^* of the instance. This had no impact on our approximation algorithms. However, it may happen that the first machine has more potential work to do than the others, leading the practical heuristic to make non-optimal decisions and fail to find a feasible solution with a small makespan. In DSLFJ, we choose the start of the ring such that the first machine has the least potential work to perform, i.e., we choose the machine i such that $\sum_{j \in J \text{ s.t. } i \in I_{\langle a_j, b_j \rangle}} p_j$ is minimized.

Second, we use the partial assignment performed in the first round to make better decisions in the second round by taking into account the load of each machine. For all machines i, we create a new dummy job j feasible only on i (i.e., $a_j = b_j = i$), with a processing time p_j that is equal to the total work assigned to i during the first round (i.e., $p_j = \sum_{j' \in J \text{ s.t. } \mu_{j'} = i} p_{j'}$).

By a simple analysis, we find that the worst-case time complexity of DSLFJ is $O(m^2 + n \log n + c \cdot mn)$, where c is the worst-case number of iterations in SLFJ, which directly depends on the function UPDATEMAKESPAN. The complexity of DSLFJ corresponds to the complexity of SLFJ (as it dominates all the other steps):

- $\tilde{w}_{\max}$ is computed in time $O(m^2 + n)$,
- the job sorting step of ELFJ may be done only once outside the loop in SLFJ, in time $O(n \log n)$, and
- the assignment step of ELFJ is done in time $O(mn)$, and is repeated at most c times.

In the following, we consider two variants of DSLFJ, according to the function UPDATEMAKESPAN:

1) ARITHMETICALLY-SEARCHED LFJ (ASLFJ), which increments δ in each iteration (i.e., $\delta \leftarrow \delta + 1$ at line 4 of Algorithm 6), and
2) GEOMETRICALLY-SEARCHED LFJ (GSLFJ), which doubles δ in each iteration (i.e., $\delta \leftarrow \max(1, 2\delta)$ at line 4 of Algorithm 6).

Their time complexities are respectively $O(m^2 + n \log n + mn \cdot p_{\max})$ and $O(m^2 + n \log n + mn \cdot \log p_{\max})$.

B. Experimental Settings

We test the quality of our two heuristics ASLFJ and GSLFJ in simulations. The key-value store is characterized by a number of machines m and a replication factor k, which defines the size of each interval of machines. The usual replication factor in practice is 3, i.e., each data item is available on 3 different servers [12]. We generate a dataset of 100 000 keys, and we uniformly assign each key to a random machine.

Each key κ is associated a corresponding service time t_κ, which is drawn from an exponential distribution with mean 12. This setting models a workload where the service time of each key is small with high probability (e.g., the probability that a given key is serviced within 30 time units is higher than 0.9), which corresponds to realistic measures in key-value stores [1]. The processing time of each job is set to the service time of the corresponding requested key (i.e., $p_j = t_\kappa$ if the job j reads the key κ).

Each multi-get request is parameterized by the number of keys n that are requested, and the chosen keys that are drawn according to a given popularity distribution. In the following, we consider two popularity distributions: the uniform distribution, where each key has the same probability to be chosen, and the Zipf distribution, where the probability of choosing a key is inversely proportional to its rank in the generated key list. The Zipf distribution is the default in most benchmarks [3], as it generates a skewed popularity distribution, e.g., the key with rank 1 has 2 more chances to be chosen than the key with rank 2, which has itself 1.5 more chances to be chosen than the key with rank 3, etc.

We compare our two heuristics ASLFJ and GSLFJ with the following algorithms:

- RANDOM, which randomly assigns each job to a compatible machine,
- EFT-MIN, which assigns each job to the first compatible machine that completes the job the earliest, and
- EFT-RAND, which assigns each job to a random compatible machine among the ones that complete the job the earliest.

EFT-MIN is a strategy that actual key-value stores tend to use, even if it is never perfectly implemented in practice due to the usual constraints of distributed systems [17]. When the instance size is not prohibitive, we also compare our heuristics with the optimal solution computed by a Mixed Integer Linear Program (MILP) solver.

C. Results

In the following, we evaluate the performance from two different perspectives: the response time of individual requests, and the maximum attainable throughput of the system on a stream of requests.

Response time of individual requests. In Figure 9, we schedule one multi-get request made of several jobs, and we measure the ratio between the makespan $C_{\max}$ of the schedule

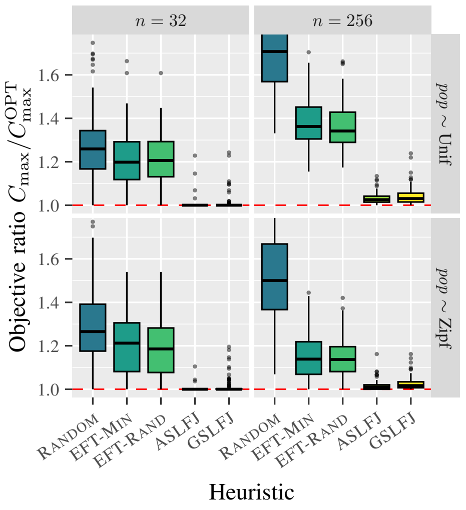


Fig. 9: Ratio between the makespan $C_{\max}$ given by each heuristic and the optimal makespan $C_{\max}^{\mathrm{OPT}}$ in different settings (the lower the better). Each boxplot aggregates 100 iterations. The red line represents the baseline (optimal makespan).

computed by each heuristic and the makespan $C_{\max}^{\mathrm{OPT}}$ of an optimal schedule computed by the MILP solver. Note that, for a given multi-get request, the makespan of the schedule also represents its response time, as it denotes the completion time of the slowest of its jobs. We set the number of machines to $m = 48$ and the replication factor to $k = 3$, and we consider multi-get requests of size $n = 32$ (medium size, left column) and $n = 256$ (large size, right column). The studied popularity distributions of keys are uniform ($pop \sim$ Unif, top row) and Zipf's law with bias 1.0 ($pop \sim$ Zipf, bottom row). Each boxplot aggregates 100 iterations of the experiment, and the red line represents the baseline (the optimal makespan computed with the MILP).

According to the results, we can see that our heuristics ASLFJ and GSLFJ give close-to-optimal solutions in the considered settings; i.e., the median ratio to the optimal of ASLFJ (resp. GSLFJ) is at most 1.025 (resp. 1.031), whereas EFT-MIN systematically has a median ratio between 1.139 and 1.362. Moreover, by counting the number of times each heuristic gives the best solution for each instance, we find that ASLFJ is the closest to the optimal in 99% of the 400 tested cases. Comparatively, without taking ASLFJ into account, GSLFJ gives the best solution in 94% of the cases, whereas EFT-MIN is the best only in 5.25% of the cases, and even gives the worst solution in 16.25% of the cases. This confirms that GSLFJ provides a good trade-off between quality and time complexity. Finally, we also observe that the boxplots of ASLFJ and GSLFJ are flatter than the others, which seems to indicate that they are less sensitive to the instance characteristics. Indeed, the coefficient of variation (CV) for ASLFJ (resp. GSLFJ) is at most 0.028 (resp. 0.04), whereas the CV for the other heuristics ranges from 0.076 to 0.153.

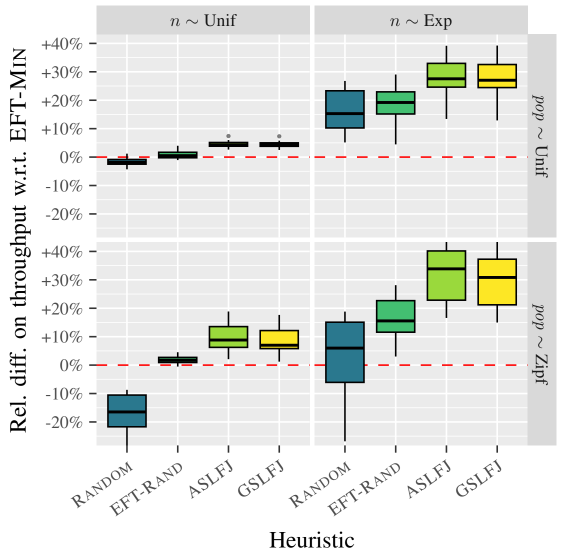


Fig. 10: Ratio between the saturating throughput given by each heuristic on 1000 multi-get requests and the saturating throughput given by EFT-MIN in different settings (the higher the better). Each boxplot aggregates 20 iterations. The red line represents the baseline (EFT-Min).

Overall, the proposed heuristics give close-to-optimal response time, where the classical EFT-MIN heuristic is between 15% and 35% slower on average, depending on the cases.

Saturating throughput of a stream of requests. In Figure 10, we study the behaviour of our heuristics from a a system point of view, to understand whether optimizing the scheduling of each individual request has an impact on the saturating throughput. To do this, we schedule a workload of 1000 multi-get request and measure the finishing time of the last request to complete. The saturating throughput is defined as the number of requests in the workload dividing by this last finish time. In this figure, we plot the ratio between the saturating throughput of each heuristic and the one of the baseline EFT-MIN. Again, we set the number of machines to $m = 48$ and the replication factor to $k = 3$, but we now make the size of multi-get requests vary according to a uniform distribution between 1 and 256 ($n \sim$ Unif, left column) and an exponential distribution with mean 32 ($n \sim$ Exp, right column). We use this last setting as a realistic workload where small multi-get requests are a lot more probable than large ones (the probability to send a multi-get request of size $n \geq 128$ is about 0.018 if $n \sim$ Exp, compared to 0.5 if $n \sim$ Unif). The studied popularity distributions of keys are uniform ($pop \sim$ Unif, top row) and Zipf's law with bias 1.0 ($pop \sim$ Zipf, bottom row). Each boxplot aggregates 20 iterations of the experiment, and the red line represents the baseline, i.e., the EFT-MIN heuristic.

We observe that ASLFJ and GSLFJ improve the maximum attainable throughput in all tested settings. However, the improvement is more significant when the size of multi-get requests follows an exponential distribution. When keys have the same probability to be requested (top row), the median saturating throughput of ASLFJ (resp. GSLFJ) is greater than the one of EFT-MIN by 4.3% (resp. 4.3%) if $n \sim \text{Unif}$, whereas it is greater by 27.5% (resp. 27%) if $n \sim \text{Exp}$. For a Zipf popularity distribution (bottom row), the median saturating throughput of ASLFJ (resp. GSLFJ) is greater than the one of EFT-MIN by 8.8% (resp. 7%) if $n \sim \text{Unif}$, whereas it is greater by 33.9% (resp. 30.8%) if $n \sim \text{Exp}$. In our experiments, we noticed that ASLFJ and GSLFJ were particularly efficient for small multi-get requests (i.e., they find an optimal solution quasi-systematically when $n \leq 10^2$), which are in majority if $n \sim \text{Exp}$. Over the 80 tested workloads, ASLFJ gives the best results in 86.25% of the cases. When ASLFJ is not taken into account, GSLFJ gives the best results in 97.5% of the cases. Interestingly, in certain cases we see that EFT-RAND also brings a significant improvement (up to 19.2%) over EFT-MIN. Indeed, EFT-MIN systematically chooses the *first* least-loaded machine in the list. This increases the chances to have a conflict with a subsequent multi-get request that could also need this particular machine. In contrast, EFT-RAND chooses a least-loaded machine at random, which tends to improve the overall load-balancing of the workload.

Overall, we notice that our heuristics not only improve the response time of individual requests, but are also able to improve the maximum load that the system is able to cope with. This is a non-trivial and interesting conclusion since throughput optimization is similar to load-balancing, which is usually an orthogonal objective to optimizing the individual performance of requests. Depending on the distribution of request sizes and key popularities, the improvement in throughput goes from 27% to 34% in realistic cases.

VIII. CONCLUSION

In this paper, we tackle the multi-get request partitioning problem that arises is modern key-value stores by modeling this as a scheduling problem, namely the Restricted Assignment problem on Intervals (RAI), and proposing approximation algorithms and heuristics to solve it. We first exhibit a $(2-1/m)$-approximation algorithm, and we further extend the RAI problem to *circular* intervals, which fit the configuration of actual replicated key-value stores. In this setting, we propose a general framework that, given an optimal algorithm for the RAI problem with at most K job types and running in time $O(f(n))$, computes an optimal solution for the RACI problem in time $O(n^K f(n))$. This enables us to revisit an optimal algorithm for the RAI problem when jobs are unitary to solve the corresponding RACI problem in time $O(m^2+n\log n+mn)$. Moreover, we derive a new $(4-2/m)$-approximation algorithm in the general case, which we use as a basis to design new practical heuristics to partition multi-get requests. We evaluate these heuristics through extensive simulations, and we show that they not only improve the response time of individual multi-get requests compared to simple greedy strategies (giving solutions with a median ratio to the optimal of 1.031), but are also able to increase the maximum attainable throughput of the system by 27%–34% in realistic cases.

As a future work, the next step would be to implement and evaluate our heuristics in a real key-value store, e.g., Apache Cassandra. On the theoretical side, it remains unknown if there exists an efficient approximation algorithm for the particular instances of RACI where circulars intervals are not necessarily totally ordered, i.e., a given circular interval may be strictly included into another. Moreover, we conjecture that there exists an efficient approximation algorithm for RACI that improves on the $4-2/m$ guaranteed factor.

REFERENCES

[1] Berk Atikoglu, Yuehai Xu, Eitan Frachtenberg, Song Jiang, and Mike Paleczny. Workload analysis of a large-scale key-value store. *ACM SIGMETRICS Performance Evaluation Review*, 40(1):53–64, 2012.

[2] Péter Biró and Eric McDermid. Matching with sizes (or scheduling with processing set restrictions). *Discrete Applied Mathematics*, 164:61–67, 2014.

[3] Brian F Cooper, Adam Silberstein, Erwin Tam, Raghu Ramakrishnan, and Russell Sears. Benchmarking cloud serving systems with ycsb. In *Proceedings of the 1st ACM symposium on Cloud computing*, pages 143–154, 2010.

[4] Jeffrey Dean and Luiz André Barroso. The tail at scale. *Communications of the ACM*, 56(2):74–80, 2013.

[5] Giuseppe DeCandia, Deniz Hastorun, Madan Jampani, Gunavardhan Kakulapati, Avinash Lakshman, Alex Pilchin, Swaminathan Sivasubramanian, Peter Vosshall, and Werner Vogels. Dynamo: amazon's highly available key-value store. In *SOSP 2007*, pages 205–220, 2007.

[6] Tomás Ebenlendr, Marek Krcál, and Jiri Sgall. Graph balancing: a special case of scheduling unrelated parallel machines. In *SODA*, volume 8, pages 483–490, 2008.

[7] Leah Epstein and Asaf Levin. Scheduling with processing set restrictions: Ptas results for several variants. *International Journal of Production Economics*, 133(2):586–595, 2011.

[8] Celia A Glass and Hans Kellerer. Parallel machine scheduling with job assignment restrictions. *Naval Research Logistics*, 54(3):250–257, 2007.

[9] Vikas Jaiman, Sonia Ben Mokhtar, and Etienne Rivière. Tailx: Scheduling heterogeneous multiget queries to improve tail latencies in key-value stores. In *IFIP International Conference on Distributed Applications and Interoperable Systems*, pages 73–92, 2020.

[10] Klaus Jansen and Lars Rohwedder. Structured instances of restricted assignment with two processing times. In *Conference on Algorithms and Discrete Applied Mathematics*, pages 230–241, 2017.

[11] Klaus Jansen and Lars Rohwedder. A quasi-polynomial approximation for the restricted assignment problem. *SIAM Journal on Computing*, 49(6):1083–1108, 2020.

[12] Avinash Lakshman and Prashant Malik. Cassandra: a decentralized structured storage system. *ACM SIGOPS Operating Systems Review*, 44(2):35–40, 2010.

[13] Jan Karel Lenstra, David B Shmoys, and Éva Tardos. Approximation algorithms for scheduling unrelated parallel machines. *Mathematical programming*, 46(1):259–271, 1990.

[14] Yixun Lin and Wenhua Li. Parallel machine scheduling of machine-dependent jobs with unit-length. *European Journal of Operational Research*, 156(1):261–266, 2004.

[15] Marten Maack and Klaus Jansen. Inapproximability results for scheduling with interval and resource restrictions. *arXiv preprint arXiv:1907.03526*, 2019.

[16] Waleed Reda, Marco Canini, Lalith Suresh, Dejan Kostić, and Sean Braithwaite. Rein: Taming tail latency in key-value stores via multiget scheduling. In *Proceedings of the Twelfth European Conference on Computer Systems*, pages 95–110, 2017.

[17] Lalith Suresh, Marco Canini, Stefan Schmid, and Anja Feldmann. C3: Cutting tail latency in cloud data stores via adaptive replica selection. In *12th USENIX Symposium on Networked Systems Design and Implementation*, NSDI, pages 513–527, 2015.

[18] Chao Wang and René Sitters. On some special cases of the restricted assignment problem. *Information Processing Letters*, 116(11):723–728, 2016.

Appendix A
Experimental Results

TABLE I: Median ratio between the makespan $C_{\max}$ given by each heuristic and the optimal makespan $C_{\max}^{\mathrm{OPT}}$ in different settings (the lower the better).

m	n	k	*pop*	Random	EFT-Min	EFT-Rand	ASLFJ	GSLFJ
$m = 12$	$n = 32$	$k = 3$	*pop* ~ Unif	1.485	1.351	1.33	**1.009**	1.019
$m = 12$	$n = 256$	$k = 3$	*pop* ~ Unif	1.443	1.169	**1.167**	1.182	1.182
$m = 48$	$n = 32$	$k = 3$	*pop* ~ Unif	1.259	1.198	1.206	**1**	**1**
$m = 48$	$n = 256$	$k = 3$	*pop* ~ Unif	1.707	1.362	1.342	**1.025**	1.031
$m = 12$	$n = 32$	$k = 3$	*pop* ~ Zipf	1.467	1.286	1.261	**1**	1.01
$m = 12$	$n = 256$	$k = 3$	*pop* ~ Zipf	1.585	1.165	1.174	**1.057**	1.059
$m = 48$	$n = 32$	$k = 3$	*pop* ~ Zipf	1.265	1.212	1.186	**1**	**1**
$m = 48$	$n = 256$	$k = 3$	*pop* ~ Zipf	1.5	1.139	1.137	**1.01**	1.017
$m = 12$	$n = 32$	$k = 6$	*pop* ~ Unif	1.495	1.374	1.38	**1**	**1**
$m = 12$	$n = 256$	$k = 6$	*pop* ~ Unif	1.443	1.152	1.159	**1.063**	**1.063**
$m = 48$	$n = 32$	$k = 6$	*pop* ~ Unif	1.278	1.199	1.216	**1**	**1**
$m = 48$	$n = 256$	$k = 6$	*pop* ~ Unif	2	1.53	1.518	**1.042**	1.054
$m = 12$	$n = 32$	$k = 6$	*pop* ~ Zipf	1.581	1.362	1.369	**1**	**1**
$m = 12$	$n = 256$	$k = 6$	*pop* ~ Zipf	1.468	1.154	1.147	**1.057**	**1.057**
$m = 48$	$n = 32$	$k = 6$	*pop* ~ Zipf	1.332	1.233	1.272	**1**	**1**
$m = 48$	$n = 256$	$k = 6$	*pop* ~ Zipf	1.86	1.349	1.336	**1.019**	1.028

TABLE II: Number of times each heuristic gives the best or worst solution for each instance. On the right, we present the table without taking ASLFJ into account to properly see the improvement brought by GSLFJ.

(a) With ASLFJ.

Heuristic	Samples	#best	#worst	%best	%worst
Random	1600	6	1317	0.38%	82.31%
EFT-Min	1600	63	190	3.94%	11.88%
EFT-Rand	1600	57	188	3.56%	11.75%
ASLFJ	1600	1500	0	93.75%	0%
GSLFJ	1600	1124	3	70.25%	0.19%

(b) Without ASLFJ.

Heuristic	Samples	#best	#worst	%best	%worst
Random	1600	10	1317	0.62%	82.31%
EFT-Min	1600	72	190	4.5%	11.88%
EFT-Rand	1600	71	188	4.44%	11.75%
GSLFJ	1600	1489	3	93.06%	0.19%

TABLE III: Coefficient of variation (CV) of the makespan $C_{\max}$ given by each heuristic over 100 experiments in the tested settings.

m	n	k	*pop*	Random	EFT-Min	EFT-Rand	ASLFJ	GSLFJ
$m = 12$	$n = 32$	$k = 3$	*pop* ~ Unif	0.136	0.105	0.107	**0.03**	0.043
$m = 12$	$n = 256$	$k = 3$	*pop* ~ Unif	0.085	0.046	0.045	**0.036**	**0.036**
$m = 48$	$n = 32$	$k = 3$	*pop* ~ Unif	0.131	0.105	0.098	**0.028**	0.038
$m = 48$	$n = 256$	$k = 3$	*pop* ~ Unif	0.127	0.081	0.076	**0.027**	0.04
$m = 12$	$n = 32$	$k = 3$	*pop* ~ Zipf	0.132	0.098	0.096	**0.03**	0.048
$m = 12$	$n = 256$	$k = 3$	*pop* ~ Zipf	0.126	**0.048**	0.049	0.058	0.057
$m = 48$	$n = 32$	$k = 3$	*pop* ~ Zipf	0.153	0.118	0.107	**0.012**	0.036
$m = 48$	$n = 256$	$k = 3$	*pop* ~ Zipf	0.147	0.089	0.084	**0.023**	0.029
$m = 12$	$n = 32$	$k = 6$	*pop* ~ Unif	0.169	0.086	0.094	**0.044**	0.059
$m = 12$	$n = 256$	$k = 6$	*pop* ~ Unif	0.108	0.044	0.036	**0.005**	**0.005**
$m = 48$	$n = 32$	$k = 6$	*pop* ~ Unif	0.129	0.117	0.114	**0.008**	0.009
$m = 48$	$n = 256$	$k = 6$	*pop* ~ Unif	0.122	0.08	0.075	**0.054**	0.057
$m = 12$	$n = 32$	$k = 6$	*pop* ~ Zipf	0.188	0.117	0.122	**0.037**	0.046
$m = 12$	$n = 256$	$k = 6$	*pop* ~ Zipf	0.116	0.046	0.042	**0.009**	**0.009**
$m = 48$	$n = 32$	$k = 6$	*pop* ~ Zipf	0.18	0.117	0.121	**0.01**	0.017
$m = 48$	$n = 256$	$k = 6$	*pop* ~ Zipf	0.162	0.116	0.129	**0.023**	0.039

TABLE IV: Median relative improvement of the saturating throughput given by each heuristic on the baseline EFT-MIN in different settings (the higher the better)

m	n	p_j	pop	RANDOM	EFT-RAND	ASLFJ	GSLFJ
$m = 12$	$n \sim$ Unif	$p_j \sim$ Exp2	$pop \sim$ Unif	-0.9%	+1.1%	**+1.6%**	**+1.6%**
$m = 12$	$n \sim$ Unif	$p_j \sim$ Exp2	$pop \sim$ Zipf	-20%	+0.1%	**+9%**	+8.9%
$m = 12$	$n \sim$ Exp	$p_j \sim$ Exp2	$pop \sim$ Unif	+3.4%	+5.7%	**+7.4%**	**+7.4%**
$m = 12$	$n \sim$ Exp	$p_j \sim$ Exp2	$pop \sim$ Zipf	-13.6%	+1.7%	+16.5%	**+16.6%**
$m = 48$	$n \sim$ Unif	$p_j \sim$ Exp2	$pop \sim$ Unif	+1.3%	+3.7%	**+7.1%**	**+7.1%**
$m = 48$	$n \sim$ Unif	$p_j \sim$ Exp2	$pop \sim$ Zipf	-13.8%	+2.3%	**+10.1%**	+9.6%
$m = 48$	$n \sim$ Exp	$p_j \sim$ Exp2	$pop \sim$ Unif	+23.6%	+26.8%	**+35.9%**	+35.7%
$m = 48$	$n \sim$ Exp	$p_j \sim$ Exp2	$pop \sim$ Zipf	+1.4%	+12.4%	**+24.4%**	+24.2%
$m = 12$	$n \sim$ Unif	$p_j \sim$ Exp12	$pop \sim$ Unif	-1.3%	+0.4%	**+1%**	**+1%**
$m = 12$	$n \sim$ Unif	$p_j \sim$ Exp12	$pop \sim$ Zipf	-18.4%	+0.1%	**+10.3%**	+10%
$m = 12$	$n \sim$ Exp	$p_j \sim$ Exp12	$pop \sim$ Unif	+1.8%	+3.4%	**+5.6%**	**+5.6%**
$m = 12$	$n \sim$ Exp	$p_j \sim$ Exp12	$pop \sim$ Zipf	-11.3%	+2.8%	**+18.1%**	+17.1%
$m = 48$	$n \sim$ Unif	$p_j \sim$ Exp12	$pop \sim$ Unif	-1.9%	+0.5%	**+4.3%**	**+4.3%**
$m = 48$	$n \sim$ Unif	$p_j \sim$ Exp12	$pop \sim$ Zipf	-16.5%	+1.7%	**+8.8%**	+7%
$m = 48$	$n \sim$ Exp	$p_j \sim$ Exp12	$pop \sim$ Unif	+15.3%	+19.2%	**+27.5%**	+27%
$m = 48$	$n \sim$ Exp	$p_j \sim$ Exp12	$pop \sim$ Zipf	+6%	+15.6%	**+33.9%**	+30.8%

TABLE V: Number of times each heuristic gives the best or worst throughput for each instance. On the right, we present the table without taking ASLFJ into account to properly see the improvement brought by GSLFJ.

(a) With ASLFJ.

Heuristic	Samples	#best	#worst	%best	%worst
RANDOM	320	0	312	0%	97.5%
EFT-RAND	320	3	8	0.94%	2.5%
ASLFJ	320	259	0	80.94%	0%
GSLFJ	320	134	0	41.88%	0%

(b) Without ASLFJ.

Heuristic	Samples	#best	#worst	%best	%worst
RANDOM	320	0	312	0%	97.5%
EFT-RAND	320	3	8	0.94%	2.5%
GSLFJ	320	317	0	99.06%	0%

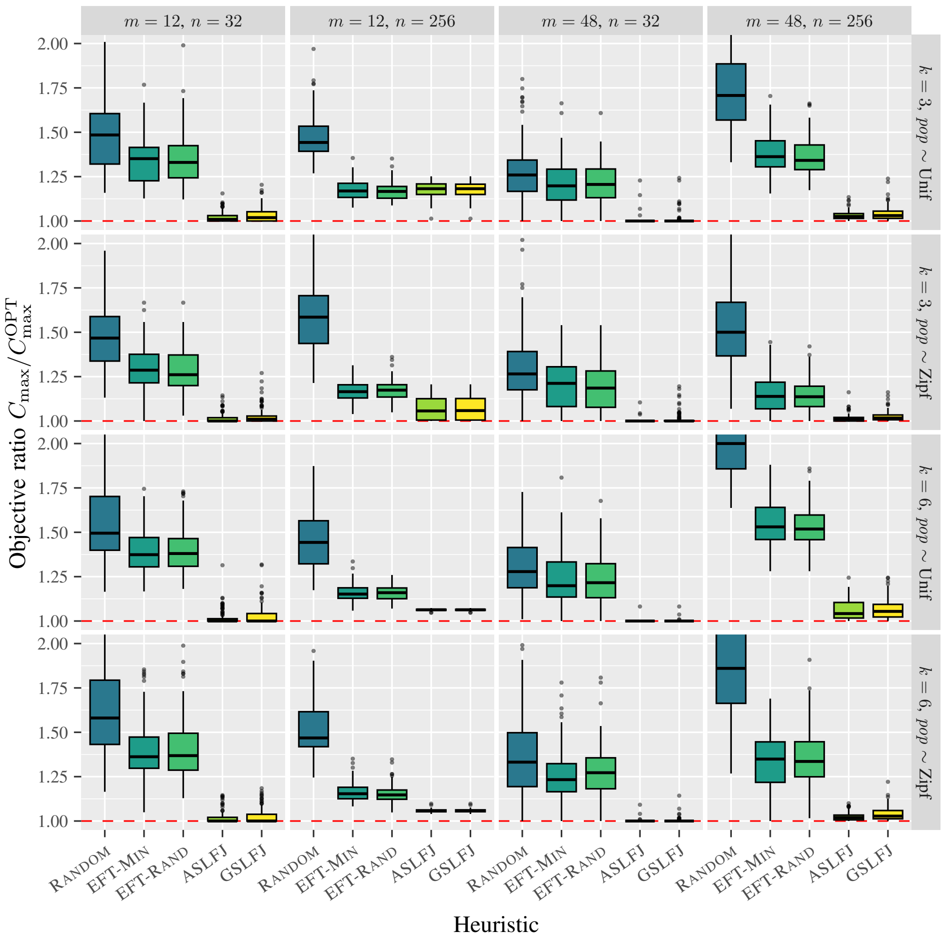


Fig. 11: Ratio between the makespan $C_{\max}$ given by each heuristic and the optimal makespan $C_{\max}^{\mathrm{OPT}}$ in different settings (the lower the better). Each boxplot aggregates 100 iterations. The red line represents the baseline.

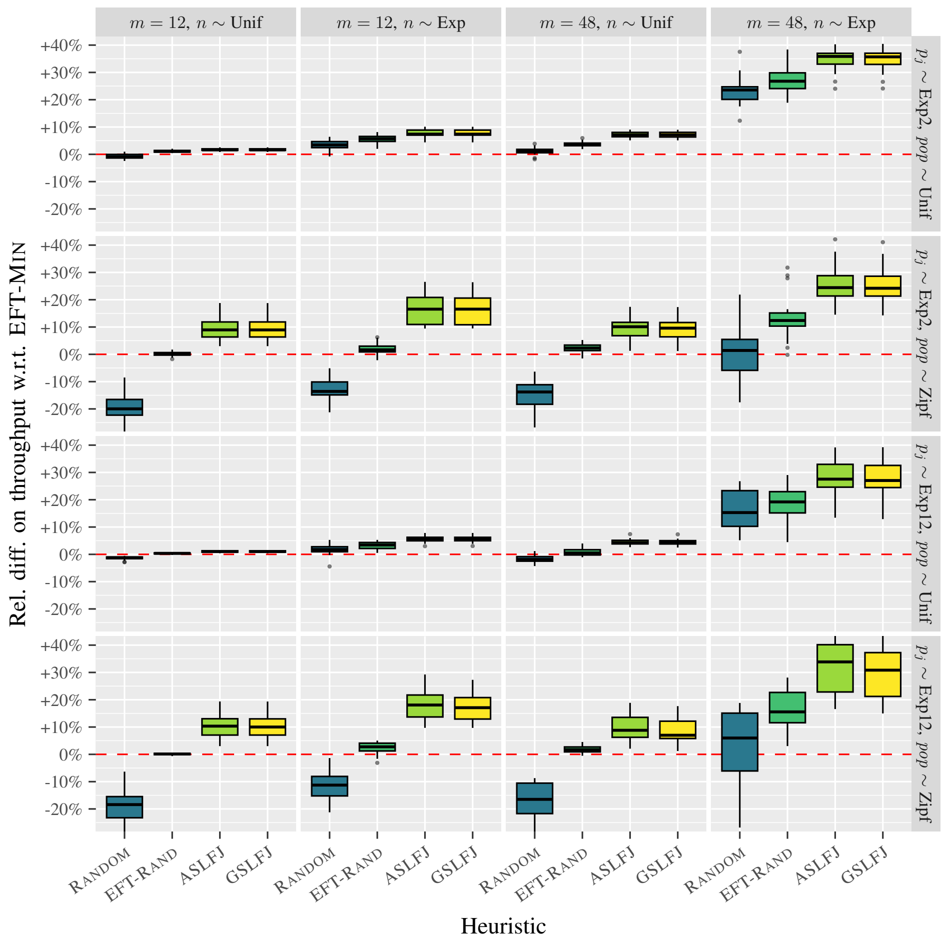


Fig. 12: Ratio between the saturating throughput given by each heuristic on 1000 multi-get requests and the saturating throughput given by EFT-MIN in different settings (the higher the better). Each boxplot aggregates 20 iterations. The red line represents the baseline.