University of Wrocław

PhD Thesis

Algorithmic aspects of contemporary networks

Maciej Pacut

supervised by
Dr hab. Marcin Bieńkowski

April 2019

Contents

Chapter 1

Introduction

In the last decades, we witnessed a growing demand for performing large-scale computations, such as protein folding, fluid dynamics, weather and market prediction, or production process optimization. The scale of such computations exceeds abilities of a single computer, hence they need to be performed on large sets of machines that cooperate over an interconnecting network. Owning and maintaining such large-scale computing infrastructure is often impractical and expensive, and parties look for alternative ways to perform computations. In comparison, outsourcing computations provides a wide range of benefits. First of all, it mitigates the costs of infrastructure management and maintenance. This is crucial especially for computational tasks that arise occasionally, such as high-quality rendering, computer verification of products with long development time or analysis of human-harvested data. Second, such an approach dismisses the need to foresee the appropriate demand for resources. If such demand increases unexpectedly, it can be immediately provided without physical extension of the infrastructure. This led to a shift of computations to large-scale remote facilities that contain computing machines with their support infrastructure, the so-called *data centers*. Performing computations in these external data centers provides the impression of unlimited computational power on demand, and is called the *cloud computing*.

The demand for outsourcing computations to the cloud created a whole market for such services. Modern suppliers of processing power such as Microsoft Azure [AZU], Amazon Web Services [AWS] or Google Compute Engine [GCE] provide convenient on-demand computational power while hiding most of the details concerning resource management. Processing capabilities are quickly and conveniently accessible to every interested party.

Computational tasks require multiple types of resources to complete: CPU time, memory, I/O operations and network bandwidth. Often the demand for these resources varies in time and is unpredictable. For this reason, a data center that performs just one task at the time would waste resources. In contrast, the co-existence of multiple tasks in the data center allows to compensate for the variable demand for resources by resource-aware scheduling. Such techniques are especially useful in (but not limited to) computationally-intensive applications, where the response time is not the primary concern.

The first part of this thesis assumes the perspective of a data center owner, who wants to use owned resources in an efficient manner. For example, the processing speed can be scaled down to save energy, memory can be shared or distributed, and cooperating processes can be migrated closer to each other in the network to save bandwidth. In the first part of this thesis, we focus on the last aspect and we show how it leads to *efficient usage of an interconnecting network* in a data center. Optimization of this resource is critical for performing efficient large-scale computations, as these involve multiple machines that cooperate over the network. To this end, we will make use of a sophisticated control system, called *virtualization*.

In the second part of this thesis, we shift our attention away from the optimization of data center network and focus on fundamental aspects of data transmission in the modern Internet. The transmitted data is split into portions called packets, which are sent independently, and the task of relaying a packet to its destination, called *packet forwarding*, is performed by network devices called *routers*. A single network is usually connected with multiple adjacent networks, and at each intermediate network, a bordering router needs to determine the next router on the way of the packet. To this end, such device directs packets based on the set of its forwarding rules, each corresponding to some network. The number of forwarding rules stored in core Internet routers is almost as numerous as the total number of networks, which leads to enormous forwarding tables to manage.

The second part of this thesis assumes the perspective of an Internet Service Provider (ISP) that owns and maintains the connecting physical infrastructure, such as routers. Typically, routers located near the core of the Internet, e.g., in top-tier networks owned by large ISP, store a sizeable number of forwarding rules, and this number continues to grow with the size of the Internet. As the size of forwarding tables grows, it inevitably *exceeds the available memory of the router*. One of the goals of the ISP is to utilize existing devices in the most efficient way and to delay the need for an upgrade. The obvious but expensive solution is to provide additional memory for the device. We focus on an alternative approach, where the router continues to operate with insufficient memory to store the whole forwarding table. In this approach, it is important to preserve the correctness and efficiency of packet forwarding: both are crucial in minimizing data transfer latency and maximizing the throughput.

1.1 Machine Virtualization in Data Centers

To use the data center's interconnecting network efficiently, cooperating computational tasks should be placed close to each other and close to the data they process. Algorithmic techniques presented in the first part of this thesis rely upon logical isolation of a computation from the physical machine that performs the computation. This gives a possibility to manage the physical placement of a computation in a way that is transparent to the computation. A particular piece of technology that provides the flexibility in placement of computations is *virtualization*.

Virtualization provides an abstraction layer, called the *virtual machine*, for the underlying hardware of a computer system. Virtual machine mimics the functionality of the physical hardware so closely that it can be used as an environment for a complete operating system.

Such operating system, running on a virtual machine is called the *guest operating system*. It operates in addition to the *host operating system*, which runs directly on the physical hardware. In a data center, the main purpose of virtualization is to provide a complete and non-restricted environment for the client that is isolated from the management software and other clients' tasks. The guest operating system is restricted to the virtualized environment: it has the perspective of housing a whole computer system.

1.1.1 Machine Migration

Besides providing an abstraction layer, mature virtualization solutions suited for data centers such as Xen [XEN], KVM [KVM], Hyper-V [HyV], VMware ESXi [VME], provide several control features. In particular, absolute control over the underlying virtual hardware allows to suspend and resume the execution of the guest operating system at will. Such functionality provides building blocks for the feature of *migration*, which transfers the complete virtual machine to a different physical machine. This is possible without shutting down the guest operating system, and hence it provides a powerful resource management tool that is transparent to clients.

Distributed cloud applications, including batch processing applications such as MapReduce, streaming applications such as Apache Flink or Apache Spark, and scale-out databases and key-value stores such as Cassandra, generate a significant amount of network traffic and a considerable fraction of their runtime is due to network activity [MP12]. For example, traces of jobs from a Facebook data center reveal that network transfers on average account for 33% of the execution time [CZM+11]. In such applications, it is desirable that frequently communicating virtual machines are *collocated*, i.e., mapped to the same physical server: communication across the network (i.e., inter-server communication) induces network load and latency. However, migrating virtual machines between servers also comes at a price: the state transfer is bandwidth intensive, and may even lead to short service interruptions. Therefore the goal is to design online algorithms that find a good trade-off between the inter-server communication cost and the migration cost. The problem central to the first part of this thesis is stated as follows:

How to assign virtual machines to physical machines to optimize network usage?

We elaborate more in the subsequent subsection.

1.1.2 Virtual Network Embedding

The computing power of a single virtual machine is usually insufficient for the client, as the resources of a virtual machine are limited by resources available to its host. Therefore, data centers provide their resources as a sizeable set of virtual machines connected by a network. Collectively, the virtual machines with their interconnecting network are called a *virtual network*, where the cooperating virtual machines are referred to as *nodes* of a virtual network. To guarantee a certain quality of service (QoS) for a multitude of co-existing virtual networks, up-front bandwidth reservations are required. However, the generality of performed calculations results in an unpredictability of communication patterns and poses a challenge in

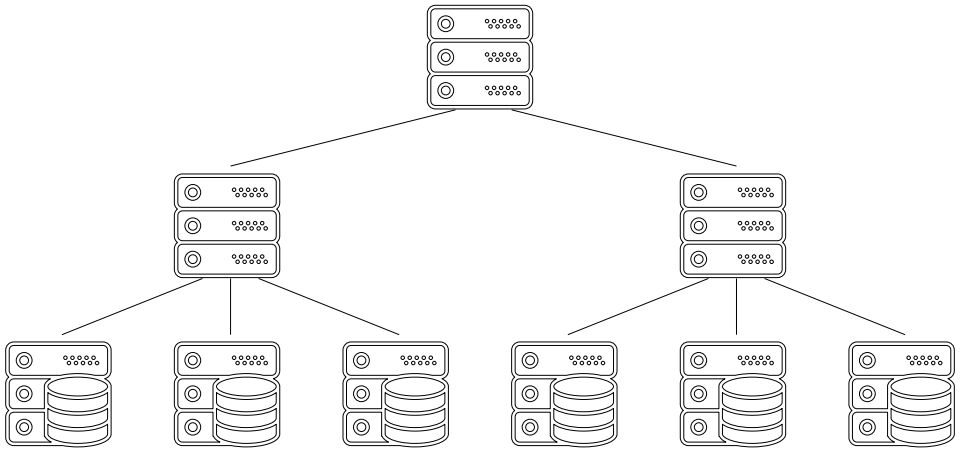

Figure 1.1: The model of typical data center with a tree-like network topology. We distinguish two types of tree nodes: the the intermediate nodes that transmit communication, and the computing machines, located at the leaves of a tree. Network links between nodes are depicted as solid lines.

the optimization of bandwidth reservations. In this thesis, we provide algorithms for an efficient management of network reservations without any assumptions about communication patterns.

To measure the quality of resource management strategy, in Chapter 2 and Chapter 3 we state formal optimization problems; for now, we only briefly sketch it. Physical components of a data center are modeled as a graph called a *substrate network*, in which vertices correspond to physical machines, and edges correspond to an interconnecting network. A communication cost between a pair of physical machines is proportional to edge-distance in substrate network (the number of *hops* in the substrate network). A communication pattern among virtual machines is also modeled as a graph, called a *communication graph*. In such settings, the communication among virtual machines running on certain physical machines can be viewed as a *graph embedding* of the communication graph into a substrate network [GKK+01]. The main objective is to find an embedding that locates closely the virtual machines that communicate often.

In this thesis, we study substrate networks in the form of a tree, which closely models the popular Fat-Tree topology [Lei85]. In this tree topology, only leaves can host virtual machines, and the sole role of intermediate tree nodes is to transmit data between leaves, see Figure 1.1.

1.1.3 Our Contributions

We consider two scenarios regarding virtual network embeddings:

1. *The static scenario*, where virtual machines are irrevocably assigned to their physical machines.

2. *The dynamic scenario*, where virtual machines can migrate between physical machines.

We investigate the static scenario in Chapter 2 and the dynamic scenario in Chapter 3. Although the problems are related by their practical motivations, their combinatorial structure differs substantially. In particular, the static scenario is not an offline version of the problem

considered in the dynamic one. In both cases, we assume that the communication pattern among virtual nodes is not known in advance. In the static scenario, we reserve a portion of bandwidth between every pair of virtual nodes to allow for any possible communication pattern. On the other hand, in the dynamic scenario, we react to changes in the communication pattern by migrating machines on the fly. In addition, the problem considered in the static scenario is enriched to model the needs of batch-processing applications that use distributed file systems.

Static Mapping of Virtual Networks

In the static scenario studied in Chapter 2, to guarantee a certain quality of service (QoS), we acquire network reservations for all pairs of cooperating virtual machines. The combinatorial problem that we consider in this scenario is essentially a variant of the minimum-cost embedding of a clique (the communication graph) in a tree (the substrate network). In addition, the scenario is designed to model certain aspects of Map-Reduce [DG04], which is a predominant framework for performing large-scale parallel data processing. We consider the wide range of possible extensions that model certain aspects of Map-Reduce applications, most notably:

- *Data chunk processing.* Virtual machines process large amounts of data chunks that are stored in a distributed file system. Each chunk of data must be assigned to a virtual machine. Data chunk transfers require their own network reservations.

- *Data chunk replication.* Distributed file systems often store redundant copies of data chunks, called *chunk replicas*. Only one copy of each data chunk replica must be processed, and we are free to choose the replica to be used based on its placement.

- *Bandwidth constraints.* Each link in the substrate network has its capacity. For the embedding to be feasible, the total network reservations have to obey link capacities.

We decompose the general optimization problem into its fundamental aspects, such as assignment of chunks, replica selection, and flexible virtual machine placement, and answer questions such as:

- How to efficiently embed virtual machines and their inter-connecting network?

- Which data chunks to assign to which virtual machine?

- How to exploit redundancy and select good data chunk replicas?

We draw a complete picture of the problem space: we show that some problem variants (also those exhibiting multiple degrees of freedom in terms of replica selection and embedding), can be solved in polynomial time. For all other variants, we show limitations of their computational tractability, by proving their NP-completeness. Interestingly, our hardness results also hold in *uncapacitated* substrate networks of small-diameter networks (as they are widely used today [ALV08]).

Dynamic Mapping of Virtual Networks

In Chapter 3, we study virtual network embeddings in the scenario where virtual machines can be migrated during runtime to another physical machine. The possibility of migration allows reacting to unpredictable communication patterns. For example, if some distant nodes communicate often, it is vital to reduce their distance to save network bandwidth. The objective is to minimize the total network bandwidth used for communication and for migration.

We assume that the communication patterns are not known in advance to our algorithm. We measure the quality of presented algorithmic solutions by competitive analysis [BE98], which is well-suited for problems that are online by their nature. In the competitive analysis, the goal is to optimize *the competitive ratio* of a given online algorithm: the ratio of its cost to the cost of an optimal offline algorithm that knows the whole input sequence in advance.

In the dynamic scenario, we assume that the physical substrate network is a tree of height one. That is, every physical machine (leaf) is connected directly to the root (that has no hosting capabilities). A single physical machine hosts a fixed number of virtual machines. The model restricted to such networks becomes a variant of online graph clustering. That is, we are given a set of n nodes (virtual machines) with time-varying pairwise communication patterns, which have to be partitioned into ℓ physical machines, each of capacity $k = n/\ell$.

Intuitively, we would like to minimize inter-machine interactions by mapping frequently communicating nodes to the same physical machine Since communication patterns change over time, the nodes should be *repartitioned*, in an online manner, by *migrating* them between physical machines. The objective is to minimize the weighted sum of inter-machine communication and repartitioning costs. The former is defined as the number of communication requests between nodes placed at distinct physical machines, and the latter as the number of migrations.

The possibility to perform a migration uncovers algorithmic challenges:

- *Serve remotely or migrate?* For a brief communication pattern, it may not be worthwhile to collocate the nodes: the migration cost might be too large in comparison to communication costs.

- *Where to migrate, and what?* If an algorithm decides to collocate nodes x and y, the question becomes how. Should x be migrated to the physical machine holding y, y to the one holding x, or should both nodes be migrated to a new machine?

- *Which nodes to evict?* The space of the desired destination physical machine may not be sufficient. In this case, the algorithm needs to decide which nodes to "evict" (migrate to other machines), to free up space.

In the model described above, every physical machine fully utilizes its processing capabilities — it hosts the maximum possible number of virtual machines, i.e., $k = n/\ell$. Hence, the migration is not possible without further reconfigurations: to respect physical machine capacity, we need to decide which virtual machines to swap. For this setting, we show a deterministic lower bound of k, where k is the physical machines hosting capacity. We also present a constant-competitive algorithm for the scenario restricted to physical machines that can host two virtual machines ($k = 2$).

In Chapter 3, we also consider the resource-augmented scenario, where we relax the above assumption: now the total hosting capacity of physical machines exceeds the total number of virtual machines, i.e., $k > n/\ell$. Surprisingly, the lower bound remains k also in this setting. The main contribution of this part is an $O(k \cdot \log k)$-competitive algorithm for the scenario with a small resource augmentation.

1.1.4 Related Work

Recently, there has been much interest in programming models and distributed system architectures for processing and analysis of big data (see, e.g., [DG04, ABB+12, XRZ+13]). Such applications generate large amounts of network traffic [CZM+11, MP12, MEC], and over the last years, several systems have been proposed that provide a provable network performance. To guarantee a certain performance level, these systems reserve a portion of bandwidth among cooperating nodes. There are two major approaches to bandwidth reservations: some systems depend on supplying the possibly different volume of bandwidth communication between each pair of nodes [PKC+12, PYB+13, SKGK10], while other systems allocate abundant bandwidth of equal volume among all nodes [BCKR11, GLW+10, RVR+07, RST+11, XDHK12]. In this thesis we research both approaches: in Chapter 2 we pre-reserve a portion of bandwidth among all cooperating nodes, while in Chapter 3 we perform the reservation as pairs of nodes communicate.

In Chapter 2, we study virtual network embeddings, a problem of embedding a weighted *guest* graph into a capacitated *host* graph. The virtual network embedding problem is related to classic VPN graph embedding problems [GKK+01, EGOS05, GKR03, GOS08], a problem that is NP-hard, and is constant-factor approximable even for asymmetric traffic demands [FOST10]. The VPN problem requires finding a graph embedding with fixed endpoints, while in virtual network embedding problems, studied in this thesis, the embedding endpoints are also subject to optimization. In this respect, the virtual network embedding problem can also be seen as related to classic Minimum Linear Arrangement problem [ENRS99, RR04] which asks for the embedding of communication graphs on simple line topologies.

We consider a variant of virtual network embedding with extensions motivated by batch-processing applications. The most popular virtual network abstraction for batch-processing applications [DG04] is a virtual network that forms a clique [BCKR11, MP12, FSSC16, RFS15, XDHK12]. Existing embedding algorithms often ignore a crucial dimension of the problem, namely data locality: an input to a batch-processing application is typically stored in a distributed and sometimes redundant file system. Since moving data is costly, an embedding algorithm should be aware of the data placement, and allocate computational resources close to the data. Redundant storage gives additional optimization possibilities. In this thesis, we study data locality and replica-aware virtual network embeddings in tree topologies.

In Chapter 3, we study an online balanced partitioning problem. The static offline version of the problem, i.e., a problem variant where migration is not allowed, where all requests are known in advance, and where the goal is to find an assignment of n nodes to ℓ physical machines, each of capacity n/ℓ, is known as the *ℓ-balanced graph partitioning problem.*

The problem is NP-complete, and cannot even be approximated within any finite factor unless P = NP [AR06]. The static variant where $\ell = 2$ corresponds to the minimum bisection problem, which is already NP-hard [GJS76], and the currently best approximation ratio is $O(\log n)$ [SV95, AKK99, FKN00, FK02, KF06, Räc08]. The inapproximability of the static variant for general values of ℓ motivated research on the bicriteria variant, which can be seen as the offline counterpart of our capacity augmentation approach. Here, the goal is to compute a graph partitioning into ℓ components of size at most k (where $k > n/\ell$) and the cost of the cut is compared to the optimal (non-augmented) solution where all components are of a size at most n/k. The variant where $n \geq 2{\cdot}k{\cdot}\ell$ was considered in [LMT90, ST97, ENRS00, ENRS99, KNS09]. So far, the best result is an $O(\sqrt{\log n \cdot \log \ell})$-approximation algorithm [KNS09].

Our dynamic model is related to online caching [ST85, FKL+91, MS91, ACN00], sometimes also referred to as online caching, where requests for data items (nodes) arrive over time and need to be served from a cache of finite capacity, and where the number of cache misses must be minimized. Classic problem variants usually boil down to finding a smart eviction strategy, such as Least Recently Used (LRU) [ST85]. In our setting, requests can be served remotely (i.e., without fetching the corresponding nodes to a single physical machine). In this light, our model is more reminiscent of caching models *with bypassing* [EILNG11, EILN15a, Ira02a]. As a side result, we show that our problem is capable of emulating online caching. A major difference between these problems is that in the caching problems, each request involves a single element of the universe, while in our model *both* endpoints of a communication request are subject to optimization.

1.2 Router Memory Optimization

In the second part of this thesis, we consider the fundamental problem of *packet forwarding*. We focus on a single router, that physically connects different networks and is responsible for passing packets between them. Upon receiving a data packet, the router forwards it to a specific output port leading to a neighboring network. To choose an appropriate port, the router stores a *forwarding table*, which consists of rules describing how to map the packet destination addresses to appropriate ports. Typically, a router stores a single rule for each network it knows about.

The router maintains the forwarding table in its memory. Only a small restricted set of operations is performed on such memory: lookups and updates. Nowadays, routers perform millions of lookup operations and thousands of updates per second (see, e.g., a report [BUP]), and use specialized hardware for the efficiency. Hence, instead of using the general-purpose memory such as RAM, the specialized memory units such as TCAM are utilized [PS06]. The TCAM memory is associative memory storage that enables hardware supported pattern-matching lookup that closely matches the way the forwarding rules are used.

1.2.1 Growth of Forwarding Tables

Most routers located at the Internet core forward packets among a large number of networks and have the knowledge about virtually all Internet networks. Their forwarding tables are

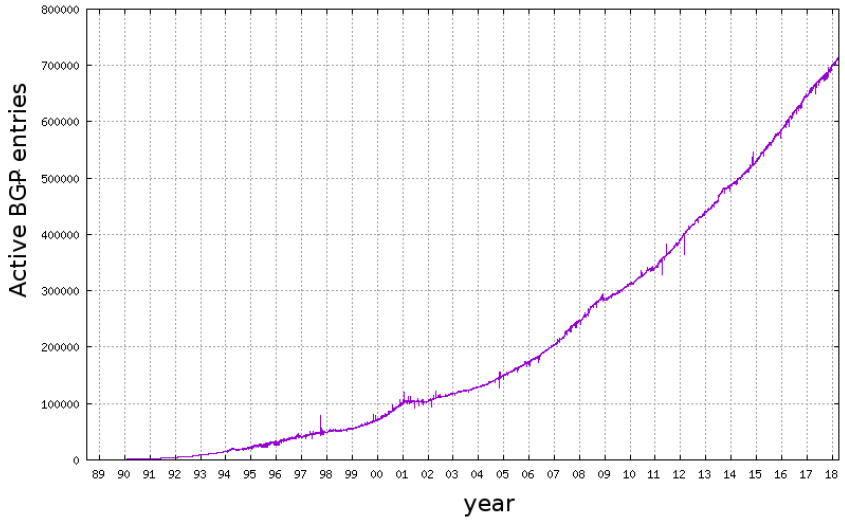

Figure 1.2: The number of entries in the global forwarding table. The entries are built on the basis of informations exchanged via Border Gateway Protocol. The graph presents the growth of the global forwarding table from 1988 to 2018.

usually referred to as *the global forwarding table* (Figure 1.2 depicts the growth of its size) [BGP]. Therefore, those routers have to store an enormous number of forwarding rules: the number of rules has doubled in the last six years [BRV] and the super-linear growth is likely to be sustained [CMU+10]. It is worth noting that some of these networks are virtual. The concept of virtual networks described in Part I contributed significantly to the partitioning of the Internet into subnetworks, and to further growth of forwarding tables.

In recent years, the size of forwarding tables started to exceed the amount of available TCAM memory in a typical router (the phenomenon called the TCAM exhaustion [EXH]). Sophisticated electronics circuits such as TCAM are very expensive and power-hungry in comparison to RAM [STT03]. Moreover, typical operations performed on TCAM memory examine the whole contents stored in memory, which requires closely connected physical structure among memory cells — as a result, available memory is expensive to expand.

The limited size of memory and expanding the size of the content to store brings new challenges in memory management of the routers. In the second part of this thesis, we investigate the following problem, using the tools provided by contemporary routers:

How to efficiently manage the memory of forwarding devices?

1.2.2 Our Contributions

In Chapter 4, we study a novel solution for the management of a growing set of forwarding rules. The idea that could delay the need for expensive or impossible memory upgrades in routers, is to store only a subset of rules in the actual router and to store all rules on a secondary device (for example a commodity server with a large, but slow memory) [KARW16, KCGR09, Liu01, LLW15, SUF+12]. We propose a theoretical model for studying algorithmic solutions for

such a setting. This setting is similar to the caching problem: some rules are stored in a fast memory of a router. We provide a natural online algorithm that dynamically manages the set of forwarding rules. Our algorithm, when applied in the context of such architectures, can be used to prolong the lifetime of some routers.

Although our theoretical model resembles the caching problem, the hierarchical structure of forwarding rules enforces some restrictions of cache configuration feasibility. Forwarding rules form a tree, and the child rule describes the exception to the parent rule. To model this issue, we introduce a variant of caching, where the universe of elements (forwarding rules) form a tree. The child-parent relations express dependencies between cached elements: to preserve the semantics of the forwarding rules set, no parent rule can be in the cache without its child rules. In other words, every valid cache configuration is a sub-forest. We give a formal problem definition in Chapter 4.

We present a deterministic online algorithm for cache management with hierarchical dependencies, proving that it is $O(h(T) \cdot k)$-competitive, where $h(T)$ is the height of the forwarding rules tree (the maximum nesting in the forwarding table), and k is the size of the available cache. Our result is optimal up to the factor $O(h(T))$: we show that the lower bound for the caching problem [ST85] implies an $\Omega(k)$ lower bound for our problem. While in theory $h(T)$ can be as large as the length of IP address, for the actual forwarding tables, it is a small constant (around 4–7). In addition, we consider the online tree caching problem within the resource augmentation paradigm: we assume that cache sizes of the online algorithm (k_{ONL}) and the optimal offline algorithm (k_{OPT}) may differ. For this setting, we show that our algorithm is $O(h(T) \cdot k_{\mathrm{ONL}}/(k_{\mathrm{ONL}} - k_{\mathrm{OPT}} + 1))$-competitive.

The performance of our algorithm is not degraded if the model is enhanced to handle rule updates (an important aspect of router operation). Finally, we show that our algorithm can be implemented efficiently and without changes to the router software and hardware.

1.2.3 Related Work

We introduced a variant of caching with dependencies motivated by the structure of forwarding table. In the framework of the competitive analysis, the caching problem was first analyzed by Sleator and Tarjan [ST85], who presented k-competitive algorithms (where k is the cache size) and a matching lower bound. The problem was later generalized to allow different fetching costs (weighted caching) [CKPV91, You94] and additionally different item sizes (file caching) [You02], with the same competitive ratio.

An important aspect of our problem is the dependency between forwarding rules. So far, the papers on caching of forwarding tables avoided this issue, either assuming that rules do not overlap (a tree has a single level) [KCGR09] or by preprocessing the forwarding table, so that the rules become non-overlapping [Liu01, LLW15]. Unfortunately, this could lead to large inflation of the routing table. A notable exception is a recent solution called CacheFlow [KARW16]. The CacheFlow model supports rule dependencies even in the form of directed acyclic graphs. However, CacheFlow was evaluated only experimentally, and no worst-case guarantees were given on the overall cost of caching. Our work provides theoretical foundations for respecting rule dependencies.

Other approaches for minimizing the number of stored rules were mostly based on *rules compression (aggregation)*, where the set of rules was replaced by an another equivalent and smaller set. Optimal aggregation of a fixed routing table can be achieved by dynamic programming [DKVZ99, SSW03], but the main challenge lies in balancing the achieved compression and the number of changes to the routing table in the presence of updates to this table. While many practical heuristics have been devised by the networking community for this problem [KCR+12, LZW13, LZN+10, LXS+13, RTK+13, UNT+11, ZLWZ10], worst-case analyses were presented only for some restricted scenarios [BSSU14, BS13]. Finally, combining rules compression and rules caching is so far an unexplored area.

1.3 Bibliographic notes and acknowledgements

The results of this thesis were published by its author in various conferences and journals. Parts of Chapter 2 appeared previously in the proceedings of 23rd IEEE International Conference on Network Protocols (ICNP 2015) [FPCS15] and in Theoretical Computer Science, vol. 697 [FPS17]. Some of the results from Chapter 2 appeared in the Ph.D. thesis of my co-author Carlo Fuerst. Parts of Chapter 3 appeared previously in the proceedings of 30th International Symposium on Distributed Computing (DISC 2016) [ALPS16]. Chapter 3 contains a thoroughly rewritten and improved revision of the results published in these proceedings. Finally, parts of Chapter 4 were published in the proceedings of 29th ACM Symposium on Parallelism in Algorithms and Architectures (SPAA 2017) [BMP+17]. Some preliminary results from Chapter 4 appeared in the master thesis of my co-author Aleksandra Spyra.

This thesis was supported by the grant Preludium 2016/23/N/ST6/03412 for research on "Online Algorithms for Packing and Covering Problems" awarded by Polish National Science Centre.

Part I

Mapping Virtual Networks

Chapter 2

Virtual Networks with Static Topology

For executing a certain computation in a data center, one places an adequate number of workers, whose task is to select their data portion, process them, and aggregate the result by cooperating with other workers. Workers establish network connections called a virtual network, formed to perform a given computational task, and in such setting, the worker is referred to as a node of the virtual network. While these nodes can be placed at any feasible physical machine in the data center, the processing performance is heavily influenced by their positions. Placing nodes closely to each other reduces network latency and bandwidth reservations in the data center. In this chapter, we investigate mapping the virtual network onto the physical infrastructure: a task of assigning the nodes of the virtual network to physical machines in a network-efficient way.

2.1 Problem Definition

As described informally in the introduction (see Section 1.1.2), the model combines three components: (1) the substrate network (the servers and the connecting physical network), (2) the virtual network (the virtual machines and the logical network connecting the machines to each other as well as to the data), and (3) the data divided into data chunks that need to be assigned to nodes that process them. For the remainder of this chapter, we restrict our attention to the substrate networks that form a tree, and a particular virtual network topology that is basically a fully connected graph (a clique). We refer to the aforementioned problem as the CLIQUE-IN-TREE EMBEDDING problem, abbreviated CTE. Below we provide the details of CTE components. Figure 2.1 gives an overview of our model.

The Substrate Network. The substrate network (also known as the *host graph*) represents the physical resources: a set S of $n_S = |S|$ servers interconnected by a network consisting of a set R of routers (or switches) and a set E of (symmetric) links. We often refer to the elements of $S \cup R$ as the *vertices*. We assume that the interconnecting network forms an arbitrary rooted tree T, where the servers are located at the tree leaves and routers at inner nodes. Depending on the available capacity $\mathsf{cap}(s)$ of server s, multiple virtual machines may be hosted on s. Each

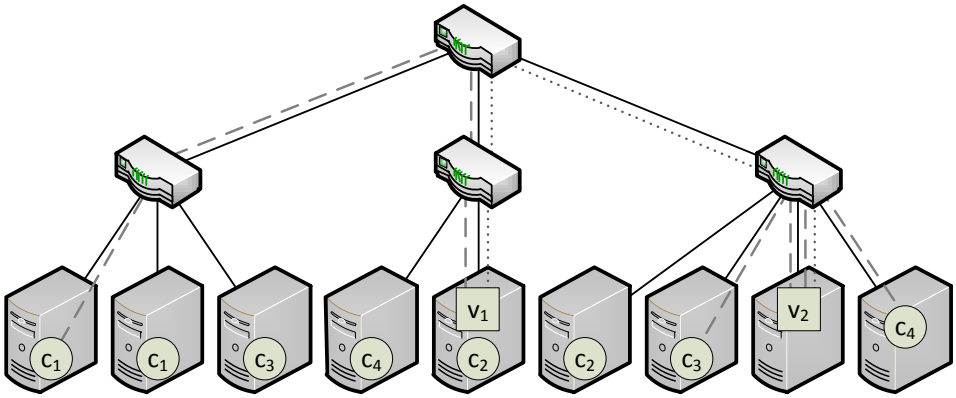

Figure 2.1: Overview: a 9-server data center storing $\tau = 4$ different chunks $\{c_1, \ldots, c_4\}$ (depicted as *circles*), each having two replicas. The replicas need to be selected and assigned to the two virtual machines v_1 and v_2; the virtual machines are depicted as *squares*, and the network connecting them to chunks (using bandwidth b_1) is *dashed*. In addition, the virtual machines are interconnected among each other using bandwidth b_2 (*dotted*). The objective of the embedding algorithm is to minimize the overall bandwidth allocation.

link $e \in E$ in the substrate network has a certain bandwidth capacity $\mathsf{cap}(e)$.

The Data. The data to be processed constitutes the input to the batch-processing application. The data is stored in a distributed filesystem spread across the servers; this spatial distribution is given and is not subject to optimization. The input data consists of τ different *chunks* $C = \{c_1, \ldots, c_\tau\}$, where each chunk c_i can have $r_i \geq 1$ instances (replicas) $c_i^{(1)}, \ldots, c_i^{(r_i)}$, stored at different servers. A single server may host multiple chunks. It is sufficient to assign one replica of each chunk, and we refer to this replica as the *active* (or selected) replica.

The Virtual Network. The virtual network V consists of $n_V = |V|$ virtual machines, called *nodes*. Each node can be placed (or, synonymously, embedded) at an arbitrary server and this placement is subject to optimization. Furthermore, to enable nodes to process the data, network connections need to be established. The *(node) interconnect network* forms a complete network (a clique) between nodes, and the *(chunk) access network* consists of paths from each active replica to the node that is assigned to process it. In order to ensure predictable application performance, both these networks require certain minimum bandwidth guarantees. Concretely, we assume that each chunk is connected to its assigned node at bandwidth b_1, and each node is connected to any other node at bandwidth b_2.

The choice of replica and the node assigned is subject to optimization, and μ denotes the assignment of chunks to nodes. Furthermore, we assume that the number of chunks τ is divisible by the number of nodes n_V, and the assignment μ is *balanced*: each node has exactly τ/n_V chunks assigned. Collectively, the nodes with the interconnect network and the access network form the *virtual network*. Note that our definition extends the notion of the virtual network studied by others [BCKR11, MP12, FSSC16, RFS15, XDHK12] by incorporating the chunk access network.

Optimization Objective. Our goal is to design algorithms that minimize the resource *footprint*, the most common objective function considered in the literature [FBB+13]. Formally, let $\mathsf{dist}(v_1, v_2)$ denote the distance in the underlying physical network T between the two nodes v_1

and v_2. For each chunk c, $\mu(c)$ is the node to which chunk c is assigned, c^* is the replica of c selected for processing, and $\mathsf{dist}(\mu(c), c^*)$ denotes the distance between the active replica of chunk c and the node to which c is assigned. The objective is to minimize the footprint of the virtual network, defined as

$$\underbrace{\sum_{c \in C} b_1 \cdot \mathsf{dist}(\mu(c), c^*)}_{\text{chunk access cost}} + \underbrace{\frac{1}{2} \cdot \sum_{v \in V} \sum_{v' \in V \setminus \{v\}} b_2 \cdot \mathsf{dist}(v, v')}_{\text{node interconnect cost}} .$$

The solution must obey the capacity of the substrate network: (1) the total number of nodes hosted at each server s must not exceed $\mathsf{cap}(s)$, and (2) the total bandwidth allocated at each link e must not exceed its capacity $\mathsf{cap}(e)$.

2.1.1 Problem Decomposition

We introduced the CTE problem in its full generality. To fully chart the algorithmic complexity of CTE, we decompose the problem into its fundamental components that can be activated or deactivated independently of each other, and we consider all possible variants. Concretely, we consider 5 aspects of CTE, namely multiple chunk assignment (MA), replica selection (RS), flexible node placement (FP), node interconnect (NI), and bandwidth constraints (BW), as described below.

Multiple Assignment (MA). In most applications, the number of chunks τ is larger than the number of nodes, and the objective is to assign multiple chunks to each node. In each variant (regardless of MA), we investigate assignments that balance chunks among nodes, i.e., where the number of chunks is divisible by the number of nodes, and each node has an identical integer number of chunks assigned. We require that the number of chunks assigned to each node is equal to $m = \tau / n_V$, and we call m the multi-assignment factor. If the number of chunks exceeds the number of nodes, i.e., $m > 1$, then we refer to such scenario as MA. In the CTE variant without MA, each node has exactly one chunk assigned, i.e., $m = 1$.

Replica Selection (RS). Distributed filesystems often utilize data redundancy for corruption detection and correction. A redundant data chunk has multiple *replicas*, stored on different physical machines. For each data chunk, it is sufficient to assign only one of its replicas. By RS we denote the flexibility of choosing which replica of each data chunk to assign. In the problem variant without RS, each chunk has a single replica ($r_i = 1$ for all i).

Flexible Placement (FP). By FP we denote the flexibility of assignment of nodes to physical machines. In the problem variant without FP, the assignment of nodes to physical machines is given as an input.

Node Interconnect (NI). In some computational tasks, it is sufficient to process data chunks independently of each other. However, more often the result of processing the individual chunks is combined afterward. By NI we refer to the latter scenario, where the computation requires the nodes to cooperate. We investigate the scenario, where we reserve a bandwidth of volume b_2 between each pair of nodes, i.e., the node interconnect is modeled as a complete graph,

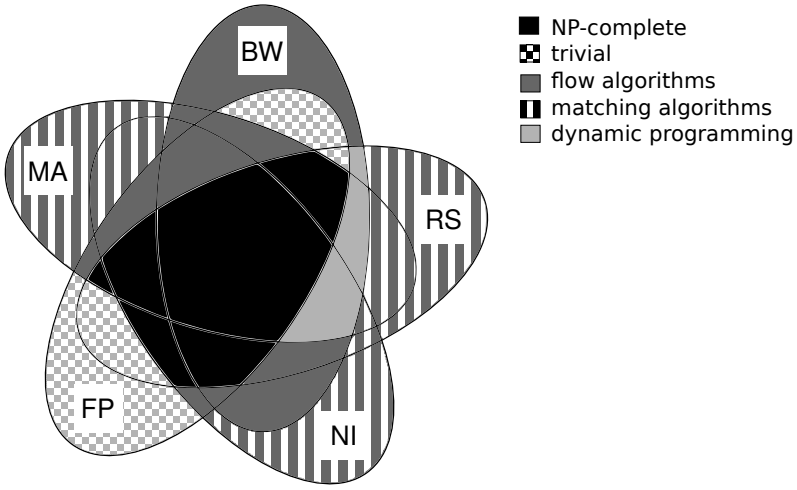

Figure 2.2: Fastest algorithms for different respective variants. Variants depicted by solid black are NP-hard, and variants depicted by checked filling are trivially solvable. For the remainder of variants we marked the fastest method.

to account for the all-to-all communication patterns of batch processing applications such as MapReduce. In the problem variant without NI, we optimize only the cost of assigning chunks to nodes, and the inter-node communication is set to zero, i.e., $b_2 = 0$.

Bandwidth Capacities (BW). We distinguish between an uncapacitated and a capacitated scenario where each link e of the substrate network comes with a bandwidth constraint $\mathsf{cap}(e)$, and we refer to the bandwidth-constrained version by BW. Note that capacity constraints introduce infeasible problem instances, where it is impossible to allocate sufficient resources to embed the virtual network. In the problem variant without BW, each link has infinite capacity, i.e., $\mathsf{cap}(e) = \infty$ for each edge e.

2.2 Polynomial-Time Algorithms

Despite the various degrees of freedom in terms of embedding and replica selection, we can solve many problem variants efficiently. This section introduces three general techniques, which can roughly be categorized into the *flow* (Section 2.2.1), *matching* (Section 2.2.2) and *dynamic programming* (Section 2.2.3) approaches. In Figure 2.2, we marked either the fastest method to solve the computational problem or its computational intractability.

First, let us make a simplifying observation:

Observation 1. *In* CTE *variants without flexible placement (FP), the bandwidth required for the interconnect network (NI) can be allocated upfront, as it does not depend on the replica selection and the assignment. Accordingly, we can reduce the $RS + MA + NI + BW$ variant (as well as all its subproblems) to $RS + MA + BW$ (resp. its subproblems).*

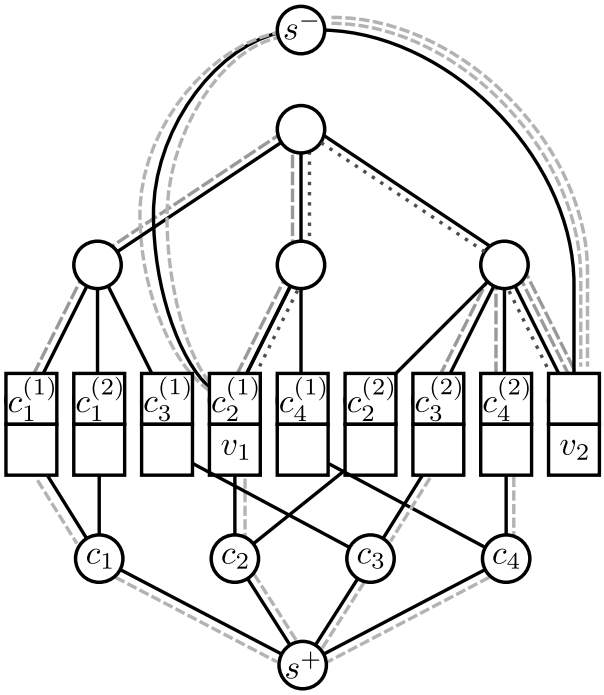

Figure 2.3: An example of the extended substrate network T^*: The sink s^- is connected to the two leaves that host the nodes. The artificial nodes are depicted below the leaves, are labeled with their respective chunks (e.g., c_1), and are connected to the source s^+ as well as to the leaves that contain replicas of their chunk. The maximum flow of minimum cost is indicated by the dashed lines: each line represents one unit of flow. The dotted lines indicate links which have reduced capacity due to NI.

2.2.1 Flow-Based Algorithm

We first present an algorithm to solve the RS+MA+NI+BW variant of the CTE problem. Recall that in this problem variant, we are given a set of redundant chunks (RS) and a set of nodes at fixed locations (no FP). The number of chunks may be larger than the number of nodes (MA), and each node needs to be connected to its selected chunks as well as to other nodes (NI), while respecting capacity constraints (BW). As we see in the following, we can use a flow approach to solve this problem variant.

Construction of the Artificial Graph. In order to solve the RS+MA+NI+BW variant of the CTE problem, we first remove the NI component using Observation 1. Then, we construct an artificial graph T^*, extending the substrate network T. We transform bandwidth capacities so that they correspond to the maximal number of chunks that we can transfer through the link. Concretely, for each link $e \in E(T)$, we set its new capacity in T^* to $\lfloor \mathsf{cap}(e)/b_1 \rfloor$. After this normalization, we extend the topology of T by introducing an artificial vertex for each of τ chunks. Each of these artificial vertices is connected to each leaf (i.e., server) in T where a replica of the respective chunk is located, connecting the replica by a link of capacity 1. In addition, we construct a *super-source* s^+, and connect it to each of the artificial chunk vertices with a link of capacity 1. Moreover, we construct an artificial *super-sink* s^- and connect it to every leaf containing at least one node; the link capacity represents the number of nodes this server hosts times the multi-assignment factor m. We additionally assign the following costs to edges of T^*:

every edge of the original substrate network costs one unit, and all other artificial edges cost nothing. A solution to the RS + MA + BW variant can now be computed from a solution to the *min-cost max-flow* problem between super-source s^+ and super-sink s^- on the artificial graph T^*. An illustration of this construction is presented in Figure 2.3.

Algorithm. Our algorithm to solve the RS + MA + NI + BW variant consists of three parts: First, we construct the extended graph T^* described above and compute a min-cost max-flow solution. State-of-the-art min-cost max-flow algorithm is the double scaling algorithm [AGOT92], which is based on the scaling technique [GT89, Tar85].

Second, we have to *round* the resulting, possibly fractional flow, to integer values. Due to the *integrality theorem* [AMO93], there always exists an optimal integer solution on graphs with integer capacities. However, min-cost max-flow algorithms may yield fractional solutions which need to be rounded to integral solutions (of the same cost) [Mad13].

Third, given an integer min-cost max-flow solution, we need to decompose the integer flow into paths representing matched chunk-node pairs: The assignment can be obtained by decomposing the flow allocated in the original substrate network. In order to identify a chunk-node assignment, we take an arbitrary (loop-free) path p carrying a flow of value at least 1 from s^+ to s^-: the first hop represents the chosen chunk, the second hop the chosen replica, and the penultimate hop represents the server: we assign the replica to an arbitrary node on this server that has less than m chunks assigned. Having found this pair, we reduce the flow along the path p by one unit. We continue the pairing process until every chunk is assigned.

Analysis. The runtime of our algorithm consists of four parts: construction of T^*, computation of the min-cost max-flow, flow rounding, and decomposition. The dominant term in the asymptotic runtime is the flow computation. Using the double scaling algorithm for min-cost max-flow [AGOT92], we get a runtime of $\mathcal{O}(|E|^2 \cdot \log\log U \cdot \log|E|)$ where $|E| = \mathcal{O}(n_S + \sum_{i=1}^{\tau} r_i + n_V)$ is the number of T^* edges, and U is the maximal link capacity. Note that in networks with high capacity and uncapacitated networks, we can simply set $U = \tau$, so the resulting overall runtime is $\mathcal{O}(|E|^2 \cdot \log\log\tau \cdot \log|E|)$. The algorithm has a quadratic dependence on the size of substrate network n_S, and in the next section, we propose an alternative algorithm with linear dependence on n_S.

2.2.2 Matching Algorithms

This section presents an alternative algorithm to solve RS + MA + NI and MA + NI + BW variants of the CTE problem. First, we consider the RS + MA + NI variant. Recall that in this variant, we are given a set of redundant chunks (RS) and a set of nodes at fixed locations (no FP). The number of chunks may be larger than the number of nodes (MA), and each node needs to be connected to its chunks as well as to other nodes (NI).

Algorithm. Due to Observation 1, the RS + MA + NI variant degenerates to RS + MA. In order to solve the latter, we construct a bipartite graph between the set of nodes and the set of chunks. First, we clone each node m times as each node needs to have m chunks assigned, and we aggregate all replicas of a given chunk in a single super-vertex. Second, we link (cloned) nodes to (aggregated) replicas. In instances without BW, assigning any replica to any node is

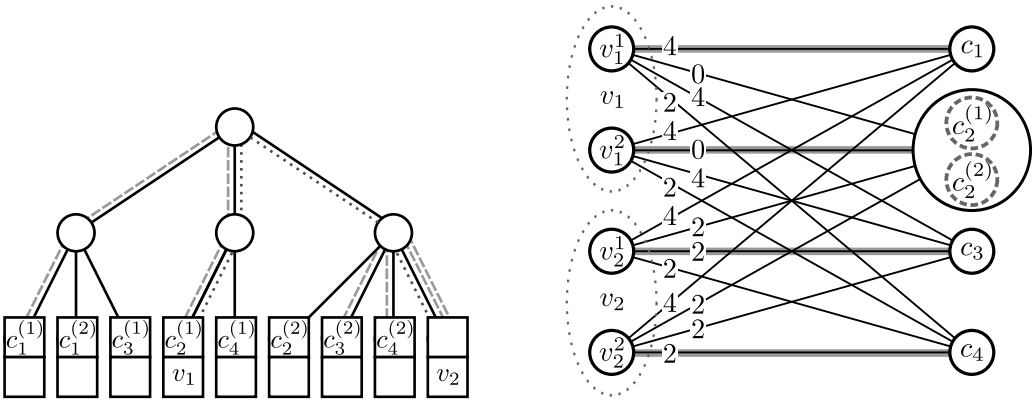

Figure 2.4: The RS + MA variant on the left is converted into a matching problem instance on the right. The figure illustrates an instance where two nodes are cloned into $m = 2$ nodes each, resulting in a total of four nodes in the matching representation. The two replicas of each chunk c_j are aggregated into a single super-vertex in the matching instance, and this gives a total of four super-vertices in the matching graph.

feasible, and hence, for a given assignment μ of chunks to nodes, the minimum bandwidth is utilized if for each c, node $\mu(c)$ serves the closest replica of c. Therefore, for each node v and chunk c, we connect each copy of v and the super-vertex c with the link of cost $\min_i \mathsf{dist}(v, c^{(i)})$, i.e., the cost of reaching the closest replica. Finally, on the resulting bipartite graph, we compute a *minimum weight perfect matching*: the resulting matching describes the optimal assignment of chunks to nodes. An example instance is presented in Figure 2.4.

Analysis. The runtime consists of two parts: the construction of the matching graph and the actual matching computation. The constructed graph consists of $(m \cdot n_V) \cdot \tau = \tau^2$ many edges, and for each edge, we compute its weight. The shortest distances in a tree of size n_S can be computed in time $\mathcal{O}(n_S + q)$ [GT83], where q is the number of queried pairs, which translates to the overall construction time $\mathcal{O}(n_s + n_v \cdot \sum_{i=1}^{\tau} r_i)$. The state-of-the-art algorithm to compute matchings in bipartite graph [CMSV17] has a running time of $\tilde{\mathcal{O}}(|E|^{10/7} \cdot \log W)$, where $|E|$ is the number of edges, W is the maximum weight of an edge, and $\tilde{\mathcal{O}}$ hides polylogarithmic (in terms of $|E|$) factors. The total running time is then $\tilde{\mathcal{O}}(\tau^{20/7} \cdot \log(n_s)) + \mathcal{O}(n_s + n_v \cdot \sum_{i=1}^{\tau} r_i)$. Note that the matching-based algorithm has a linear dependence on the size of the substrate network n_S, whereas the flow-based algorithm runtime contains the term $\mathcal{O}(n_S^2)$.

Local matching algorithm

Now, we present the way to solve the MA + NI variant even faster by using a greedy approach. Moreover, we show that we can even solve MA+NI+BW variants by simply verifying feasibility. In the following, due to Observation 1, we can focus on the MA and MA + BW variants, respectively. We start by lower-bounding the required bandwidth allocation. The *uplink* of a proper subtree T', denoted $\textsc{uplink}(T')$ is an edge from the root of T' to its parent.

Lemma 2.1. *In the* MA *variant of* CTE*, for a proper subtree T' containing $\tau(T')$ chunks and x nodes, the bandwidth allocation on the uplink of T' is at least $b_1 \cdot |\tau(T') - x \cdot m|$.*

Proof. In case the number of chunks equals the processing capacities of the nodes in the given subtree, the bandwidth allocation inflicted by the chunk access network on the uplink is zero since we can assign all chunks to nodes in the same subtree. Otherwise, we distinguish between two cases. If there are more chunks in the subtree, at least all excess chunks have to be assigned to nodes outside T', which inflicts costs b_1 per excess chunk on the uplink of T'. Similar situation occurs if the processing capabilities exceed the number of available chunks. Hence, the minimum bandwidth allocation for the chunk access on the uplink is the absolute difference between the number of chunks and the processing capabilities of the subtree, i.e., $|\tau(T') - x \cdot m|$ times b_1.□

Algorithm. Our proposed algorithm for the MA variant of CTE proceeds in a bottom-up fashion, traversing the substrate network T from the leaves toward the root. For each subtree T', we maintain two sets S_C, S_V in order to map unassigned chunks S_C in the subtree T' to nodes S_V in T'. Both sets are initially empty. We associate a counter with each node that enters S_V, and we initialize it to m, the multi-assignment factor.

We first process all the leaves, in an arbitrary order; subsequently, we process inner vertices of T whenever all their children have been processed. We process any leaf ℓ by adding any nodes or chunks which are located on ℓ to the corresponding sets S_C and S_V. A non-leaf vertex u is processed in the following way: we take the union of the sets corresponding to u's children, i.e., the sets containing the unmatched chunks and nodes in this subtree. For both leaves and inner nodes, whenever both sets are non-empty, we greedily assign an arbitrary chunk c in S_C with an arbitrary node v in S_V. Then, we remove c from S_C, and we decrement the counter of v. If the counter of v reaches zero, we remove v from S_V.

Analysis. For each vertex in the substrate graph, we build the union of the children's sets. The number of all remove operations is equal to the number of chunks $\mathcal{O}(\tau)$. Hence, the overall complexity of this construction amounts to $\mathcal{O}(n_S + \tau)$. The local matching algorithm outperforms the previous algorithm that relied upon calculating the minimum-weight perfect matching.

The algorithm allocates the minimum possible bandwidth at an uplink of every subtree, as stated in Lemma 2.1, and hence the algorithm computes an optimal solution. Combined with a simple post-processing step, this approach can also solve MA + BW variant. It is sufficient to observe that the algorithm allocates the minimal bandwidth on each individual edge. In consequence, any bandwidth constraint lower than the requirement given in Lemma 2.1 renders the problem infeasible. Hence, it is sufficient to temporarily omit the bandwidth limitations, compute an optimal assignment for an MA instance, and verify that the resulting allocations do not violate any capacities. The post-processing step scales linearly with the number of edges in the substrate graph.

2.2.3 Dynamic Programming

We now show how to solve the MA + FP + NI + BW variant in polynomial time. Note that this variant requires to find a tradeoff between the desire to place nodes as close as possible to each other (in order to minimize communication costs), and the desire to place nodes as close as possible to the chunk locations.

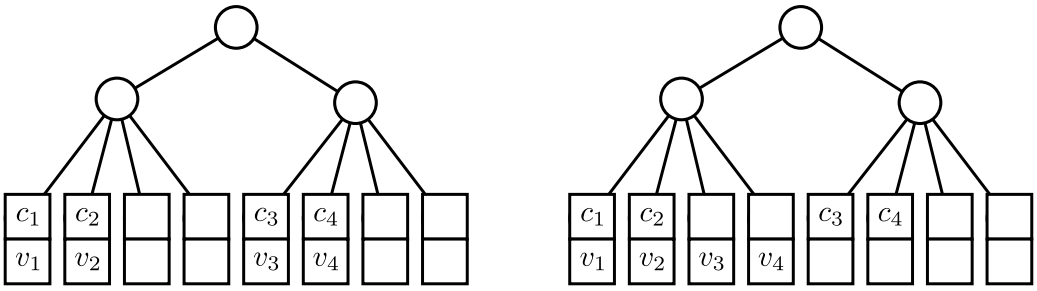

Figure 2.5: Two different node placements for the same substrate graph and chunk locations. For $b_1 = b_2$, both solutions have an identical footprint. In other cases, one solution outperforms the other.

Example. Figure 2.5 shows an example: one extreme solution is to minimize the distance between chunks and nodes, see mapping π_1 in the left picture in Figure 2.5: the four nodes are all collocated with chunks, resulting in a zero-cost chunk access network. As a result, the paths between the individual nodes are longer than in alternative node placements: each node is at a distance of two hops to one other node, and four hops to two other nodes. Hence the resulting allocations for the node interconnect sum up to $20 \cdot b_2$.

The right picture in Figure 2.5 shows a different node mapping π_2, which seeks to minimize the interconnect costs between the nodes, and places all nodes in one subtree. The distance between all nodes is 2, which results in a total bandwidth allocation of $12 \cdot b_2$ for the interconnect. However, this reduced price comes at additional costs in the access network: c_3 and c_4 have to be assigned to v_3 and v_4, which requires a total bandwidth allocation of $8 \cdot b_1$.

Algorithm. Our proposed approach is based on dynamic programming and leverages the *optimal substructure property* of the MA + FP + NI + BW variant: as we will see, optimal solutions for subtrees can be efficiently combined into optimal solutions for the whole tree. First, we transform the substrate network T into a binary tree: we clone every higher-degree node, iteratively attaching additional clones as right children and original children as left descendants. New edges between cloned vertices constructed during the binarization have infinite capacity.

As usual in dynamic programs, we define, over the structure of the tree, a recursive formula f for the minimal cost solution given any possible number of nodes embedded in a given subtree (the actual set of nodes does not matter, due to symmetry). The value of $f(T', x)$ corresponds to the minimum cost of placing x nodes in a subtree T'. It is defined as the cost of the bandwidth allocated inside T' plus the bandwidth allocated on the uplink of T'. The latter is equal to

$$bw(T', x) = b_1 \cdot |\tau(T') - x \cdot m| + b_2 \cdot (n_V - x) \cdot x \ ,$$

where $\tau(T')$ is the number of chunks in a subtree T'. If $bw(T', x)$ exceeds the capacity of the uplink of T', we set $bw(T', x) = \infty$. The first term of bw represents the bandwidth necessary to transport the chunks in T' from their location to nodes outside of T' (see Lemma 2.1 for similar argument). The second term of bw represents the required bandwidth for the communication between the x nodes inside T', and the $n_V - x$ nodes in the remaining parts of the substrate network. For the coherence of the description, we assume that the whole tree T also has a dummy uplink of zero capacity. Our goal is to compute $f(T, n_V)$; note that by our definition $bw(T, n_V) = 0$ as expected.

Then, the formula to calculate the function f for a subtree containing only a leaf ℓ is

$$f(\{\ell\}, x) = \begin{cases} \infty & \text{if } x > \mathsf{cap}(\ell) \ , \\ bw(\{\ell\}, x) & \text{otherwise} \ . \end{cases}$$

That is, we set $f(\{\ell\}, x)$ to infinity if x exceeds the node hosting capacity of the server ℓ.

To calculate the value of $f(T', x)$ for non-leaf subtree T', we split x nodes into two non-negative integer values, r and $x - r$, and we put r in the right subtree and $x - r$ in the left one. That is, we take the optimal cost (given recursively) of placing r nodes in the right subtree $\textsc{Ri}(T')$ of T' and $x - r$ nodes in the left subtree $\textsc{Le}(T')$ of T'. Given the cheapest combination, we add the bandwidth requirements on the uplink of T' to generate the overall costs for placing x nodes in T', obtaining

$$f(T', x) = \min_{0 \le r \le x} \{ f\left(\textsc{Le}(T'), x - r\right) + f\left(\textsc{Ri}(T'), r\right) \} + bw(T', x) \ .$$

To compute $f(T, n_V)$, we evaluate f a bottom-up manner. To finally find an actual optimal embedding, we traverse the computed minimal-cost path backward, according to the optimal values found for f during the bottom-up computation.

Analysis. The substructure optimality follows from the observation that costs can be accounted on the uplink and the fact that we check each possible node distribution. For each substrate vertex (n_S many) we have to check the cost of all possible splits, resulting in an overall complexity of $\mathcal{O}(n_S \cdot n_V^2)$. The runtime to binarize T is asymptotically negligible in comparison.

2.2.4 Simple Variants

For the sake of completeness, we also observe that there are several variants that admit a trivial solution. Concretely, variants with FP plus any combination of RS and BW (but without MA and NI) can easily be solved by assigning nodes to chunk locations.

2.3 NP-Hardness Results

We have seen that even variants with multiple dimensions of flexibility can be solved optimally in polynomial time. This section now points out fundamental limitations in terms of computational tractability. In particular, we show that variants become NP-hard if flexibly placeable nodes (FP) have to be assigned to one of the multiple replicas (RS), either with multiple chunks per node (MA in Section 2.3.2) or with communication among nodes (NI in Section 2.3.4). Both results hold even in uncapacitated networks, and even in small-diameter substrate networks (namely two- or three-level trees). Clearly, the hardness of the FP+RS+MA and FP + RS + NI variants imply the hardness of more general ones.

In addition to the hardness results for the aforementioned CTE variants, we study the influence of restricting the number of replicas of a chunk on computational tractability on these variants. Namely, we introduce the restricted replica selection component RS(r_{max}), parameterized by an integer r_{max}, that restricts the number of replicas of any chunk: for

every chunk c_i we have $r_i \leq r_{max}$. We consider RS(2) component in relation with multi-assignment (MA): we first present the NP-hardness of the unrestricted scenario FP + RS + MA as a warm-up, and then we present more refined hardness result for FP + RS(2) + MA.

2.3.1 Introduction to 3D Perfect Matching

The hardness of both FP + RS + MA and FP + RS + NI variants of the CTE problem uses a reduction from the NP-complete *3D Perfect Matching* problem [CKH+00], which can be seen as a generalization of bipartite matchings to 3-uniform hypergraphs. We refer to this problem as 3-DM and we review it quickly for completeness. 3-DM is defined as follows: we are given three finite and disjoint sets X, Y, and Z of cardinality k, as well as a subset of triples $P \subseteq X \times Y \times Z$. Set $M \subseteq P$ is a 3-dimensional matching if and only if, for any two distinct triples $t_1 = \langle x_1, y_1, z_1 \rangle \in M$ and $t_2 = \langle x_2, y_2, z_2 \rangle \in M$, it holds that $x_1 \neq x_2$, $y_1 \neq y_2$, and $z_1 \neq z_2$. The goal is to decide if there exists a subset of triples $M \subseteq P$ that is *perfect*, i.e., covers all elements of $X \cup Y \cup Z$ exactly once.

2.3.2 Hardness of Multi-Assignments

The proof of hardness of the FP + RS + MA variant of the CTE problem is based on the following construction. Let $I_{\text{3D-M}}$ be an instance of 3-DM with p triples and set cardinality $k = |X| = |Y| = |Z|$. We create an instance I_{CTE} of the CTE problem in the following way:

1. *Substrate network:* We construct a substrate network in a form of a tree consisting of a root, and for each triple from P, we construct a *triple gadget* which we directly attach as a child of the root. The gadget is of height 2 and consists of an inner node and three leaves.

2. *Chunks:* For each element e in X, Y and Z, we construct a chunk ($3 \cdot k$ chunks in total). Every gadget contains three chunk replicas (one per leaf), corresponding to the elements of the triple. Note that the number of replicas for a chunk for element e corresponds to the number of triples that contain e.

3. *Other properties of the instance:* To transform the optimization problem into a decision problem, we say that the solution is feasible if it has a cost at most $\xi = 4 \cdot k$ (the ξ is the acceptance threshold). We fix the number of to-be-embedded nodes $n_V = k$, bandwidth $b_1 = 1$, and the the multi-assignment factor $m = 3$.

Figure 2.6 shows an example of our construction. Given these concepts, we can now show the computational hardness of the considered variant of CTE.

Theorem 2.2. *The* FP + RS + MA *variant of the* CTE *problem is NP-hard.*

Proof. Fix an instance $I_{\text{3D-M}}$ of 3-DM and construct the I_{CTE} instance of the FP + RS + MA variant in the way described above. We prove that I_{CTE} has a solution of cost at most $\xi = 4 \cdot k$ if ($\Leftarrow$) and only if ($\Rightarrow$) $I_{\text{3D-M}}$ has a perfect matching (of size k).

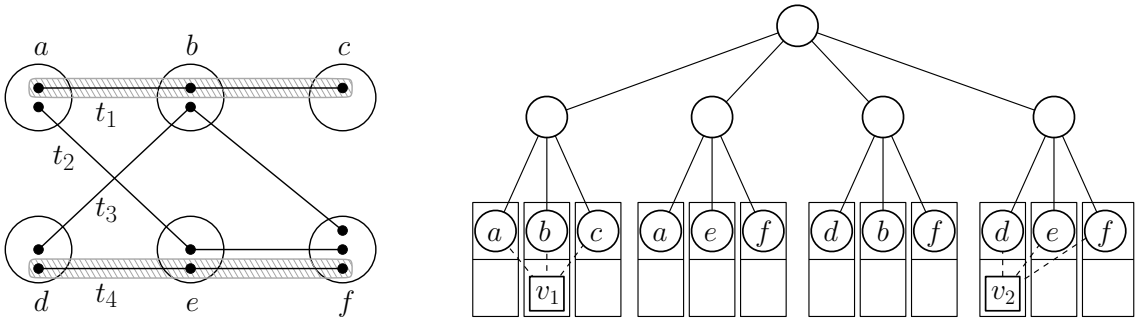

Figure 2.6: *Left:* A 3-DM instance of cardinality $k = 2$ with four triples: $t_1 = \langle a, b, c\rangle$, $t_2 = \langle a, e, f\rangle$, $t_3 = \langle d, b, f\rangle$, and $t_4 = \langle d, e, f\rangle$. The solution is indicated by the grey triples t_1 and t_4. *Right:* The corresponding instance and an optimal solution for the FP + MA + RS variant of the CTE problem. In specifying the chunk placement, we omitted replica superscripts. Each triple corresponds to a single triple gadget.

($\Leftarrow$) Fix a solution $S_{\text{3D-M}}$ for $I_{\text{3D-M}}$, i.e., the set of triples that form a perfect matching. To construct the solution S_{CTE} to I_{CTE}, we place a single node in every gadget that corresponds to a triple from $S_{\text{3D-M}}$ (at an arbitrary leaf of this gadget). In each gadget, we assign all three chunks to the node located in this gadget. This solution costs exactly $\xi = 4 \cdot k$: each of k nodes is assigned one chunk collocated with it and two chunks at distance 2. The solution $S_{\text{3D-M}}$ matches every element of the universe $X \cup Y \cup Z$, hence every chunk has a replica assigned to exactly one node, and S_{CTE} is feasible.

($\Rightarrow$) Fix a solution S_{CTE} for I_{CTE} with cost at most ξ. We call the triple gadget *active* if it hosts a single node at one of its leaves. We claim that S_{CTE} places at most one node in each triple gadget. For a contradiction, suppose that two nodes x_1, x_2 are placed in the same triple gadget. First, we lower-bound the cost incurred by x_1 and x_2 that collectively have $2 \cdot m = 6$ replicas assigned. At most two of these chunks are co-located with x_1 or x_2 and cost 0, and remaining four chunks incur a cost at least 2 each. Moreover, at least three of these chunks are placed outside of this triple gadget and incur the cost at least 4 each. In total, chunks assigned for x_1 and x_2 incur the cost at least $2 \cdot 0 + 2 \cdot 1 + 3 \cdot 4 = 14$. Second, we lower-bound the cost incurred by remaining $k - 2$ nodes. Each node can have at most 1 of its $m = 3$ replicas assigned for free (every leaf hosts exactly one chunk), and it incurs the cost at least 4 for the remaining 2 chunks that are at distance at least 2. The total chunk assignment cost is then $4 \cdot (k - 2) + 14 = 4 \cdot k + 6$ and exceeds the threshold $\xi = 4 \cdot k$, a contradiction. As $n_V = k$, we conclude that there are exactly k active gadgets in S_{CTE}.

To construct the solution $S_{\text{3D-M}}$ to $I_{\text{3D-M}}$, we pick the set of triples whose gadgets are active. The solution $S_{\text{3D-M}}$ consists then of k triples. Since each chunk is assigned to exactly one node, every element of X, Y and Z is matched and $S_{\text{3D-M}}$ forms a 3-dimensional perfect matching.□

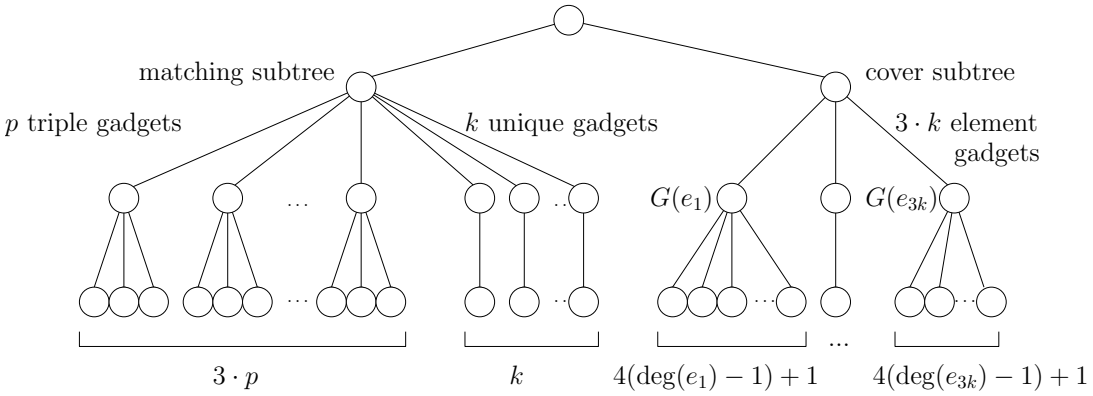

Figure 2.7: Overview of the substrate network in the proof of the NP-hardness of the FP + RS(2) + MA variant of CTE.

2.3.3 Hardness of Multi-Assignments with at Most Two Replicas

Now we provide a more detailed look at this hardness result and explore the minimal requirements for rendering replica selection hard. Concretely, we show that already two replicas for each chunk are sufficient for computational intractability.

For an arbitrary instance $I_{\text{3D-M}}$ of the 3-DM problem, we construct a RS(2) + MA + FP variant instance I_{CTE} the following way. Let $k = |X| = |Y| = |Z|$ and $p = |P|$. For each element $e \in X \cup Y \cup Z$, by P_e we denote the set of all triples that contain e. Let $\deg(e) = |P_e|$, and note that $\sum_e \deg(e) = 3 \cdot p$. We proceed with the construction as follows.

Substrate network. We construct a substrate network that consists of two subtrees connected to the root: a *matching subtree* and a *cover subtree*. The matching subtree consists of p *triple gadgets* (one per each triple from P) and k *unique gadgets*. The cover subtree consists of $3 \cdot k$ *element gadgets* $G(e)$, one for each element $e \in X \cup Y \cup Z$. The construction is depicted in Figure 2.7. Gadgets are constructed in the following way.

1. Each triple gadget consists of four vertices: three leaves and the root of the gadget, as defined in the reduction for the FP + RS + MA variant in Section 2.3.2.

2. Each unique gadget consists of two vertices: the leaf and the root of the gadget. This keeps the tree balanced, and also keeps leaves of unique gadgets far from each other.

3. The element gadget $G(e)$ for element e consists of the gadget root and $4 \cdot (\deg(e) - 1) + 1$ leaves. There are $3 \cdot k$ elements numbered from e_1 to $e_{3 \cdot k}$, and $3 \cdot k$ corresponding element gadgets $G(e_1), \ldots, G(e_{3 \cdot k})$.

Chunks. We construct three sets of chunks. The first set corresponds to elements of the universe $X \cup Y \cup Z$. This time we cannot repeat the construction of chunks from Section 2.3.2, because we need to obey the restricted replication factor. The construction for the 3D-M instance from Figure 2.6 is given in Figure 2.8.

1. *Element chunks.* For each triple $t = \langle x, y, z \rangle \in P$, we construct 3 chunks: x_t, y_t, and z_t, i.e., $\sum_e \deg(e) = 3 \cdot p$ chunks in total. Then, each element e has $\deg(e)$ element chunks e_t

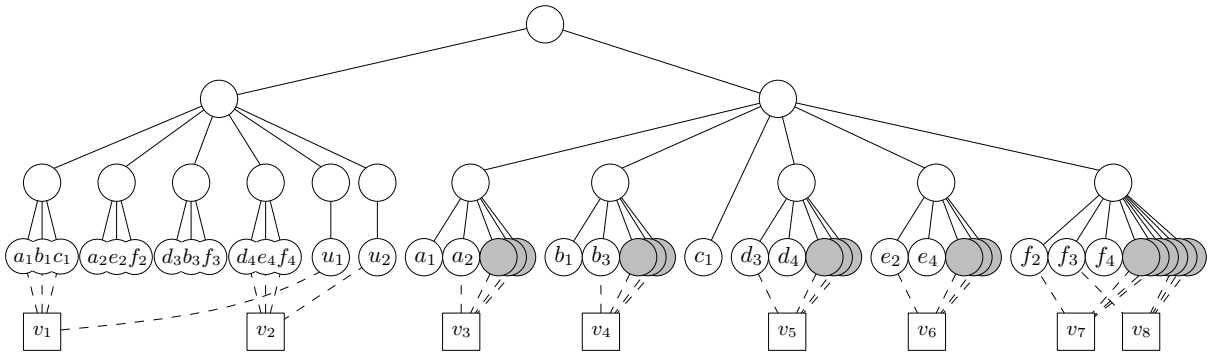

Figure 2.8: An instance of FP + MA + RS(2) variant of CTE corresponding to the 3-DM instance from Figure 2.6. Leaves containing dummy chunks are grey. An optimal solution assigns chunks to nodes $v_1, \dots v_8$ by the dashed lines. In specifying the chunk placement we omit replica superscripts.

for each $t \in P_e$, collectively called C_e. Each element chunk has two replicas. First replicas are placed in the matching subtree: we place three replicas: $x_t^{(1)}, y_t^{(1)}$ and $z_t^{(1)}$ in the triple gadget for triple $t \in P$, one per each leaf. Second replicas are placed in the cover subtree: for each element $e \in X \cup Y \cup Z$, for each $t \in P_e$, we place replicas $e_t^{(2)}$ at different leaves of the gadget $G(e)$. In total, we place $\deg(e)$ replicas of element chunks at the leaves of the gadget $G(e)$.

2. *Unique chunks.* We construct k unique chunks $u_1, \dots, u_k$ with one replica each, and we place these replicas at the leaves of unique gadgets (one chunk per leaf).

3. *Dummy chunks.* For each element $e \in X \cup Y \cup Z$, we construct an additional set of $3 \cdot (\deg(e) - 1)$ dummy chunks, with one replica each, and we place these replicas in $3 \cdot (\deg(e) - 1)$ leaves of $G(e)$ (one chunk per leaf), not occupied by (the second) replicas of element chunks. Recall that the remaining $\deg(e)$ leaves of $G(e)$ contain replicas $e_t^{(2)}$ for each $t \in P_e$, i.e., there is one replica at each leaf.

Other properties of the instance. The multi-assignment factor is $m = 4$, and the number of nodes to embed is $n_V = k + \sum_e(\deg(e) - 1) = k + \sum_e \deg(e) - 3 \cdot k = 3 \cdot p - 2 \cdot k$. We set the cost threshold to $\xi = 8 \cdot k + 6 \cdot (n_V - k) = 18 \cdot p - 10 \cdot k$.

Given any 3D-M instance $I_{\text{3D-M}}$, we produce the corresponding instance I_{CTE} of the RS(2)+ MA + FP variant, in the way described above. We illustrate an instance of CTE corresponding to the 3D-M instance from Figure 2.6 in Figure 2.8. The reduction (Theorem 2.9) consists of two parts: (1) given a solution for $I_{\text{3D-M}}$, we construct a solution to I_{CTE} of cost at most ξ, and (2) given a solution for I_{CTE} of cost at most ξ, we construct a solution for $I_{\text{3D-M}}$. The first part is easier, and we present it now.

Lemma 2.3. *Fix an instance $I_{\text{3D-M}}$ of* 3D-M *and construct an instance I_{CTE} of* CTE *in the way described in this section. If there exists a perfect 3D matching for $I_{\text{3D-M}}$, then I_{CTE} has a solution of cost at most ξ.*

Proof. Fix a feasible solution $S_{\text{3D-M}}$ for $I_{\text{3D-M}}$. In the solution S_{CTE} for I_{CTE} we need to place $n_V = k + \sum_e(\deg(e) - 1)$ nodes and assign 4 chunks to each of these.

1. We place k nodes in k triple gadgets (one per gadget, at an arbitrary leaf) that correspond to triples chosen in the solution $S_{\text{3D-M}}$. To each such node, we assign 3 chunk replicas from its gadget and one arbitrary, unassigned unique chunk.

2. For each element e, in $G(e)$ we place $\deg(e) - 1$ nodes, and to each of them we assign 3 arbitrary dummy chunks and 1 replica of an arbitrary element chunk (in this gadget) whose another replica is not assigned in any triple gadget. We co-locate the node with one of its assigned replicas, hence one of the assignments is for free, and the other three assignments incur the cost 2 each.

Triples selected in $S_{\text{3D-M}}$ match each element exactly once, and hence nodes in triple gadgets have exactly one chunk assigned from C_e for each element e. For each element e, by e_* we denote the element chunk e_t that is assigned in a triple gadget, i.e., the replica $e_*^{(1)}$ is assigned in a triple gadget. The remaining $\deg(e) - 1$ chunks from $C_e \setminus \{e_*\}$ (their second replicas) are assigned to $\deg(e) - 1$ nodes in $G(e)$. Finally, each of k unique chunks is assigned to a node in a triple gadget, and each dummy chunk is assigned to some node in its element gadget. This shows that each chunk is assigned to a node.

To see that the solution cost does not exceed the threshold ξ, we sum up the total assignment cost. The k nodes placed in triple gadgets have three assignments to chunk replicas within the triple gadget of the total cost $0+2+2 = 4$, and one assignment of cost 4 (to some unique gadget). The remaining $n_V - k$ nodes placed in the cover subtree have four assignments within element gadgets for the total cost $0 + 3 \cdot 2 = 6$. Summing up, the incurred cost is $8 \cdot k + 6 \cdot (n_V - k) = \xi$, i.e., the solution is feasible. □

To construct a solution for $I_{\text{3D-M}}$ from a solution to I_{CTE}, we show that every feasible solution for I_{CTE} has a certain structure. We call a triple gadget *active*, if it contains a single node at one of its leafs, and we call a node *active* if it is placed in an active triple gadget. In the lemmas below, we show in particular that in feasible solutions to I_{CTE}, exactly k triple gadgets are active (similarly to Theorem 2.2).

In the instance I_{CTE}, chunks can be assigned to nodes at distance $0, 2, 4$ or 6. We call the assignments at distance 0 or 2 *short*, and the ones at distance 4 or 6 *long*.

Lemma 2.4. *Any solution to I_{CTE} with more than k long assignments is infeasible.*

Proof. Consider a solution that uses $k + i$ long assignments for $i > 0$. The solution consists of $m \cdot n_V = 4 \cdot n_V$ assignments, and $k + i$ of them incur cost at least $4 \cdot (k + i)$. As every leaf of the substrate network of I_{CTE} contains one replica, at most n_V assignments are free. As the minimum distance between leaves is 2, the remaining $3 \cdot n_V - k - i$ assignments incur the cost 2 each. In total, the cost of the solution is then $4 \cdot (k + i) + 2 \cdot (3 \cdot n_V - k - i) = 6 \cdot n_V + 2 \cdot k + 2 \cdot i = \xi + 2 \cdot i$, i.e., the solution cost exceeds the threshold ξ. □

Lemma 2.5. *Any feasible solution to I_{CTE} places exactly k nodes in the matching subtree and $\deg(e) - 1$ nodes in gadget $G(e)$ for each $e \in X \cup Y \cup Z$.*

Proof. First, we show that at most k nodes are placed in the matching subtree. Each node placed in the matching subtree has at most 3 leaves in the distance at most 2, hence at least one of its $m = 4$ assignments is long. Therefore, more than k nodes in the matching subtree would lead to more than k long assignments, and by Lemma 2.4, the solution would be infeasible.

As at most k nodes are placed in the matching subtree, at least $n_V - k$ nodes are placed in the cover subtree. Now we claim that the cover subtree contains exactly $n_V - k$ of them. Suppose the contrary, i.e., that $n_V - k + i$ nodes were placed in the cover subtree for $i > 0$. With each element gadget $G(e)$ we associate its *volume* $\deg(e) - 1$. The volume corresponds to the maximum number of nodes that can be placed inside the gadget without causing a long assignment. More concretely, if the number of nodes in $G(e)$ exceeds its volume, i.e., $\deg(e) - 1 + j_e$ nodes are placed in $G(e)$ for $j_e > 0$, then the number of chunks assigned to these nodes is $4 \cdot (\deg(e) - 1 + j_e)$, and at most $4 \cdot (\deg(e) - 1) + 1$ of them can be assigned within the gadget $G(e)$. For the remaining $4 \cdot j_e - 1 \geq 3 \cdot j_e$ chunks, long assignments are used. By the pigeon-hole principle, at least $i = \sum_e j_e$ nodes surpass the volume of their gadgets, causing at least $3 \cdot i$ long assignments. As each of $k - i$ nodes in the matching subtree causes one long assignment, in total we have at least $k + 2 \cdot i$ long assignments and, by Lemma 2.4, the solution is infeasible, a contradiction. Therefore, exactly k nodes are placed in the matching subtree, and exactly $\deg(e) - 1$ nodes are placed in each element gadget $G(e)$. □

Lemma 2.6. *A feasible solution S_{CTE} to I_{CTE} has no nodes placed in unique gadgets.*

Proof. By Lemma 2.5, exactly k nodes were placed in the matching subtree. Suppose that $j > 0$ of these nodes were placed in the unique gadgets. The leaf of every unique gadget is at distance at least 4 from every other leaf of the substrate network. Hence, each of j nodes placed in a unique gadget has at least 3 out of its $m = 4$ chunks assigned by a long assignment. As triple gadgets consists of 3 leaves, each of remaining $k - j$ nodes placed in the triple gadget uses at least 1 long assignment. In total, the number of long assignments would be then at least $k - j + 3 \cdot j > k$, which by Lemma 2.4 would lead to infeasibility of S_{CTE}. □

Lemma 2.7. *In a feasible solution S_{CTE} to I_{CTE}, exactly k triple gadgets are active.*

Proof. By Lemma 2.5, exactly k nodes were placed in the matching subtree. By Lemma 2.6, these k nodes are placed in the triple gadgets and not in the unique gadgets. As there are exactly 3 replicas in each triple gadget, placing more than one node in a single triple gadget causes at least additional 3 long assignments (the argument is the same as in the proof of Theorem 2.2). Therefore, exactly k triple gadgets are active. □

Lemma 2.8. *In a feasible solution S_{CTE} to I_{CTE}, each active node has exactly 1 long assignment (to some unique chunk) and 3 short assignments, and remaining nodes have 4 short assignments.*

Proof. Consider an active node (in an active gadget). Only three chunks are placed at distance at most 2 to this node. Hence, at least 1 of its $m = 4$ chunks is assigned at distance at least 4, i.e., by the long assignment. If an active node would have more than 1 long assignment, then

by Lemma 2.7, the total number of long assignments would exceed k, and by Lemma 2.4 the solution would be infeasible. Therefore, each active node has exactly 1 long assignment. By Lemma 2.4, in S_{CTE} there are at most k long assignments, and as these are used by k active nodes, the remaining nodes (in the cover subtree) have no long assignments (they have 4 short assignments). Finally, by Lemma 2.6, k unique chunks have no node placed at them, hence they can be assigned only by a long assignment, and the only nodes that have long assignments are the active nodes that have k of them in total. □

Theorem 2.9. *The* RS(2) + MA + FP *variant of the* CTE *problem is NP-hard.*

Proof. Fix an instance $I_{\text{3D-M}}$ of 3D-M. We show that I_{CTE} constructed on the basis of $I_{\text{3D-M}}$ has a solution of cost $\leq \xi$ if ($\Leftarrow$) and only if ($\Rightarrow$) there exists a perfect 3D matching in $I_{\text{3D-M}}$. By Lemma 2.3, ($\Leftarrow$) holds.

To show ($\Rightarrow$), fix a feasible solution S_{CTE} for I_{CTE} in the way described in the construction section. By Lemma 2.7, exactly k triple gadgets are active. We construct the solution $S_{\text{3D-M}}$ from the set of k triples from P that correspond to active triple gadgets.

Now, we show that $S_{\text{3D-M}}$ is feasible, i.e., it matches every element of $X \cup Y \cup Z$. For every element $e \in X \cup Y \cup Z$, we constructed a set of chunks C_e, each chunk having two replicas: one in the matching subtree and one in the cover subtree. By Lemma 2.5, in each element gadget $G(e)$ we have exactly $\deg(e) - 1$ nodes, and by Lemma 2.8, these nodes have short assignments only. Hence, they have $4 \cdot (\deg(e) - 1)$ chunks from $G(e)$ assigned, i.e., one chunk is not assigned. Dummy chunks have one replica, and by Lemma 2.8, long assignments are used only for unique chunks, and thus dummy chunks must be assigned by nodes placed in the element gadget. The one remaining chunk is an element chunk. As its second replica (in the cover subtree) is not assigned, its first replica (in the matching subtree) must be assigned to some node. Therefore, active nodes have one of $\deg(e)$ element chunks assigned for each $e \in X \cup Y \cup Z$, and thus $S_{\text{3D-M}}$ matches each $e \in X \cup Y \cup Z$. □

2.3.4 Hardness of Interconnects

Next, we prove that the joint optimization of node placement and replica selection is NP-hard if an interconnect has to be established between nodes. In our terminology, this is the FP+ RS + NI variant. The proof is similar in spirit to the proof of the FP + RS + MA variant, however, we modify the construction to account for the absence of MA: we place bandwidth constraints on certain links in the substrate network. Additionally, we choose a high value for b_1 (the cost of chunk assignment), so that nodes are directly collocated with their assigned chunk replicas.

Construction. Let $I_{\text{3D-M}}$ be an instance of 3-DM with p triples and set cardinality $k = |X| = |Y| = |Z|$. We use the identical substrate network as in Section 2.3.2 with identical chunk replicas placed in the same way. This time, however, the interconnect cost is $b_2 = 1$, and the number of nodes (virtual machines) is $n_V = 3 \cdot k$, where k is the set cardinality. The threshold value is $\xi := 2 \cdot (3 \cdot k) + 4 \cdot (\binom{3 \cdot k}{2} - (3 \cdot k)) = 6 \cdot k + 18 \cdot (k - 1) \cdot k$, and we set the access cost b_1 to $\xi + 1$. High assignment cost forces each node to be collocated with the replica assigned to it. We formalize this observation in Lemma 2.10.

Lemma 2.10. *In any feasible solution to I_{CTE}, k gadgets have exactly 3 nodes each, and $p-k$ gadgets remain empty.*

Proof. The $n_V = 3 \cdot k$ nodes have to be placed directly at different $3 \cdot k$ leaves, and thus at $3 \cdot k$ chunks, as otherwise, the access cost would immediately exceed the threshold ξ. The $3 \cdot k$ nodes are distributed among at least k gadgets as each gadget can host at most 3 nodes. In total there are $\binom{3 \cdot k}{2}$ pairs of nodes, and the interconnect cost is 2 for each intra-gadget pair and 4 for each inter-gadget pair. If a gadget contains 3 nodes, then it has 3 intra-gadget pairs.

Suppose that nodes were distributed among more than k gadgets. Then, some gadgets contain less than 3 nodes, and the number of intra-gadget pairs is $3 \cdot k - i$ for some $i > 0$ (each of cost 2). The remaining pairs of nodes are connected between gadgets (of cost 4 each). Hence, the total cost is $2 \cdot (3 \cdot k - i) + 4 \cdot (\binom{3 \cdot k}{2} - (3 \cdot k - i)) = 6 \cdot k + 4 \cdot (\binom{3 \cdot k}{2} - 3 \cdot k) + 2 \cdot i = \xi + 2 \cdot i$, i.e., the solution exceeds the threshold ξ, a contradiction. □

Theorem 2.11. *The* FP + RS + NI *variant of* CTE *problem is NP-hard.*

Proof. Let $I_{\text{3D-M}}$ be the corresponding instance of 3-DM and let I_{CTE} be an instance of the FP+ RS + NI variant constructed as described above. We prove that I_{CTE} has solution of cost at most ξ if ($\Leftarrow$) and only if ($\Rightarrow$) $I_{\text{3D-M}}$ has a solution.

($\Leftarrow$) To compute a solution for I_{CTE} given a solution for $I_{\text{3D-M}}$, we proceed as follows. Given a perfect matching $S_{\text{3D-M}}$ consisting of triples $\{t_1, t_2, \ldots, t_k\}$, we place three nodes in each gadget that corresponds to every triple of $S_{\text{3D-M}}$ (one node per leaf). Chunks are assigned to the nodes which are co-located in the same server, and thus each chunk is assigned.

The interconnect costs inside all gadgets are $2 \cdot (3 \cdot k)$, as the distance between every pair these of nodes is 2, and there are 3 intra-gadget pairs in each of k gadgets. The remaining cost is $4 \cdot (\binom{3 \cdot k}{2} - (3 \cdot k))$, as the remaining $\binom{3 \cdot k}{2} - (3 \cdot k)$ pairs are intra-gadget pairs, and these incur the cost 4 each. The total cost is then $2 \cdot (3 \cdot k) + 4 \cdot (\binom{3 \cdot k}{2} - (3 \cdot k)) = \xi$.

($\Rightarrow$) Given a solution for I_{CTE}, we use Lemma 2.10 to construct a solution for $I_{\text{3D-M}}$: in any solution of cost at most ξ, k gadgets contain exactly 3 nodes. These gadgets correspond to a valid 3D perfect matching: as exactly one replica of every chunk is assigned, every element is covered exactly once. □

Remark on two replicas and interconnects. It is worth noting that the FP + RS(2) + NI + BW variant is also NP-hard [FPS17]. The reduction uses similar construction to the reduction from Section 2.3.2, with the help of bandwidth constraints that allow to control the number of nodes placed in a subtree.

2.4 Conclusions

In this chapter, we have shown that despite the multiple dimensions of flexibility in terms of chunk assignment and node placement, and despite the large scale of modern data centers, many variants can be solved efficiently. However, we have also shown that several embedding

variants are NP-hard already in two- and three-level trees—a practically relevant result given today's data center topologies [ALV08].

Our results are summarized in Figure 2.2. One interesting take away from this figure regards the question of which properties render the problem NP-hard. For instance, we see that BW does not influence the hardness of any variant, while RS is crucial for NP-hardness. MA only affects hardness if combined with RS. NI is trivial without FP, and FP requires more sophisticated algorithms when combined with NI or MA; in combination with RS and MA or NI, FP renders the problem NP-hard.

Chapter 3

Virtual Networks with Dynamic Topology

This chapter further explores algorithmic challenges of network-efficient mapping of virtual networks in data centers. In Chapter 2, we considered the static mapping, where nodes of the virtual network (virtual machines) are placed at vertices of the physical substrate network until the computation task is done. However, communication patterns in virtual networks are difficult to predict, and static mappings may result in sub-optimal network utilization. In this chapter, we study the dynamic mapping, where it is possible to *migrate* a virtual machine from one physical machine to another, incurring a fixed cost. We focus on a particular substrate network topology, namely a 1-level tree.

In contrast to Chapter 2, where we focused on fixed virtual topology suitable for batch processing applications, now we study general virtual networks. The communication pattern is not known in advance, and upon receiving a communication request it is possible to reconfigure a mapping by migrating virtual machines. By migrating repeatedly communicating virtual machines closer together in the network, it is possible to reduce the overall network utilization. To emphasize the different focus of this chapter, we refer to vertices of the substrate network as clusters.

3.1 Problem Definition

Formally, the online *Balanced RePartitioning* problem (BRP) is defined as follows. There is a set of n nodes, initially distributed arbitrarily across ℓ clusters, each of size k. We call two nodes $u, v \in V$ *collocated* if they are in the same cluster.

An input to the problem is a sequence of communication requests $\sigma = (u_1, v_1), (u_2, v_2), (u_3, v_3), \ldots$, where pair (u_t, v_t) means that the nodes u_t, v_t exchange a fixed amount of data. For succinctness of later descriptions, we assume that a request (u_t, v_t) occurs at time $t \geq 1$. At any time $t \geq 1$, an online algorithm needs to serve the communication request (u_t, v_t). Right before serving the request, the online algorithm can repartition the nodes into new clusters. We assume that a communication request between two collocated nodes costs 0. The cost of a communication request between two nodes located in different clusters is normalized to 1,

and the cost of migrating a node from one cluster to another is $\alpha \geq 1$, where α is a parameter (an integer). For any algorithm ALG, we denote its total cost (consisting of communication plus migration costs) on sequence σ by $\text{ALG}(\sigma)$.

The description of some algorithms (in particular the ones in Sections 3.2 and 3.3) is more natural if they first serve a request and then optionally migrate. Clearly, this modification can be implemented at no extra cost by postponing the migration to the next step.

As already mentioned in the introduction, we will estimate the performance of an online algorithm by comparing to the performance of an optimal offline algorithm. Formally, let $\text{ONL}(\sigma)$, resp. $\text{OPT}(\sigma)$, be the cost incurred by an online algorithm ONL, resp. by an optimal offline algorithm OPT, for a given sequence of requests σ. In contrast to ONL, which learns the requests one-by-one as it serves them, OPT has complete knowledge of the entire request sequence σ *ahead of time*. The goal is to design online repartitioning algorithms that provide worst-case guarantees. In particular, ONL is said to be *ρ-competitive* if there is a constant β, such that for any input sequence σ it holds that

$$\text{ONL}(\sigma) \leq \rho \cdot \text{OPT}(\sigma) + \beta.$$

Note that β cannot depend on input σ but can depend on other parameters of the problem, such as the number of nodes or the number of clusters. The minimum ρ for which ONL is ρ-competitive is called the *competitive ratio* of ONL.

As highlighted in the introduction, we consider two different settings:

Without augmentation: The nodes fit perfectly into the clusters, i.e., $n = k \cdot \ell$. Note that in this setting, due to cluster capacity constraints, a node can never be migrated alone, but it must be *swapped* with another node at a cost of $2 \cdot \alpha$. We also assume that when an algorithm wants to migrate more than two nodes, this has to be done using several swaps, each involving two nodes.

With augmentation: An online algorithm has access to additional space in each cluster. We say that an algorithm is δ-augmented if the size of each cluster is $k' = \delta \cdot k$, whereas the total number of nodes remains $n = k \cdot \ell < k' \cdot \ell$. As usual in competitive analysis, the augmented online algorithm is compared to the optimal offline algorithm with (non-augmented) cluster capacity k.

3.2 A Simple Upper Bound

As a warm-up and to present the model, we start with a straightforward $O(k^2 \cdot \ell^2)$-competitive deterministic algorithm DET. At any time, DET serves a request, adjusts its internal structures (defined below) accordingly and then possibly migrates some nodes. DET operates in phases, and each phase is analyzed separately. The first phase starts with the first request.

In a single phase, DET maintains a helper structure: a complete graph on all $\ell \cdot k$ nodes, with an edge present between each pair of nodes. We say that a communication request is *paid* (by DET) if it occurs between nodes from different clusters, and thus entails a cost for DET.

For each edge between nodes x and y, we define its weight $w(x, y)$ to be the number of paid communication requests between x and y since the beginning of the current phase.

Whenever an edge weight reaches α, it is called *saturated*. If a request causes the corresponding edge to become saturated, DET computes a new placement of nodes (potentially for all of them), so that all saturated edges are inside clusters (there is only one new saturated edge). If this is not possible, node positions are not changed, the current phase ends with the current request, and a new phase begins with the next request. Note that all edge weights are reset to zero at the beginning of a phase.

Theorem 3.1. DET *is* $O(k^2 \cdot \ell^2)$*-competitive.*

Proof. We bound the costs of DET and OPT in a single phase. First, observe that whenever an edge weight reaches α, its endpoint nodes will be collocated until the end of the phase, and therefore its weight is not incremented anymore. Hence the weight of any edge is at most α.

Second, observe that the graph induced by saturated edges always constitutes a forest. Suppose that, at a time t, two nodes x and y, which are not connected by a saturated edge, become connected by a path of saturated edges. From that time onward, DET stores them in a single cluster. Hence, the weight $w(x, y)$ cannot increase at subsequent time points, and (x, y) may not become saturated. The forest property implies that the number of saturated edges is smaller than $k \cdot \ell$.

The two observations above allow us to bound the cost of DET in a single phase. The number of reorganizations is at most the number of saturated edges, i.e., at most $k \cdot \ell$. As the cost associated with a single reorganization is $O(k \cdot \ell \cdot \alpha)$, the total cost of all node migrations in a single phase is at most $O(k^2 \cdot \ell^2 \cdot \alpha)$. The communication cost itself is equal to the total weight of all edges, and by the first observation, it is at most $\binom{k \cdot \ell}{2} \cdot \alpha < k^2 \cdot \ell^2 \cdot \alpha$. Hence, for any phase P (also for the last one), it holds that $\text{DET}(P) = O(k^2 \cdot \ell^2 \cdot \alpha)$.

Now we lower-bound the cost of OPT on any phase P but the last one. If OPT performs a node swap in P, it pays $2 \cdot \alpha$. Otherwise its assignment of nodes to clusters is fixed throughout P. Recall that at the end of P, DET failed to reorganize the nodes. This means that for any static mapping of the nodes to clusters (in particular the one chosen by OPT), there is a saturated inter-cluster edge. The communication cost over such an edge incurred by OPT is at least α (it can be also strictly greater than α as the edge weight only counts the communication requests paid by DET).

Therefore, the DET-to-OPT cost ratio in any phase but the last one is at most $O(k^2 \cdot \ell^2)$ and the cost of DET on the last phase is at most $O(k^2 \cdot \ell^2 \cdot \alpha)$. Hence, $\text{DET}(\sigma) \leq O(k^2 \cdot \ell^2) \cdot \text{OPT}(\sigma) + O(k^2 \cdot \ell^2 \cdot \alpha)$ for any input σ. □

3.3 Algorithm CREP

In this section, we present the main result of this chapter, a *Component-based REPartitioning algorithm (*CREP*)* which achieves a competitive ratio of $O((1+1/\epsilon) \cdot k \log k)$ with augmentation $2 + \epsilon$, for any $\epsilon \geq 1/k$ (i.e., the augmented cluster is of size at least $2k + 1$). For technical

convenience, we assume that $\epsilon \leq 2$. This assumption is without loss of generality: if the augmentation is $2 + \epsilon > 4$, CREP simply uses each cluster only up to capacity $4k$.

CREP maintains a similar graph structure as the simple deterministic $O(k^2 \cdot \ell^2)$-competitive algorithm DET from the previous section, i.e., it keeps counters denoting how many times it paid for a communication between two nodes. Similarly, at any time t, CREP serves the current request, adjusts its internal structures, and then possibly migrates nodes. Unlike DET, however, the execution of CREP is *not* partitioned into global phases: the reset of counters to zero can occur at different times.

3.3.1 Algorithm Definition

We describe the construction of CREP in two stages. The first stage uses an intermediate concept of *communication components*, which are groups of at most k nodes. In the second stage, we show how components are assigned to clusters so that all nodes from any single component are always stored in a single cluster.

Stage 1: Maintaining Components

Roughly speaking, nodes are grouped into components if they communicated a lot recently. At the very beginning, each node is in a singleton component. Once the cumulative communication cost between nodes distributed across s components exceeds $\alpha \cdot (s-1)$, CREP merges them into a single component. If a resulting component size exceeds k, it becomes split into singleton components.

More precisely, the algorithm maintains a time-varying *partition of all nodes into components*. As a helper structure, CREP keeps a complete graph on all $k \cdot \ell$ nodes, with an edge present between each pair of nodes. For each edge between nodes x and y, CREP maintains its weight $w(x, y)$. We say that a communication request is *paid* (by CREP) if it occurs between nodes from different clusters, and thus entails a cost for CREP. If x and y belong to the same component, then $w(x, y) = 0$. Otherwise, $w(x, y)$ is equal to the number of paid communication requests between x and y since the last time when they were placed in *different components* by CREP. It is worth emphasizing that during an execution of CREP, it is possible that $w(x, y) > 0$ even when x and y belong to the same cluster.

For any subset of components $\mathcal{S} = \{C_1, C_2, \ldots, C_{|\mathcal{S}|}\}$ (called *component-set*), by $w(\mathcal{S})$ we denote the total weight of all edges between nodes of $\mathcal{S}$. Note that positive weight edges occur only between different components of $\mathcal{S}$. We call a component-set *trivial* if it contains only one component; $w(\mathcal{S}) = 0$ in this case.

Initially, all components are singleton components and all edge weights are zero. At time t, upon a communication request between a pair of nodes x and y, if x and y lie in the same cluster, the corresponding cost is 0 and CREP does nothing. Otherwise, the cost entailed to CREP is 1, nodes x and y lie in different clusters (and hence also in different components), and the following updates of weights and components are performed.

1. *Weight increment.* Weight $w(x, y)$ is incremented.

2. *Merge actions.* We say that a non-trivial component-set $\mathcal{S} = \{C_1, \ldots, C_{|\mathcal{S}|}\}$ is *mergeable* if $w(\mathcal{S}) \geq (|\mathcal{S}| - 1) \cdot \alpha$. If a mergeable component-set S exists, then all its components are merged into a single one. If multiple mergeable component-sets exist, CREP picks the one with the maximum number of components, breaking ties arbitrarily. Weights of all intra-$\mathcal{S}$ edges are reset to zero, and thus intra-component edge weights are always zero. A mergeable set $\mathcal{S}$ induces a sequence of $|\mathcal{S}| - 1$ *merge actions*: CREP iteratively replaces two arbitrary components from $\mathcal{S}$ by a component being their union (this constitutes a single merge action).

3. *Split action.* If the component resulting from merge action(s) has more than k nodes, it is split into singleton components. Note that weights of edges between these singleton components are all zero as they have been reset by the preceding merge actions.

We say that merge actions are *real* if they are not followed by a split action (at the same time point) and *artificial* otherwise.

Stage 2: Assigning Components to Clusters

At time t, CREP processes a communication request and recomputes components as described in the first stage. Recall that we require that nodes of a single component are always stored in a single cluster. To maintain this property for artificial merge actions, no actual migration is necessary. The property may, however, be violated by real merge actions. Hence, in the following, we assume that in the first stage CREP found a mergeable component set $\mathcal{S} = \{C_1, \ldots, C_{|\mathcal{S}|}\}$ that triggers $|\mathcal{S}| - 1$ merge actions not followed by a split action.

CREP consecutively processes each real merge action by migrating some nodes. We describe this process for a single real merge action involving two components C_x and C_y. As a split action was not executed, $|C_x| + |C_y| \leq k$, where $|C|$ denotes the number of component C nodes. Without loss of generality, $|C_x| \leq |C_y|$.

We may assume that C_x and C_y are in different clusters as otherwise, CREP does nothing. If the cluster containing C_y has $|C_x|$ free space, then C_x is migrated to this cluster. Otherwise, CREP finds a cluster that has at most k nodes, and moves both C_x and C_y there. We call the corresponding actions *component migrations*. By an averaging argument, there always exists a cluster that has at most k nodes, and hence, with $(2+\epsilon)$-augmentation, component migrations are always feasible.

3.3.2 Analysis: Structural Properties

We start with a structural property of components and edge weights. The property states that immediately after CREP merges (and possibly splits) a component-set, no other component-set is mergeable. This property holds independently of the actual placement of components in particular clusters.

Lemma 3.2. *At any time t, after* CREP *performs all its actions, $w(\mathcal{S}) < \alpha \cdot (|\mathcal{S}| - 1)$ for any non-trivial component-set $\mathcal{S}$.*

Proof. We prove the lemma by induction on steps. Clearly, the lemma holds before an input sequence starts as then $w(\mathcal{S}) = 0 \leq \alpha - 1 < \alpha \cdot (|\mathcal{S}| - 1)$ for any non-trivial set $\mathcal{S}$. We assume that it holds at time $t-1$ and show it for time t.

At time t, only a single weight, say $w(x, y)$, may be incremented. If after the increment, Crep does not merge any component, then clearly $w(\mathcal{S}) < \alpha \cdot (|\mathcal{S}| - 1)$ for any non-trivial set $\mathcal{S}$. Otherwise, at time t, Crep merges a component-set $\mathcal{A}$ into a new component $C_{\mathcal{A}}$, and then possibly splits $C_{\mathcal{A}}$ into singleton components. We show that the lemma statement holds then for any non-trivial component-set $\mathcal{S}$. We consider three cases.

1. Component-sets $\mathcal{A}$ and $\mathcal{S}$ do not share any common node. Then, $\mathcal{A}$ and $\mathcal{S}$ consist only of components that were present already right before time t and they are all disjoint. The edge (x, y) involved in communication at time t is contained in $\mathcal{A}$, and hence does not contribute to the weight of $w(\mathcal{S})$. By the inductive assumption, the inequality $w(\mathcal{S}) < \alpha \cdot (|\mathcal{S}| - 1)$ held right before time t. As $w(\mathcal{S})$ is not affected by Crep's actions at step t, the inequality holds also right after time t.

2. Crep does not split $C_{\mathcal{A}}$ and $C_{\mathcal{A}} \in \mathcal{S}$. Let $\mathcal{X} = \mathcal{S} \setminus \{C_{\mathcal{A}}\}$. Let $w(\mathcal{A}, \mathcal{X})$ denote the total weight of all edges with one endpoint in $\mathcal{A}$ and another in $\mathcal{X}$. Recall that Crep always merges a mergeable component-set with the maximum number of components. As Crep merged component-set $\mathcal{A}$ and did not merge (larger) component-set $\mathcal{A} \uplus \mathcal{X}$, $\mathcal{A}$ was mergeable ($w(\mathcal{A}) \geq \alpha \cdot (|\mathcal{A}| - 1)$), while $\mathcal{A} \uplus \mathcal{X}$ was not, i.e., $w(\mathcal{A}) + w(\mathcal{A}, \mathcal{X}) + w(\mathcal{X}) = w(\mathcal{A} \uplus \mathcal{X}) < \alpha \cdot (|\mathcal{A}| + |\mathcal{X}| - 1)$. Therefore, $w(\mathcal{A}, \mathcal{X}) + w(\mathcal{X}) < \alpha \cdot |\mathcal{X}|$ right after weight $w(x, y)$ is incremented at time t. Observe that when component-set $\mathcal{A}$ is merged and all intra-$\mathcal{A}$ edges have their weights reset to zero, neither $w(\mathcal{A}, \mathcal{X})$ nor $w(\mathcal{X})$ is affected. Therefore after Crep merges $\mathcal{A}$ into $C_{\mathcal{A}}$, $w(\mathcal{S}) = w(\mathcal{A}, \mathcal{X}) + w(\mathcal{X}) < \alpha \cdot |\mathcal{X}| = \alpha \cdot (|\mathcal{S}| - 1)$.

3. Crep splits $C_{\mathcal{A}}$ into singleton components $B_1, B_2, \ldots, B_r$ and some of these components belong to set $\mathcal{S}$. This time, we define $\mathcal{X}$ to be the subset of $\mathcal{S}$ not containing these components ($\mathcal{X}$ might be also an empty set). In the same way as in the previous case, we may show that $w(\mathcal{A}, \mathcal{X}) + w(\mathcal{X}) < \alpha \cdot |\mathcal{X}|$ after Crep performs all its operations at time t. Hence, at this time $w(\mathcal{S}) \leq w(\mathcal{A}, \mathcal{X}) + w(\mathcal{X}) < \alpha \cdot |\mathcal{X}| \leq \alpha \cdot (|\mathcal{S}| - 1)$. The last inequality follows as $\mathcal{S}$ has strictly more components than $\mathcal{X}$. □

Since only one request is given at a time, and since all weights and α are integers, Lemma 3.2 immediately implies the following result.

Corollary 3.3. *Fix any time t and consider weights right after they are updated by* Crep*, but before* Crep *performs merge actions. Then, $w(\mathcal{S}) \leq (|\mathcal{S}| - 1) \cdot \alpha$ for any component-set $\mathcal{S}$. In particular, $w(\mathcal{S}) = (|\mathcal{S}| - 1) \cdot \alpha$ for a mergeable component-set $\mathcal{S}$.*

3.3.3 Analysis: Overview

In the remaining part of the analysis, we fix an input sequence σ and consider a set $\textsc{sp}(\sigma)$ of all components that are split by Crep, i.e., components that were created by merge actions, but because of their size, they were immediately split into singleton components. Our goal is to

compare both the cost of OPT and CREP to $\sum_{C \in \mathrm{SP}(\sigma)} |C|$. Below provide the main intuitions for our approach.

For each component $C \in \mathrm{SP}(\sigma)$, we may track the history of how it was created. This history corresponds to a tree $\mathcal{T}(C)$ whose root is C, the leaves are the singleton components containing nodes of C, and the internal nodes correspond to components that are created by merging their children. Note that for any two sets in $\mathrm{SP}(\sigma)$ their trees contain disjoint subsets of components. Hence, for any $C \in \mathrm{SP}(\sigma)$, we want to relate the costs of OPT and CREP due to processing, to the requests related to the components of $\mathcal{T}(C)$. (Some components may not belong to any tree, but the related cost can be universally bounded by a constant independent of input σ.)

In Section 3.3.4, we lower-bound the cost of OPT. Assume first that OPT does not migrate nodes. Fix any component $C \in \mathrm{SP}(\sigma)$. As its size is greater than k, it spans $\Omega(|C|/k)$ clusters in the solution of OPT. Note that Corollary 3.3 lower-bounds the number of requests between siblings in $\mathcal{T}(C)$. Then, for any assignment of nodes of C to the clusters, $\Omega(|C| \cdot \alpha/k)$ requests are between clusters. Additionally, if OPT migrates nodes, then the amount of request-related cost that OPT saves, is dominated by the migration cost. In total, the cost of OPT related to $\mathcal{T}(C)$ is at least $\Omega(|C| \cdot \alpha/k)$.

In Sections 3.3.5 and 3.3.6, we upper-bound the cost of CREP. Its request cost is asymptotically dominated by its migration cost, and hence it is sufficient to bound the latter. If CREP was always able to migrate the smaller component to the cluster holding the larger component, then the total migration cost related to components from $\mathcal{T}(C)$ could be bounded by $\Omega(|C| \cdot \alpha \cdot \log k)$. (This bound is easy to observe when $\mathcal{T}(C)$ is a fully balanced binary tree and all merged components are of equal size.) Unfortunately, CREP may sometimes need to migrate both components. However, if such migrations are expensive, then the distribution of nodes in clusters becomes significantly more even. Consequently, the cost of expensive migrations can be charged to the cost of other migrations, at the expense of an extra $O(1+1/\epsilon)$ factor in the cost. In total, the (amortized) cost of CREP related to $\mathcal{T}(C)$ is at most $\Omega((1+1/\epsilon) \cdot |C| \cdot \alpha \cdot \log k)$.

Finally, comparing bounds on CREP and OPT yields the desired bound on the competitive ratio.

3.3.4 Analysis: Lower Bound on OPT

In our analysis, we conceptually replace any swap of two nodes performed by OPT into two migrations of the corresponding nodes.

For any component C maintained by CREP, let $\tau(C)$ be the time of its creation. A non-singleton component C is created at $\tau(C)$ by the merge of a component-set, henceforth denoted by $\mathcal{S}(C)$. For a singleton component, $\tau(C)$ is the time when the component that previously contained the sole node of C was split; $\tau(C) = 0$ if C existed at the beginning of input σ. We use time 0 as an artificial time point that occurred before an actual input sequence.

For a non-singleton component C, we define $\mathcal{F}(C)$ as the set of the following (node, time) pairs:

$$\mathcal{F}(C) = \biguplus_{B \in \mathcal{S}(C)} \{B\} \times \{\tau(B)+1, \ldots, \tau(C)\}\,.$$

Intuitively, $\mathcal{F}(C)$ tracks the history of all nodes of C from the time (exclusively) they started belonging to some previous component B, until the time (inclusively) they become members of C. Note that for any two components C_1, C_2, sets $\mathcal{F}(C_1)$ and $\mathcal{F}(C_2)$ are disjoint. The union of all $\mathcal{F}(C)$ (over all components C) cover all possible node-time pairs (except for time zero).

For a given component C, we say that a communication request between nodes x and y at time t *is contained* in $F(C)$ if both $(x,t) \in \mathcal{F}(C)$ and $(y,t) \in \mathcal{F}(C)$. Note that only the requests contained in $\mathcal{F}(C)$ could contribute towards the later creation of C by Crep. In fact, by Corollary 3.3, the number of these requests that entailed an actual cost to Crep is exactly $(|\mathcal{S}(C)| - 1) \cdot \alpha$.

We say that a migration of node x performed by Opt at time t *is contained* in $\mathcal{F}(C)$ if $(x,t) \in \mathcal{F}(C)$. For any component C, we define Opt(C) as the cost incurred by Opt due to requests contained in $\mathcal{F}(C)$, plus the cost of Opt migrations contained in $\mathcal{F}(C)$. The total cost of Opt can then be lower-bounded by the sum of Opt(C) over all components C. (The cost of Opt can be larger as $\sum_C$ Opt(C) does not account for communication requests not contained in $\mathcal{F}(C)$ for any component C.)

Lemma 3.4. *Fix any component C and partition $\mathcal{S}(C)$ into a set of $g \geq 2$ disjoint component-sets $\mathcal{S}_1, \mathcal{S}_2, \ldots, \mathcal{S}_g$. The number of communication requests in $\mathcal{F}(C)$ that are between sets $\mathcal{S}_i$ is at least $(g-1) \cdot \alpha$.*

Proof. Let w be the weight measured right after its increment at time $\tau(C)$. Observe that the number of all communication requests from $\mathcal{F}(C)$ that were between sets $\mathcal{S}_i$ and that *were paid* by Crep is $w(\mathcal{S}(C)) - \sum_{i=1}^{g} w(\mathcal{S}_i)$. It suffices to show that this amount is at least $(g-1) \cdot \alpha$. By Corollary 3.3, $w(\mathcal{S}(C)) = (|\mathcal{S}(C)| - 1) \cdot \alpha$ and $w(\mathcal{S}_i) \leq (|\mathcal{S}_i| - 1) \cdot \alpha$. Therefore, $w(\mathcal{S}(C)) - \sum_{i=1}^{g} w(\mathcal{S}_i) \geq (|\mathcal{S}(C)| - 1) \cdot \alpha - \sum_{i=1}^{g} (|\mathcal{S}_i| - 1) \cdot \alpha = (g-1) \cdot \alpha$. □

For any component C maintained by Crep, let Y_C denote the set of clusters containing nodes of C in the solution of Opt after Opt performs its migrations (if any) at time $\tau(C)$. In particular, if $\tau(C) = 0$, then Y_C consists of only one cluster that contained the sole node of C at the beginning of an input sequence.

Lemma 3.5. *For any non-trivial component C, it holds that*

$$\text{Opt}(C) \geq (|Y_C| - 1) \cdot \alpha - \sum_{B \in \mathcal{S}(C)} (|Y_B| - 1) \cdot \alpha \ .$$

Proof. Fix a component $B \in \mathcal{S}(C)$ and any node $x \in B$. Let opt-mig(x) be the number of Opt migrations of node x at times $t \in \{\tau(B)+1, \ldots, \tau(C)\}$. Furthermore, let Y'_x be the set of clusters that contained x at some moment of a time $t \in \{\tau(B)+1, \ldots, \tau(C)\}$ (in the solution of Opt). We extend these notions to components: opt-mig$(B) = \sum_{x \in B}$ opt-mig(x) and $Y'_B = \bigcup_{x \in B} Y'_x$. Observe that $|Y'_B| \leq |Y_B| +$ opt-mig(B). We say that Y'_B are the clusters that were *touched* by component B (in the solution of Opt).

By Corollary 3.3, the number of communication requests between components of $\mathcal{S}(C)$ is $(|\mathcal{S}(C)| - 1) \cdot \alpha$. However, Opt$(C)$ includes the cost only due to these requests that are between different clusters. Hence, to lower-bound Opt(C), we aggregate components of $\mathcal{S}(C)$

into component-sets called *bundles*, so that any two bundles have their nodes always in disjoint clusters. This way, any communication between nodes from different bundles incurs a cost to OPT.

The bundles with the desired property can be created by a natural iterative process. We start from $|\mathcal{S}(C)|$ bundles, each containing just a single component from $\mathcal{S}(C)$. Then, we iterate over all clusters touched by any component of $\mathcal{S}(C)$, i.e., over all clusters from $\bigcup_{B\in\mathcal{S}(C)} Y'_B$. For each such cluster V, let H_V be the set of all components of $\mathcal{S}(C)$ that touched V. We then aggregate all bundles containing any component from H_V into a single bundle.

On the basis of this construction, we may lower-bound the number of bundles. Initially, we have $|\mathcal{S}(C)|$ bundles. When we process a cluster $V \in \bigcup_{B\in\mathcal{S}(C)} Y'_B$, we aggregate at most $|H_V|$ bundles, and thus the total number of bundles drops at most by $|H_V| - 1$. Therefore, the final number of bundles is

$$\begin{aligned}
p &\geq |\mathcal{S}(C)| - \textstyle\sum_{V\in\bigcup_{B\in\mathcal{S}(C)} Y'_B} (|H_V| - 1) \\
&= |\textstyle\bigcup_{B\in\mathcal{S}(C)} Y'_B| + |\mathcal{S}(C)| - \sum_{V\in\bigcup_{B\in\mathcal{S}(C)} Y'_B} |H_V| \\
&= |\textstyle\bigcup_{B\in\mathcal{S}(C)} Y'_B| + |\mathcal{S}(C)| - \sum_{B\in\mathcal{S}(C)} |Y'_B| \\
&= |\textstyle\bigcup_{B\in\mathcal{S}(C)} Y'_B| - \sum_{B\in\mathcal{S}(C)} (|Y'_B| - 1) \\
&\geq |Y_C| - \textstyle\sum_{B\in\mathcal{S}(C)} (|Y'_B| - 1) \\
&\geq |Y_C| - \textstyle\sum_{B\in\mathcal{S}(C)} (|Y_B| - 1) - \sum_{B\in\mathcal{S}(C)} \textsc{opt-mig}(B),
\end{aligned}$$

where the second inequality follows as $Y_C \subseteq \bigcup_{B\in\mathcal{S}(C)} Y'_B$.

By Lemma 3.4, the number of communication requests in $\mathcal{F}(C)$ that are between different bundles is at least $(p-1)\cdot\alpha$, and each of these requests is paid by OPT. Additionally, $\textsc{Opt}(C)$ involves $\sum_{B\in\mathcal{S}(C)} \textsc{opt-mig}(B)$ node migrations in $\mathcal{F}(C)$, and therefore

$$\textsc{Opt}(C) \geq (p-1)\cdot\alpha + \sum_{B\in\mathcal{S}(C)} \textsc{opt-mig}(B)\cdot\alpha \geq (|Y_C| - 1)\cdot\alpha - \sum_{B\in\mathcal{S}(C)} (|Y_B| - 1)\cdot\alpha \ . \qquad \square$$

Lemma 3.6. *For any input σ, $\textsc{Opt}(\sigma) \geq \sum_{C\in\textsc{sp}(\sigma)} |C|/(2k)\cdot\alpha$.*

Proof. Fix any component $C \in \textsc{sp}(\sigma)$. Recall that $\mathcal{T}(C)$ is a tree describing how C was created: the leaves of $\mathcal{T}(C)$ are singleton components containing nodes of C, the root is C itself, and each internal node corresponds to a component created at a specific time, by merging its children.

We now sum $\textsc{Opt}(B)$ over all components B from $\mathcal{T}(C)$, including the root C and the leaves $L(\mathcal{T}(C))$. The lower bound given by Lemma 3.5 sums telescopically, i.e.,

$$\begin{aligned}
\textstyle\sum_{B\in\mathcal{T}(C)} \textsc{Opt}(B) &\geq (|Y_C| - 1)\cdot\alpha - \sum_{B\in L(\mathcal{T}(C))} (|Y_B| - 1)\cdot\alpha \\
&= (|Y_C| - 1)\cdot\alpha,
\end{aligned}$$

where the equality follows as any $B \in L(\mathcal{T}(C))$ is a singleton component, and therefore $|Y_B| = 1$. As C has $|C|$ nodes, it has to span at least $\lceil |C|/k \rceil$ clusters of OPT, and therefore $\sum_{B\in\mathcal{T}(C)} \textsc{Opt}(B) \geq (\lceil |C|/k \rceil - 1)\cdot\alpha \geq |C|/(2k)\cdot\alpha$, where the second inequality follows because $C \in \textsc{sp}(\sigma)$ and thus $|C| > k$.

The proof is concluded by observing that, for any two components C_1 and C_2 from $\textsc{sp}(\sigma)$, the corresponding trees $\mathcal{T}(C_1)$ and $\mathcal{T}(C_2)$ do not share common components, and therefore

$$\textsc{Opt}(\sigma) \geq \sum_{C \in \textsc{sp}(\sigma)} \sum_{B \in \mathcal{T}(C)} \textsc{Opt}(B) \geq \sum_{C \in \textsc{sp}(\sigma)} |C|/(2k) \cdot \alpha \ . \qquad \square$$

3.3.5 Analysis: Upper Bound on CREP

To bound the cost of Crep, we fix any input σ and introduce the following notions. Let $M(\sigma)$ be the sequence of merge actions (real and artificial ones) performed by Crep. For any real merge action $m \in M(\sigma)$, by $\textsc{size}(m)$ we denote the size of the smaller component that was merged. For an artificial merge action, we set $\textsc{size}(m) = 0$.

Let $\textsc{fin}(\sigma)$ be the set of all components that exist when Crep finishes sequence σ. Note that $w(\textsc{fin}(\sigma))$ is the total weight of all edges after processing σ. We split $\textsc{Crep}(\sigma)$ into two parts: the cost of serving requests, $\textsc{Crep}^{\text{req}}(\sigma)$, and the cost of node migrations, $\textsc{Crep}^{\text{mig}}(\sigma)$.

Lemma 3.7. *For any input σ, $\textsc{Crep}^{\text{req}}(\sigma) = |M(\sigma)| \cdot \alpha + w(\textsc{fin}(\sigma))$.*

Proof. The proof follows by induction on all requests of σ. Whenever Crep pays for the communication request, the corresponding edge weight is incremented and both sides increase by 1. At a time when s components are merged, $s-1$ merge actions are executed and, by Corollary 3.3, the sum of all edge weights decreases exactly by $(s-1) \cdot \alpha$. Then, the value of both sides remains unchanged. $\square$

Lemma 3.8. *For any input σ with $(2+\epsilon)$-augmentation, it holds that*

$$\textsc{Crep}^{\text{mig}}(\sigma) \leq (1 + 4/\epsilon) \cdot \alpha \cdot \sum_{m \in M(\sigma)} \textsc{size}(m) \ .$$

Proof. If Crep has more than $2k$ nodes in cluster V_i (for $i \in \{1, \ldots, \ell\}$), then we call the excess $|V_i| - 2k$ the *overflow* of V_i; otherwise, the overflow of V_i is zero. We denote the overflow of cluster V_i measured right after processing sequence σ by $\textsc{ovr}^{\sigma}(V_i)$. It is sufficient to show the following relation for any sequence σ:

$$\textsc{Crep}^{\text{mig}}(\sigma) \ + \ \sum_{j=1}^{\ell} (4/\epsilon) \cdot \alpha \cdot \textsc{ovr}^{\sigma}(V_j) \ \leq \ (1 + 4/\epsilon) \cdot \alpha \cdot \sum_{m \in M(\sigma)} \textsc{size}(m). \tag{3.1}$$

As the second summand of (3.1) is always non-negative, (3.1) will imply the lemma. In other words, the lemma will be shown using amortized analysis, where the amount $\sum_{j=1}^{\ell} (4/\epsilon) \cdot \alpha \cdot \textsc{ovr}^{\sigma}(V_j)$ serves as a potential function.

The proof of (3.1) follows by induction on all requests in σ. Clearly, (3.1) holds trivially at the beginning, as there are no overflows, and thus both sides of (3.1) are zero. Assume that (3.1) holds for a sequence σ and we show it for sequence $\sigma \cup \{r\}$, where r is some request.

We may focus on a request r which triggers the migration of component(s), as otherwise (3.1) holds trivially. Such a migration is triggered by a real merge action m of two components C_x

and C_y. We assume that $|C_x| \leq |C_y|$, and hence $\textsc{size}(m) = |C_x|$. Note that $|C_x| + |C_y| \leq k$, as otherwise the resulting component would be split and no migration would be performed.

Let V_x and V_y denote the cluster that held components C_x and C_y, respectively, and V_z be the destination cluster for C_x and C_y (it is possible that $V_z = V_y$). For any cluster V, we denote the change in its overflow by $\Delta\textsc{ovr}(V) = \textsc{ovr}^{\sigma\cup\{r\}}(V) - \textsc{ovr}^{\sigma}(V)$. It suffices to show that the change of the left-hand side of (3.1) is at most the increase of its right-hand side, i.e.,

$$\textsc{Crep}^{\text{mig}}(r) \; + \sum_{V\in\{V_x,V_y,V_z\}} (4/\epsilon)\cdot\alpha\cdot\Delta\textsc{ovr}(V) \;\leq\; (1+4/\epsilon)\cdot|C_x|\cdot\alpha. \tag{3.2}$$

For the proof, we consider three cases.

1. V_y had at least $|C_x|$ empty slots. In this case, Crep simply migrates C_x to V_y paying $|C_x|\cdot\alpha$. Then, $\Delta\textsc{ovr}(V_x) \leq 0$, $\Delta\textsc{ovr}(V_y) \leq |C_x|$, $V_z = V_y$, and thus (3.2) follows.

2. V_y had less than $|C_x|$ empty slots and $|C_y| \leq (2/\epsilon)\cdot|C_x|$. Crep migrates both C_x and C_y to component V_z and the incurred cost is

$$\textsc{Crep}^{\text{mig}}(r) = (|C_x| + |C_y|)\cdot\alpha \leq (1+2/\epsilon)\cdot|C_x|\cdot\alpha < (1+4/\epsilon)\cdot|C_x|\cdot\alpha \;.$$

   It remains to show that the second summand of (3.2) is at most 0. Clearly, $\Delta\textsc{ovr}(V_x) \leq 0$ and $\Delta\textsc{ovr}(V_y) \leq 0$. Furthermore, the number of nodes in V_z was at most k before the migration by the definition of Crep, and thus is at most $k + |C_x| + |C_y| \leq 2k$ after the migration. This implies that $\Delta\textsc{ovr}(V_z) = 0 - 0 = 0$.

3. V_y had less than $|C_x|$ empty slots and $|C_y| > (2/\epsilon)\cdot|C_x|$. As in the previous case, Crep migrates C_x and C_y to component V_z, paying $\textsc{Crep}^{\text{mig}}(r) = (|C_x| + |C_y|)\cdot\alpha < 2\cdot|C_y|\cdot\alpha$. This time, $\textsc{Crep}^{\text{mig}}(r)$ can be much larger than the right-hand side of (3.2), and thus we resort to showing that the second summand of (3.2) is at most $-2\cdot|C_y|\cdot\alpha$.

   As in the previous case, $\Delta\textsc{ovr}(V_x) \leq 0$ and $\Delta\textsc{ovr}(V_z) = 0$. Observe that $|C_x| < (\epsilon/2)\cdot|C_y| \leq (\epsilon/2)\cdot k$. As the migration of C_x to V_y was not possible, the initial number of nodes in V_y was greater than $(2+\epsilon)\cdot k - |C_x| \geq (2+\epsilon/2)\cdot k$, i.e., $\textsc{ovr}^{\sigma}(V_y) \geq (\epsilon/2)\cdot k \geq (\epsilon/2)\cdot|C_y|$. As component C_y was migrated out of V_y, the number of overflow nodes in V_y changes by

$$\Delta\textsc{ovr}(V_y) = -\min\{\,\textsc{ovr}^{\sigma}(V_y),\,|C_y|\,\} \leq -(\epsilon/2)\cdot|C_y|.$$

   In the inequality above, we used $\epsilon \leq 2$. Therefore, the second summand of (3.2) is at most $(4/\epsilon)\cdot\alpha\cdot\Delta\textsc{ovr}(V_y) \leq -(4/\epsilon)\cdot\alpha\cdot(\epsilon/2)\cdot|C_y| = -2\cdot|C_y|\cdot\alpha$ as desired. □

3.3.6 Analysis: Competitive Ratio

In the previous two subsections, we related $\textsc{Opt}(\sigma)$ to the total size of components that are split by Crep (cf. Lemma 3.6) and $\textsc{Crep}(\sigma)$ to $\sum_{m\in M(\sigma)}\textsc{size}(m)$, where the latter amount is related to the merging actions performed by Crep (cf. Lemma 3.8). Now we link these two amounts. Note that each split action corresponds to preceding merge actions that led to the creation of the split component.

Lemma 3.9. *For any input σ, it holds that*

$$\sum_{m \in M(\sigma)} \textsc{size}(m) \leq \sum_{C \in \textsc{sp}(\sigma)} |C| \cdot \log k + \sum_{C \in \textsc{fin}(\sigma)} |C| \cdot \log |C|,$$

where all logarithms are binary.

Proof. We prove the lemma by induction on all requests of σ. At the very beginning, both sides of the lemma inequality are zero, and hence the induction basis holds trivially. We assume that the lemma inequality is preserved for a sequence σ and we show it for sequence $\sigma \cup \{r\}$, where r is an arbitrary request. We may assume that r triggers some merge actions, otherwise the claim follows trivially.

First, assume r triggered a sequence of real merge actions. We show that the lemma inequality is preserved after processing each merge action. Consider a merge action and let C_x and C_y be the components that are merged, with sizes $p = |C_x|$ and $q = |C_y|$, where $p \leq q$ without loss of generality. Due to the merge action, the right-hand side of the lemma inequality increases by

$$\begin{aligned}(p+q) \cdot \log(p+q) - p \cdot \log p - q \cdot \log q & \\ &= p \cdot (\log(p+q) - \log p) + q \cdot (\log(p+q) - \log q) \\ &\geq p \cdot \log(p+q)/p \\ &\geq p \cdot \log 2 = p.\end{aligned}$$

As the left-hand side of the inequality changes exactly by p, the inductive hypothesis holds.

Second, assume r triggered a sequence of artificial merge actions (i.e., followed by a split action) and let $C_1, C_2, \ldots, C_g$ denote components that were merged to create a component C that was immediately split. Then, the right hand side of the lemma inequality changes by $-\sum_{i=1}^{g} |C_i| \cdot \log |C_i| + |C| \cdot \log k \geq -\sum_{i=1}^{g} |C_i| \cdot \log k + |C| \cdot \log k = 0$. As the left-hand side of the lemma inequality is unaffected by artificial merge actions, the inductive hypothesis follows also in this case. □

Theorem 3.10. *With augmentation at least $2 + \epsilon$,* Crep *is $O((1 + 1/\epsilon) \cdot k \cdot \log k)$-competitive.*

Proof. Fix any input sequence σ. By Lemmas 3.7 and 3.8,

$$\begin{aligned}\textsc{Crep}(\sigma) &= \textsc{Crep}^{\text{mig}}(\sigma) + \textsc{Crep}^{\text{req}}(\sigma) \\ &\leq (1 + 4/\epsilon) \cdot \alpha \cdot \textstyle\sum_{m \in M(\sigma)} \textsc{size}(m) + |M(\sigma)| \cdot \alpha + w(\textsc{fin}(\sigma)).\end{aligned}$$

Regarding a bound for $|M(\sigma)|$, we observe the following. First, if Crep executes artificial merge actions, then they are immediately followed by a split action of the resulting component C. The number of artificial merge actions is clearly at most $|C| - 1 \leq |C|$, and thus the total number of all artificial actions in $M(\sigma)$ is at most $\sum_{C \in \textsc{sp}(\sigma)} |C|$. Second, if Crep executes a real merge action m, then $\textsc{size}(m) \geq 1$. Combining these two bounds yields $|M(\sigma)| \leq \sum_{m \in M(\sigma)} \textsc{size}(m) + \sum_{C \in \textsc{sp}(\sigma)} |C|$. We use this inequality and later apply Lemma 3.9 to bound

$\sum_{m \in M(\sigma)} \textsc{size}(m)$ obtaining

$$\begin{aligned}
&\textsc{Crep}(\sigma) - w(\textsc{fin}(\sigma)) \\
&\quad \leq (1 + 4/\epsilon) \cdot \alpha \cdot \textstyle\sum_{m \in M(\sigma)} \textsc{size}(m) + |M(\sigma)| \cdot \alpha \\
&\quad \leq (2 + 4/\epsilon) \cdot \alpha \cdot \textstyle\sum_{m \in M(\sigma)} \textsc{size}(m) + \alpha \cdot \sum_{C \in \textsc{sp}(\sigma)} |C| \\
&\quad \leq (2 + 4/\epsilon) \cdot \alpha \cdot \left(\textstyle\sum_{C \in \textsc{sp}(\sigma)} |C| \cdot \log k + \sum_{C \in \textsc{fin}(\sigma)} |C| \cdot \log |C| \right) + \alpha \cdot \sum_{C \in \textsc{sp}(\sigma)} |C| \\
&\quad \leq (3 + 4/\epsilon) \cdot \alpha \cdot \textstyle\sum_{C \in \textsc{sp}(\sigma)} |C| \cdot \log k + (2 + 4/\epsilon) \cdot \alpha \cdot \sum_{C \in \textsc{fin}(\sigma)} |C| \cdot \log |C|.
\end{aligned}$$

By Lemma 3.6, $\sum_{C \in \textsc{sp}(\sigma)} |C| \cdot \alpha \leq 2k \cdot \textsc{Opt}(\sigma)$. This yields

$$\textsc{Crep}(\sigma) \leq O(1 + 1/\epsilon) \cdot k \cdot \log k \cdot \textsc{Opt}(\sigma) + \beta,$$

where

$$\beta = O(1 + 1/\epsilon) \cdot \alpha \cdot \sum_{C \in \textsc{fin}(\sigma)} |C| \cdot \log |C| + w(\textsc{fin}(\sigma)).$$

To bound β, observe that the component-set $\textsc{fin}(\sigma)$ contains at most $k \cdot \ell$ components, and hence by Lemma 3.2, $w(\textsc{fin}(\sigma)) < k \cdot \ell \cdot \alpha$. Furthermore, the maximum of $\sum_{C \in \textsc{fin}(\sigma)} |C| \cdot \log |C|$ is achieved when all nodes in a specific cluster constitute a single component. Thus, $\sum_{C \in \textsc{fin}(\sigma)} |C| \cdot \log |C| \leq \ell \cdot ((2+\epsilon) \cdot k) \cdot \log((2+\epsilon) \cdot k) = O(\ell \cdot k \cdot \log k)$. In total, $\beta = O((1+1/\epsilon) \cdot \alpha \cdot \ell \cdot k \cdot \log k)$, i.e., it can be upper-bounded by a constant independent of input sequence σ, which concludes the proof. □

3.4 Online Rematching

Let us now consider the special case where clusters are of size two ($k = 2$, arbitrary ℓ). This can be viewed as an online maximal (re)matching problem: clusters of size two contain ("match") exactly one pair of nodes, and maximizing pairwise communication within each cluster is equivalent to minimizing inter-cluster communication.

3.4.1 Greedy Algorithm

We define a natural greedy online algorithm Greedy, parameterized by a real positive number λ. Similarly to our other algorithms, Greedy maintains an edge weight for each pair of nodes. The weights of all edges are initially zero. Weights of intra-cluster edges are always zero and weights of inter-cluster edges are related to the number of paid communication requests between edge endpoints.

When facing an inter-cluster request between nodes x and y, Greedy increments the weight $w(e)$, where $e = (x, y)$. Let x' and y' be the nodes collocated with x and y, respectively. If after the weight increase, it holds that $w(x, y) + w(x', y') \geq \lambda \cdot \alpha$, Greedy performs a swap: it places x and y in one cluster and x' and y' in another; afterward, it resets the weights of edges (x, y) and (x', y') to 0. Finally, Greedy pays for the request between x and y. Note that if the request triggered a migration, then Greedy does not pay its communication cost.

3.4.2 Analysis

We use E to denote the set of all edges. Let M^{GR} (M^{OPT}) denote the set of all edges $e = (u, v)$, such that u and v are collocated by Greedy (Opt). Note that M^{GR} and M^{OPT} are perfect matchings on the set of all nodes.

To estimate the total cost of Greedy, we use amortized analysis with an appropriately defined potential function. First, we associate the following edge-potential with any edge e:

$$\Phi(e) = \begin{cases} 0 & \text{if } e \in M^{\mathrm{GR}}, \\ -w(e) & \text{if } e \in M^{\mathrm{OPT}} \setminus M^{\mathrm{GR}}, \\ f \cdot w(e) & \text{if } e \notin M^{\mathrm{OPT}} \text{ and } e \notin M^{\mathrm{GR}}, \end{cases}$$

where $f \geq 0$ is a constant that will be defined later.

The union of M^{GR} and M^{OPT} constitutes a set of alternating cycles: an alternating cycle of length $2j$ (for some $j \geq 1$) consists of $2j$ nodes, j edges from M^{GR} and j edges from M^{OPT}, interleaved. The case $j = 1$ is degenerate: such a cycle consists of a single edge from $M^{\mathrm{GR}} \cap M^{\mathrm{OPT}}$, but we still count it as a cycle of length 2. It turns out that the number of these alternating cycles is a good measure of similarity between matchings of Greedy and Opt (when these matchings are equal, the number of cycles is maximized). We define the cycle-potential as

$$\Psi = -g \cdot K \cdot \alpha,$$

where K is the number of all alternating cycles and $g \geq 0$ is a constant that will be defined later.

To simplify the analysis, we slightly modify the way weights are increased by Greedy. The modification is applied only when the weight increment triggers a node migration. Recall that this happens when there is an inter-cluster request between nodes x and y. The corresponding weight $w(x, y)$ is then increased by 1. After the increase, it holds that $w(x, y) + w(x', y') \geq \lambda \cdot \alpha$. (Nodes x' and y' are those collocated with x and y, respectively.) Instead, we increase $w(x, y)$ possibly by a smaller amount, so that $w(x, y) + w(x', y')$ becomes *equal* to $\lambda \cdot \alpha$. This modification allows for a more streamlined analysis and is local: before and after the modification, Greedy performs a migration and right after that, it resets weight $w(x, y)$ to zero.

We split the processing of a communication request (x, y) into three stages. In the first stage, Opt performs an arbitrary number of migrations. In the second stage, the weight $w(x, y)$ is increased accordingly and both Opt and Greedy serve the request. It is possible that the weight increase triggers a node swap of Greedy, in which case its serving cost is zero. Finally, in the third stage, Greedy may perform a node swap.

We show that for an appropriate choice of λ, f, and g, for all three stages described above the following inequality holds:

$$\Delta\textsc{Greedy} + \Delta\Psi + \textstyle\sum_{e \in E} \Delta\Phi(e) \leq 7 \cdot \Delta\textsc{Opt}. \tag{3.3}$$

Here, ΔGreedy and ΔOpt denote the increases of Greedy's and Opt's cost, respectively. $\Delta\Psi$ and $\Delta\Phi(e)$ are the changes of the potentials Ψ and $\Phi(e)$. The 7-competitiveness then immediately follows from summing (3.3) and bounding the initial and final values of the potentials.

Lemma 3.11. *If* $2 \cdot (f+1) \cdot \lambda + g \leq 14$*, then* (3.3) *holds for the first stage.*

Proof. We consider any node swap performed by Opt. Clearly, for such an event $\Delta\textsc{Greedy} = 0$ and $\Delta\textsc{Opt} = 2 \cdot \alpha$. The number of cycles decreases at most by one, and thus $\Delta\Psi \leq g \cdot \alpha$.

We now upper-bound the change in the edge-potentials. Let e_1^{old} and e_2^{old} be the edges that were removed from M^{OPT} by the swap and let e_1^{new} and e_2^{new} be the edges added to M^{OPT}. For any $i \in \{1,2\}$, $\Delta\Phi(e_i^{\text{new}}) \leq 0$ as the initial value of $\Phi(e_i^{\text{new}})$ is at least 0 and the final value of $\Phi(e_i^{\text{new}})$ is at most 0. Similarly, $\Delta\Phi(e_i^{\text{old}}) \leq (f+1) \cdot w(e_i^{\text{old}})$ as the initial value of $\Phi(e_i^{\text{old}})$ is at least $-w(e_i^{\text{old}})$ and the final value of $\Phi(e_i^{\text{old}})$ is at most $f \cdot w(e_i^{\text{old}})$.

Summing up, $\sum_{e \in E} \Delta\Phi \leq (f+1) \cdot (w(e_1^{\text{old}}) + w(e_2^{\text{old}})) \leq 2 \cdot (f+1) \cdot \lambda \cdot \alpha$ as the weight of each edge is at most $\lambda \cdot \alpha$. By combining the bounds above and using the lemma assumption, we obtain $\Delta\textsc{Greedy} + \sum_{e \in E} \Delta\Phi(e) + \Delta\Psi \leq 0 + 2 \cdot (f+1) \cdot \lambda \cdot \alpha + g \cdot \alpha \leq 14 \cdot \alpha = 7 \cdot \Delta\textsc{Opt}$. □

Lemma 3.12. *If* $f \leq 6$*, then* (3.3) *holds for the second stage.*

Proof. In this stage, both Greedy and Opt serve a communication request between nodes x and y. Let $e_c = (x, y)$. As neither Greedy nor Opt migrates any nodes in this stage, the structure of alternating cycles remains unchanged, i.e., $\Delta\Psi = 0$. Furthermore, only edge e_c may change its weight, and therefore, among all edges, only the edge-potential of e_c may change. We consider two cases.

1. If $e_c \in M^{\text{GR}}$, then $\Delta\textsc{Greedy} = 0$ and $\Delta\textsc{Opt} \geq 0$. As $w(e_c)$ is unchanged, $\Delta\Phi(e_c) = 0$, and therefore $\Delta\textsc{Greedy} + \Delta\Phi(e_c) = 0 \leq \Delta\textsc{Opt}$.

2. If $e_c \notin M^{\text{GR}}$, then let $\Delta w(e_c) \leq 1$ denote the increase of the weight of edge e_c. Note that $\Delta\textsc{Greedy} \leq \Delta w(e_c)$: either no migration is triggered and $\Delta\textsc{Greedy} = \Delta w(e_c) = 1$ or a migration is triggered and then Greedy does not pay for the request.

   If $e_c \in M^{\text{OPT}}$, then $\Delta\textsc{Opt} = 0$ and $\Delta\Phi(e_c) = -w(e_c)$. Thus, $\Delta\textsc{Greedy} + \Delta\Phi(e_c) \leq 0 = \Delta\textsc{Opt}$. Otherwise, $e_c \notin M^{\text{OPT}}$, in which case $\Delta\textsc{Opt} = 1$. Furthermore, $\Delta\Phi(e_c) = f \cdot \Delta w(e_c)$, and thus $\Delta\textsc{Greedy} + \Delta\Phi(e_c) = (f+1) \cdot \Delta w(e_c) \leq f + 1 = (f+1) \cdot \Delta\textsc{Opt}$.

Therefore, in the second stage, $\Delta\textsc{Greedy} + \Delta\Psi + \sum_{e \in E} \Delta\Phi(e) \leq (f+1) \cdot \Delta\textsc{Opt}$, which implies (3.3) as we assumed $f \leq 6$. □

Lemma 3.13. *If* $2 + \lambda \leq g \leq f \cdot \lambda - 2$*, then* (3.3) *holds for the third stage.*

Proof. In the third stage (if it is present), Greedy performs a swap. Clearly, for such an event $\Delta\textsc{Greedy} = 2 \cdot \alpha$ and $\Delta\textsc{Opt} = 0$.

There are four edges involved in a swap: let (x, x') and (y, y') be the edges that were in M^{GR} before the swap and let (x, y) and (y, y') be the new edges in M^{GR} after the swap. Note that $w(x, x') = w(y, y') = 0$ before and after the swap. By the definition of Greedy and our modification of weight updates, $w(x, y) + w(x', y') = \lambda \cdot \alpha$ before the swap, and after the swap, these weights are reset to zero.

For any edge e, let $w^{\text{S}}(e)$ and $\Phi^{\text{S}}(e)$ denote the weight and the edge-potential of e right before the swap. By the bounds above, $\Delta\textsc{Greedy} + \sum_{e \in E} \Delta\Phi(e) + \Delta\Psi = 2 \cdot \alpha - \Phi^{\text{S}}(x, y) -$

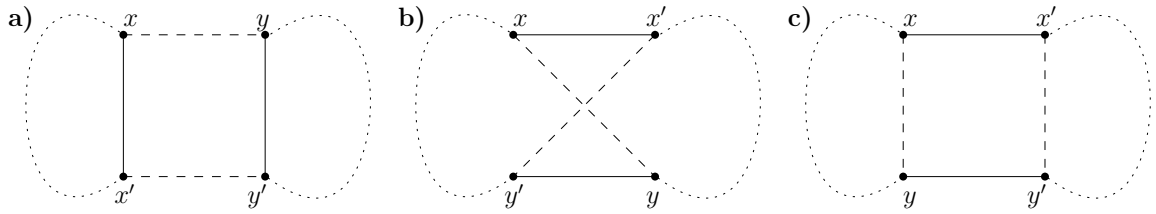

Figure 3.1: Three cases in the analysis of the third stage (a swap performed by GREEDY). Solid edges denote edges that were removed from M^{GR} because of the swap, dashed ones denote the ones that were added to M^{GR}. Dotted paths denote the remaining parts of the involved alternating cycle(s).

$\Phi^{\mathrm{S}}(x', y') + \Delta\Psi$, and hence to show (3.3) it suffices to show that the latter amount is at most $7 \cdot \Delta\mathrm{OPT} = 0$. We consider three cases.

1. Assume that edges (x, x') and (y, y') were in different alternating cycles before the swap, see Figure 3.1a. Then the number of alternating cycles decreases by one, and hence $\Delta\Psi = g \cdot \alpha$. Let C be the cycle that contained edge (x, x'). Node x is adjacent to an edge from C that belongs to M^{OPT}. (It is possible that this edge is (x, x'); this occurs in the degenerate case when C is of length 2.) As M^{OPT} is a matching, $(x, y) \notin M^{\mathrm{OPT}}$. Analogously, $(x', y') \notin M^{\mathrm{OPT}}$. Therefore, $\Phi^{\mathrm{S}}(x, y) + \Phi^{\mathrm{S}}(x', y') = f \cdot w(x, y) + f \cdot w(x', y') = f \cdot \lambda \cdot \alpha$. Using the lemma assumption, $\Delta\mathrm{GREEDY} + \sum_{e \in E} \Delta\Phi(e) + \Delta\Psi = (2 + g - f \cdot \lambda) \cdot \alpha \leq 0$.

2. Assume that edges (x, x') and (y, y') belonged to the same cycle and it contained the nodes in the order $x, x', \ldots, y, y', \ldots$, see Figure 3.1b. In this case, it holds that $\Delta\Psi = 0$, since the number of alternating cycles is unaffected by the swap. By similar reasoning as in the previous case, neither (x, y) nor (x', y') belong to M^{OPT}, and thus again, $\Phi^{\mathrm{S}}(x, y) + \Phi^{\mathrm{S}}(x', y') = f \cdot w(x, y) + f \cdot w(x', y') = f \cdot \lambda \cdot \alpha$. In this case, $\Delta\mathrm{GREEDY} + \sum_{e \in E} \Delta\Phi(e) + \Delta\Psi = (2 - f \cdot \lambda) \cdot \alpha \leq (2 + g - f \cdot \lambda) \cdot \alpha \leq 0$.

3. Assume that edges (x, x') and (y, y') belonged to the same cycle and it contained the nodes in the order $x, x', \ldots, y', y, \ldots$, see Figure 3.1c. When the swap is performed, the number of alternating cycles decreases, and thus $\Delta\Psi = -g \cdot \alpha$. Unlike the previous cases, here it is possible that (x, y) belonged to M^{OPT}. (This happens when x and y were adjacent on the alternating cycle.) Similarly, it is possible that $(x', y') \in M^{\mathrm{OPT}}$. But even in such a case, we may lower-bound the initial values of the corresponding edge-potentials: $\Phi^{\mathrm{S}}(x, y) + \Phi^{\mathrm{S}}(x', y') \geq -w^{\mathrm{S}}(x, y) - w^{\mathrm{S}}(x', y') = -\lambda \cdot \alpha$. Using the lemma assumption, $\Delta\mathrm{GREEDY} + \sum_{e \in E} \Delta\Phi(e) + \Delta\Psi \leq (2 - g + \lambda) \cdot \alpha \leq 0$. □

Theorem 3.14. *For $\lambda = 4/5$,* GREEDY *is 7-competitive.*

Proof. We choose $f = 6$ and $g = 14/5$. The chosen values of λ, f, and g satisfy the conditions of Lemmas 3.11, 3.12, and 3.13. Summing these inequalities over all stages occurring while serving an input sequence σ yields

$$\mathrm{GREEDY}(\sigma) + (\Psi_{\mathrm{final}} - \Psi_{\mathrm{initial}}) + \sum_{e \in E} (\Phi_{\mathrm{final}}(e) - \Phi_{\mathrm{initial}}(e)) \leq 7 \cdot \mathrm{OPT}(\sigma),$$

where Ψ_{final} and $\Phi_{\text{final}}(e)$ denote the final values of the potentials and Ψ_{initial} and $\Phi_{\text{initial}}(e)$ their initial values. We observe that all the potentials occurring in the inequality above are lower-bounded and upper-bounded by values that are independent of the input sequence σ. That is, $\Psi_{\text{final}} - \Psi_{\text{initial}} \geq -g \cdot \ell \cdot \alpha$ (as the number of alternating cycles is at most ℓ) and $\Phi_{\text{final}}(e) - \Phi_{\text{initial}}(e) \geq -(f+1) \cdot w(e) \geq -(f+1) \cdot \lambda \cdot \alpha$ (as all edge weights are always at most $\lambda \cdot \alpha$). The number of edges is exactly $\binom{2\cdot\ell}{2}$, and therefore

$$\begin{aligned}\textsc{Greedy}(\sigma) &\leq 7 \cdot \textsc{Opt}(\sigma) + g \cdot \ell \cdot \alpha + \tbinom{2\cdot\ell}{2} \cdot (f+1) \cdot \lambda \cdot \alpha \\ &\leq 7 \cdot \textsc{Opt}(\sigma) + O(\ell^2 \cdot \alpha).\end{aligned}$$

This concludes the proof. □

3.5 Lower Bounds

In order to shed light on the optimality of the presented online algorithm, we next investigate lower bounds on the competitive ratio achievable by any (deterministic) online algorithm. We start by showing a reduction of the BRP problem to online caching, which will imply that already for two clusters the competitive ratio of the problem is at least $k-1$. We strengthen this bound, providing a lower bound of k that holds for any amount of augmentation, as long as the augmentation does not allow to put all nodes in a single cluster. The proof uses the averaging argument. We refine this approach for a special case of online rematching ($k = 2$ without augmentation), for which we present a lower bound of 3.

3.5.1 Lower Bound by Reduction to Online Caching

Theorem 3.15. *Fix any k. If there exists a γ-competitive deterministic algorithm B for BRP for two clusters, each of size k, then there exists a γ-competitive deterministic algorithm P for the caching problem with cache size $k-1$ and where the number of different pages is k.*

Proof. The pages are denoted by $p_1, p_2, \ldots, p_k$. Without loss of generality, we assume that the initial cache is equal to $p_1, p_2, \ldots, p_{k-1}$. We fix any input sequence $\sigma^P = (\sigma_1^P, \sigma_2^P, \sigma_3^P, \ldots)$ for the caching problem, where σ_t^P denotes the t-th accessed page. We show how to construct, an online algorithm P for the caching problem that proceeds in the following online manner. The algorithm internally runs the algorithm B, starting on the initial assignment of nodes to clusters that will be defined below. For a requested page σ_t^P, it creates a subsequence of communication requests for the BRP problem, runs B on them, and serves σ_t^P on the basis of B's responses.

We use the following $2k$ nodes for the BRP problem: caching nodes $p_1, p_2, \ldots, p_k$, auxiliary nodes $a_1, a_2, \ldots, a_{k-1}$, and a special node s. We say that the node clustering is *well-aligned* if one cluster contains the node s and $k-1$ caching nodes, and the other cluster contains one caching node and all auxiliary nodes. There is a natural bijection between possible cache contents and well-aligned configurations: the cache consists of the $k-1$ caching nodes that are in the same cluster as node s. (Without loss of generality, we may assume that the cache of any caching algorithm is always full, i.e., consists of $k-1$ pages.) If the configuration c of a BRP algorithm is well-aligned, $\textsc{cache}(c)$ denotes the corresponding cache contents. The

initial configuration for the BRP problem is the well-aligned configuration corresponding to the initial cache (pages $p_1, p_2, \ldots, p_{k-1}$ in the cache).

For any caching node p, let $\textsc{comm}(p)$ be a subsequence of communication requests for the BRP problem, consisting of the request (p, s), followed by $\binom{k-1}{2}$ requests to all pairs of auxiliary nodes. Given an input sequence σ^P for online caching, we construct the input sequence σ^B for the BRP problem in the following way: For a request σ_t^P, we repeat a subsequence $\textsc{comm}(\sigma_t^P)$ till the node clustering maintained by B becomes well-aligned and σ_t^P becomes collocated with s. Note that B must eventually achieve such a node configuration: otherwise its cost would be arbitrarily large while a sequence of repeated $\textsc{comm}(\sigma_t^P)$ subsequences can be served at a constant cost—the competitive ratio of B would then be unbounded. We denote the resulting sequence of $\textsc{comm}(\sigma_t^P)$ subsequences by $\textsc{comm}_t(\sigma_t^P)$.

To construct the response to the caching request σ_t^P, the algorithm P runs B on $\textsc{comm}_t(\sigma_t^P)$. Right after processing $\textsc{comm}_t(\sigma_t^P)$, the node configuration c of B is well-aligned and σ_t^P is collocated with s. Hence, P may change its cache configuration to $\textsc{cache}(c)$: such a response is feasible, because since σ_t^P is collocated with s, it is included by P in the cache. Furthermore, we may relate the cost of P to the cost of B: If P modifies the cache contents, the corresponding cost is 1, as exactly one page has to be fetched. Such a change occurs only if B changed the clustering (at a cost of at least $2 \cdot \alpha$). Therefore, $2 \cdot \alpha \cdot P(\sigma_t^P) \leq B(\textsc{comm}_t(\sigma_t^P))$, which, summed over all requests from sequence σ^P, yields $2 \cdot \alpha \cdot P(\sigma^P) \leq B(\sigma^B)$.

Now we show that there exists an (offline) solution $\textsc{Off}$ to σ^B, whose cost is exactly $2 \cdot \alpha \cdot \textsc{Opt}(\sigma^P)$. Recall that, for a caching request σ_t^P, σ^B contains the corresponding sequence $\textsc{comm}_t(\sigma_t^P)$. Before serving the first request of $\textsc{comm}_t(\sigma_t^P)$, $\textsc{Off}$ changes its state to a well-aligned configuration corresponding to the cache of $\textsc{Opt}$ right after serving caching request σ_t^P. This ensures that the subsequence $\textsc{comm}_t(\sigma_t^P)$ is free for $\textsc{Off}$. Furthermore, the cost of node migration of $\textsc{Off}$ is $2 \cdot \alpha$ (two caching nodes are swapped) if $\textsc{Opt}$ performs a fetch, and 0 if $\textsc{Opt}$ does not change its cache contents. Therefore, $\textsc{Off}(\textsc{comm}_t(\sigma_t^P)) = 2 \cdot \alpha \cdot \textsc{Opt}(\sigma_t^P)$, which summed over the entire sequence σ^P yields $\textsc{Off}(\sigma^B) = 2 \cdot \alpha \cdot \textsc{Opt}(\sigma^P)$.

As B is ρ-competitive for the BRP problem, there exists a constant β, such that for any sequence σ^P and the corresponding sequence σ^B, it holds that $B(\sigma^B) \leq \gamma \cdot \textsc{Off}(\sigma^B) + \beta$. Combining this inequality with the inequalities between P and B and between $\textsc{Off}$ and $\textsc{Opt}$ yields

$$2 \cdot \alpha \cdot P(\sigma^P) \leq B(\sigma^B) \leq \gamma \cdot \textsc{Off}(\sigma^B) + \beta = \gamma \cdot 2 \cdot \alpha \cdot \textsc{Opt}(\sigma^P) + \beta,$$

and therefore P is γ-competitive. □

As any deterministic algorithm for the caching problem with cache size $k-1$ has a competitive ratio of at least $k-1$ [ST85], we obtain the following result.

Corollary 3.16. *The competitive ratio of the BRP problem on two clusters is at least $k-1$.*

3.5.2 Additional Lower Bounds

Theorem 3.17. *No δ-augmented deterministic online algorithm* $\textsc{Onl}$ *can achieve a competitive ratio smaller than k, as long as $\delta < \ell$.*

Proof. In our construction, all nodes are numbered from v_0 to v_{n-1}. All presented requests are edges in a ring graph on these nodes with edge e_i defined as $(v_i, v_{(i+1) \mod n})$ for $i = 0, \ldots, n-1$. At any time, the adversary gives a communication request between an arbitrary pair of nodes not collocated by ONL. As $\delta < \ell$, ONL cannot fit the entire ring in a single cluster, and hence such a pair always exists. Such a request entails a cost of at least 1 for ONL. This way, we may define an input sequence σ of arbitrary length, such that $\text{ONL}(\sigma) \geq |\sigma|$.

Now we present k offline algorithms $\text{OFF}_1, \text{OFF}_2, \ldots, \text{OFF}_k$, such that, neglecting an initial node reorganization they perform before the input sequence starts, the sum of their total costs on σ is exactly $|\sigma|$. Toward this end, for any $j \in \{0, \ldots, k-1\}$, we define a set $\text{CUT}(j) = \{e_j, e_{j+k}, e_{j+2k}, \ldots, e_{j+(\ell-1)\cdot k}\}$. For any j, set $\text{CUT}(j)$ defines a natural partitioning of all nodes into clusters, each containing k nodes. Before processing σ, the algorithm OFF_j first migrates its nodes (paying at most $n \cdot \alpha$) to the clustering defined by $\text{CUT}(j)$ and then never changes the node placement.

As all sets $\text{CUT}(j)$ are pairwise disjoint, for any request σ_t, exactly one algorithm OFF_j pays for the request, and thus $\sum_{j=1}^{k} \text{OFF}_j(\sigma_t) = 1$. Therefore, taking the initial node reorganization into account, we obtain that $\sum_{j=1}^{k} \text{OFF}_j(\sigma) \leq k \cdot n \cdot \alpha + \text{ONL}(\sigma)$. By the averaging argument, there exists an offline algorithm OFF_j, such that $\text{OFF}_j(\sigma) \leq \frac{1}{k} \cdot \sum_{j=1}^{k} \text{OFF}_j(\sigma) \leq n \cdot \alpha + \text{ONL}(\sigma)/k$. Thus, $\text{ONL}(\sigma) \geq k \cdot \text{OFF}_j(\sigma) - k \cdot n \cdot \alpha \geq k \cdot \text{OPT}(\sigma) - k \cdot n \cdot \alpha$. The theorem follows because the additive constant $k \cdot n \cdot \alpha$ becomes negligible as the length of σ grows. □

Theorem 3.18. *No deterministic online algorithm* ONL *can achieve a competitive ratio smaller than* 3 *for the case* $k = 2$ *(without augmentation).*

Proof. As in the previous proof, we number the nodes from v_0 to v_{n-1}. We distinguish three types of node clusterings. Configuration A: v_0 collocated with v_1, v_2 collocated with v_3, other nodes collocated arbitrarily; configuration B: v_1 collocated with v_2, v_3 collocated with v_0, other nodes collocated arbitrarily; configuration C: all remaining clusterings.

Similarly to the proof of Theorem 3.17, the adversary always requests a communication between two nodes not collocated by ONL. This time the exact choice of such nodes is relevant: ONL receives a request to (v_1, v_2) in configuration A, and to (v_0, v_1) in configurations B and C.

We define three offline algorithms. They keep nodes $\{v_0, \ldots, v_3\}$ in the first two clusters and the remaining nodes in the remaining clusters (the remaining nodes never change their clusters). More concretely, OFF_1 keeps nodes $\{v_0, \ldots, v_3\}$ always in configuration A and OFF_2 always in configuration B. Furthermore, we define the third algorithm OFF_3 that is in configuration B if ONL is in configuration A, and is in configuration A if ONL is in configuration B or C.

We split the cost of ONL into the cost for serving requests, ONL^{req}, and the cost paid for its migrations, ONL^{mig}. Observe that, for any request σ_t, $\text{OFF}_1(\sigma_t) + \text{OFF}_2(\sigma_t) = \text{ONL}^{\text{req}}(\sigma_t)$. Moreover, as OFF_3 does not pay for any request and migrates only when ONL does, $\text{OFF}_3(\sigma_t) \leq \text{ONL}^{\text{mig}}(\sigma_t)$. Summing up, $\sum_{j=1}^{3} \text{OFF}_j(\sigma_t) \leq \text{ONL}(\sigma_t)$ for any request σ_t. Taking into account the initial reconfiguration of nodes in OFF_j solutions (which involves at most one swap of cost $2 \cdot \alpha$), we obtain that $\sum_{j=1}^{3} \text{OFF}_j(\sigma) \leq 2 \cdot \alpha + \text{ONL}(\sigma)$. Hence, by the averaging argument, there exists $j \in \{1, 2, 3\}$, such that $\text{ONL}(\sigma) \geq 3 \cdot \text{OFF}_j(\sigma) - 2 \cdot \alpha \geq 3 \cdot \text{OPT}(\sigma) - 2 \cdot \alpha$. This concludes the proof, as $2 \cdot \alpha$ becomes negligible as the length of σ grows. □

3.6 Conclusions

In this chapter, we introduced a formal model for studying a natural dynamic partitioning problem which finds applications, e.g., in the context of virtualized distributed systems subject to changing communication patterns. We derived upper and lower bounds, both for the general case as well as for a special case of small clusters, related to a dynamic matching problem. The natural research direction is to develop better deterministic algorithms for the non-augmented variant of the general case, improving over the straightforward $O(k^2 \cdot \ell^2)$-competitive algorithm given in Section 3.2. While the linear dependency on k is inevitable (cf. Section 3.5), it is not known whether an algorithm whose competitive ratio is independent of ℓ is possible. We resolved this issue for the $O(1)$-augmented variant, for which we gave an $O(k \log k)$-competitive algorithm. The disparity between the performance of the algorithm and the lower bound encourages future research on improving either one of those.

In our model, we assumed a simplified view of the substrate network topology, by assuming that the distance among all physical machines is equal. Capturing real-world data center topologies can lead to more network-efficient virtual machine placement. It would be vital to either focus on a specific topology (such as e.g. a Fat Tree), or to develop a general solution for an arbitrary metric.

Part II

Managing Resources in Routers

Chapter 4

Caching of Forwarding Tables

To model the problem of maintaining large forwarding tables, we introduce a natural extension of online caching, where items have inter-dependencies. In the classic online caching problem, items of some universe are requested by a processing entity (e.g., blocks of RAM are requested by the processor). To speed up the access, computers use a faster memory, called *cache*, capable of accommodating k such items. Upon a request to a non-cached item, the algorithm has to fetch it into the cache, paying a fixed cost, while a request to a cached item is free. If the cache is full, the algorithm has to free some space by evicting an arbitrary subset of items from the cache. In the traditional caching problem, the cache elements are independent: it is always feasible to pull in or out the elements of the cache regardless of cache configuration.

Dependencies among to-be-cached items arise in numerous settings and are a natural refinement of many caching problems. As highlighted in the introduction, an important application for caching with tree-based dependencies arises in the context of forwarding rules in routers. We begin with a technical overview, where we explain the setup for storing only a part of forwarding rules in a router. We model this setup by introducing an abstract problem of *tree caching*, for which we propose a competitive algorithm. Finally, we show that the proposed algorithm can be efficiently implemented.

4.1 Technical Setup

As highlighted in the introduction, routers have finite memory, and in an increasingly common scenario the size of a forwarding table exceeds the size of available memory. One of the solutions is to store only a part of a forwarding table in the router, which acts as a cache for the complete forwarding table, stored at another (slow) device. This solution is particularly attractive with the advent of Software-Defined Networking (SDN) technology, which allows managing the router forwarding table using a software controller. Such an approach is used in real-world architectures like [KARW16, KCGR09, Liu01, LLW15, SUF+12], which we describe below in detail. Our model formalizes the underlying operational problems of such architectures. Our algorithm, when applied in the context of such architectures, can hence be used to prolong the lifetime of IP routers.

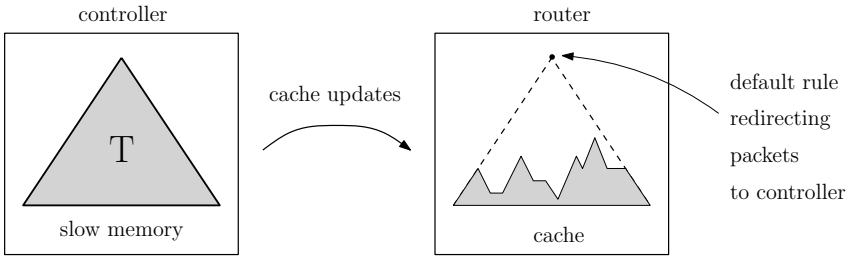

Figure 4.1: The router (*right*) caches only a subset of all rules, and rules that are not cached are answered by the controller (*left*) that keeps the whole tree of rules. Updates to the rules are passed by the controller to the router.

The setup depicted in Figure 4.1 consists of two entities: a single router (e.g., an OpenFlow switch) which caches only a subset of all forwarding rules, and the (SDN) controller, which keeps all rules in its less expensive and slower memory (see [SUF+12] for a more technical discussion). It is also possible that these two entities are parts of a single device. In such a case, they are called forwarding plane and control plane, respectively.

4.1.1 Processing (Forwarding) Packets

Forwarding rules form a tree. That is, they are prefixes of IP addresses (bit strings). Whenever a packet arrives, the router follows the longest matching prefix (LMP) scheme: it searches for the rule that is a prefix of the destination IP of the packet, and among matching rules, it chooses the longest one. In other words, if the prefixes corresponding to rules are stored in the tree, then the tree is traversed from the root downwards, and the last found rule is used (we do not have to assume that they are actually stored in a tree; this tree is implicit in the LMP scheme).

Dependencies among the rules impose certain restrictions on feasible cache configurations. Leaving a less specific rule on the router while evicting a more specific one (i.e., keeping a tree node in the cache while evicting its descendant) would result in a situation where packets are forwarded according to the less specific rule, and hence potentially exit through the wrong port. To guarantee that the router can forward the package correctly with partial forwarding table, we require that cached nodes form a subforest. To see that this is sufficient, consider a packet that, according to the complete forwarding table, should be forwarded using the rule v. If the rule v is present in the portion of the forwarding table cached at the router, then LMP scheme selects v, even if some ancestors of v are not present in the cache. However, if v is not present in the cache, neither are its predecessors, and hence no rule present in the cache matches the packet.

To handle such packet, we want to redirect it to the controller, where an appropriate forwarding rule exists. To this end, we simply add an artificial rule at the tree root in the router (matching an empty prefix). This rule is the least specific rule for any packet, hence as long as any rule that matches the packet is present in the cache, the LMP scheme never selects the artificial rule. When no actual matching rule is found in the cache, the packet will be forwarded according to this artificial rule to the controller that stores all the rules and can handle all

packets appropriately.

Technically speaking, the packet cannot be sent out to the appropriate network by the controller, hence it is supplemented with forwarding informations and sent back to the router. Then, the router overrides the usual procedure of forwarding table lookup, and instead it uses the supplemented information to forward the packet. For this indirection in packet forwarding (a cache miss), we assign a fixed cost 1.

After forwarding a packet, the caching algorithm run at the controller may decide to remove or add rules to the cache. Any such change entails a fixed cost α; this cost corresponds to the transmission of a message from the controller to the router as well as the update of internal data structures of the router. Such an update of proprietary and vendor-dependent structures can be quite costly [HYS13], but the empirical studies show it to be independent of the rule being updated [FFEB05]. Standard caching-related problems arise, such as: which entry (entries) to evict to provide space to store the new one. An immediate update might seem a rational strategy, but in scenarios with highly dynamic networks that rearrange often, it might be beneficial to delay the update. A premature update might result in the situation, where more time is spent alternating the cache configuration than processing packets (this is similar to *thrashing* in virtual memory systems).

4.1.2 Forced Cache Modifications

Additionally, a rule may need to be updated. For example, due to a change communicated by a dynamic routing protocol run at the controller (e.g., BGP) the action defined by a rule has to be modified. In either case, we have to update the rules at the controller: we assume that this cost is zero (this cost is unavoidable for any algorithm, so such an assumption makes our problem only more difficult). Furthermore, if the rule is also stored at the router (the cache), then we charge a fixed cost α for updating the router. We ignore such forced modifications for now, and we come back to it in Section 4.5, where we show that such penalties can be easily simulated in our model.

4.2 Problem Definition

In our abstract problem, we assume that the universe is an arbitrary (not necessarily binary) rooted tree T and the requested items are its nodes. For any tree node v, $T(v) \subseteq T$ is a subtree rooted at v containing v and all its descendants. We require the following property: if a node v is in the cache, then all nodes of $T(v)$ are also cached. In other words, we require that *the cache is a subforest of* T, i.e., a union of disjoint subtrees of T. We call this problem *online tree caching*.

We assume discrete time slotted into rounds, with round $t \geq 1$ corresponding to the time interval $(t-1, t)$. In round t, the algorithm is given one request to exactly one tree node and has to process it, i.e., pay associated costs (if any). We distinguish between two types of requests: a request can be either *positive* or *negative*. The positive requests correspond to "normal" requests known from caching problems: we pay 1 if the node is not cached; for a negative request, we pay 1 if the corresponding request is cached. The ability to process

negative requests will be crucial in using our algorithm to handle forced cache modifications (cf. Section 4.1.2), and we elaborate on this issue in Section 4.5.

Right after round t, at time t, the algorithm may arbitrarily reorganize its cache, (i) ensuring that the resulting cache is a subforest of T (i.e., if the cache contains node v, then it contains the entire $T(v)$) and (ii) preserving the cache capacity constraint. An algorithm pays α for a single node fetch or eviction, where $\alpha \geq 1$ is an integer and a parameter of the problem. We denote the contents of the cache at round t by C_t (as the cache changes contents only between rounds, C_t is well defined). We assume that α is an even integer (this assumption may change costs at most by a constant factor). We assume that the algorithm starts with the empty cache.

Note that in the standard caching problem, fetching is mandatory: the requested item must be present in the cache to serve the request, and if it is not, we must reconfigure the cache to include it. The variant of caching, where this requirement is relaxed is called the *bypassing model* [EILN15b, Ira02b], where the item fetching is optional: and the requested item can be served without being in the cache, for a fixed cost. In the tree caching model, we also assume a bypassing model, where we pay 1 for serving the requested item not present in cache, and afterward we can reconfigure the cache, paying α for each altered item.

4.3 Algorithm TC

The algorithm Tree Caching (TC) presented in the following is a simple scheme that follows a *rent-or-buy paradigm*: it fetches (or evicts) a changeset X if the cost associated with requests at X reaches the cost of such fetch or eviction. More concretely, TC operates in multiple phases. The first phase starts at time 0. TC starts each phase with the empty cache and proceeds as follows. Within a phase, every node keeps a counter, which is initially zero. If at round t it pays 1 for serving the request, it increments its counter. Whenever a node is fetched or evicted from the cache, its counter is reset to zero. Note that this implies that the counter of v is equal to the number of negative (positive) requests to v since its last fetching to the cache (eviction from the cache). For a set $A \subseteq T$, we denote the sum of all counters in A at time t by $\mathrm{cnt}_t(A)$.

We call a non-empty set X a *valid positive changeset* for cache C if $X \cap C = \emptyset$ and $C \cup X$ is a subforest of T, and a *valid negative changeset* if $X \subseteq C$ and $C \setminus X$ is a subforest of T. We call X a *valid changeset* if it is either valid positive or negative changeset. Note that the union of positive (negative) changesets is also a valid positive (negative) changeset. We say that the algorithm applies changeset X, if it fetches all nodes from X (for a positive changeset) and evicts all nodes from X (for a negative one). Note that not all valid changesets may be applied as the algorithm is also limited by its cache capacity (k_{ONL} for an online algorithm and k_{OPT} for the optimal offline one).

At time t, TC verifies whether there exists a valid changeset X, such that

- *(saturation property)* $\mathrm{cnt}_t(X) \geq |X| \cdot \alpha$ and
- *(maximality property)* $\mathrm{cnt}_t(Y) < |Y| \cdot \alpha$ for any valid changeset $Y \supsetneq X$.

In this case, the algorithm modifies its cache applying X. If, at time t, TC is supposed to fetch some set X, but by doing so it would exceed the cache capacity k_{ONL}, it evicts all nodes from the cache instead, and starts a new phase at time t. Such a *final eviction* might not be present in the last phase, in which case we call it *unfinished*.

In Lemma 4.2 (below), we show that at any time, all valid changesets satisfying both properties of TC are either all positive or all negative. Furthermore, right after the algorithm applies a changeset, no valid changeset satisfies saturation property.

4.4 Analysis

We denote the height of T by $h(T)$. A *tree cap* rooted at v is "an upper part" of $T(v)$, i.e., it contains v and if it contains node u, then it also contains all nodes on the path from u to v. If $A \subseteq B$ are both tree caps rooted at v, then we say that A is a tree cap of B.

Throughout this section, we fix an input I, its partition into phases, and analyze both TC and Opt on a single fixed phase P. We denote the times at which P starts and ends by $\text{begin}(P)$ and $\text{end}(P)$, respectively, i.e., rounds in P are numbered from $\text{begin}(P)+1$ to $\text{end}(P)$.

We assume that no positive requests are given to nodes inside the cache and no negative ones to nodes outside of it (this does not change the behavior of TC and can only decrease the cost of Opt).

For the sake of analysis, we assume that at time $\text{end}(P)$, TC actually performs a cache fetch (exceeding the cache size limit) and then, at the same time instant, empties the cache. This replacement only increases the cost of TC. Let k_P denote the number of nodes in the cache of TC at $\text{end}(P)$. In a finished phase, we measure it after the artificial fetch, but right before the final eviction, and thus $k_P \geq k_{\text{ONL}} + 1$; in an unfinished phase $k_P \leq k_{\text{ONL}}$.

The crucial part of our analysis that culminates in Section 4.4.3 is the technique of shifting requests. Namely, we modify the input sequence by shifting requests up or down the tree, so that the resulting input sequence (i) is not harder for Opt and (ii) is more structured: we may lower bound the cost of Opt on each node separately and relate it to the cost of TC.

4.4.1 Properties of Valid Changesets

We state that no valid changeset contains too many requests to its rules. The proof follows by induction and uses the following technical claim.

Lemma 4.1. *For any phase P, the following invariants hold for any time $t > \text{begin}(P)$:*

1. *$\text{cnt}_{t-1}(X) < |X| \cdot \alpha$ for a valid changeset X for C_t,*
2. *$\text{cnt}_t(X) \leq |X| \cdot \alpha$ for a valid changeset X for C_t,*
3. *any changeset X with property $\text{cnt}_t(X) = |X| \cdot \alpha$ contains the node requested at round t.*

Proof. First, observe that Invariant 1 (for time t) along with the fact that round t contains only one request immediately implies that $\text{cnt}_t(X) \leq \text{cnt}_{t-1}(X) + 1 \leq (|X| \cdot \alpha - 1) + 1 = |X| \cdot \alpha$, i.e.,

Invariant 2 for time t. Furthermore, the equality may hold only for changesets containing the node requested at round t, which implies Invariant 3 for time t.

It remains to show that Invariant 1 holds for any step $t > \text{begin}(P)$. It is trivially true for $t = \text{begin}(P) + 1$ as $\text{cnt}_{t-1}(X) = 0$ then. Let $t+1$ be the earliest time in phase P for which Invariant 1 does not hold; we will then show a contradiction with the definition of ALG or a contradiction with other Invariants at time t. That is, we assume that there exists a positive changeset X for C_{t+1} such that $\text{cnt}_t(X) \geq |X| \cdot \alpha$ (the proof for a negative changeset is analogous). Note that ALG must have performed an action (fetch or eviction) at time t as otherwise X would be also a changeset for $C_t = C_{t+1}$ with $\text{cnt}_t(X) \geq |X| \cdot \alpha$, which means that X should have been applied by ALG at time t. We consider two cases.

If ALG fetches a positive changeset Y at time t, $C_{t+1} = C_t \sqcup Y$ and $\text{cnt}_t(Y) = |Y| \cdot \alpha$. Then, $Y \sqcup X$ is a changeset for C_t, and $\text{cnt}_t(Y \sqcup X) \geq |Y \sqcup X| \cdot \alpha$. This contradicts the maximality property of set Y chosen at time t by ALG.

If ALG evicts a negative changeset Y at time t, $C_{t+1} = C_t \setminus Y$. Invariant 2 and the definition of ALG implies $\text{cnt}_t(Y) = |Y| \cdot \alpha$, and thus, by Invariant 3, Y contains the node requested at round t. As $X \cap Y \subseteq C_t$, $X \cap Y$ does not have any positive requests at time t, and therefore $\text{cnt}_t(X \setminus Y) = \text{cnt}_t(X) \geq |X| \cdot \alpha \geq |X \setminus Y| \cdot \alpha$. By Invariant 2, $\text{cnt}_t(X \setminus Y) \leq |X \setminus Y| \cdot \alpha$, and hence $\text{cnt}_t(X \setminus Y) = |X \setminus Y| \cdot \alpha$. This contradicts Invariant 3 as $X \setminus Y$ cannot contain the node requested at round t (because Y contains this node). □

Lemma 4.2. *Fix any time $t > \text{begin}(P)$. For any valid changeset X for C_t, it holds that $\text{cnt}_t(X) \leq |X| \cdot \alpha$. If a changeset X is applied at time t, the following properties hold:*

1. *X contains the node requested at round t,*
2. *$\text{cnt}_t(X) = |X| \cdot \alpha$,*
3. *$\text{cnt}_t(Y) < |Y| \cdot \alpha$ for any valid changeset Y for C_{t+1} (note that C_{t+1} is the cache state right after application of X),*
4. *X is a tree cap of a tree from C_{t+1} if X is positive and it is a tree cap of a tree from C_t if X is negative.*

Proof. The inequality $\text{cnt}_t(X) \leq |X| \cdot \alpha$ is equivalent to Invariant 2 of Lemma 4.1. Assume now that X is applied at time t. By the definition of ALG, $\text{cnt}_t(X) \geq |X| \cdot \alpha$, and thus $\text{cnt}_t(X) = |X| \cdot \alpha$, i.e., Property 2 follows. Then, Invariant 3 of Lemma 4.1 implies Property 1. Finally, Invariant 1 of Lemma 4.1 for time $t+1$ is equivalent to Property 3.

To show Property 4, observe that the changeset X applied at time t cannot be a disjoint union of two (or more) valid changesets X_1 and X_2. By Property 2, $|X| \cdot \alpha = \text{cnt}_t(X) = \text{cnt}_t(X_1) + \text{cnt}_t(X_2)$. If $\text{cnt}_t(X_1) < |X_1| \cdot \alpha$ or $\text{cnt}_t(X_2) < |X_2| \cdot \alpha$, then $\text{cnt}_t(X_1) + \text{cnt}_t(X_2) < (|X_1| + |X_2|) \cdot \alpha = |X| \cdot \alpha$, a contradiction. Therefore, $\text{cnt}_t(X_1) = |X_1| \cdot \alpha$ and $\text{cnt}_t(X_2) = |X_2| \cdot \alpha$. But then Invariant 3 of Lemma 4.1 would imply that both X_1 and X_2 contain a node requested at time t, which is a contradiction as they are disjoint.

Therefore, if X is a positive changeset applied at t, then X is a single tree cap of a tree from subforest C_{t+1}, and likewise, if X is negative, then X is a single tree cap of a tree from subforest C_t. □

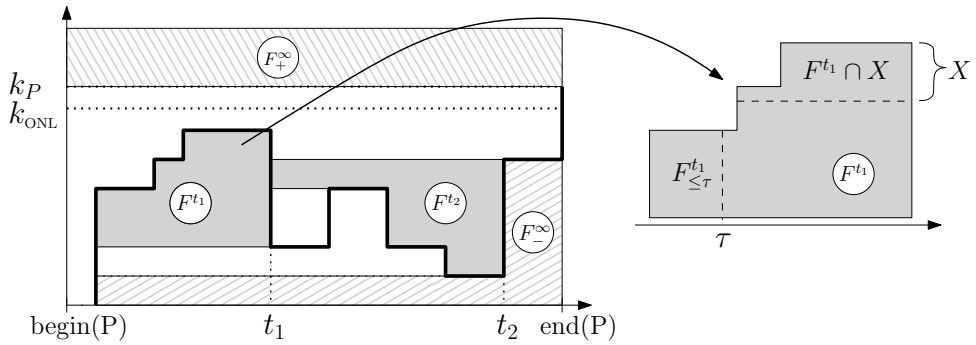

Figure 4.2: Partitioning of a single phase into fields for a line (a tree with no branches). The thick line represents cache contents. Possible final eviction at end(P) is not depicted. F^{t_1} is a negative field and F^{t_2} is a positive one. In the particular depicted example, nodes are ordered from the leaf (bottom) to the root (top of the picture). We emphasize that for a general, branched tree, some notions (in particular fields) no longer have nice geometric interpretations.

4.4.2 Event Space and Fields

In our analysis, we look at a two-dimensional, discrete, spatial-temporal space, called the *event space*. The first dimension is indexed by tree nodes, whose order is an arbitrary extension of the partial order given by the tree. That is, the parent of a node v is always "above" v. The second dimension is indexed by round numbers of phase P. The space elements are called *slots*. Some slots are occupied by requests: a request at node v given at round t occupies slot (v, t). From now on, we will identify P with a set of requests occupying some slots in the event space.

We partition slots of the whole event space into disjoint parts, called *fields*, and we show how this partition is related to the costs of TC and Opt. For any node v and time t, $\text{last}_v(t)$ denotes the last time strictly before t, when node v changed state from cached to non-cached or vice versa; $\text{last}_v(t) = \text{begin}(P)$ if v did not change its state before t in phase P. For a changeset X_t applied by TC at time t, we define the field F^t as

$$F^t = \{\ (v, r) : v \in X_t \ \wedge\ \text{last}_v(t) + 1 \leq r \leq t\ \}\,.$$

That is, field F^t contains all the requests that eventually trigger the application of X_t at time t. We say that F^t ends at t. We call field F^t *positive* (*negative*) if X_t is a positive (negative) changeset. An example of a partitioning into fields is given in Figure 4.2. We define $\text{req}(F^t)$ as the number of requests belonging to slots of F^t and let $\text{size}(F^t)$ be the number of involved nodes (note that $\text{size}(F^t) = |X_t|$). The observation below follows immediately by Lemma 4.2.

Observation 4.3. *For any field F, $\text{req}(F) = \text{size}(F) \cdot \alpha$. All these requests are positive (negative) if F is positive (negative).*

Finally, we call the rest of the event space defined by phase P *open field* and denote it by F^{∞}. The set of all fields except F^{∞} is denoted by $\mathcal{F}$. Let $\text{size}(\mathcal{F}) = \sum_{F \in \mathcal{F}} \text{size}(F)$.

Lemma 4.4. *For any phase P partitioned into a set of fields $\mathcal{F} \cup \{F^{\infty}\}$, it holds that* $\text{TC}(P) \leq 2\alpha \cdot \text{size}(\mathcal{F}) + \text{req}(F^{\infty}) + k_P \cdot \alpha$.

Proof. By Observation 4.3, the cost associated with serving the requests from all fields from $\mathcal{F}$ is $\sum_{F\in\mathcal{F}} \alpha \cdot \text{size}(F) = \alpha \cdot \text{size}(\mathcal{F})$. The cost of the cache reorganization at the fields' ends is exactly the same. The term $\text{req}(F^\infty)$ represents the cost of serving the requests from F^∞ and $k_P \cdot \alpha$ upper-bounds the cost of the final eviction (not present in an unfinished phase). □

4.4.3 Shifting Requests

The actual challenge in the proof is to relate the structure of the fields to the cost of Opt. The rationale behind our construction is based on the following thought experiment. Assume that the phase is unfinished (for example, when the cache is so large that the whole input corresponds to a single phase). Recall that the number of requests in each field $F \in \mathcal{F}$ is equal to $\text{size}(F) \cdot \alpha$. Assume that these requests are evenly distributed among the nodes of F (each node from F receives α requests in the slots of F). Then, the history of any node v is alternating between periods spent in positive fields and periods spent in negative fields. By our even distribution assumption, each such a period contains exactly α requests. Hence, for any two consecutive periods of a single node, Opt has to pay at least α (either α for positive requests or α for negative ones, or α for changing the cached/non-cached state of v). Essentially, this shows that Opt has to pay an amount that can be easily related to $\alpha \cdot \text{size}(\mathcal{F})$.

Unfortunately, the requests may not be evenly distributed among the nodes. To alleviate this problem, we will modify the requests in phase P, so that the newly created phase P' is not harder for Opt and will "almost" have the even distribution property. In this construction, the time frame of P and its fields are fixed.

Legal Shifts

We say that a request placed originally (in phase P) at slot (v, t) is *legally shifted* if its new slot is $(m(v), t)$, where (i) for a positive request, $m(v)$ is either equal to v or is one of its descendants and (ii) for a negative request, $m(v)$ is either equal to v or is one of its ancestors. For any fixed sequence of fetches and evictions within phase P, the associated cost may only decrease when these actions are replayed on the modified requests.

Observation 4.5. *If P' is created from P by legally shifting the requests, then* $\text{Opt}(P') \leq \text{Opt}(P)$.

The main difficulty is however in keeping the legally shifted requests within the field they originally belonged to. For example, a negative request from F shifted at round t from node u to its parent may fall out of F as the parent may still be outside the cache at round t. In effect, a careless shifting of requests may lead to a situation where, for a single node v, requests do not create interleaved periods of positive and negative requests, and hence we cannot argue that $\text{Opt}(P')$ is sufficiently large.

In the following subsections, we show that it is possible to legally shift the requests of any field $F \in \mathcal{F}$ (i.e., shift positive requests down and negative requests up), so that they remain within F, and they will be either exactly or approximately evenly distributed among nodes of F. This will create P' with an appropriately large cost for Opt.

Notation

We start with some general definitions and remarks. For any field F and set of nodes A, let $F \cap A = \{(v,t) \in F : v \in A\}$. Analogously, if L is a set of rounds, then let $F \cap L = \{(v,t) \in F : t \in L\}$. For any field F^t and time τ, we define

$$F^t_{\leq \tau} = F^t \cap \{t' : t' \leq \tau\}.$$

It is convenient to think that F^t evolves with time and $F^t_{\leq \tau}$ is the snapshot of F^t at time τ. Note that F^t may have some nodes not included in $F^t_{\leq \tau}$. These objects are depicted in Figure 4.2.

We may extend the notions of req and size to arbitrary subsets of fields in a natural way. For any subset $S \subseteq F$, we call it *over-requested* if $\mathrm{req}(S) > \mathrm{size}(S) \cdot \alpha$.

Lemma 4.6. *Fix any field F^t, the corresponding changeset X_t, and any time τ.*

1. *If F^t is negative, then for any tree cap D of X_t, the set $F^t_{\leq \tau} \cap D$ is not over-requested.*
2. *If F^t is positive, then for any subtree $T' \subseteq T$, the set $F^t_{\leq \tau} \cap T'$ is not over-requested.*

Proof. As the nodes from $F^t_{\leq \tau} \cap D$ form a valid changeset at time τ, Lemma 4.2 implies $\mathrm{req}(F^t_{\leq \tau} \cap D) = \mathrm{cnt}_\tau(F^t_{\leq \tau} \cap D) \leq |F^t_{\leq \tau} \cap D| \cdot \alpha$.

The proof of the second property is identical: As $F^t_{\leq \tau} \cap T'$ is also a valid changeset at time τ, by Lemma 4.2, $\mathrm{req}(F^t_{\leq \tau} \cap T') = \mathrm{cnt}_\tau(F^t_{\leq \tau} \cap T') \leq |F^t_{\leq \tau} \cap T'| \cdot \alpha$. □

By Lemma 4.6 applied at $\tau = t$ and Observation 4.3, we deduct the following corollary.

Corollary 4.7. *Fix any field F^t, the corresponding changeset X_t and any tree cap D of X_t.*

1. *If F^t is positive, then $\mathrm{req}(F^t \cap D) \geq \alpha \cdot |D|$.*
2. *If F^t is negative, then $\mathrm{req}(F^t \cap (X_t \setminus D)) \geq \alpha \cdot |X_t \setminus D|$.*

Informally speaking, the corollary above states that the average amount of requests in a positive field is *at least as large at the top of the field as at its bottom*. For a negative field this relation is reversed.

Shifting Negative Requests Up

Fix a valid negative changeset X_t applied at time t and the corresponding field F^t. We call a tree cap $Y \subseteq X_t$ *proper* if

1. $\mathrm{req}(F^t \cap Y) = |Y| \cdot \alpha$ and
2. $F^t_{\leq \tau} \cap D$ is not over-requested for any tree cap $D \subseteq Y$ and any time $\tau \leq t$.

The first property of Lemma 4.6 states that before we shift the requests of F_t, the set X_t is proper. We start with $Y = X_t$, and proceed in a bottom-up fashion, inductively using the lemma below. We take care of a single node of Y at a time and ensure that after the shift the number of requests at this node is exactly α and the remaining part of Y remains proper.

Lemma 4.8. *Given a negative field F^t, the corresponding changeset X_t and a proper tree cap $Y \subseteq X_t$, it is possible to choose a leaf v and legally shift some requests inside Y, so that in result $req(v) = \alpha$ and $Y \setminus \{v\}$ is proper.*

Proof. As $\text{req}(F^t \cap Y) = |Y| \cdot \alpha$, Corollary 4.7 implies that any leaf of Y was requested at least α times inside F^t. We pick an arbitrary leaf v, and let $r \geq \alpha$ be the number of requests to v in F^t.

We look at all the requests to v in F^t ordered by their round. Let s be the round when $(\alpha+1)$-th of them arrives. We will now show that at round s, TC already has $p(v)$ in its cache. If it had not, $\{v\}$ would be a tree cap of $F^t_{\leq s}$, and by the first property of Lemma 4.6, it would contain at most α requests, which is a contradiction. Hence, if we shift the chronologically last $r - \alpha$ requests from v to $p(v)$, these requests stay within F^t.

It remains to show that $Y \setminus \{v\}$ is proper after such a shift. We choose any tree cap $D \subseteq Y$ and any time $\tau \leq t$. If D does not contain $p(v)$ or $\tau < s$, then the number of requests in $F^t_{\leq \tau} \cap D$ was not changed by the shift, and hence $F^t_{\leq \tau} \cap D$ is not over-requested. Otherwise, $D \cup \{v\}$ was a tree cap in Y and by the lemma assumption, $F^t_{\leq \tau} \cap (D \cup \{v\})$ was not over-requested. As $F^t_{\leq \tau} \cap D$ has now exactly α fewer requests than $F^t_{\leq \tau} \cap (D \cup \{v\})$ had, it is not over-requested, either. □

Corollary 4.9. *For any negative field F^t, it is possible to legally shift its requests up, so that they remain within F^t and after the modification, each node is requested exactly α times.*

Shifting Positive Requests Down

We will now focus on the problem of shifting the positive requests down in a single positive field F^t, corresponding to a single fetch of TC at the time t. Our goal is to devise a shifting strategy, that will result in at least $\Omega(\text{size}(F^t)/h(T))$ nodes having $\alpha/2$ requests each. First, we prove that from any node v in the field, we can shift down a constant fraction of its requests within the field, distributing them to different nodes.

Lemma 4.10. *Let F^t be a positive field and let X_t be the corresponding changeset fetched to the cache at time t. Fix any node $v \in X_t$ that has been requested at least $c \cdot (\alpha/2)$ times in F^t, where c is an integer. It is possible to shift down its requests to the nodes of $T(v) \cap X_t$, so that these requests remain inside F^t and $\lceil c/2 \rceil$ nodes of $T(v)$ get $\alpha/2$ requests each.*

Proof. We order the nodes $u_1, u_2, \ldots u_{|T(v) \cap X_t|}$ of $T(v) \cap X_t$, so that $\text{last}_{u_i}(t) \leq \text{last}_{u_{i+1}}(t)$ for all i. In case of a tie, we place nodes that are closer to v first. Note that this linear ordering is an extension of the partial order defined by the tree: the parent of a node cannot be evicted later than the node itself (otherwise the cache would cease to be a subforest of T). In particular, it holds that $u_1 = v$.

We number $c \cdot (\alpha/2)$ requests to v chronologically, starting from 1. For any $j \in \{1, \ldots, \lceil c/2 \rceil\}$ we look at round τ_j with the $((j-1) \cdot \alpha + 1)$-th request to v. When this request arrives, the node u_j is already present in the cache. Otherwise, we would have at least $j \cdot \alpha + 1$ requests in $F^t_{\leq \tau_j} \cap \{u_1, \ldots, u_j\}$ (already in $F^t_{\leq \tau_j} \cap \{u_1\}$ alone), which would make it over-requested, and

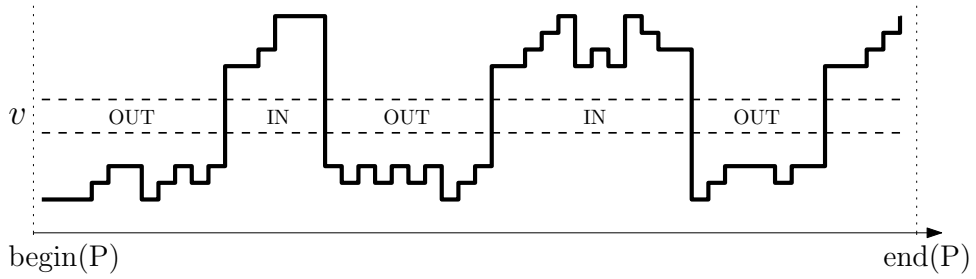

Figure 4.3: Partitioning of the phase into interleaving IN and OUT periods for node v. The thick line represents cache contents. The *leftover* OUT period (the last one) is present for node v as it has finished phase P inside TC's cache. The periods can be followed by requests contained in F^∞.

thus contradict the second property of Lemma 4.6. Hence, we may take requests numbered from $(j-1)\cdot\alpha+1$ to $(j-1)\cdot\alpha+\alpha/2$, shift them down from v to u_j, and after such modification these requests are still inside F^t. Note that for $j=1$ requests are not really shifted, as u_1 is v itself. We perform such shift for any $j \in \{1,\ldots,\lceil c/2\rceil\}$, which yields the lemma. □

Lemma 4.11. *For any positive field F^t, it is possible to legally shift its requests down, so that they remain within F^t and after the modification at least size$(F^t)/(2h(T))$ nodes in F^t have at least $\alpha/2$ requests each.*

Proof. Let X_t be the changeset corresponding to field F^t, which is fetched to the cache at time t. By Observation 4.3, $\mathrm{req}(F^t) = |X_t|\cdot\alpha$. We gather the requests at every node into groups of $\alpha/2$ consecutive requests. In every node at most $\alpha/2$ requests remain not grouped. Let $\overline{\mathrm{req}}(X)$ denote the number of grouped requests in the set X. Clearly, $\overline{\mathrm{req}}(F^t) \geq |X_t|\cdot\alpha/2$, i.e., there are at least $|X_t|$ groups of requests in set X_t.

Let $X_t = X_t^1 \sqcup X_t^2 \sqcup \cdots \sqcup X_t^{h(T)}$ be a partition of the nodes of the tree X_t into layers according to their distance to the root. By the pigeonhole principle, there is a layer X_t^i containing at least $\lceil |X_t|/h(T)\rceil$ groups of requests (each group has $\alpha/2$ requests).

Nodes of X_t^i are independent, i.e., for $u,v \in X_t^i$ the trees $T(u)$ and $T(v)$ are disjoint. Therefore, we may use the shifting strategy described in Lemma 4.10 for each node of X_t^i separately. After such modification, at least $\lceil |X_t|/(2h(T))\rceil \geq \mathrm{size}(F_t)/(2h(T))$ nodes have at least $\alpha/2$ requests each. □

Using Request Shifting for Bounding OPT

Finally, we may use our request shifting to relate $\mathrm{size}(\mathcal{F}) = \sum_{F\in\mathcal{F}}\mathrm{size}(F)$ to the cost of OPT in a single phase P. Recall that k_P denotes the size of TC's cache at the end of P. We assume that OPT may start the phase with an arbitrary state of the cache.

Lemma 4.12. *For any phase P, $\mathrm{OPT}(P) \geq (\mathrm{size}(\mathcal{F})/(4h(T)) - k_P)\cdot\alpha/2$.*

Proof. We transform P using legal shifts that are described in this section. That is, we create a corresponding phase P' that satisfies both Corollary 4.9 and Lemma 4.11. By Observation 4.5, it is sufficient to show that $\mathrm{OPT}(P') \geq (\mathrm{size}(\mathcal{F})/(4h(T)) - k_P)\cdot\alpha/2$.

We focus on a single node v. We cut its history into interleaved periods: OUT *periods* when v is outside the cache and receives positive requests, and IN *periods* when TC keeps v in the cache and v receives negative requests. A final (possibly empty) part corresponding to the time when v is in the F^∞ field is not accounted in OUT or IN periods, i.e., each IN or OUT period corresponds to some field $F \in \mathcal{F}$. Let p^{IN} and p^{OUT} denote the total number of IN and OUT periods (respectively) for all nodes during the phase. An example is given in Figure 4.3.

Recall that TC starts each phase with an empty cache, and hence each node starts with an OUT period. For k_P nodes that are in TC's cache at the end of the phase (and only for them) their history ends with an OUT period not followed by an IN period. We call them *leftover periods*. Thus, $p^{\text{OUT}} = p^{\text{IN}} + k_P$. The total number of periods $(p^{\text{IN}} + p^{\text{OUT}})$ is equal to the total size of all *fields*, $\text{size}(\mathcal{F})$, and thus $p^{\text{OUT}} \geq \text{size}(\mathcal{F})/2$.

We call a period *full* if it has at least $\alpha/2$ requests. The shifting strategies described in the previous section ensure that all IN periods are full and at least $1/(2h(T))$ of all OUT periods are full. Thus, there are at least $p^{\text{OUT}}/(2h(T)) - k_P$ full non-leftover OUT periods; each of them together with the following IN period constitutes a *full* OUT-IN *pair*.

OPT has to pay at least $\alpha/2$ for the node in the course of the history described by a full OUT-IN pair: it pays α either for changing the cached/non-cached state of a node, or $\alpha/2$ for all positive requests or $\alpha/2$ for all negative ones. Thus,

$$\text{OPT}(P') \geq (p^{\text{OUT}}/(2h(T)) - k_P) \cdot \alpha/2 \geq (\text{size}(\mathcal{F})/(4h(T)) - k_P) \cdot \alpha/2 \ . \qquad \square$$

4.4.4 Competitive Ratio

To relate the cost of OPT to TC in a single phase P, we still need to upper-bound $\text{req}(F^\infty)$ and relate $k_P \cdot \alpha$ to the cost of OPT (i.e., compare the bounds on TC and OPT provided by Lemma 4.4 and Lemma 4.12, respectively).

For the next two lemmas, we define V_{OPT} as the set of all nodes that were in OPT cache at some time of P and let $V^{\text{c}}_{\text{OPT}} = T \setminus V_{\text{OPT}}$. Note that V_{OPT} is a union of subforests (nodes present in OPT's cache at consecutive times), and hence a subforest itself.

Lemma 4.13. *For any phase P, it holds that $\text{req}(F^\infty) \leq 2 \cdot k_{\text{ONL}} \cdot \alpha + 2 \cdot \text{OPT}(P)$.*

Proof. We assume first that P is a finished phase. Then, P ends with an artificial fetch of $X_{\text{end}(P)}$ at time $\text{end}(P)$ (followed by the final eviction). We split F^∞ into two disjoint parts (see Figure 4.2):

$$F^\infty_- = \{(v,t) : v \in C_{\text{end}(P)}, t \geq \text{last}_v(\text{end}(P))\},$$
$$F^\infty_+ = \{(v,t) : v \notin C_{\text{end}(P)} \sqcup X_{\text{end}(P)}, \ t \geq \text{last}_v(\text{end}(P))\}.$$

Note that F^∞_- contains only negative requests and F^∞_+ only positive ones. As $\text{req}(F^\infty) = \text{req}(F^\infty_-) + \text{req}(F^\infty_+ \cap V^{\text{c}}_{\text{OPT}}) + \text{req}(F^\infty_+ \cap V_{\text{OPT}})$, we estimate each of these summands separately.

- Nodes from F^∞_- are in the cache $C_{\text{end}(P)}$ and were not evicted from the cache. Thus, $\text{req}(F^\infty_-) \leq |C_{\text{end}(P)}| \cdot \alpha \leq k_{\text{ONL}} \cdot \alpha$.

- All the requests from $V^{\mathrm{c}}_{\mathrm{OPT}}$ are paid by Opt, and hence $\mathrm{req}(F^{\infty}_{+} \cap V^{\mathrm{c}}_{\mathrm{OPT}}) \leq \mathrm{req}(V^{\mathrm{c}}_{\mathrm{OPT}}) \leq \textsc{Opt}(P)$.
- F^{∞}_{+} is a valid changeset for cache $C_{\mathrm{end}(P)} \sqcup X_{\mathrm{end}(P)}$. As V_{OPT} is a subforest of T, $F^{\infty}_{+} \cap V_{\mathrm{OPT}}$ is also a valid changeset for the cache $C_{\mathrm{end}(P)} \sqcup X_{\mathrm{end}(P)}$. Therefore, $\mathrm{req}(F^{\infty}_{+} \cap V_{\mathrm{OPT}}) \leq \mathrm{size}(F^{\infty}_{+} \cap V_{\mathrm{OPT}}) \cdot \alpha$, as otherwise the set fetched at time $\mathrm{end}(P)$ would not be maximal. (TC could then fetch $X_{\mathrm{end}(P)} \sqcup (F^{\infty}_{+} \cap V_{\mathrm{OPT}})$ instead of $X_{\mathrm{end}(P)}$.) Thus, $\mathrm{req}(F^{\infty}_{+} \cap V_{\mathrm{OPT}}) \leq |V_{\mathrm{OPT}}| \cdot \alpha = k_{\mathrm{OPT}} \cdot \alpha + (|V_{\mathrm{OPT}}| - k_{\mathrm{OPT}}) \cdot \alpha \leq k_{\mathrm{ONL}} \cdot \alpha + \textsc{Opt}(P)$. The last inequality follows as — independently of the initial state — Opt needs to fetch at least $|V_{\mathrm{OPT}}| - k_{\mathrm{OPT}}$ nodes to the cache during P.

Hence, in total, $\mathrm{req}(F^{\infty}) \leq 2 \cdot k_{\mathrm{ONL}} \cdot \alpha + 2 \cdot \textsc{Opt}(P)$ for a finished phase P.

We note that if there was no cache change at $\mathrm{end}(P)$, the analysis above would hold with $X_{\mathrm{end}(P)} = \emptyset$ with virtually no change. Therefore, for an unfinished phase P ending with a fetch or ending without cache change at $\mathrm{end}(P)$, the bound on $\mathrm{req}(F^{\infty})$ still holds. However, if an unfinished phase P ends with an eviction, then we look at the last eviction-free time τ of P. We now observe the evolution of field F^{∞} from time τ till $\mathrm{end}(P)$. At time τ, $\mathrm{req}(F^{\infty}) \leq 2 \cdot k_{\mathrm{ONL}} \cdot \alpha + 2 \cdot \textsc{Opt}(P)$. Furthermore, in subsequent times, it may only decrease: at any round F^{∞} gets an additional request, but on eviction $\mathrm{req}(F^{\infty})$ decreases by α times the number of evicted nodes (i.e., at least by $\alpha \geq 1$). Hence, the value of $\mathrm{req}(F^{\infty})$ at $\mathrm{end}(P)$ is also at most $2 \cdot k_{\mathrm{ONL}} \cdot \alpha + 2 \cdot \textsc{Opt}(P)$. □

By combining Lemma 4.4, Lemma 4.12 and Lemma 4.13, we immediately obtain the following corollary (holding for both finished and unfinished phases).

Corollary 4.14. *For any phase P, it holds that* $\mathrm{TC}(P) \leq O(h(T)) \cdot \textsc{Opt}(P) + O(h(T) \cdot (k_P + k_{\mathrm{ONL}}) \cdot \alpha)$.

Using the corollary above, its remains to bound the value of k_P. This is easy for an unfinished phase, as $k_P \leq k_{\mathrm{ONL}}$ there. For a finished phase, we provide another bound.

Lemma 4.15. *For any finished phase P, it holds that* $k_P \cdot \alpha \leq \textsc{Opt}(P) \cdot (k_{\mathrm{ONL}} + 1)/(k_{\mathrm{ONL}} + 1 - k_{\mathrm{OPT}})$.

Proof. First, we compute the number of positive requests in $V^{\mathrm{c}}_{\mathrm{OPT}}$. Let $X_{t_1}, X_{t_2}, \ldots, X_{t_s}$ be all positive changesets applied by TC in P. For any t, let $X'_t = X_t \setminus V_{\mathrm{OPT}}$. As X_t is some tree cap and V_{OPT} is a subforest of T, X'_t is a tree cap of X_t. By Corollary 4.7, the number of requests to nodes of X'_t in field F^t is at least $|X'_t| \cdot \alpha$. These requests for different changesets X_t are disjoint and they are all outside of V_{OPT}. Hence the total number of positive requests outside of V_{OPT} is at least $\sum_{i=1}^{s} |X'_{t_i}| \cdot \alpha$, where $\sum_{i=1}^{s} |X'_{t_i}| \geq |\bigcup_{i=1}^{s} X'_{t_i}| = |(\bigcup_{i=1}^{s} X_{t_i}) \setminus V_{\mathrm{OPT}}| \geq |\bigcup_{i=1}^{s} X_{t_i}| - |V_{\mathrm{OPT}}| \geq k_P - |V_{\mathrm{OPT}}|$.

Now $\textsc{Opt}(P)$ can be split into the cost associated with nodes from V_{OPT} and $V^{\mathrm{c}}_{\mathrm{OPT}}$, respectively. For the former part, Opt has to pay at least $(|V_{\mathrm{OPT}}| - k_{\mathrm{OPT}}) \cdot \alpha$ for the fetches alone. For the latter part, it has to pay 1 for each of at least $(k_P - |V_{\mathrm{OPT}}|) \cdot \alpha$ positive requests outside of V_{OPT}. Hence, $\textsc{Opt}(P) \geq (|V_{\mathrm{OPT}}| - k_{\mathrm{OPT}}) \cdot \alpha + (k_P - |V_{\mathrm{OPT}}|) \cdot \alpha = (k_P - k_{\mathrm{OPT}}) \cdot \alpha$.

Then, $k_P \cdot \alpha \leq k_P \cdot \textsc{Opt}(P)/(k_P - k_{\textsc{Opt}})$. As the phase is finished, $k_P \geq k_{\textsc{Onl}} + 1$, and thus $k_P \cdot \alpha \leq (k_{\textsc{Onl}} + 1) \cdot \textsc{Opt}(P)/(k_{\textsc{Onl}} + 1 - k_{\textsc{Opt}})$. □

Theorem 4.16. *The algorithm* TC *is* $O(h(T) \cdot k_{\textsc{Onl}}/(k_{\textsc{Onl}} - k_{\textsc{Opt}} + 1))$*-competitive.*

Proof. Let $R = h(T) \cdot k_{\textsc{Onl}}/(k_{\textsc{Onl}} - k_{\textsc{Opt}} + 1)$. We split an input I into a sequence of finished phases followed by a single unfinished phase (which may not be present). For a finished phase P, we have $k_P > k_{\textsc{Onl}}$, and hence Corollary 4.14 and Lemma 4.15 imply that $\textsc{Tc}(P) \leq O(R) \cdot \textsc{Opt}(P)$. For an unfinished phase $k_P \leq k_{\textsc{Onl}}$, and therefore, by Corollary 4.14, $\textsc{Tc}(P) \leq O(h(T)) \cdot \textsc{Opt}(P) + O(h(T) \cdot k_{\textsc{Onl}} \cdot \alpha)$. Summing over all phases of I yields $\textsc{Tc}(I) \leq O(R) \cdot \textsc{Opt}(I) + O(h(T) \cdot k_{\textsc{Onl}} \cdot \alpha)$. □

4.5 Handling Forced Cache Modifications

In this section, we show how to handle forced cache modifications (cf. Section 4.1.2). That is, we present a formal argument showing why we can use any q-competitive online algorithm A_T for the tree caching problem to obtain a $2q$-competitive online algorithm A for the tree caching problem with forced cache modifications. Recall that they cost α when the requested item is present in the cache, and 0 if it is not. We call the variant with forced cache modifications the *extended tree caching problem*.

Namely, we take any input I for the extended tree caching problem and create, in an online fashion, an input I_T for the non-extended tree caching problem. To this end, we repeat positive and negative requests from I, and upon encountering a forced cache modification of some node, we insert to I_T a sequence α negative requests for it, called a *chunk*. For any solution for I_T, we may replay its actions (fetches and evictions) on I and vice versa. However, there is one place, where these solutions may have different costs. As forced rule modifications of node v in I is mapped to a chunk of α negative requests to v in I_T, and it is then possible that an algorithm for I_T modifies the cache *during* a chunk. An algorithm that never performs such an action is called *canonical*.

To alleviate this issue, we first note that any algorithm B for I_T can be transformed into a canonical solution B' by postponing all cache modifications that occur during some chunk to the time right after it. Such a transformation may increase the cost of a solution on a chunk at most by α and such an increase occurs only when B modifies a cache within this chunk. Hence, the additional cost of transformation can be mapped to the already existing cost of B, and thus the cost of B' is at most by a factor of 2 larger than that of B.

Furthermore, note that there is a natural cost-preserving bijection between solutions to I and canonical solutions to I_T (solutions perform same cache modifications). Hence, the algorithm A for I runs A_T on I_T, transforms it in an online manner into the canonical solution $A'_T(I_T)$, and replays its cache modification on I. Then, $A(I) = A'_T(I_T) \leq 2 \cdot A_T(I_T) \leq 2q \cdot \textsc{Opt}(I_T) \leq 2q \cdot \textsc{Opt}(I)$.

The second inequality follows immediately by the q-competitiveness of A_T. The third inequality follows by replaying cache modifications as well, but this time we take solution $\textsc{Opt}(I)$

and replay its actions on I_T, creating a canonical (not necessarily optimal) solution of the same cost.

4.6 Lower Bound on the Competitive Ratio

Theorem 4.17. *For any $\alpha \geq 1$, the competitive ratio of any deterministic online algorithm for the online tree caching problem is at least $\Omega(k_{\mathrm{ONL}}/(k_{\mathrm{ONL}} - k_{\mathrm{OPT}} + 1))$*

Proof. We show the lower bound by a reduction to the (standard) caching problem. We will assume that in the tree caching problem, evictions are free (this changes the cost by at most by a factor of two). We consider a tree whose leaves correspond to the set of all elements in the caching problem. The rest of the tree will be irrelevant.

For any input sequence I for the caching problem, we may create a sequence I_{T} for tree caching, where a request to an element is replaced by α requests to the corresponding leaf. Now, we claim that any solution A for I of cost c can be transformed, in an online manner, into a solution A_{T} for I_{T} of cost $\Theta(\alpha \cdot c)$ and vice versa.

If upon a request r, an algorithm A fetches r to the cache and evicts some elements, then A_{T} pays for α corresponding requests to leaf r, fetches r afterward and evicts the corresponding leaves, paying $O(\alpha)$ times the cost of A. By doing it iteratively, A_{T} ensures that its cache is equivalent to that of A. In particular, a request free for A is also free for A_{T}.

Now take any algorithm A_{T} for I_{T}. It can be transformed into the algorithm A'_{T} that (i) keeps only leaves of the tree in the cache and (ii) performs actions only at times that are multiplicities of α (losing at most a constant factor in comparison to A_{T}). Then, fix any chunk of α requests to some leaf r' immediately followed by some fetches and evictions of A'_{T} leaves. Upon seeing the corresponding request r' in I, the algorithm A performs fetches and evictions on the corresponding elements. In effect, the cost of A is $O(1/\alpha)$ times the cost of A_{T}.

The bidirectional reduction described above preserves competitive ratios up to a constant factor. Hence, applying the adversarial strategy for the caching problem that enforces the competitive ratio $R = k_{\mathrm{ONL}}/(k_{\mathrm{ONL}} - k_{\mathrm{OPT}} + 1)$ [ST85] immediately implies the lower bound of $\Omega(R)$ on the competitive ratio for the tree caching problem. □

4.7 Implementation

The whole input fed to a tree caching algorithm is created at the controller: positive requests are caused by cache misses (which redirect packet to the controller) and batches of α negative requests are caused by forced cache modifications. Therefore, the whole tree caching algorithm can be implemented in software in the controller only. The algorithm TC is a simple counter-based scheme, and in this section we show that it can be implemented efficiently.

Recall that at each time t, TC verifies the existence of a valid changeset that satisfies saturation and maximality properties (see the definition of TC in Section 4.3). Here, we show that this operation can be performed efficiently. In particular, in the following two subsections, we will prove the following theorem.

Theorem 4.18. TC *can be implemented using* $O(|T|)$ *additional memory, so that to make a decision at time* t*, it performs* $O(h(T) + \max\{h(T), deg(T)\} \cdot |X_t|)$ *operations, where* $deg(T)$ *is a maximum node degree in* T *and* X_t *is the changeset applied at time* t *(*$|X_t| = 0$ *if no changeset is applied).*

Let v_t be the node requested at round t. Note that we may restrict our attention to requests that entail a cost for TC, as otherwise its counters remain unchanged and certainly TC does not change cache contents. We use Lemma 4.2 to restrict possible candidates for changesets that can be applied at time t. First, we note that if a node v_t requested at round t is outside the cache, then, at time t, TC may only fetch some changeset, and otherwise it may only evict some changeset. Therefore, we may construct two separate schemes, one governing fetches and one for evictions.

In Section 4.7.1, using Lemma 4.2, we show that after processing a positive request, TC needs to verify at most $h(T)$ possible positive changesets, each in constant time, using an auxiliary data structure. The cost of updating this structure at time t is $O(h(T) + h(T) \cdot |X_t|)$.

The situation for negative changesets is more complex as even after applying Lemma 4.2 there are still exponentially many valid negative changesets to consider. In Section 4.7.2, we construct an auxiliary data structure that returns a viable candidate in time $O(h(T) + \deg(T) \cdot |X_t|)$. The update of this structure at time t can be also done in $O(h(T) + \deg(T) \cdot |X_t|)$ operations.

4.7.1 Positive Requests and Fetches

At any time t and for any non-cached node u, we may define $P_t(u)$ as a tree cap rooted at u containing all non-cached nodes from $T(u)$. During an execution of TC, we maintain two values for each non-cached node u: $\mathrm{cnt}_t(P_t(u))$ and $|P_t(u)|$. When a counter at node v_t is incremented, we update $\mathrm{cnt}_t(P_t(u))$ for each ancestor u of v (at most $h(T)$ updated values). Furthermore, if a node v changes its state from cached to non-cached (or vice versa), we update the value of $|P_t(u)|$ for any ancestor u of v (at most $h(T)$ updates per each node that changes the state). Therefore, the total cost of updating these structures at time t is at most $O(h(T) + h(T) \cdot |X_t|)$.

By Lemma 4.2, a positive valid changeset fetched at time t has to contain v_t and is a single tree cap. Such a tree cap has to be equal to $P_t(u)$ for u being an ancestor of v_t. Hence, we may iterate over all ancestors u of v_t, starting from the tree root and ending at v_t, and we stop at the first node u, for which $P_t(u)$ is saturated (i.e., $\mathrm{cnt}_t(P_t(u)) \geq |P_t(u)| \cdot \alpha$). If such a u is found, the corresponding set $P_t(u)$ satisfies also the maximality condition (cf. the definition of TC) as all valid changesets that are supersets of $P_t(u)$ were already verified to be non-saturated. Therefore, in such a case, TC fetches $P_t(u)$. Otherwise, if no saturated changeset is found, TC does nothing. Checking all ancestors of v_t can be performed in time $O(h(T))$.

4.7.2 Negative Requests and Evictions

Handling evictions is more complex. If the request to node v_t at round t was negative, Lemma 4.2 tells us only that the negative changeset evicted by TC has to be a tree cap rooted at u, where u is the root of the cached tree containing v_t. There are exponentially many such tree

caps, and hence their naïve verification is intractable. To alleviate this problem, we introduce the following helper notion. For any set of cached nodes A and any time t, let

$$\mathrm{val}_t(A) = \mathrm{cnt}_t(A) - |A| \cdot \alpha + \frac{|A|}{|T|+1}.$$

Note that for any non-empty set A, $\mathrm{val}_t(A) \neq 0$ as the first two terms are integers and $|A|/(|T|+1) \in (0,1)$. Furthermore, val_t is additive: for two disjoint sets A and B, $\mathrm{val}_t(A \sqcup B) = \mathrm{val}_t(A) + \mathrm{val}_t(B)$. For any time t and a cached node u, we define

$$H_t(u) = \arg\max_D \{\mathrm{val}_t(D) : \ D \text{ is a non-empty tree cap} \quad \text{rooted at } u\}.$$

Our scheme maintains the value $H_t(u)$ for any cached node u. To this end, we observe that $H_t(u)$ can be defined recursively as follows. Let $H'_t(u) = H_t(u)$ if $\mathrm{val}_t(H_t(u)) > 0$ and $H'_t(u) = \emptyset$ otherwise. Then, for any node v and time t, by the additivity of val_t,

$$H_t(u) = \{u\} \sqcup \bigsqcup_{w \text{ is a child of } u} H'_t(w).$$

Each cached node u keeps the value $\mathrm{val}_t(H_t(u))$. Note that set $H_t(u)$ itself can be recovered from this information: we iterate over all children of u (at most $\deg(T)$ of them) and for each child w, if $\mathrm{val}_t(H_t(w)) > 0$, we recursively compute set $H_t(w)$. Thus, the total time for constructing $H_t(u)$ is $O(\deg(T) \cdot |H_t(u)|)$.

During an execution of TC, we update stored values accordingly. That is, whenever a counter at a cached node v_t is incremented, we update $\mathrm{val}_t(H_t(u))$ values for each cached ancestor u of v_t, starting from $u = v_t$ and proceeding towards the cached tree root. Any such update can be performed in constant time, and the total time is thus $O(h(T))$. For a cache change, we process nodes from the changeset iteratively, starting with nodes closest to the root in case of an eviction and furthest from the root in case of a fetch. For any such node u, we appropriately stop or start maintaining the corresponding value of $\mathrm{val}_t(H_t(u))$. The latter requires looking up the stored values at all its children. As u does not have cached ancestors, sets H_t (and hence also the stored values) at other nodes remain unchanged. In total, the cost of updating all H_t values at time t is at most $O(h(T) + \deg(T) \cdot |X_t|)$.

Finally, we show how to use sets H_t to quickly choose a valid changeset for eviction. Recall that for a negative request v_t, the changeset to be evicted has to be a tree cap rooted at u, where u is the root of a cached subtree containing v_t. For succinctness, we use H^u to denote $H_t(u)$. We show that if $\mathrm{val}_t(H^u) < 0$, then there is no valid negative changeset that is saturated, and hence TC does not perform any action, and if $\mathrm{val}_t(H^u) > 0$, then H^u is both saturated and maximal, and hence TC may evict H^u.

1. First, assume that $\mathrm{val}_t(H^u) < 0$. Then, for any tree cap X rooted at u, it holds that $\mathrm{cnt}_t(X) - |X| \cdot \alpha < \mathrm{val}_t(X) \leq \mathrm{val}_t(H^u) < 0$, i.e., X is not saturated, and hence cannot be evicted by TC.

2. Second, assume that $\mathrm{val}_t(H^u) > 0$. As $\mathrm{cnt}_t(H^u) - |H^u| \cdot \alpha$ is an integer and $|H^u|/(|T|+1) < 1$, it holds that $\mathrm{cnt}_t(H^u) - |H^u| \cdot \alpha \geq 0$, i.e., H^u is saturated. Moreover, by Lemma 4.2,

$\mathrm{cnt}_t(H^u) \leq |H^u| \cdot \alpha$, and therefore $\mathrm{cnt}_t(H^u) - |H^u| \cdot \alpha = 0$, i.e., $\mathrm{val}_t(H^u) = |H^u|/(|T|+1)$. It remains to show that H^u is maximal, i.e., there is no valid saturated changeset $Y \supsetneq H^u$. By Lemma 4.2, Y has to be a tree cap rooted at u as well. If Y was saturated, $\mathrm{val}_t(Y) = \mathrm{cnt}_t(Y) - |Y| \cdot \alpha + |Y|/(|T|+1) \geq |Y|/(|T|+1) > |H^u|/(|T|+1) = \mathrm{val}_t(H^u)$, which would contradict the definition of H^u.

Note that node u can be found in time $O(h(T))$, and the actual set H^u (of size $|X_t|$) can be computed in time $O(\deg(T) \cdot |X_t|)$. Therefore the total time for finding set $|X_t|$ is $O(h(T) + \deg(T) \cdot |X_t|)$.

4.8 Conclusions

In this chapter, we defined a novel variant of online caching which finds applications in the context of IP routing networks where forwarding rules can be cached. We presented a deterministic online algorithm that achieves a provably competitive trade-off between the benefit of caching and update costs.

It is worth noting that, in the offline setting, choosing the best static cache in the presence of only positive requests is known as a *tree sparsity* problem and can be solved in $O(|T|^2)$ time [BIS17].

We believe that our work opens interesting directions for future research. Most importantly, it will be interesting to study the optimality of the derived result; we conjecture that the true competitive ratio does not depend on the tree height. In particular, primal-dual approaches that were successfully applied for other caching problems [You94, ACER12, BBN12] may turn out to be useful also for the considered variant.

Bibliography

[ABB+12] Ioannis Alagiannis, Renata Borovica, Miguel Branco, Stratos Idreos, and Anastasia Ailamaki. NoDB: Efficient query execution on raw data files. In *Proc. ACM SIGMOD*, pages 241–252, 2012.

[ACER12] Anna Adamaszek, Artur Czumaj, Matthias Englert, and Harald Räcke. An $O(\log k)$-competitive algorithm for generalized caching. In *23rd ACM-SIAM Symposium on Discrete Algorithms (SODA)*, pages 1681–1689, 2012.

[ACN00] Dimitris Achlioptas, Marek Chrobak, and John Noga. Competitive analysis of randomized paging algorithms. *Theoretical Computer Science*, 234(1–2):203–218, 2000.

[AGOT92] Ravindra Ahuja, Andrew Goldberg, James Orlin, and Robert Tarjan. Finding minimum-cost flows by double scaling. *Mathematical Programming*, 53(1–3):243–266, 1992.

[AKK99] Sanjeev Arora, David R. Karger, and Marek Karpinski. Polynomial time approximation schemes for dense instances of NP-hard problems. *Journal of Computer and System Sciences*, 58(1):193–210, 1999.

[ALPS16] Chen Avin, Andreas Loukas, Maciej Pacut, and Stefan Schmid. Online balanced repartitioning. *International Conference on Distributed Computing*, pages 243–256, 2016.

[ALV08] Mohammad Al-Fares, Alexander Loukissas, and Amin Vahdat. A scalable, commodity data center network architecture. In *Proc. ACM SIGCOMM*, pages 63–74, 2008.

[AMO93] Ravindra Ahuja, Thomas Magnanti, and James Orlin. *Network Flows: Theory, Algorithms, and Applications*. Prentice-Hall, Inc., 1993.

[AR06] Konstantin Andreev and Harald Räcke. Balanced graph partitioning. *Theory of Computing Systems*, 39(6):929–939, 2006.

[AWS] Amazon Web Services. URL: https://aws.amazon.com/ec2/.

[AZU] Microsoft Azure. URL: http://azure.microsoft.com.

[BBN12] Nikhil Bansal, Niv Buchbinder, and Joseph Naor. Randomized competitive algorithms for generalized caching. *SIAM Journal on Computing*, 41(2):391–414, 2012.

[BCKR11] Hitesh Ballani, Paolo Costa, Thomas Karagiannis, and Antony Rowstron. Towards predictable datacenter networks. *Proc. ACM SIGCOMM*, 41(4):242–253, 2011.

[BE98] Allan Borodin and Ran El-Yaniv. Online computation and competitive analysis. *Cambridge University Press*, 1998.

[BGP] CIDR Report on BGP Table Size. URL: https://www.cidr-report.org/cgi-bin/plota?file=%2fvar%2fdata%2fbgp%2fas2.0%2fbgp%2dactive%2etxt&descr=Active%20BGP%20entries%20%28FIB%29&ylabel=Active%20BGP%20entries%20%28FIB%29&with=step.

[BIS17] Arturs Backurs, Piotr Indyk, and Ludwig Schmidt. Better approximations for tree sparsity in nearly-linear time. In *Proc. 28th ACM-SIAM Symposium on Discrete Algorithms (SODA)*, pages 2215–2229, 2017.

[BMP+17] Marcin Bienkowski, Jan Marcinkowski, Maciej Pacut, Stefan Schmid, and Aleksandra Spyra. Online tree caching. *ACM Symposium on Parallelism in Algorithms and Architectures*, pages 329–338, 2017.

[BRV] BGP statistics from route-views data. http://bgp.potaroo.net/bgprpts/rva-index.html.

[BS13] Marcin Bienkowski and Stefan Schmid. Competitive FIB aggregation for independent prefixes: Online ski rental on the trie. In *Proc. 20th Int. Colloq. on Structural Information and Communication Complexity (SIROCCO)*, volume 8179 of *Lecture Notes in Computer Science*, pages 92–103. Springer, 2013.

[BSSU14] Marcin Bienkowski, Nadi Sarrar, Stefan Schmid, and Steve Uhlig. Competitive FIB aggregation without update churn. In *Proc. 34th IEEE Int. Conf. on Distributed Computing Systems (ICDCS)*, pages 607–616, 2014.

[BUP] The BGP instability report. URL: http://bgpupdates.potaroo.net/instability/bgpupd.html.

[CKH+00] Pierluigi Crescenzi, Viggo Kann, Magnus Halldorsson, Marek Karpinski, and Gerhard Woeginger. Maximum 3-dimensional matching. *A Compendium of NP Optimization Problems*, 2000.

[CKPV91] Marek Chrobak, Howard J. Karloff, Thomas H. Payne, and Sundar Vishwanathan. New results on server problems. *SIAM Journal on Discrete Mathematics*, 4(2):172–181, 1991.

[CMSV17] Michael Cohen, Aleksander Madry, Piotr Sankowski, and Adrian Vladu. Negative-Weight Shortest Paths and Unit Capacity Minimum Cost Flow in O (m(10/7) log

W) Time. *Proceedings of the 28th ACM-SIAM Symposium on Discrete Algorithms (SODA)*, pages 752–771, 2017.

[CMU+10] Luca Cittadini, Wolfgang Muhlbauer, Steve Uhlig, Randy Bushy, Pierre Francois, and Olaf Maennel. Evolution of internet address space deaggregation: myths and reality. *IEEE Journal of Selected Areas in Communications*, 28(8):1238–1249, 2010.

[CZM+11] M. Chowdhury, M. Zaharia, J. Ma, M. I. Jordan, and I. Stoica. Managing Data Transfers in Computer Clusters with Orchestra. *ACM SIGCOMM*, 41(4):98–109, 2011.

[DG04] Jeffrey Dean and Sanjay Ghemawat. Mapreduce: Simplified data processing on large clusters. In *Proc. USENIX OSDI*, pages 137–150, 2004.

[DKVZ99] Richard P. Draves, Christopher King, Srinivasan Venkatachary, and Brian D. Zill. Constructing optimal IP routing tables. In *Proc. IEEE Int. Conference on Computer Communications (INFOCOM)*, pages 88–97, 1999.

[EGOS05] Friedrich Eisenberg, Fabrizio Grandoni, Gianpaolo Oriolo, and Martin Skutella. New Approaches for Virtual Private Network Design. *Int. Colloq. on Automata, Languages and Programming (ICALP)*, pages 1151–1162, 2005.

[EILN15a] Leah Epstein, Csanád Imreh, Asaf Levin, and Judit Nagy-György. Online file caching with rejection penalties. *Algorithmica*, 71(2):279–306, 2015.

[EILN15b] Leah Epstein, Csanád Imreh, Asaf Levin, and Judit Nagy-György. Online file caching with rejection penalties. *Algorithmica*, 71(2):279–306, 2015.

[EILNG11] Leah Epstein, Csanád Imreh, Asaf Levin, and Judit Nagy-György. On variants of file caching. In *Proc. 38th Int. Colloq. on Automata, Languages and Programming (ICALP)*, pages 195–206, 2011.

[ENRS99] Guy Even, Joseph Naor, Satish Rao, and Baruch Schieber. Fast approximate graph partitioning algorithms. *SIAM Journal on Computing*, 28(6):2187–2214, 1999.

[ENRS00] Guy Even, Joseph Naor, Satish Rao, and Baruch Schieber. Divide-and-conquer approximation algorithms via spreading metrics. *Journal of the ACM*, 47(4):585–616, 2000.

[EXH] ACL and QoS TCAM Exhaustion Avoidance. URL: https://www.cisco.com/c/en/us/support/docs/switches/catalyst-4000-series-switches/66978-tcam-cat-4500.html.

[FBB+13] Andreas Fischer, Juan Botero, Michael Beck, Hermann DeMeer, and Xavier Hesselbach. Virtual network embedding: A survey. *IEEE Communications Surveys and Tutorials*, 15(4):1888–1906, 2013.

[FFEB05] Pierre François, Clarence Filsfils, John Evans, and Olivier Bonaventure. Achieving sub-second IGP convergence in large IP networks. volume 35, pages 35–44, 2005.

[FK02] Uriel Feige and Robert Krauthgamer. A polylogarithmic approximation of the minimum bisection. *SIAM Journal on Computing*, 31(4):1090–1118, 2002.

[FKL+91] Amos Fiat, Richard M. Karp, Michael Luby, Lyle A. McGeoch, Daniel D. Sleator, and Neal E. Young. Competitive paging algorithms. *Journal of Algorithms*, 12(4):685–699, 1991.

[FKN00] Uriel Feige, Robert Krauthgamer, and Kobbi Nissim. Approximating the minimum bisection size (extended abstract). In *Proc. 32nd ACM Symposium on Theory of Computing (STOC)*, pages 530–536, 2000.

[FOST10] Samuel Fiorini, Gianpaolo Oriolo, Laura Sanità, and Dirk Oliver Theis. The VPN Problem with Concave Costs. *SIAM Journal of Discrete Mathematics. 24(3)*, pages 1080–1090, 2010.

[FPCS15] Carlo Fuerst, Maciej Pacut, Paolo Costa, and Stefan Schmid. How hard can it be? Understanding the complexity of replica aware virtual cluster embeddings. *International Conference on Network Protocols (ICNP)*, pages 11–21, 2015.

[FPS17] Carlo Fuerst, Maciej Pacut, and Stefan Schmid. Data locality and replica aware virtual cluster embeddings. *Journal of Theoretical Computer Science 697*, pages 37–57, 2017.

[FSSC16] Carlo Fuerst, Stefan Schmid, Lalith Suresh, and Paolo Costa. Kraken: Online and elastic resource reservations for multi-tenant datacenters. *Journal IEEE/ACM Transactions on Networking (TON)*, 26(1):422–435, 2016.

[GCE] Google Compute Engine. URL: http://cloud.google.com.

[GJS76] M. R. Garey, David S. Johnson, and Larry J. Stockmeyer. Some Simplified NP-Complete Graph Problems. *Theoretical Computer Science*, 1(3):237–267, 1976.

[GKK+01] Anupam Gupta, Jon Kleinberg, Amit Kumar, Rajeev Rastogi, and Bulent Yener. Provisioning a virtual private network. *Proc. ACM Symposium on Theory of Computing (STOC)*, pages 389–398, 2001.

[GKR03] Anupam Gupta, Amit Kumar, and Tim Roughgarden. Simpler and better approximation algorithms for network design. *Proc. of the ACM symposium on Theory of Computing (STOC)*, pages 365–372, 2003.

[GLW+10] Chuanxiong Guo, Guohan Lu, Helen J. Wang, Shuang Yang, Chao Kong, Peng Sun, Wenfei Wu, and Yongguang Zhang. SecondNet: A data center network virtualization architecture with bandwidth guarantees. In *Proc. ACM CoNEXT*, pages 101–112, 2010.

[GOS08] Navin Goyal, Neil Olver, and F B. Shepherd. The VPN conjecture is true. In *Proc. 40th Annual ACM Symposium on Theory of Computing (STOC)*, pages 443–450, 2008.

[GT83] Harold Gabow and Robert Tarjan. A linear-time algorithm for a special case of disjoint set union. *Proceedings of 50th annual ACM Symposium on Theory of computing (STOC)*, pages 246–251, 1983.

[GT89] Andrew Goldberg and Robert Tarjan. Finding minimum-cost circulations by canceling negative cycles. *ACM Symposium on Theory of Computing (STOC)*, 36(4):873–886, 1989.

[HYS13] Danny Yuxing Huang, Ken Yocum, and Alex C. Snoeren. High-fidelity switch models for software-defined network emulation. In *Proc. 2ndACM SIGCOMM Workshop on Hot Topics in Software Defined Networking (HotSDN)*, pages 43–48, 2013.

[HyV] Hyper-V. URL: https://www.microsoft.com/en-us/cloud-platform/server-virtualization.

[Ira02a] Sandy Irani. Page replacement with multi-size pages and applications to web caching. *Algorithmica*, 33(3):384–409, 2002.

[Ira02b] Sandy Irani. Page replacement with multi-size pages and applications to web caching. *Algorithmica*, 33(3):384–409, 2002.

[KARW16] Naga Katta, Omid Alipourfard, Jennifer Rexford, and David Walker. Cacheflow: Dependency-aware rule-caching for software-defined networks. In *Proc. ACM Symposium on SDN Research (SOSR)*, 2016.

[KCGR09] Changhoon Kim, Matthew Caesar, Alexandre Gerber, and Jennifer Rexford. Revisiting route caching: The world should be flat. In *Proc. 10th Int. Conf. on Passive and Active Network Measurement (PAM)*, pages 3–12, 2009.

[KCR+12] Elliott Karpilovsky, Matthew Caesar, Jennifer Rexford, Aman Shaikh, and Jacobus E. van der Merwe. Practical network-wide compression of IP routing tables. *IEEE Transactions on Network and Service Management*, 9(4):446–458, 2012.

[KF06] Robert Krauthgamer and Uriel Feige. A polylogarithmic approximation of the minimum bisection. *SIAM Review*, 48(1):99–130, 2006.

[KNS09] Robert Krauthgamer, Joseph Naor, and Roy Schwartz. Partitioning graphs into balanced components. In *Proc. 20th ACM-SIAM Symposium on Discrete Algorithms (SODA)*, pages 942–949, 2009.

[KVM] Kernel-based Virtual Machine. URL: http://www.linux-kvm.org.

[Lei85] Charles Leiserson. Fat-trees: universal networks for hardware-efficient supercomputing. *IEEE Transactions on Computers*, 34(10):892–901, 1985.

[Liu01] Huan Liu. Routing prefix caching in network processor design. In *Proc. 10th Int. Conf. on Computer Communications and Networks (ICCCN)*, pages 18–23, 2001.

[LLW15] Yaoqing Liu, Vince Lehman, and Lan Wang. Efficient FIB caching using minimal non-overlapping prefixes. *Computer Networks*, 83:85–99, 2015.

[LMT90] Tom Leighton, Filia Makedon, and S. G. Tragoudas. Approximation algorithms for VLSI partition problems. In *Proc. IEEE International Symposium on Circuits and Systems (ISCAS)*, volume 4, pages 2865–2868, 1990.

[LXS+13] Layong Luo, Gaogang Xie, Kavé Salamatian, Steve Uhlig, Laurent Mathy, and Yingke Xie. A trie merging approach with incremental updates for virtual routers. In *Proc. 32nd IEEE Int. Conf. on Computer Communications (INFOCOM)*, pages 1222–1230, 2013.

[LZN+10] Yaoqing Liu, Xin Zhao, Kyuhan Nam, Lan Wang, and Beichuan Zhang. Incremental forwarding table aggregation. In *Proc. Global Communications Conference (GLOBECOM)*, pages 1–6, 2010.

[LZW13] Yaoqing Liu, Beichuan Zhang, and Lan Wang. Fast incremental FIB aggregation. pages 1213–1221, 2013.

[Mad13] Aleksander Madry. Navigating Central Path with Electrical Flows: From Flows to Matchings, and Back. *2013 IEEE 54th Annual Symposium on Foundations of Computer Science (FOCS)*, pages 253–262, 2013.

[MEC] Measuring EC2 system performance. http://goo.gl/V5zhEd.

[MP12] Jeffrey C Mogul and Lucian Popa. What we talk about when we talk about cloud network performance. *ACM SIGCOMM Computer Communication Review*, 42(5):44–48, 2012.

[MS91] Lyle McGeoch and Daniel Sleator. A strongly competitive randomized paging algorithm. *Algorithmica*, 6(6):816–825, 1991.

[PKC+12] Lucian Popa, Gautam Kumar, Mosharaf Chowdhury, Arvind Krishnamurthy, Sylvia Ratnasamy, and Ion Stoica. Faircloud: sharing the network in cloud computing. *Proceedings of the ACM SIGCOMM 2012 Conference on Applications, Technologies, Architectures, and Protocols for Computer Communication*, pages 187–198, 2012.

[PS06] Kostas Pagiamtis and Ali Sheikholeslami. Content-Addressable Memory (CAM) circuits and architectures: a tutorial and survey. *IEEE Journal of Solid-State Circuits. 41(3)*, pages 712–727, 2006.

[PYB+13] Lucian Popa, Praveen Yalagandula, Sujata Banerjee, Jeffrey C. Mogul, Yoshio Turner, and Jose Renato Santos. Elasticswitch: Practical work-conserving bandwidth guarantees for cloud computing. In *Proc. ACM SIGCOMM*, pages 351–362, 2013.

[Räc08] Harald Räcke. Optimal hierarchica decompositions for congestion minimization in networks. In *Proc. 40th ACM Symposium on Theory of Computing (STOC)*, pages 255–264, 2008.

[RFS15] Matthias Rost, Carlo Fuerst, and Stefan Schmid. Beyond the stars: Revisiting virtual cluster embeddings. *Proc. ACM SIGCOMM Computer Communication Review (CCR)*, 45(3):12–18, 2015.

[RR04] Satish Rao and Andréa W. Richa. New Approximation Techniques for Some Linear Ordering Problems. *SIAM Journal on Computing. 34(2)*, pages 388–404, 2004.

[RST+11] Henrique Rodrigues, Jose Renato Santos, Yoshio Turner, Paolo Soares, and Dorgival Guedes. Gatekeeper: Supporting bandwidth guarantees for multi-tenant datacenter networks. In *Proc. 3rd Conference on I/O Virtualization (WIOV)*, pages 6–6, 2011.

[RTK+13] Gábor Rétvári, János Tapolcai, Attila Korösi, András Majdán, and Zalán Heszberger. Compressing IP forwarding tables: towards entropy bounds and beyond. In *Proc. ACM SIGCOMM Conference*, pages 111–122, 2013.

[RVR+07] Barath Raghavan, Kashi Vishwanath, Sriram Ramabhadran, Kenneth Yocum, and Alex C. Snoeren. Cloud control with distributed rate limiting. In *Proc. ACM SIGCOMM*, pages 337–348, 2007.

[SKGK10] Alan Shieh, Srikanth Kandula, Albert Greenberg, and Changhoon Kim. Seawall: Performance isolation for cloud datacenter networks. In *Proc. USENIX HotCloud*, pages 1–1, 2010.

[SSW03] Subhash Suri, Tuomas Sandholm, and Priyank Ramesh Warkhede. Compressing two-dimensional routing tables. *Algorithmica*, 35(4):287–300, 2003.

[ST85] Daniel Sleator and Robert Tarjan. Amortized efficiency of list update and paging rules. *Communications of the ACM*, 28(2):202–208, 1985.

[ST97] Horst Simon and Shang-Hua Teng. How good is recursive bisection? *SIAM Journal on Computing*, 18(5):1436–1445, 1997.

[STT03] Ed Spitznagel, David Taylor, and Jonathan Turner. Packet classification using extended tcams. In *Proc. 11th IEEE Int. Conf. on Network Protocols (ICNP)*, pages 120–131, 2003.

[SUF+12] Nadi Sarrar, Steve Uhlig, Anja Feldmann, Rob Sherwood, and Xin Huang. Leveraging Zipf's law for traffic offloading. *ACM SIGCOMM Computer Communication Review*, 42(1):16–22, 2012.

[SV95] Huzur Saran and Vijay Vazirani. Finding k cuts within twice the optimal. *SIAM Journal on Computing*, 24(1):101–108, 1995.

[Tar85] Éva Tardos. A strongly polynomial minimum cost circulation algorithm. *Combinatorica*, 5(3):247–255, July 1985.

[UNT+11] Zartash Afzal Uzmi, Markus Nebel, Ahsan Tariq, Sana Jawad, Ruichuan Chen, Aman Shaikh, Jia Wang, and Paul Francis. SMALTA: Practical and near-optimal

FIB aggregation. In *Proc. of the 7th Int. Conference on Emerging Networking Experiments and Technologies (CoNEXT)*, 2011.

[VME] VMware ESXi. URL: http://vmware.com/products/esxi-and-esx/.

[XDHK12] Di Xie, Ning Ding, Y Charlie Hu, and Ramana Kompella. The only constant is change: incorporating time-varying network reservations in data centers. *ACM SIGCOMM Computer Communication Review (CCR)*, pages 199–210, 2012.

[XEN] Xen Project. URL: http://www.xenproject.org.

[XRZ+13] Reynold Xin, Josh Rosen, Matei Zaharia, Michael Franklin, Scott Shenker, and Ion Stoica. Shark: SQL and Rich Analytics at Scale. In *Proc. ACM SIGMOD*, pages 13–24, 2013.

[You94] Neal Young. The k-server dual and loose competitiveness for paging. *Algorithmica*, 11(6):525–541, 1994.

[You02] Neal Young. On-line file caching. *Algorithmica*, 33(3):371–383, 2002. Also appeared in *Proc. of the 9th ACM-SIAM Symposium on Discrete Algorithms (SODA)*, pages 82–86, 1998.

[ZLWZ10] Xin Zhao, Yaoqing Liu, Lan Wang, and Beichuan Zhang. On the aggregatability of router forwarding tables. In *Proc. 29th IEEE Int. Conference on Computer Communications (INFOCOM)*, pages 848–856, 2010.