
University of Wrocław

PhD Thesis

# Algorithmic aspects of contemporary networks

**Maciej Pacut**

supervised by
Dr hab. Marcin Bieńkowski

November 2018

# Contents


**1 Introduction** **5**

1.1 Machine Virtualization in Data Centers . . . 6

1.1.1 Machine Migration . . . 7

1.1.2 Virtual Network Embedding . . . 7

1.1.3 Our Contributions . . . 8

1.1.4 Related Work . . . 11

1.2 Router Memory Optimization . . . 12

1.2.1 Forwarding Tables . . . 12

1.2.2 Growth of the Internet . . . 12

1.2.3 Our Contributions . . . 13

1.2.4 Related Work . . . 14

1.3 Bibliographic notes and acknowledgements . . . 15

**I Mapping Virtual Networks** **17**

**2 Virtual Networks with Static Topology** **19**

2.1 Problem Definition . . . 19

2.1.1 Optimization Objective . . . 20

2.1.2 Problem Decomposition . . . 20

2.2 Polynomial-Time Algorithms . . . 21

2.2.1 Flow Algorithms . . . 22

2.2.2 Matching Algorithms . . . 24

2.2.3 Dynamic Programming . . . 28

2.2.4 Simple Problems . . . 30

2.3 NP-Hardness Results . . . 30

2.3.1 Introduction to 3D Perfect Matching . . . 31

2.3.2 Hardness of Multi-Assignments . . . 31

2.3.3 Hardness of Inter-connects . . . 33

2.4 A Detailed Study of Replica Selection Hardness . . . 34

2.4.1 Two Replicas without Bandwidth Constraints . . . 35

2.4.2 Two replicas without Multiple Assignment . . . 40

2.5 Conclusions . . . 46

**3 Virtual Networks with Dynamic Topology** **47**
3.1 Problem Definition . . . 47
3.2 A Simple Upper Bound . . . 48
3.3 Algorithm Crep . . . 49
3.3.1 Algorithm Definition . . . 49
3.3.2 Analysis: Structural Properties . . . 51
3.3.3 Analysis: Lower Bound on OPT . . . 52
3.3.4 Analysis: Upper Bound on CREP . . . 54
3.3.5 Analysis: Competitive Ratio . . . 56
3.4 Online Rematching . . . 57
3.4.1 Greedy Algorithm . . . 58
3.4.2 Analysis . . . 58
3.5 Lower Bounds . . . 61
3.5.1 Lower Bound by Reduction to Online Paging . . . 61
3.5.2 Additional Lower Bounds . . . 63
3.6 Conclusions . . . 64

**II Managing Resources in Routers** **65**

**4 Caching of Routing Tables** **67**
4.1 Problem Definition . . . 69
4.2 Algorithm . . . 70
4.3 Analysis of TC . . . 70
4.3.1 Event Space and Fields . . . 71
4.3.2 Shifting Requests . . . 72
4.3.3 Competitive Ratio . . . 77
4.4 No over-requested changesets . . . 78
4.5 Implementation of TC . . . 79
4.5.1 Positive Requests and Fetches . . . 80
4.5.2 Negative Requests and Evictions . . . 80
4.6 Cache Updates with Fixed Cost . . . 82
4.7 Lower Bound on the Competitive Ratio . . . 82
4.8 Conclusions . . . 83

# 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, collectively called the *computer cluster*. Owning and maintaing 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 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 facilites that contain computer clusters 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 Elastic Cloud Computing EC2 [AWS] or Google Compute Engine [GCE] provide convenient on-demand computational power while hiding most of 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 bandwith. 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 efficient manner. For example, 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 those involve multiple machines that cooperate over 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 relying 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. We investigate the increasingly common scenario, where the number of rules exceeds the available memory capacity.

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 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 additon to the *host operating system*, which runs directly on the physical hardware. In a data center,

the main purpose of virtualization is to provide the 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.

Such mechanisms play an important role in load balancing in the data center and allow for sophisticated optimizations such as *reducing network distance between communicating virtual machines*. In this thesis, we focus on migration capabilities provided by modern virtualization techologies used for efficient usage of important resource in the data center — the network bandwidth. 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 its 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 refered to as *nodes* of a virtual network. To guarantee certain quality of service ($QoS$) for multitude of co-existing virtual networks, up-front bandwidth reservations are required. However, the generality of performed calculations results in unpredictibility of communication patterns and poses a challenge in optimization of bandwidth reservations. In this thesis, we provide algorithms for efficient management of network reservations without any assumptions about communication patterns.

To measure the quality of resource management strategy, in Part I we state formal optimization problems; for now, we only briefly sketch it. Physical components of a data center are modelled 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 modelled 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*

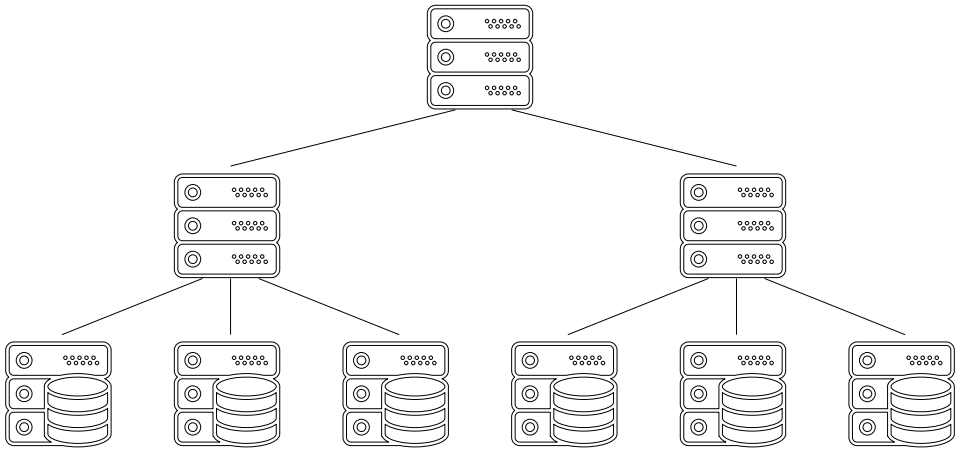

**Figure 1.1:** The model of typical data center with a tree-like network topology. We distinguish two types of tree nodes: the the intermediate nodes that transmit communication, and the computing machines, located at the leaves of a tree. Network links between nodes are depicted as solid lines.

of 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 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 a 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 by certain important properties of batch-processing applications that use distributed file systems.

#### Static Mapping of Virtual Networks

In the static scenario studied in Chapter 2, to guarantee certain quality of service ($QoS$), we need to acquire network reservations for all pairs of cooperating virtual machines. The combinatorial problem that we consider in the static 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 MapReduce [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.* In Map-Reduce applications, virtual machines process large amounts of data chunks that are stored in a distributed file system. Each chunk of data must be assigned and transferred to a virtual machine. Data chunk transfer requires its 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 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:

- Which chunks to assign to which virtual machine?
- How to exploit redundancy and select good replicas?
- How to efficiently embed virtual machines and their inter-connecting network?

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 prove limitations of their computational tractability, by proving their NP-completeness. Interestingly, our hardness results also hold in *uncapacitated* 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. Possibility of migration provides efficient tools that allow to react to unpredictible communication patterns. For example, if some distant nodes communicate often, it is vital to reduce the 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. 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 by comparing its performance to an optimal offline algorithm that knows the whole input sequence in advance. To obtain the competitive

ratio for a minimization problem, we take the maximum (over all input sequences) of the cost of an online algorithm divided by the cost of an offline algorithm.

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 virtual machine 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 $\ell$ clusters (physical machines) each of capacity $k = n/\ell$.

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

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 cluster holding $y$, $y$ to the one holding $x$, or should both nodes be migrated to a new cluster?

- *Which nodes to evict?* There may not exist sufficient space in the desired destination cluster. In this case, the algorithm needs to decide which nodes to "evict" (migrate to other clusters), to free up space.

In the model described above, every physical machine fully utilizes its processing capabilities — it hosts 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 present 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 which provide a provable network performance. To guarantee a certain performance level, we reserve a portion of a bandwidth among cooperating nodes. We distinguish between two major approaches to bandwidth reservations. Relative reservations [PKC+12, PYB+13, SKGK10] depend on supplying the volume of bandwidth communication between each pair of nodes, while in absolute reservations [BCKR11, GLW+10, RVR+07, RST+11, XDHK12] we guarantee sufficient bandwidth for any communication pattern by acquiring sufficient bandwidth among all nodes. In this thesis we research both approaches: in Chapter 2 we study absolute reservations, and in Chapter 3 we study relative reservations.

In Chapter 2, we study virtual network embeddings. The most popular virtual network abstraction for batch-processing applications [DG04] today is the *virtual cluster* [BCKR11], later studied by many others [MP12, FSSC16, RFS15, XDHK12]. The virtual network embedding problem is related to classic VPN graph embedding problems [EGOS05, GKR03, GOS08, GKK+01]. The VPN problem 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 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 guest graphs on a simple *line topology* (rather than tree-like topologies as studied in this thesis).

We consider a variant of virtual network embedding enriched by motivations that follow from batch-processing applications. Existing virtual cluster embedding algorithms often ignore a crucial dimension of the problem, namely data locality: an input to a batch-processing application such as MapReduce is typically stored in a distributed, and sometimes redundant, file system. Since moving data is costly, an embedding algorithm should be data locality aware, and allocate computational resources close to the data; in case of redundant storage, the algorithm should also optimize the replica selection. Note that our variant is not strictly a generalization of a virtual network embedding, as we took simplifing assumptions about the traffic demands.

In Chapter 3, we study an online balanced partition 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 best node assignment to $\ell$ clusters, is known as the *$\ell$-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 inaproximability of the static variant for general values of $\ell$ motivated research on the bicriteria variant, which can be seen as the offline counterpart of our cluster-size augmentation

approach. Here, the goal is to develop $(\ell, \delta)$-balanced graph partitioning, where the graph has to be partitioned into $\ell$ components of size less than $\delta \cdot (n/\ell)$ and the cost of the cut is compared to the optimal (non-augmented) solution where all components are of size at most $k$. The variant where $\delta \geq 2$ was considered in [LMT90, ST97, ENRS00, ENRS99, KNS09]. So far, the best result is an $O(\sqrt{\log n \cdot \log \ell})$-approximation by Krauthgamer et al. [KNS09].

Our model is related to online paging [ST85a, 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) [ST85b]. In our setting, requests can be served remotely (i.e., without fetching the corresponding nodes to a single cluster). In this light, our model is more reminiscent of caching models *with bypassing* [EILNG11, EILN15, Ira02]. As a side result, we show that our problem is capable of emulating online paging. There is a major difference between the caching problems and online partition problems. In the caching problem, the request relate to a sigle element of the universe. In contrast, in our model, *both* end-points 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 neighbouring network. To choose the appropriate port, the router stores of 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.

### 1.2.1 Forwarding Tables

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 milions of lookup operations and thousands updates (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 [PS06] are utilized. The TCAM memory is an associative memory storage that enables hardware supported pattern-matching lookup that closely matches the way the forwarding rules are used.

### 1.2.2 Growth of the Internet

Certain routers are located near the core of the Internet, and forward packets among large number of networks and have the knowledge about virtually all Internet networks. Their forwarding tables are usually refered to as *global forwarding table*. In Figure 1.2, we see the growth of the number of entries in the global forwarding table [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 superlinear 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 partitioning of the Internet into subnetworks, and to further growth of forwarding tables.

In the last years, the size of forwarding table 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 the result, it is expensive to expand available memory.

Limited size of memory and expanding size of the content to store brings new challenges in memory management of the routers. In Chapter 4, we investigate the following problem, using the tools provided by contemporary routers:

*How to efficiently manage the memory of forwarding devices?*

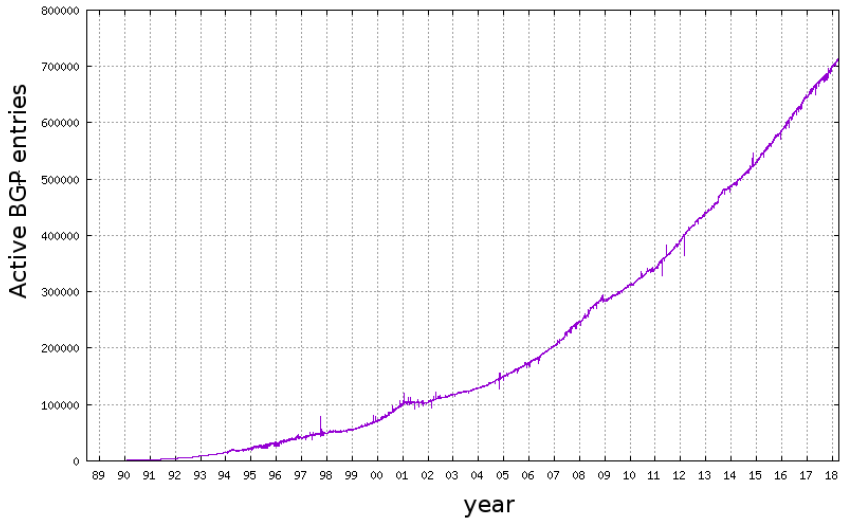


**Figure 1.2:** The number of entries in the global forwarding table. The global forwarding table is built upon the informations exchanged via Border Gateway Protocol. The graph presents the growth of the global forwarding table from 1988 to 2018.

### 1.2.3 Our Contributions

In Chapter 4, we study a novel solution for management of growing set of forwarding rules. This approach 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 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 setting. 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 IP routers.

Although this setting resemble 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 problem, 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 cache without its child rules. In other words, every valid cache configuration is a subforest. We elaborate more in Chapter 4.

We present a deterministic online algorithm for cache management with hierarchical dependencies. We prove that the algorithm 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 available cache. Our result is optimal up to the factor $O(h(T))$: we show that the lower bound for the paging problem [ST85b] implies an $\Omega(k)$ lower bound for our problem. In addition, we consider the online tree caching problem within the resource augmentation paradigm: we assume that cache sizes of the online algorithm ($k_{\mathrm{ONL}}$) and the optimal offline algorithm ($k_{\mathrm{OPT}}$) 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 the algorithm is not degraded if the model is enhanced to handle rule updates, that is an important aspect of router operation. Whenever the particular rule is updated, the algorithm incurs the cost if the rule is present in cache. Finally, we show that Alg can be implemented efficiently.

### 1.2.4 Related Work

So far, the papers on IP rule caching avoided dependencies 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 a large inflation of the routing table. A notable exception is a recent solution called CacheFlow [KARW16]. The CacheFlow model supports 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 tree 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 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 amount 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]. 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 the author of this thesis 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 PhD 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], and parts of Chapter 4 — in the proceedings of 29th ACM Symposium on Parallelism in Algorithmics and Architectures (ACM SPAA 2017) [BMP+17]. Preliminary results from Chapter 4 appeared in the master thesis of my co-author Aleksandra Spyra.

For some figures in this thesis, icons by Smashicons from www.flaticons.com were used.

# Part I

# Mapping Virtual Networks

# Chapter 2

# Virtual Networks with Static Topology

## 2.1 Problem Definition

As described informally in the introduction, the model combines three components: (1) the substrate network (the servers and the connecting physical network), (2) the input which needs to be processed (divided into data chunks), and (3) the virtual network (the virtual machines and the logical network connecting the machines to each other as well as to the chunks).

**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 will often refer to the elements in $S \cup R$ as the *vertices*. We will assume that the inter-connecting network forms an (arbitrary, not necessarily balanced or regular) tree, where the servers are located at the tree leaves. Each server $s \in S$ can host a certain number of virtual machines (available server capacity $\mathsf{cap(s)}$), and each link $e \in E$ has a certain bandwidth capacity $\mathsf{cap(e)}$.

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

**The Virtual Network.** The virtual network consists of a set $V$ of $n_V = |V|$ virtual machines, henceforth often simply called *nodes*. Each node $v \in V$ can be placed (or, synonymously, *embedded*) on a server; this placement can be subject to optimization.

Depending on the available capacity $\mathsf{cap(s)}$ of server $s$, multiple nodes may be hosted on $s$. We will denote the server $s$ hosting node $v$ by $\pi(v) = s$. Since these nodes process the input data, they need to be assigned and connected to the chunks. Concretely, for each chunk type $c_i$, exactly one replica $c_i^{(j)}$ must be processed by exactly one node $v$; which replica $c_i^{(k)}$ is chosen is subject to optimization, and we will denote by $\mu$ the assignment of nodes to chunks.

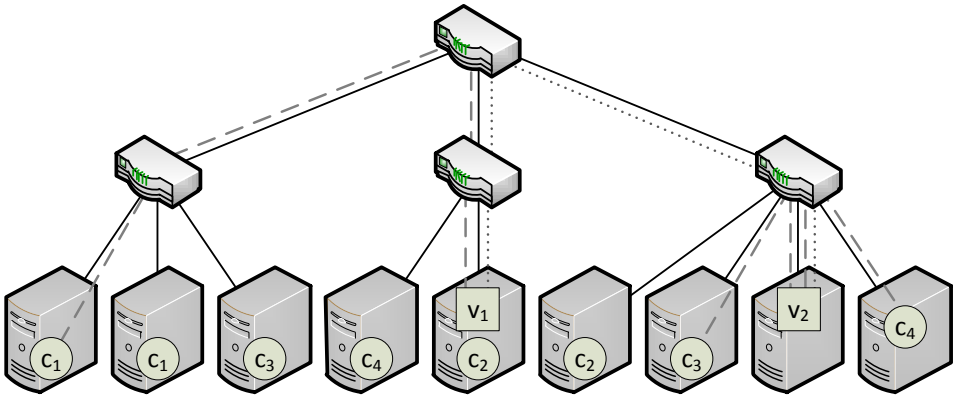


**Figure 2.1:** Overview: a 9-server datacenter storing $\tau = 4$ different chunk types $\{c_1, \ldots, c_4\}$ (depicted as *circles*). The chunk replicas need to be selected and assigned to the two virtual machines $v_1$ and $v_2$; the virtual machines are depicted as *squares*, and the network connecting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

In order to ensure a predictable application performance, both the connection to the chunks as well as the interconnection between the nodes may have to ensure certain minimal bandwidth guarantees; we will refer to the first type of virtual network as the *(chunk) access network*, and to the second type of virtual network as the *(node) inter-connect*; the latter is modeled as a complete network (a *clique*). Concretely, we assume that an active chunk is connected to its node at a minimal (guaranteed) bandwidth $b_1$, and a node is connected to any other node at minimal (guaranteed) bandwidth $b_2$. Figure 2.1 gives an overview of our model.

### 2.1.1 Optimization Objective

Our goal is to develop algorithms which accept and embed a request whenever this is possible, and minimize the *resource footprint*: the amount of resources which have to be dedicated to a request, in order to realize its guarantees. Essentially, the footprint captures the overall resource allocation, and is the most common objective function considered in the literature (a.k.a. as the min-sum objective guarantee) [FBB+13].

Formally, let $dist(v, c)$ denote the distance (in the underlying physical network $T$) between a node $v$ and its assigned (active) chunk replica $c$, and let $dist(v_1, v_2)$ denote the distance between the two nodes $v_1$ and $v_2$. We define the *footprint* $\mathrm{F}(v)$ of a node $v$ as follows:

$$\mathrm{F}(v) = \sum_{c \in \mu(v)} b_1 \cdot dist(v, c) \underbrace{+ \frac{1}{2} \cdot \sum_{v' \in V \setminus \{v\}} b_2 \cdot dist(v, v')}_{\text{only for inter-connect}},$$

where $\mu(v)$ is the set of chunks assigned to $v$. Our goal is to minimize the overall footprint $\mathrm{F} = \sum_{v \in V} \mathrm{F}(v)$.

### 2.1.2 Problem Decomposition

In order to chart the landscape of the computational tractability and intractability of different problem variants, we decompose our problem into its fundamental aspects, namely replica

selection (RS), multiple chunk assignment (MA), flexible node placement (FP), node interconnect (NI), and bandwidth constraints (BW), as described in the following. In this chapter, we will consider all possible 32 problem variants, where each of these five aspects can either be enabled or disabled.

**Replica Selection (RS).** The first fundamental problem is replica selection: if the input data is stored redundantly, the algorithm has the freedom to choose a replica for each chunk type, and assign it to a virtual machine (i.e., *node*). In the following, we will refer to a scenario with redundant chunks by RS; in the RS-only scenario, the number of chunk types is equal to the number of nodes. Otherwise, we will add the +MA property discussed next.

**Multiple Assignment (MA).** If the number of chunk types $\tau$ is larger than the number of nodes, each node needs to be assigned multiple chunks. We will refer to such a scenario by MA. Since all nodes are identical and no additional information regarding the chunks is available at request time, we assume that each node will process an identical integer number of chunks $m = \tau/n_V$.

**Flexible Placement (FP).** While the nodes are placed a priori in some cases, the node placement (or synonymously: *embedding*) of nodes on physical servers can also be subject to optimization. We will refer to this degree of freedom by FP.

**Node Interconnect (NI).** We distinguish between scenarios where bandwidth needs to be reserved both from each node to its assigned chunks as well as to the other nodes (i.e., $b_1 > 0$ and $b_2 > 0$), and scenarios where only the (chunk) access network requires bandwidth reservation (i.e., $b_1 > 0$ and $b_2 = 0$). We will refer to the former scenario where bandwidth needs to be reserved also for the inter-connect, by NI. The node interconnect is modelled as a complete graph, to account for the all to all communication patterns of batch processing applications such as MapReduce.

**Bandwidth Capacities (BW).** We distinguish between an uncapacitated and a capacitated scenario where the links of the substrate network come with bandwidth constraints, and will refer to the bandwidth-constrained version by BW; the capacity of servers (the number of nodes which can be hosted concurrently) is always limited. Note that capacity constraints introduce infeasible problem instances, where it is impossible to allocate sufficient resources to satisfy an embedding request.

## 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 *flow* (Section 2.2.1), *matching* (Section 2.2.2) and *dynamic programming* (Section 2.2.3) approaches. First, let us make a simplifying observation:

**Observation 1.** *In problems without flexible placement (*FP*), the bandwidth required for the inter-connect network (*NI*) can be allocated* upfront*, as it does not depend on the replica selection*

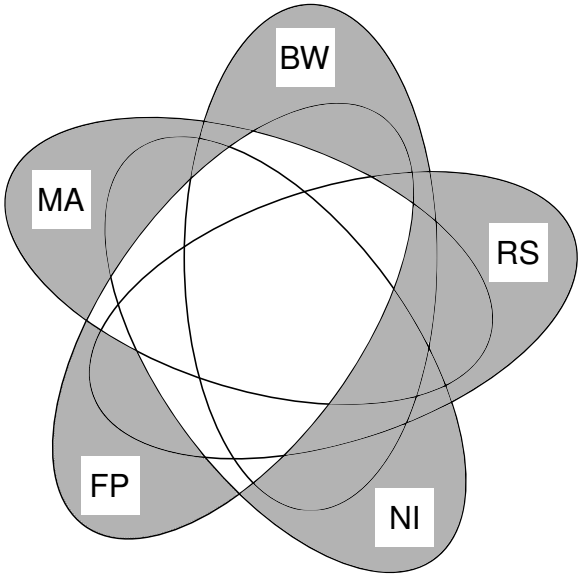


**Figure 2.2:** Variants solved by flow approach.

*and assignment. Accordingly, we can reduce problem variant* RS + MA + NI + BW *(as well as all its subproblems) to* RS + MA + BW *(resp. its subproblems).*

### 2.2.1 Flow Algorithms

We first present an algorithm to solve the RS + MA + NI + BW problem. Recall that in this problem variant, we are given a set of redundant chunks (RS) and a set of nodes (the *nodes*) at fixed locations (no FP). The number of chunk types is 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). Our goal is to minimize the resource footprint F, consisting of the bandwidth reservations in the (chunk) access network and the (node) inter-connect. As we will see in the following, we can use a flow approach to solve this problem variant.

**Construction of Artificial Graph.** In order to solve the RS + MA + NI + BW problem, we first remove the NI property using Observation 1. We then construct an artificial graph $T^*$, extending the substrate network $T$ and normalizing bandwidth capacities, as follows. For $T^*$, we normalize the bandwidth of $T$ to integer multiples of $b_1$, i.e., for each link $e \in E(T)$, we set its new capacity in $T^*$ to $\lfloor \mathsf{cap(e)}/\mathsf{b_1} \rfloor$. After this normalization, we extend the topology $T$ by introducing an artificial vertex for each chunk type. These artificial vertices are connected to each leaf (i.e., server) in $T$ where a replica of the respective chunk type is located, connecting the replica of the respective chunk type by a link of capacity 1. In addition, we create a *super-source* $s^+$, and connect it to each of the artificial chunk type vertices (with a link of capacity 1). Moreover, we create an artificial *super-sink* $s^-$ and connect it to every leaf containing at least one node; the link capacity represents the number of nodes $x$ hosted on this server, 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.

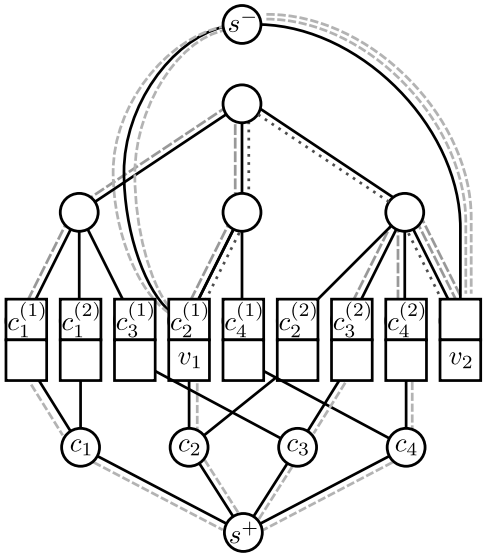


**Figure 2.3:** Example of flow construction: Problem instance with two nodes, four chunk types, and two replicas per type. The min-cost-max-flow is indicated by the dashed lines: each line represents one unit of flow.

A solution to the RS+MA+BW problem 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 $T^*$.

**Example.** Figure 2.3 shows an example of the extended substrate network $T^*$: The sink $s^-$ is connected to the two leaves, which host the nodes. The artificial nodes are depicted below the leaves, are labeled with their respective chunk types (e.g., $c_1$), and are connected to the source $s^+$ as well as to the leaves which contain replicas of their chunk type. The maximum flow with minimal costs 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.

**Algorithm.** Our algorithm to solve RS + MA + NI + BW consists of three parts: *First*, we construct the normalized and extended graph $T^*$ described above and compute a min-cost-max-flow solution, e.g., using [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, while algorithms like the successive shortest path algorithm [KK12] directly give us such an integral solution (in polynomial time), the fastest min-cost-max-flow algorithms (e.g., based on double-scaling methods [GT89] or minimum mean-cost cycle algorithms [Tar85], may yield fractional solutions which need to be rounded to integral solutions (of the same cost). In order to compute integral solutions, we proceed as follows: we iteratively pick an arbitrary (loop-free) path currently having a fractional allocation of value $f$ ($f > 0$), and distribute its flow $f$ among all other fractional paths of the same length; due to the optimality of the fractional solution and due to the integrality theorem, such paths must always exist. After distributing this flow, the total allocation on this path will be 0, and we have increased the number of integer paths by at least one. We proceed until we constructed the perfect matching. *Third*, given an integer min-cost-max-flow solution, we need to decompose the integer flow into the paths representing matched chunk-node pairs: The assignment can be obtained by decomposing the flow allocated

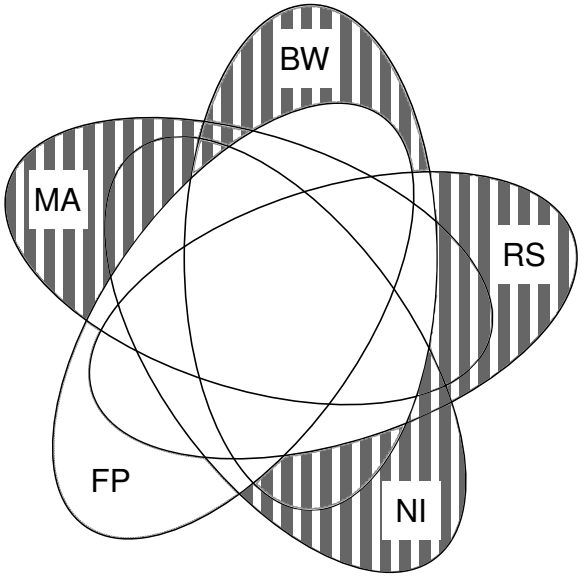


**Figure 2.4:** Variants solved by matching approaches.

in the original substrate network. In order to identify a matched chunk-node pair, we take an arbitrary (loop-free) path $p$ carrying a flow of value $\geq 1$ from $s^+$ to $s^-$: the first hop represents the chosen chunk type, the second hop the chosen replica, and the last but one hop represents the server: we will assign the replica to an arbitrary unused node on this server. Having found this pair, we reduce the flow along the path $p$ by one unit. We continue the pairing process until every chunk type is assigned.

**Analysis.** The correctness of our approach follows from our construction of $T^*$, using integer capacities (in our case $\lfloor \mathsf{cap(e)}/\mathsf{b_1} \rfloor$), and the fact that cost optimal integral solutions always exist [AMO93]. 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 state-of-the-art min-cost-max-flow algorithms [GT89, Tar85] we get a runtime of $\mathcal{O}(n_S^2 \cdot \log\log \min\{U, \tau\})$ where $U$ is the maximal link capacity; note that in networks with high capacity and uncapacitated networks, we can simply set $U = \tau$.

### 2.2.2 Matching Algorithms

This section presents faster algorithms to solve the two problem variants RS + MA + NI and MA + NI + BW which can also be solved with the flow approach introduced above. In general, we refer to the algorithms presented in this section as matching approaches.

#### RS + MA + NI

Let us first consider the RS + MA + NI variant. Recall that in this problem, we are given a set of redundant chunks (RS) and a set of nodes at fixed locations. The number of chunk types is larger than the number of nodes (MA), and each node needs to be connected to its chunks

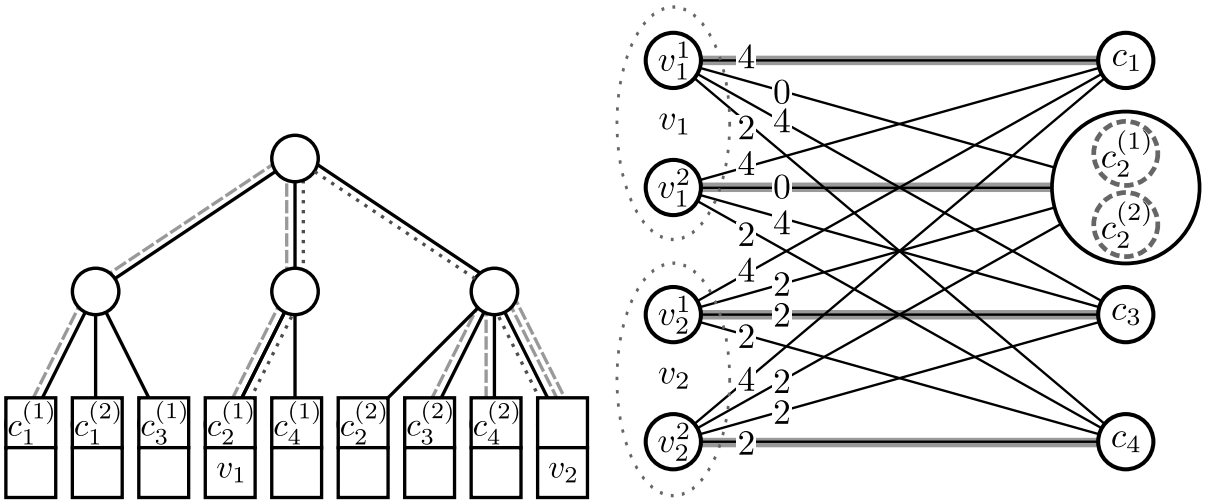


**Figure 2.5:** The RS + MA problem on the *left* is converted into a matching problem on the *right*. Since each node has to process two chunks, the nodes are replicated in the matching representation. The two replicas of each chunk type are represented by a single node, and all edges connecting to this node have a weight according to the shorter distance to one of the replicas. This is visualized for $c_2$.

as well as to other nodes (NI). Our goal is to minimize the resource footprint F, consisting of the bandwidth reservations in the access network and the inter-connect.

**Algorithm.** Due to Observation 1, RS + MA + NI degenerates to RS + MA. In order to solve the RS + MA problem variant, we construct a bipartite graph between the set $V$ of nodes and the set of chunks. Concretely, we clone each node $m$ times, as each node needs to process $m$ chunk types, and we collect all copies of a given chunk type in a single "super-node". We connect each node to all chunk types using the *lowest hop count* to one of the copies as the cost metric (the link weight). On the resulting bipartite graph, we can now compute a *Minimum Weight Perfect Matching* [Gab85]: the resulting matching describes the optimal assignment of chunks to nodes.

**Example.** Before analyzing our algorithm, let us consider a small example. Figure 2.5 illustrates an instance where two nodes are cloned into $m = 2$ nodes each, resulting in a total of four nodes in the matching problem representation. The two replicas of each chunk type are aggregated into a single chunk type vertex $c_j$ in the matching problem; this gives a total of four chunk type vertices in the matching graph. The costs on the links between all clones of a specific vertex and a chunk type are set to the minimum distance. We can observe this for instance at the edges connecting the two clones of $v_1$ to $c_2$: both weights are 0.

**Analysis.** The correctness of our algorithm follows from the construction and the optimal solution of the minimum matching. The runtime consists of two parts: the construction of the matching graph and the actual matching computation. The constructed graph consists of $m \cdot n_V \cdot \tau$ many edges, and for each edge we need to compute its cost, i.e., the shortest distance which in a tree can be computed in time $n_S$; thus, the overall construction time is $\mathcal{O}(n_S \cdot \tau^2)$. The state of the art algorithm to compute matchings are based on scaling techniques [DS12]. The runtime translates to $\mathcal{O}(\tau^{5/2} \cdot \log(\tau \cdot n_S))$; recall that $\tau = m \cdot n_V$.

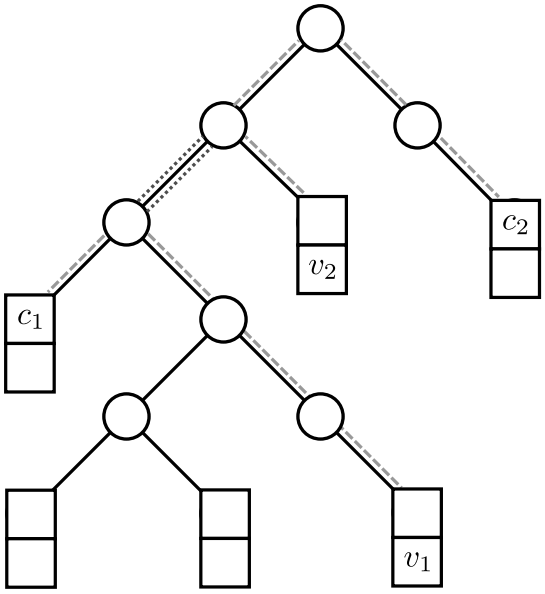


**Figure 2.6:** Illustration of local assignment: The dashed lines indicate bandwidth allocations, which occur independently of the chosen assignment. The dotted lines indicate bandwidth allocation which occur only if $c_2$ is assigned to $v_1$.

### Faster MA + NI and MA + NI + BW

We now show that we can solve MA + NI even faster, by exploiting locality. Moreover, we will show that we can even solve MA + NI + BW problem variants by simply verifying feasibility. In the following, due to Observation 1, we can focus on the MA resp. MA + BW problem.

We first introduce the following definition.

**Definition 1 (Local Assignment (LA)).** *We define an assignment $\mu$ to be* local in a specific subtree $T'$*, iff $\mu$ assigns the maximum number of chunks in the subtree to nodes in the same subtree. We define $\mu$ to be* local *when it is local with respect to all possible subtrees of the substrate network.*

**Example.** Figure 2.6 illustrates the concept of local assignment: The closest chunk to $v_2$ is $c_1$, and the closest node to $c_1$ is $v_2$. However, a subtree $T'$ exists such that $v_1 \in T'$ and $c_1 \in T'$, but $v_2 \notin T'$. Therefore, a local assignment cannot assign $c_1$ to $v_2$.

We will see later that optimal solutions to MA have a local assignment. We exploit this in our algorithms described in the following.

**Algorithm.** Our proposed algorithm for MA 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 $S_1, S_2$ in order to match unmatched chunks $S_1$ in the subtree $T'$ to unmatched nodes $S_2$ in $T'$. Both sets are initially empty.

We first process all the leaves, in an arbitrary order; subsequently, we process arbitrary inner vertices of $T$, whenever all their children have been processed. We process any leaf $\ell$ by adding any nodes or chunks which are located on $\ell$ to the corresponding sets $S_1$ and $S_2$. A non-leaf

vertex $u$ is processed in the following way: we take the union of the sets of $u$'s children, i.e., the sets contain the unmatched chunks and nodes in this subtree. For both leaves and inner nodes, whenever both sets are non-empty, we greedily match an arbitrary chunk in $S_1$ with an arbitrary node in $S_2$, and remove them from the sets.

**Analysis.** On a given vertex $u$, emptying one of the sets, results in a *local assignment* (cf Definition 1) in the subtree rooted at $u$. The bottom-up strategy ensures that this works for every subtree in the substrate, rendering the resulting assignment local. The complexity of this construction is low: For each vertex in the substrate graph, we build the union of the children's sets, and since each vertex can only be the child of one vertex, the amortized runtime per vertex is constant; and hence the overall runtime $\mathcal{O}(n_S)$. The sum of all remove operations, is equal to the number of chunk types $\mathcal{O}(\tau)$. Hence the overall complexity of this construction amounts to $\mathcal{O}(n_S + \tau)$.

It remains to prove optimality of such local assignments. By *uplink* of a subtree with root $r$ we denote the edge from $parent(r)$ to $r$ (if it exists). We first characterize the bandwidth allocation on uplinks of subtrees.

**Lemma 1.** *Given an* MA *problem and a subtree $T'$ containing $x$ chunks and $y$ nodes, the minimal bandwidth allocation of any assignment $\mu$ on the uplink of $T'$ is $|x - y \cdot m| \cdot b_1$.*

*Proof.* In case the number of chunk types equals the processing capacities of the nodes in the given subtree, the bandwidth allocation inflicted by the chunk access network on the uplink can be zero, since we can assign all chunks to nodes in the same subtree. Otherwise, we distinguish between two cases: Recall, that in instances without RS, all chunks have to be processed. In case there are more chunks in the subtree, at least all of the excess chunks have to be transferred to a different subtree, which will inflict costs $b_1$ per excess chunk on the uplink connecting $T'$ with the remaining parts of $T$, which will inflict costs $b_1$ per excess chunk on the uplink of root of $T'$. Similarly, if the processing capabilities exceed the amount of available chunks, excess chunks from other subtrees will have to be transferred to nodes in the subtree $T'$, inflicting bandwidth costs of $b_1$ each. Hence, the minimum bandwidth allocation for the chunk access on the uplink is the difference between the number of chunks and the processing capabilities of the subtree $|x - y \cdot m|$ times the amount of bandwidth needed, for a single transfer $b_1$. □

**Theorem 1.** *Given an* MA + NI *problem instance, a feasible assignment $\mu$ is optimal iff it is local.*

*Proof.* Local assignments generate exactly the minimal allocations on all links, as the assignments which generate the minimal bandwidth allocations described in the proof of Lemma 1 are local in the given subtree. Hence each local assignment has to be optimal. A non-local assignment, has at least one subtree, in which it is not local. This subtree will have a higher allocation on the uplink. Since the local assignment has minimal allocations on all other links, the non local assignment has a larger footprint. □

Combined with a simple postprocessing step, this approach can also solve MA + BW. The central idea of this extension, is that *local* assignments allocate the minimal bandwidth on each

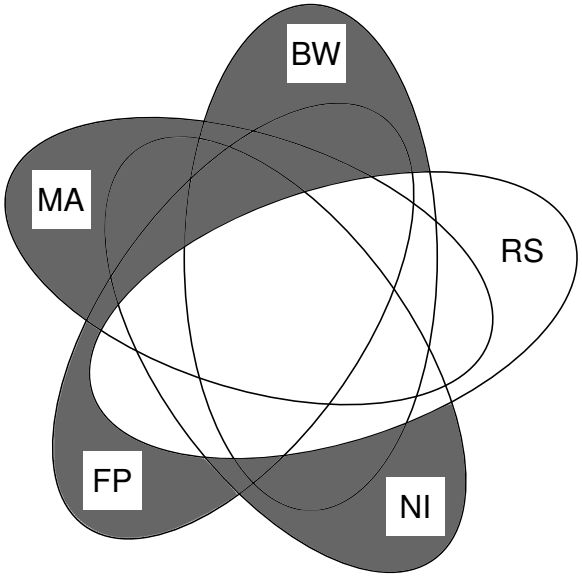


**Figure 2.7:** Variants solved by dynamic programming approach.

individual edge. In consequence, each bandwidth constraint which is lower than the allocation of a local assignment on one link, 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 postprocessing 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 problem variant in polynomial time. Note that this problem 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.

**Example.** Figure 2.8 shows an example: one extreme solution is to minimize the distance between chunks and nodes, see mapping $\pi_1$ in Figure 2.8 *(left)*: 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 has 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 $20 \cdot b_2$.

Figure 2.8 *(right)* shows a different node mapping $\pi_2$, which seeks to minimize the communication costs between the nodes, and places all nodes in one subtree. The distance between all nodes is two, 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 communicated to $v_3$ and $v_4$, which requires a total bandwidth allocation of $8 \cdot b_1$.

**Basic ideas.** Our proposed approach is based on dynamic programming, and leverages the *optimal substructure property* of MA + FP + NI + BW: as we will see, optimal solutions for subproblems (namely subtrees) can efficiently be combined into optimal solutions for larger

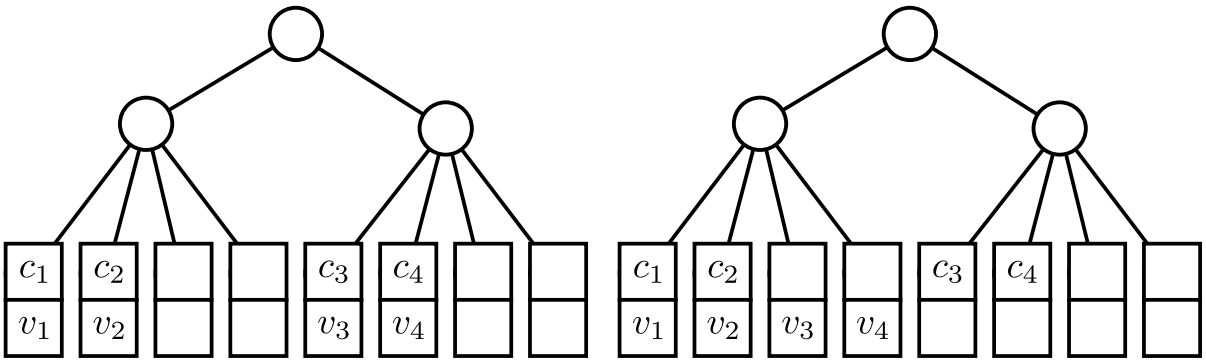


**Figure 2.8:** Two different node placements for the same substrate graph and chunk locations. For $b_1 = b_2$, both solutions have an identical footprint. In other cases, one solution outperforms the other.

problems. Indeed, the MA + FP + NI + BW problem exhibits such a structure, and we show how to exploit it to compute efficient embeddings, even in scenarios where multiple chunks need to be assigned to flexibly placeable nodes.

For ease of presentation we will transform the substrate network $T$ into a binary tree, using binarization: we clone every higher-degree node, iteratively attaching additional clones as right children and original children as left descendants.

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 does not matter, due to symmetry arguments. Our approach is to evaluate this function in a bottom-up manner. To finally compute the actual optimal embedding, we traverse the computed minimal-cost path backwards (according to the optimal values found for $f$ during the bottom-up computation).

Concretely, the first argument to function $f$ is a subtree $T'$, containing a given number of chunks $y(T')$, and the second argument is the number of nodes to be embedded in the subtree. Function $f$ is evaluated in a bottom up manner. We initialize the function at each leaf $\ell$, by $f(T_\ell, x) = \infty$ for all numbers of nodes $x$ which are larger than the server capacity $\mathsf{cap}(\ell)$; to calculate $f(T_\ell, x)$, for $x \leq \mathsf{cap}(\ell)$, we compute the bandwidth allocation on the uplink of $T_\ell$, referred to by the function $bw(T_\ell, x)$: $bw(T_l, x) = b_1 \cdot \quad |x - y(T_\ell)| + b_2 \cdot (n_V - x) \cdot x$, which accounts for the bandwidth allocation on the uplink of $T_\ell$. The first term represents the required bandwidth for the communication between the $x$ nodes on $\ell$, and the $n_V - x$ nodes in the remaining parts of the substrate network. The second term represents the bandwidth, which is necessary to transport the chunks from their location to the node which should process the data (see Lemma 1 for more details).

After initialization, we proceed to compute $f$ for non-leaf nodes in a bottom-up manner: We split the $x$ nodes into two positive integer values, and we put $r$ on the right and $x - r$ on the left subtree. 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 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 $T'$. Therefore, $f(T', x) = \min_{0 \leq r \leq x} \{f(\textsc{Le}(T'), x - r) + f(\textsc{Ri}(T'), r)\} + bw(T', x)$. Again, we set $f(T', x)$ to infinity if the required bandwidth $bw$ exceeds the capacity $\mathsf{cap}$ of the uplink of $T'$.

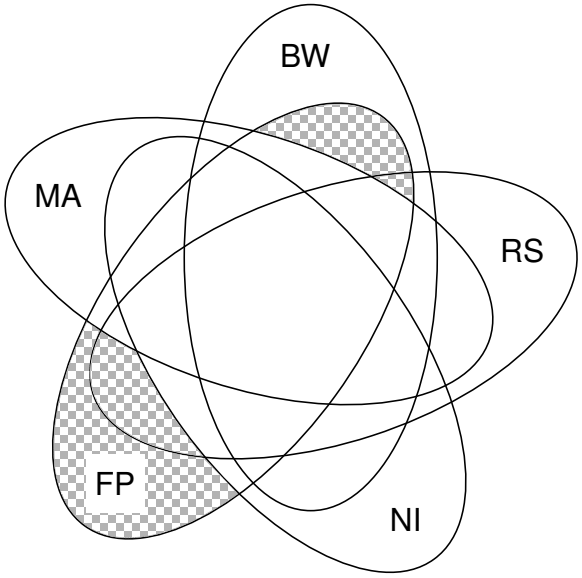


**Figure 2.9:** Trivially solvable problem variants.

**Analysis.** The correctness and optimality of our dynamic program is due to the decoupling of the costs induced by the tree structure of $T$ and the substructure optimality property. 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.

### 2.2.4 Simple Problems

For the sake of completeness, we also observe that there are several problems which allow for a trivial solution. Concretely, problems with FP plus any combination of RS and BW (but without MA and NI) can easily be solved by mapping nodes to chunk locations. Figure 2.9 shows a Venn diagram of the trivial property combinations.

## 2.3 NP-Hardness Results

We have seen that even problems 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 will show that problems become NP-hard if flexibly placeable nodes (FP) have to be assigned to one of multiple replicas ($RS$), either with multiple chunks per node (MA in Section 2.3.2) or with communication among nodes (NI in Section 2.3.3). Both results hold even in uncapacitated networks, and even in small-diameter substrate networks (namely two- or three-level trees [ALV08]). The hardness of FP + RS + MA and FP + RS + NI imply the hardness of four additional, more general models, as summarized in Figure 2.10:

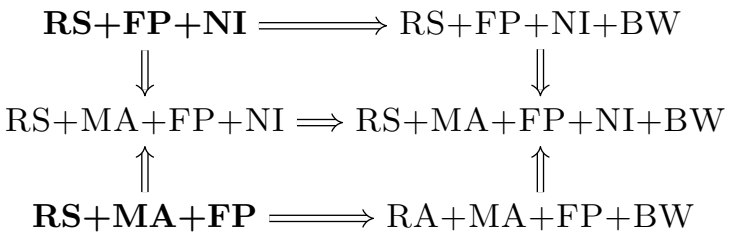


**Figure 2.10:** The NP-hardness of 2 variants implies the hardness of 4 other variants.

### 2.3.1 Introduction to 3D Perfect Matching

Both the hardness of FP + RS + MA and FP + RS + NI are shown by a reduction from the NP-complete problem of *3D Perfect Matching* [CKH+00], which we can see as a generalization of bipartite matchings to 3-uniform hypergraphs. We will refer to this problem by 3-DM, and for completeness, review it quickly: 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 $T \subseteq X \times Y \times Z$, and $t = |T|$. Set $M \subseteq T$ is a 3-dimensional matching if and only if, for any two distinct triples $t_1 = (x_1, y_1, z_1) \in M$ and $t_2 = (x_2, y_2, z_2) \in M$, it holds that $x_1 \neq x_2$, $y_1 \neq y_2$, and $z_1 \neq z_2$. Our goal is to decide if we can construct a $M \subseteq T$ which is *perfect*, that is, a subset which covers all elements of $X \cup Y \cup Z$ exactly once.

### 2.3.2 Hardness of Multi-Assignments

Our proof that FP + RS + MA is NP-hard is based on the following main ideas. We encode a 3-DM instance as an FP + RS + MA instance as follows:

- For every element in the universe $X \cup Y \cup Z$, we create a chunk type. Intuitively, in 3-DM, each element must be covered, which corresponds to the requirement of FP + RS + MA that each chunk type is processed.
- We will encode each triple as gadget with three leaves in a substrate tree $T$. The three leaves are close to each other in $T$, and the placement of chunk replicas in FP + RS + MA corresponds to the elements of the triples in these leaves.
- The node placement will correspond to the choice of triples, independently of which leaf the node is mapped to. A node will process its collocated chunk, as well as the chunks in other two leaves of the same gadget.
- In order to turn the optimization problem into a decision problem, we will use a cost threshold $\xi$. The cost threshold will be met by all assignments which assign all three chunks of each triple to a node which is collocated with one of the chunks. Assignments which connect a chunk to a node in a different triple, will have a larger footprint, and are considered to be infeasible.

**Construction.** Let $I$ be an instance of 3-DM with $t$ triples and set cardinality $k$ ($k = |X| = |Y| = |Z|$). We construct an instance $I'$ of FP + RS + MA as follows:

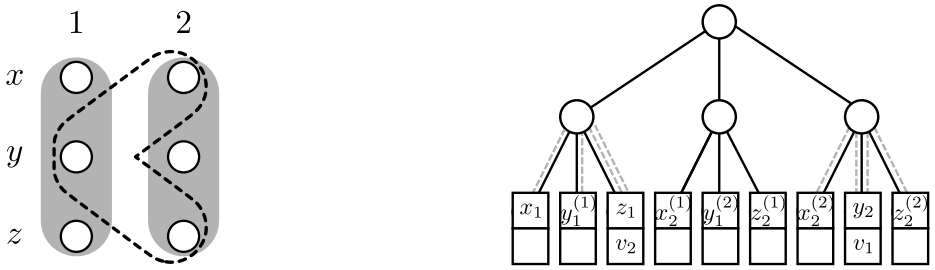


**Figure 2.11:** *Left:* A 3-DM instance with three triples: $(x_1, y_1, z_1)$, $(x_2, y_1, z_2)$, and $(x_2, y_2, z_2)$. The solution is indicated by the grey triples; the dashed triple is not used for the solution. *Right:* The corresponding problem and solution of FP + MA + RS.

- *Tree Construction:* We create a tree consisting of a root, and for each triple, we create a gadget which we directly attach as child of the root. The gadget is of height 2, and has the following form: The gadget of each triple consists of an inner node (a router) and three leaves.

- *Chunks and chunk replicas:* For each element in $X$, $Y$ and $Z$, we create a chunk type ($3 \cdot k$ in total). Every gadget contains three chunk replicas, corresponding to the elements of the triple. Each leave in a gadget, contains exactly one replica.

- *Other properties:* We set the number of to-be-embedded nodes to $k$, $b_1$ to 1, and the number of chunk slots in each node to the multi-assignment factor $m = 3$. We use a threshold $\xi = 4 \cdot k$.

**Example.** Figure 2.11 shows an example of our construction: An instance $I$ of 3-DM is given: The disjoint sets $X$, $Y$ and $Z$ have a cardinality $k = 2$. We will refer to the two elements in $X$ as $x_1$ and $x_2$, and use the same notation for the other two sets. $T$ contains the three triples $(x_1, y_1, z_1)$, $(x_2, y_1, z_2)$, and $(x_2, y_2, z_2)$. The goal of 3-DM is to find a subset $M \subseteq T$, which contains each element in each of the three sets exactly once. This instance only has one solution: $M = \{(x_1, y_1, z_1), (x_2, y_2, z_2))\}$.

To construct the corresponding instance $I'$ of FP + RS + MA, we create a gadget for each triple in $T$. For each variable which occurs in a triple, the corresponding gadget contains a chunk of the type of the variable. The triple $(x_2, y_1, z_2)$ of the instance is represented by the middle gadget in Figure 2.11. The objective of $I'$ is to spawn $k = 2$ nodes, with the smallest possible footprint. If the total footprint is at most $4 \cdot k$, we can construct a solution to $I$ from the solution to $I'$. The footprint consists of the costs which occur when a node is embedded in a gadget, and the three chunks of that gadget which are assigned to that node: one of the chunks is collocated with the node, the other two have to be transferred via two hops, inflicting unitary costs on each hop.

**Correctness.** Given these concepts, we can now show the computational hardness.

**Theorem 2.** FP + RS + MA *is NP-hard.*

*Proof.* Let $I$ be an instance of 3-DM and let $I'$ be an instance of FP + RS + MA constructed as described above. We prove that $I'$ has a solution of cost $\leq \xi$ if ($\Rightarrow$) and only if ($\Leftarrow$) $I$ has a matching of size $k$.

($\Rightarrow$) Let us take a feasible solution to 3-DM. We place a node in every gadget that corresponds to the chosen triples. In each of the corresponding gadgets, we match every chunk to the node in this gadget. This solution has cost exactly $\xi$. As every element of the universe is covered, every chunk type is processed.

($\Leftarrow$) Let us take a solution to FP + RS + MA of cost at most$\xi$. We choose triples that correspond to gadgets where there are nodes. Since all chunks are processed, every element of $X$, $Y$ and $Z$ is matched. Each node must process chunks that correspond to the triple, otherwise the cost must be larger than $\xi$ (high costs for chunk transportation). □

### 2.3.3 Hardness of Inter-connects

Next, we prove that the joint optimization of node placement and replica selection is NP-hard if an inter-connect has to be established between nodes. In our terminology, this is the FP + RS + NI problem.

The proof is similar in spirit to the proof of FP + RS + MA, however, we modify the construction to account for the absence of MA: we choose a high value for $b_1$, such that nodes will be directly collocated with their assigned chunks. We leverage the fact that any solution which does not assign 0 or 3 chunks to each gadget, will have higher communication costs.

**Construction.** Let $I$ be an instance of 3-DM with $t$ triples and set cardinality $k$ ($k = |X| = |Y| = |Z|$). We will create an instance $I'$ for FP + RS + NI as follows:

- We will construct the same tree as in previous reduction with chunk replicas placed in the same way.
- The communication cost in the inter-connect is set to $b_2 = 1$.
- The number of nodes (virtual machines) is $n_V = 3 \cdot k$, where $k$ is the set cardinality.
- Only solutions which place a node in each leaf of $k$ gadgets, can be converted into solutions for the 3-DM problem. We use the cost threshold $\xi = 6 \cdot k + 18 \cdot (k-1) \cdot k$, to verify whether a solution achieves this, transforming FP + RS + NI into a decision problem. A detailed explanation of this value can be found in the proof of Theorem 3.
- We set the access cost $b_1$ to a chunk replica to a high value $W$. This will force nodes to be collocated with the replica. One example of sufficient (and polynomial but not necessarily minimal) $W$ is the value of the threshold $\xi + 1$. Any solution not assigning chunks to collocated nodes, have cost $> \xi$: communicating a chunk inflicts costs $W = \xi + 1$ over every link.

We focus on instances with unit server capacities.

**Proof of correctness of the reduction.** Intuitively, in order to minimize embedding costs, nodes should be placed on near-by replicas. We use the following helper lemma.

**Lemma 2.** *In every valid solution of $I'$ of cost $\leq \xi$, each gadget falls in one of two categories: $k$ gadgets have exactly 3 nodes, and $t-k$ gadgets remain empty.*

*Proof.* Since $W$ is large enough, the $3 \cdot k$ nodes have to be placed directly on different chunks, resulting in 0 costs for the access network. Consider any pair of nodes communicating over the inter-connect; due to our construction, the communication cost for each such pair is either 2 hops (if they belong to the same gadget) or 4 hops (if they belong to different gadgets). The lemma then follows from the observation that $\xi$ is chosen such that it is never possible to distribute nodes among more than $k$ gadgets. □

**Theorem 3.** FP + RS + NI *is NP-hard.*

*Proof.* Let $I$ be an instance of 3-DM and let $I'$ be an instance of FP + RS + NI constructed as described above. We prove that $I'$ has solution of cost $\leq \xi$ if ($\Rightarrow$) and only if ($\Leftarrow$) $I$ has a solution.

($\Rightarrow$) In order to compute a solution for $I'$ given a solution for $I$, we proceed as follows. Given an exact covering set of triples $S = \{t_1, t_2, \ldots, t_k\}$, we place three nodes in each gadget that corresponds to every triple of $S$. Chunks are matched to the nodes which are located on the same server.

The solution has the following cost: (1) the communication cost inside a gadget is $2 \cdot \binom{3}{2}$, as every pair contributes two hops; (2) the communication cost from each gadget to all other gadgets is $4 \cdot 3 \cdot 3 \cdot (k-1)/2$, where the factor 4 is for the communication over 4 hops, the factor 3 corresponds to the number of nodes per gadget, and $3 \cdot (k-1)$ is the number of nodes in remote gadgets; as we count each pair twice, we need to divide by two in the end. Summing up over all $k$ gadgets, we get exactly $\xi$.

($\Leftarrow$) Given a solution for $I'$, we can exploit Lemma 2 to construct a solution for $I$. We know that in any solution of cost at most $\xi$, $k$ gadgets contain exactly 3 nodes. These gadgets correspond to a valid 3D Perfect Matching: exactly one replica of every chunk type is processed and hence every element is covered exactly once. □

## 2.4 A Detailed Study of Replica Selection Hardness

We have seen that replica selection flexibilities can render embeddings computationally hard. We will now provide a more detailed look at this hardness result and explore the minimal requirements for rendering replica selection hard. In particular, we will show that already two replicas for each chunk type are sufficient to introduce intractability.

Namely, we provide the NP-hardness results for two restricted variants of Virtual Cluster Embedding (Sections 2.4.1 and 2.4.2). We augment the RS variant of *VCEMB* problem in the following way: by RS($k$) we denote the problem where each chunk has the redundancy factor at most $k$. In Section 2.4.1 we provide the hardness result for RS(2) + MA + FP, and in Section 2.4.2 we provide the hardness result for RS(2) + FP + NI + BW.

Both problems are reduced from the problem 3DPM (see Section 2.3.1 with no further restrictions. The constructions are based upon the reduction of 3DPM to FP + RS + MA

(see Section 2.3.2) and the reduction of 3DPM to FP + RS + NI (see Section 2.3.3). However, in contrast to Section 2.3.3, in two replica variant without multiple assignment, we added the bandwidth constraints. It is currently unknown to the authors of this very paper, whether the hardness result holds without bandwidth constraints (namely, whether the problem RS(2) + FP + NI is NP-hard). The necessity for bandwidth constraints arises as to deal with restricted factor of replication, we need to introduce gadgets in the tree that makes the tree asymmetric. Introducing bandwidth constraints allows to control the number of nodes spawning in certain parts of the tree.

### 2.4.1 Two Replicas without Bandwidth Constraints

We now show that the 2-replica selection problem is even NP-hard without capacity constraints. In particular, we consider the problem variant RS(2) + MA(4) + FP with at most two replicas of each chunk type and assignment factor four. There are no capacity constraints on links.

Our construction consists of two major modifications to hardness result without replication factor restrictions (for that result, refer to Section 2.3.2).

**Unique chunks on the comb.** First, we provide the tools for restricting the placement of nodes in certain parts of the tree. In Section 2.3.2, due to symmetric structure of the tree, the carefully crafted threshold value allowed us to prove that e.g. no Triple Gadget ever had two or more nodes placed in it. We still use the threshold value as the placement mechanism, but in this section, due to the asymmetrical tree construction, we combine it with the concept of unique chunks on the comb (by *comb* we denote the balanced tree, where all non-root vertices have at most one child).

For an introduction to the concept of unique chunks, let us consider the following example. Suppose that within one $VCEMB$ construction, we would like to encode not one 3DPM instance, but two 3DPM instances: $M_1$ and $M_2$, with disjoint universe and different number of triples to be chosen: $n_1$ and $n_2$. We perform the following modifications to the encoding provided in Section 2.3.2. The multi-assignment factor grows by 1, that is the instance we construct is the RS + MA(4) + FP instance. We construct two subtrees $T_1$ and $T_2$, that correspond to $M_1$, resp. $M_2$; we construct two two-edge-level combs $C_1$ and $C_2$, with number of leaves $n_1$, resp. $n_2$. We attach $M_1$ and $C_1$ (resp. $M_2$ and $C_2$) to the common root and we name the resulting subtree $P_1$, resp. $P_2$. Next, we attach $P_1$ and $P_2$ to the common root. In the end, the height of the tree grew by 2. Finally, we populate both combs with unique chunks, and we set the number of to-be-placed nodes to $n_V = n_1 + n_2$. We modify the threshold to be the sum of the thresholds for constructions for $M_1$ and $M_2$ plus $4 \cdot (n_1 + n_2)$. The last substrate of the threshold value corresponds to transportation of the fourth chunk processed by each machine for the distance of four.

To see why the example indeed can solve two instances of 3DPM, we need the following observations. First, we claim that no node is ever placed in a comb. To prove this fact, we use the property of the comb that the leaves are highly separated, and the fact that each machine has to process 4 chunks. Next, we claim that the number of nodes spawned in $P_1$ (resp. $P_2$) is

$n_1$ (resp. $n_2$). To see this, consider any imbalance of the number of spawned nodes; notice that some chunks in the underpopulated comb are processed outside of their $P_i$ subtree, resulting in the solution that exceeds the threshold.

**Families of chunk types.** The second tool that we introduce allows us to express the redundancy of chunks without actually replicating chunks more than two-fold. For simplicity of introduction, we consider the scenario with no multi-assignment. For each chunk type $c$ with redundancy, we count the number of occurrences of replicas of such a chunk in the tree, and name it $r_c$. We replace the chunk type $c$ with $r$ chunk types, which we call the family $F_c$ of that chunk type. For each occurrence of replica of $c$, we replace it with a replica of any chunk type from the family (without repetitions). To this point, the redundancy factor was reduced from $r_c$ to 1. Now, we construct the gadget $G_c$ for chunk type $c$, which consists of $r_c$ leaves, each hosting the second replica of each chunk type from family $F_c$. We use the technique of unique chunks on the comb to constraint the number of nodes in $G_c$ to be exactly $r_c - 1$. We provide necessary additional $r_c - 1$ nodes to be placed. Hence, exactly 1 node is placed on a chunk type of family $F_c$ outside the gadget $G_c$, and exactly $r_c - 1$ nodes cover the remaining $r_c - 1$ chunk types inside gadget $G_c$. All chunk types are processed, the replication factor is reduced to 2, and the size of construction grows polynomially.

**Introduction to the reduction.** As we already stated, we modify the construction from Section 2.3.2. As a way to deal with replication, we use the families of chunk types using the unique chunks on the comb. We extend the construction of a gadget for chunk type with redundancy, by incorporating the fact that the multi-assignment factor is 4. For the construction to remain correct given such a multi-assignment factor, we introduce further chunks types with one chunk replica to place in the chunk gadget and use the excessive 3 data processing capacities.

**Construction.** For an arbitrary instance $I_{\text{3DPM}}$ of 3-DM we construct a RS(2)+MA(4)+FP instance $I_{\text{VCEMB}}$ the way described in the remainder of this section. Let $k = |X| = |Y| = |Z|$.

By $T$ we denote the set of all triples of $I_{\text{3DPM}}$, and let $t = |T|$. For each $e \in X \cup Y \cup Z$, by $T_e$ we denote the set of all triples that contain element $e$. Let $\deg(e) = |T_e|$, and note that $\sum_e \deg(e) = 3 \cdot t$.

We proceed with the construction as follows.

**Chunk types and replicas** We construct three sets of chunk types. The first set corresponds to elements of the universe (that is, $X \cup Y \cup Z$). The construction of such chunk types is similar to construction of chunk types in Section 2.3.2, but to take into consideration the restricted replication factor, we construct the familiy of chunk types (as described in the introduction to this section). Namely, for each element of universe $e$, we construct as many chunk types as there are occurences of $e$ in triples of $I_{\text{3DPM}}$. Each such chunk type has exactly two replicas.

The other two sets of chunk types has one replica, therefore those are called called *unique* chunks. We construct two types of unique chunks, distinguished by a different role in the construction. For unique chunks we simply co-notate the chunk type with chunk replica.

Formally, the construction of chunk types and replicas unfolds as follows:

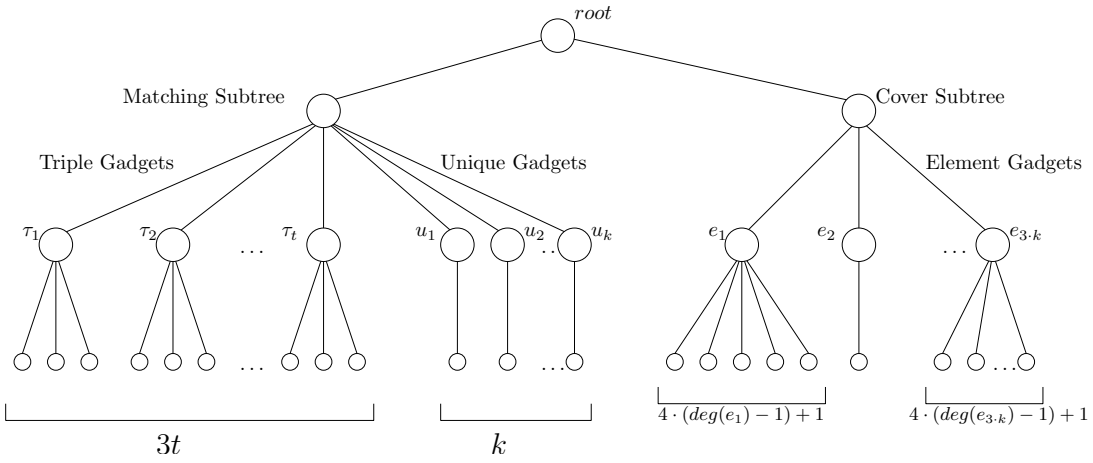


**Figure 2.12:** Overview of the substrate network.

1. For each triple $\tau \in T$, we construct 3 chunk types, with two replicas each. We construct different chunk types for each triple $\tau$, which contain element $e$ (in total $\deg(e)$ chunk types). We refer to those replicas by $ch_1(e, \tau)$ and $ch_2(e, \tau)$. In total we construct $2 \cdot \sum_e \deg(e) = 6 \cdot t$ chunk replicas.

2. We construct $k$ additional chunk types named $u_1, \ldots, u_k$ with one replica each.

3. For each element $e \in X \cup Y \cup Z$, we construct additional $3 \cdot (\deg(e) - 1)$ chunks, with one replica each. We call this set $\mathcal{U}_e$.

**Tree.** We construct the tree that has the following structure (see Figure 2.12):

1. The physical network consists of two subtrees connected to the root: the *Matching Subtree* and the *Cover Subtree*. The *Matching Subtree* consists of $t$ *Triple Gadgets*, one per each triple $\tau \in T$ and $k$ *Unique Gadgets*. The *Cover Subtree* consist of $3 \cdot k$ *Element Gadgets*, one for each element $e \in X \cup Y \cup Z$.

2. *Triple Gadget* consists of four vertices: three leaves and the root of the gadget.

3. *Unique Gadget* consists of two vertices: the leaf and the root of the gadget. We construct the root node of the gadget not only to keep the tree balanced, but also to keep leaves of *Unique Gadgets* far from leaves of other *Unique Gadgets*. Note that *Unique Gadgets* form a comb.

4. *Element Gadget* for element $e$ has a structure that depends on the number of triples that cover $e$. The *Element Gadget* consists of the root and $4 \cdot (\deg(e) - 1) + 1$ leaves.

**Chunk Placement.** The chunks are placed as follows:

1. *Chunks in the Matching Subtree:* In *Triple Gadget* of triple $\tau$ we put three replicas: $ch_1(e_X(\tau), \tau), ch_1(e_Y(\tau), \tau), ch_1(e_Z(\tau), \tau)$, one per each leaf.

2. *Chunks in the Unique Gadgets:* We place replicas $u_1, \ldots, u_k$ at the leaves of *Unique Gadgets*.

3. *Chunks in Element Gadgets:* Consider the *Element Gadget* for the element $e \in X \cup Y \cup Z$. We place two types of replicas in the leaves of the gadget. We put replicas $ch_2(\tau, e)$ for each $\tau \in T_e$. Additionally, we place all the replicas from set $\mathcal{U}_e$. In total, we place $4 \cdot (\deg(e) - 1) + 1$ replicas, one per each leaf of the gadget.

**Other properties of the instance.**

1. *Multiple assignment:* We set the multi-assignment factor to $m = 4$.

2. *Number of nodes:* We set the number of nodes to spawn to $n_V = k + \sum_e (\deg(e) - 1) = 3 \cdot t - 2 \cdot k$ nodes.

3. *Threshold:* We set the following threshold: $\xi = 18 \cdot t - 10 \cdot k$. This value corresponds to the cost of solution, where $k$ nodes are matched to 4 chunks that are in distance: $0, 2, 2$ and 4 to the node, and remaining $n_V - k$ nodes are matched to 4 chunks that are in distance: $0, 2, 2$ and 2 to the node.

**The reduction.**

Given any 3DPM instance $I_{\text{3DPM}}$, we produce corresponding instance of $VCEMB$ variant, namely the RS(2) + MA + FP instance, in the way described above. We refer to such RS(2) + MA + FP instance as the $I_{\text{VCEMB}}$.

The reduction (Theorem 4) unfolds in two stages. First, given a solution $S_{\text{3DPM}}$ to $I_{\text{3DPM}}$, we construct a solution $S_{\text{VCEMB}}$ to $I_{\text{VCEMB}}$. This part is the easier of the two, and mainly consists of placing nodes in Triple Gadgets for triples chosen in $S_{\text{3DPM}}$.

In the second stage, given $S_{\text{VCEMB}}$, we construct the $S_{\text{3DPM}}$. In this stage, the main difficulty lies in showing that $S_{\text{VCEMB}}$ has certain structure.

We call the *Triple Gadget active*, if it contains a node at any leaf, and we call the node *active* if it is placed in *Triple Gadget*. Our goal is to show that in every feasible solution, exactly $k$ *Triple Gadgets* are active (Lemma 5), and hence we can construct $S_{\text{3DPM}}$ from the triples that correspond to active *Triple Gadgets* in $S_{\text{VCEMB}}$.

In $I_{\text{VCEMB}}$, chunks can be matched to nodes at distance $0, 2, 4$ or 6. We call the matches at distance 0 the *free matches*, the matches at distance 2 the *neighbouring matches*. In addition we call the matches at distance 0 or 2 the *short matches*, and the matches at distance 4 or 6 the *long matches*. We call the distance between the pair of leaves the *short distance*, if the distance between them is at most 2, otherwise we call said distance the *long distance*.

Proving the existance of more than $k$ long matches is sufficient to show that the instance is infeasible, as its cost exceeds the threshold. To see this, note that the threshold value $\xi$ corresponds to the cost of solution, where $k$ nodes has 1 free match, 2 neighbouring matches and 1 long match at distance of 4 hops, and remaining $n_V - k$ nodes has 1 free match and 3 neighbouring matches assigned. Note that the limit of $n_V$ free matches is exhausted, hence excessive long matches cannot be compensated in any way. Hence, at most $k$ long matches are present in any feasible solution.

**Lemma 3.** *In* $S_{\text{VCEMB}}$ *there are have exactly* $k$ *nodes spawned in the* Matching Subtree.

*Proof.* We claim that each node spawned in the *Matching Subtree* results in at least one long match. This fact is a consequence of the structure of the tree and the fact that multi-assignment factor is set to 4. For each node spawned in the *Matching Subtree*, by the construction of the tree, the node has at most 3 leaves in short distance, hence at least one of the matches is long. Hence, we conclude that spawning more than $k$ nodes in the *Matching Subtree* results in more than $k$ long matches, which results in infeasibility of the solution. In addition, each node spawned in *Unique Gadget* results in at least 3 long matches, as the only leaf in short distance is the leaf collocated with the node.

As at most $k$ nodes are spawned in *Matching Subtree*, at least $n_V - k$ nodes are spawned in the *Cover Subtree*. Assume then that $n_V - k + i$ nodes spawned in the *Cover Subtree* for non-negative $i$. Now, we argue that such node configuration results in $3 \cdot i$ long matches. Consider the *Element Gadget* $g_e$ for element $e$. The gadget $g_e$ has exactly $4 \cdot (\deg(e) - 1) + 1$ leaves, each hosting exactly one chunk replica. As $4 \cdot (\deg(e) - 1) + 1 \mod 4 = 1$, spawnining $\deg(e) - 1 + j$ nodes in $g_e$ for non-negative $j$ results in at least $3 \cdot j$ long matches by the fact that there are insufficient chunk replicas in the short distance. Using the fact that $\sum_e (\deg(e) - 1) = 3 \cdot t - 3 \cdot k = n_V - k$, by pidgeon-hole principle we conclude that indeed spawning $n_V - k + i$ nodes in the *Cover Subtree* results in at least $3 \cdot i$ long matches.

Consider the configuration with $n_V - k + i$ nodes spawned in the *Cover Subtree*, and $k - i$ nodes spawned in the *Matching Subtree*. Such configuration results in at least $2 \cdot i + k$ long matches, where $3 \cdot i$ long matches come from the excessive nodes spawned in the *Cover Subtree*, and $k - i$ long matches come from $k - i$ nodes in the *Matching Subtree*. Hence we deduce that $i = 0$, as otherwise the number of long matches would exceed $k$.

**Lemma 4.** *In* $S_{\text{VCEMB}}$ *no node spawned in* Unique Gadget.

*Proof.* By Lemma 3, exactly $k$ nodes spawned in the *Matching Subtree*. Suppose that out of $k$ nodes in the *Matching Subtree*, a non-negative number of nodes $j$ spawned in the *Unique Gadgets*. From the fact that each leaf of *Unique Gadget* has long distance to every other leaf, every node spawned in *Unique Gadget* result in at least 3 long matches. Hence, the total number of long matches is at least $k - j + 3 \cdot j$. Finally, for the solution to be feasible we allow at most $k$ long matches, therefore no node spawns in the *Unique Gadget*. □

**Lemma 5.** *In* $S_{\text{VCEMB}}$ *there are have exactly* $k$ *active* Triple Gadgets.

*Proof.* By Lemmas 4 and 3 we conclude that $k$ nodes spawned in the *Triple Gadgets*. As there are exactly 3 replicas in each *Triple Gadget*, spawning more than one node in a single *Triple Gadget* results in at least additional 3 long matches. Hence, exactly $k$ *Triple Gadgets* are active. □

**Lemma 6.** *In* $S_{\text{VCEMB}}$ *every chunk replica besides* $u_1, \ldots, u_k$ *is matched by a short match.*

*Proof.* By Lemma 5, exactly $k$ nodes are spawned in *Triple Gadgets*, and by Lemma 4 we deduce that chunks $u_1, \ldots, u_k$ are matched by long matches. As at most $k$ long matches are allowed for the solution to be feasible, remaining matches are short. □

**Theorem 4.** RS(2) + MA + FP *is NP-hard.*

*Proof.* Let's take any instance $I_{\text{3DPM}}$ of 3DPM. We show that $I_{\text{VCEMB}}$ has a solution of cost $\leq \xi$ if and only if $I_{\text{3DPM}} \in$ 3DPM (there exists a perfect 3D matching in $I_{\text{3DPM}}$).

($\Leftarrow$) Let's take any feasible solution $S_{\text{3DPM}}$ to $I_{\text{3DPM}}$. We construct a solution $S_{\text{VCEMB}}$ in the following way:

1. We place $k$ nodes in $k$ *Triple Gadgets* (one per gadget) that correspond to triples in $S_{\text{3DPM}}$. The choice of exact leaf of the gadget to place a node is arbitrary. We match each such node to 3 chunk replicas in the gadget it is placed, and we match 1 arbitrary, unmatched chunk replica in *Unique Subtree*.

2. In each *Element Gadget* that corresponds to element $e$, we place $\deg(e) - 1$ nodes and match them to arbitrary chunks in this gadget, which are not yet matched in any *Triple Gadget*.

We can observe that every chunk type was processed, exactly $k + \sum_e(\deg(e) - 1)$ nodes are spawned, and each of the nodes process exactly 4 chunk replicas. To see that indeed the produced solution do not exceed the threshold $\xi$, we sum up the total transportation cost. The $k$ nodes placed in *Triple Gadgets* have 1 free match and 2 neighbouring matches to chunk replicas within the *Triple Gadget*, and 1 long match of cost 4 (to some *Unique Gadget*). The remaining $n_V - k$ nodes placed in the *Cover Subtree* have 1 free match and 3 neighbouring matches. In total, the cost incurred is $8 \cdot k + 6 \cdot (n_V - k) = \xi$. Hence, the solution is indeed feasible.

($\Rightarrow$) Let's take any feasible solution $S_{\text{VCEMB}}$ to $I_{\text{VCEMB}}$ in the way described in the construction section. By Lemma 5, exactly $k$ *Triple Gadgets* are active. We construct the solution $S_{\text{3DPM}}$ from the set of triples that correspond to active *Triple Gadgets*.

It remains to show that $S_{\text{3DPM}}$ indeed matches every element of $X \cup Y \cup Z$. By Lemma 6, each match of $ch(e, \tau)$ for each $e \in X \cup Y \cup Z$ and each $\tau \in T$ is matched by a short match. Hence, each active node processes the 3 chunks that are placed in its *Triple Gadget*. In each *Element Gadget* for element $e$, one chunk $ch(e, \tau)$ for some $\tau \in T$ is not matched. Let's call this chunk instance $\gamma(e)$, and let's call $\gamma = \cup_e \gamma(e)$. Note that $|\gamma| = 3 \cdot k$. The set $\gamma$ is covered by *active nodes*, and hence the set of triples in $S_{\text{3DPM}}$ form a 3D Perfect Matching of $X \cup Y \cup Z$.□

### 2.4.2 Two replicas without Multiple Assignment

We now show that RS(2) + FP + NI + BW is even NP-hard without multiple assignment. The proof is similar in spirit to proof of hardness of RS(2) + FP + MA.

The reduction (Theorem 4) unfolds in two stages. First, given a solution $S_{\text{3DPM}}$ to $I_{\text{3DPM}}$, we construct a solution $S_{\text{VCEMB}}$ to $I_{\text{VCEMB}}$. This part is the easier of the two, and mainly consists of placing nodes in Triple Gadgets for triples chosen in $S_{\text{3DPM}}$.

In the second stage, given $S_{\text{VCEMB}}$, we construct the $S_{\text{3DPM}}$. Again, we use the technique that we call "families of chunk types", which was introduced in previous section. The main technical difficulty lies in controlling the number of nodes that are spawned in certain parts of (asymmetric) tree. To guantee the desired number of spawned nodes, we use the bandwidth constraints. Namely, if the number of nodes to be spawned in a subtree is $k$, we set the bandwidth constraints on the uplink of the subtree to $k \cdot (m - k)$, where $m$ is the total number of machines to spawn in the instance. As we further see in Lemma 7, such bandwidth constraint in form of a quadratic expression provides both lower- and upper-bound on the number of machines spawned in such subtree. To see this, consider a simple example: regardless of the bandwidth constraint on the uplink of the subtree, capacities are not exceeded in at least two scenarios: with all $m$ nodes spawned in the subtree, and with 0 nodes spawned in the subtree. More precisely, bandwidth constraints in such form excludes configurations with number of nodes between $k$ and $m - k$.

However, we are interested only in lower-bounding the number of nodes to spawn in a subtree, and in fact the upper-bound on the number of nodes is only a liability. We make sure that the upper-bound on the number of nodes is always satisfied by artificially increasing the number of total nodes to be spawned. In this way the upper-bound on number of nodes always exceeds the number of leaves of any subtree in which we would like to have $k$ nodes spawned, see Lemmas 8 and 9. Additional nodes do not interfere with the rest of the construction, as we provide unique chunks for them to process.

**Contruction.** For an arbitrary instance $I_{\text{3DPM}}$ of 3-DM we construct a RS(2)+FP+NI+BW instance $I_{\text{VCEMB}}$ the way described in the remainder of this section. Let $k = |X| = |Y| = |Z|$. By $T$ we denote the set of all triples of $I_{\text{3DPM}}$, and let $t = |T|$. For each $e \in X \cup Y \cup Z$, by $T_e$ we denote the set of all triples that contain element $e$. Let $\deg(e) = |T_e|$, and note that $\sum_e \deg(e) = 3 \cdot t$.

We proceed with the construction as follows.

*Chunk Types.* We construct two sets of chunk types. The first set corresponds to elements of the universe (that is, $X \cup Y \cup Z$). The construction of such chunk types is similar to construction of chunk types in Section 2.3.3, but to take into consideration the restricted replication factor, we construct the familiy of chunk types (as described in the introduction to this section). Namely, for each element of universe $e$, we construct as many chunk types as there are occurences of $e$ in triples of $I_{\text{3DPM}}$. Each such chunk type has exactly two replicas.

The other set of chunk types has one replica, therefore those are called called *unique* chunks. For unique chunks we simply co-notate the chunk type with chunk replica.

Formally, the construction of chunk types and replicas unfolds as follows:

1. For each triple $\tau \in T$, we construct 3 chunk types, with two replicas each. We construct different chunk types for each triple $\tau$, which contain element $e$ (in total $\deg(e)$ chunk types). We refer to those replicas by $ch_1(e, \tau)$ and $ch_2(e, \tau)$. In total we construct $2 \cdot \sum_e \deg(e) = 6 \cdot t$ chunk replicas.

2. Additionally, we construct $\max\{3 \cdot t + 3 \cdot k + 1, \sum_e (2 \cdot \deg(e) - 1)\}$ chunk types called *unique chunks*. We refer to the set of unique chunks by $U$.

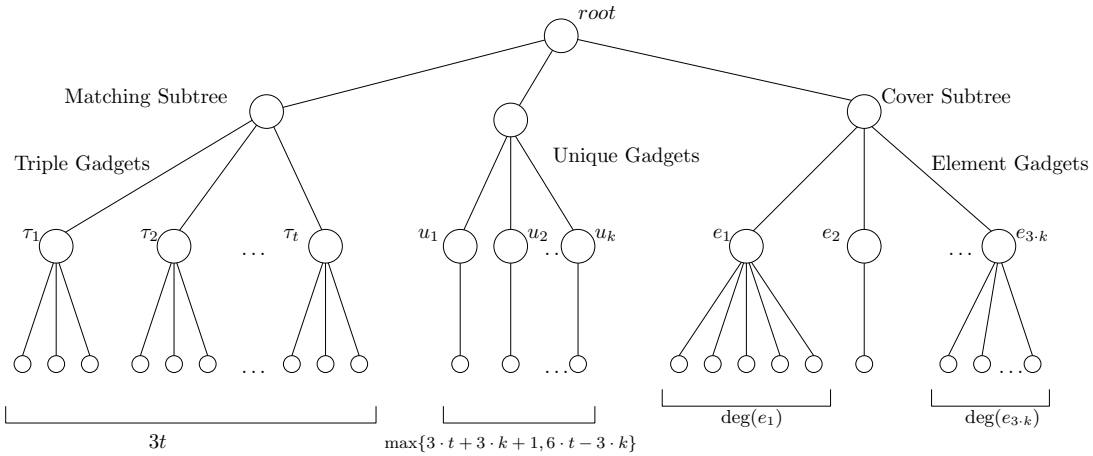


**Figure 2.13:** Overview of the substrate network.

*The substrate network.* We construct the tree that has the following structure (see Figure 2.13):

1. The physical network consists of three subtrees connected to the root: the *Matching Subtree*, the *Cover Subtree*, and a *Unique Subtree*. In the *Matching Subtree* we put $t$ *Triple Gadgets*. The *Cover Subtree* consist of $k$ element gadgets.

2. The *Unique Subtree* consist of $|U|$ leaves, and two middle nodes: a lower and an upper middle node. Note that this is different from RS(2) + FP + MA(4) NP-completeness proof, where *Unique Subtree* was placed in the *Matching Subtree*.

3. *Triple Gadget*: For each triple, we create a subtree consisisting of four vertices: three leaves and one triple root. We attach the root of the triple to the root of the matching subtree.

4. *Element Gadget*: For each element $e \in X \cup Y \cup Z$, we construct a subtree consisting of the root of the element (attached to the root of the cover subtree), and $\deg(e)$ leaves.

*Chunk placement.* The chunks are placed as follows:

1. *Chunks in the Matching Subtree:* In *Triple Gadget* of triple $\tau$ we put three replicas: $ch_1(e_X(\tau), \tau), ch_1(e_Y(\tau), \tau), ch_1(e_Z(\tau), \tau)$, one per each leaf.

2. *Chunks in the Unique Subtree:* We place replicas $U$ at the leaves of *Unique Gadgets*.

3. *Chunks in Element Gadgets:* Consider the *Element Gadget* for the element $e \in X \cup Y \cup Z$. We place two types of replicas in the leaves of the gadget. We put replicas $ch_2(\tau, e)$ for each $\tau \in T_e$. In total, we place $\deg(e)$ replicas, one per each leaf of the gadget.

*Bandwidth constraints.* We use bandwidth constraints of the form $\text{BW}(k) := k \cdot (n_V - k)$, where $n_V$ is the total number of nodes to be spawned in the instance. Namely, we set the bandwidth constraints of an uplink of an *Element Gadget* for each element $e$ to $\text{BW}(\deg(e) - 1)$, the bandwidth of an uplink of a *Matching Subtree* to $\text{BW}(n)$, and an uplink of a *Cover Subtree*

to $\text{BW}(\sum_e(\deg(e)-1))$. Note that out of $\deg(e)$ leaves of *Element Gadget* for element $e$, we allow to spawn $\deg(e)-1$ nodes.

*The threshold value and other properties of the instance.* We set the cost threshold for any solution to the following value:

$$
\begin{aligned}
\xi = {} & 2 \cdot \binom{3}{2} \cdot k && \text{(over 2 hops in the } \textit{Matching Subtree}\text{)} \\
& + 4 \cdot \binom{3 \cdot k}{2} && \text{(over 4 hops in the } \textit{Matching Subtree}\text{)} \\
& + 4 \cdot \binom{u}{2} && \text{(over 4 hops in the } \textit{Unique Gadgets}\text{)} \\
& + \sum_e 2 \cdot \binom{\deg(e)-1}{2} && \text{(over 2 hops in the } \textit{Cover Subtree}\text{)} \\
& + 4 \cdot \binom{\sum_e(\deg(e)-1)}{2} && \text{(over 4 hops in the } \textit{Cover Subtree}\text{)} \\
& + 6 \cdot \binom{n_V}{2} && \text{(over 6 hops)}
\end{aligned}
$$

where $n_V$ is the total number of nodes to be spawned in the instance, and $u = |U|$. We set $b_1$, the cost of chunk transportation to $\xi + 1$ (so that no chunk transportation happens in any feasible solution), $b_2 = 1$, and we host only one node per machine. We set the number of machines to spawn to: $n_V := 3 \cdot k + \sum_e(\deg(e)-1) + |U|$.

**Properties of the substrate network.**

**Lemma 7.** *Assume we have a* RS(2)+FP+NI+BW *instance $I$ with a subtree $T'$ with $l$ leaves and the bandwidth capacity on uplink of $T'$ is* $\text{BW}(k)$*. Assume that no chunk transportation is allowed ($b_1 = \infty$, so every node must be collocated with the chunk it processes in every feasible solution), and $b_2 = 1$. Then in any feasible solution the number $s$ of nodes spawned in $T$ satisfies $s \leq k \vee n_V - s \leq k$, and $s \leq l$.*

*Proof.* It holds that $s \leq l$ as we cannot spawn more nodes than leaves. The bandwidth allocation on the uplink of $T'$ is $\mathit{uplink}(s, T) := s \cdot (n_V - s)$, as no chunk transportation is allowed ($b_1 = \infty$), and every node in $T$ has to communicate over $T'$'s uplink with nodes spawned outside of $T'$. Therefore, in every feasible solution we have: $\mathit{uplink}(s, T') \leq \text{BW}(k)$. Let's define the remaining bandwidth on the uplink of $T'$ $\mathit{remainBw}(s) := \text{BW}(k) - \mathit{uplink}(s, T') = s^2 - s \cdot n_V - k^2 + k \cdot n_V$ . Every feasible solution fulfills $\mathit{remainBw}(s) \geq 0$, which is true for $s \leq k \vee n_V - s \leq k$ (follows from the properties of the quadratic function). □

Next, we show how to precisely control the number of nodes in the constructed subtree.

**Observation 2.** *In every feasible solution we have exactly $|U|$ nodes spawned in a* Unique Subtree *(no chunk transportation is allowed, and every chunk type must be processed).*

**Lemma 8.** *The following properties holds in $S_{\text{VCEMB}}$:*

1. *The number of nodes spawned in a* Matching Subtree *is* $3 \cdot k$.
2. *The number of nodes spawned in a* Cover Subtree *is* $\sum_e(\deg(e) - 1)$

*Proof.* From Observation 2 we know that we have $|U|$ nodes in the *Unique Subtree*. Let's refer to the number of nodes spawned in a *Matching Subtree* by $M$, and to the number of nodes spawned in *Cover Subtree* by $C$. By applying Lemma 7 to *Matching Subtree*, we know that: $M \leq 3 \cdot k \vee M \geq n_V - 3 \cdot k$. We observe that $n_V - 3 \cdot k$ is greater than the number of leaves in a *Matching Subtree*. By applying Lemma 7 to the *Cover Subtree* we know that: $C \leq \sum_e(\deg(e)-1) \vee C \geq n_V - \sum_e(\deg(e)-1)$. We observe that $n_V - \sum_e(\deg(e)-1)$ is greater than the number of leaves in the *Cover Subtree*. We also know that $n_V = |U| + C + M$. Therefore, by the pigeon-hole principle $C = \sum_e(\deg(e) - 1)$ and $M = 3 \cdot k$. □

**Lemma 9.** *In the solution* $S_{\text{VCEMB}}$*, the number of nodes spawned in Element Gadget of element* $e$ *is* $\deg(e) - 1$.

*Proof.* Let's call the number of nodes spawned in the Element Gadget of element $e$ the $x_e$. From Lemma 7, we know that $x_e \leq \deg(e)-1 \vee x_e \geq n_V - \deg(e)+1$. We observe that $n_V - \deg(e)+1$ is greater than the number of leaves of the gadget, which is $\deg(e)$. From Lemma 8, we know that the number of nodes spawned in the entire *Cover Subtree* is $\sum_e(\deg(e) - 1)$. Therefore, by the pigeon-hole principle, we have that $x_e = \deg(e) - 1$. □

From the above lemmas we know the precise number of nodes spawned in certain parts of the tree. Feasible solutions only differ in the choice of the $\deg(e) - 1$ out of $\deg(e)$ chunks in each Element Gadget, and the placement of nodes in the *Matching Subtree*.

Similar in spirit to the NP-completeness proof of RS(2) + MA(4) + FP, we call the *Triple Gadget* active if it contains exactly three nodes. Similarly, we call the *Triple Gadget* inactive if it does not contain spawned nodes, and *partially active* if it has one or two spawned nodes.

**Lemma 10.** *In* $S_{\text{VCEMB}}$*, we have exactly* $k$ *active* Triple Gadgets.

*Proof.* Since $I$ is feasible, we know that it has a solution $S$ of cost $\leq \xi$. By Lemma 8, we know that there are exactly $3 \cdot k$ spawned nodes in the *Matching Subtree*. Therefore, by the pigeon-hole principle, we know that we have at most $k$ active *Triple Gadgets*. It remains to show that there are no partially active *Triple Gadgets* in the solution of cost $\leq \xi$. Using Lemma 9, we conclude that the communication cost of nodes in the *Cover Subtree* is the same for every feasible solution (let's name that cost $P$). We also know that the communication cost between nodes in *Cover Subtree* and *Matching Subtree* is the same for every feasible solution (let's name it $Q$). Let's call the would-be cost of communication in the *Matching Subtree*, if there were exactly $k$ active gadgets, $R$. The threshold value was chosen so that $\xi = P + Q + R$. If we have at least one partially active gadget, then the cost of communication in *Matching Subtree* is greater than $R$, because we increase the number of 4-hop communications by at least one per each partially active gadget in comparison to a solution where we have exactly $k$ active gadgets. □

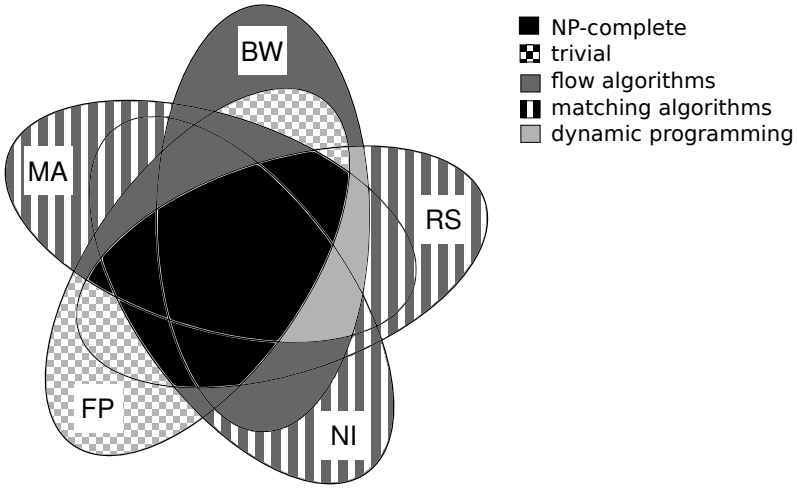


**Figure 2.14:** Fastest algorithms for different respective problem variants. Variants depicted by solid black are NP-hard, and variants depicted by checked filling are trivially solvable. For the remainder of variants we provide the fastest algorithm determined by the key.

**The reduction.** Using the properties of the substrate network, we perform the reduction in the following way.

**Theorem 5.** $\mathrm{RS}(2) + \mathrm{FP} + \mathrm{NI} + \mathrm{BW}$ *is NP-hard.*

*Proof.* Let's take any instance $I_{\mathrm{3DPM}}$ of 3DPM. We show that $I_{\mathrm{VCEMB}}$ has a solution of cost $\leq \xi$ if and only if $I_{\mathrm{3DPM}} \in$ 3DPM (there exists a perfect 3D matching in $I_{\mathrm{3DPM}}$).

Let's take an instance $I$ of 3DPM and construct an instance $I'$ of $\mathrm{RS}(2) + \mathrm{FP} + \mathrm{NI} + \mathrm{BW}$ in the way described above. We show that $I'$ has solution of cost $\leq \xi$ if and only if $I \in$ 3DPM (there exists a perfect 3D matching).

($\Leftarrow$) Let's take any feasible solution $S_{\mathrm{3DPM}}$ to $I_{\mathrm{3DPM}}$ and produce a solution $S_{\mathrm{VCEMB}}$ to $I_{\mathrm{VCEMB}}$ in the way described in the construction section. We show that the cost of $S_{\mathrm{VCEMB}}$ is indeed $\leq \xi$. For each triple $t \in T$ in $S_{\mathrm{3DPM}}$, we put 3 nodes at leaves of triple gadgets corresponding to those triples. In each element gadget (that corresponds to element $e$), we put $\deg(e) - 1$ nodes. In each element gadget there is only one leaf without the node placed in it: this node contains the chunk replica that is processed in the *Matching Subtree*. It is easy to see that $S_{\mathrm{VCEMB}}$ has cost exactly $\xi$ and no bandwidth constraint is violated. Each chunk type is processed.

($\Rightarrow$) Let's take any feasible solution $S_{\mathrm{VCEMB}}$ to $I_{\mathrm{VCEMB}}$ and produce a solution $S_{\mathrm{3DPM}}$ to $I_{\mathrm{3DPM}}$ by taking triples that correspond to active triple gadgets. Using Lemma 10, we conclude that there are exactly $k$ active triple gadgets. By feasibility of $S_{\mathrm{3DPM}}$, we know that each chunk type is processed. From Lemma 9, we know that out of $\deg(e)$ chunk types that correspond to $e \in X \cup Y \cup Z$, exactly one is processed in the *Matching Subtree*, hence each element of $X \cup Y \cup Z$ is matched. □

## 2.5 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 datacenters, many problems can be solved efficiently. However, we have also shown that several embedding problems are NP-hard already in two- and three-level trees—a practically relevant result given today's datacenter topologies [ALV08].

Our results are summarized in Figure 2.14. One interesting takeaway from this figure regards the question which properties render the problem NP-hard. For instance, we see that, BW does not influence the hardness of any problem 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

## 3.1 Problem Definition

We indroduce BRP, the online *Balanced RePartitioning* problem, which is defined as follows. There is a set of $n$ nodes, initially distributed arbitrarily across $\ell$ 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 nodes 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 $\alpha$ is a parameter (an integer). For any algorithm Alg, we denote its total cost (consisting of communication plus migration costs) on sequence $\sigma$ by $\textsc{Alg}(\sigma)$.

The description of some algorithms (in particular the ones in 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.

We are in the realm of competitive worst-case analysis and compare the performance of an online algorithm to the performance of an optimal offline algorithm. Formally, let $\textsc{Onl}(\sigma)$ resp. $\textsc{Opt}(\sigma)$ be the cost induced by $\sigma$ on an online algorithm Onl resp. on an optimal offline algorithm Opt. In contrast to Onl, which learns the requests one-by-one as it serves them, Opt has a complete knowledge of the entire request sequence $\sigma$ ahead of time. The goal is to design online repartitioning algorithms that provide worst-case guarantees. In particular, Onl is said to be $\rho$-competitive if there is a constant $\beta$ such that for any input sequence $\sigma$ it holds that

$$\textsc{Onl}(\sigma) \leq \rho \cdot \textsc{Opt}(\sigma) + \beta \ .$$

Note that $\beta$ cannot depend on input $\sigma$ but can depend on other parameters of the problem,

such as the number of nodes or the number of clusters. The minimum $\rho$ for which ONL is $\rho$-competitive is called the *competitive ratio* of ONL.

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 $2\alpha$. 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 $\delta$-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 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 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 $\alpha$, 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 6.** 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 $\alpha$, 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 $\alpha$.

Second, observe that the graph induced by saturated edges always constitutes a forest. For the sake of contradiction, 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 $2\alpha$. 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 will be a saturated intra-cluster edge. The communication cost over such an edge incurred by OPT is at least $\alpha$ (it can be also strictly greater than $\alpha$ 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 $\sigma$. □

## 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 $\epsilon \geq \frac{1}{k}$ (i.e., the augmented cluster is of size at least $2k + 1$). CREP maintains a similar graph structure as the simple deterministic $O(k^2 \cdot \ell^2)$-competitive algorithm 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 deleted and replaced by 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 $S = \{c_1, c_2, \ldots, c_{|S|}\}$ (called *component-set*), by $w(S)$ we denote the total weight of all edges between nodes of $S$. Note that positive weight edges occur only between different components of $S$. We call a component-set *trivial* if it contains only one component; $w(S) = 0$ in such a 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 $S = \{c_{i_1}, \ldots, c_{i_{|S|}}\}$ is *mergeable* if $w(S) \geq (|S| - 1) \cdot \alpha$. 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 maximum number of components, breaking ties arbitrarily. Weights of all intra-$S$ edges are reset to zero, and thus intra-component edge weights are always zero. A mergeable set $S$ induces a sequence of $|S|-1$ *merge actions*: CREP iteratively replaces two arbitrary components from $S$ by a component being their union (this constitutes a single merge action).

3. *Delete action.* If the component resulting from merge action(s) has more than $k$ nodes, it is deleted and replaced by 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 delete 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 $S = \{c_1, \ldots, c_{|S|}\}$ that triggers $|S| - 1$ merge actions not followed by a delete 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 delete 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. It states that immediately after CREP merges (and possibly deletes) a component-set, no other component-set is mergeable. This property holds independently of the actual placement of components in particular clusters.

**Lemma 11.** *At any time $t$, after* CREP *performs its merge and delete actions (if any), $w(S) < \alpha \cdot (|S| - 1)$ for any non-trivial component-set $S$.*

*Proof.* We prove the lemma by an induction on steps. Clearly, the lemma holds before an input sequence starts as then $w(S) = 0 \leq \alpha - 1 < \alpha \cdot (|S| - 1)$ for any non-trivial set $S$. 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 $w(S) < \alpha \cdot (|S| - 1)$ for any non-trivial set $S$. Otherwise, at time $t$, CREP merges a component-set $A$ into a new component $c_A$, and then possibly deletes $c_A$ and creates singleton components from its nodes. We show that the lemma statement holds then for any non-trivial component-set $S$. We consider three cases.

1. Component-sets $A$ and $S$ do not share any common node. Then, $A$ and $S$ 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 $A$, and hence does not contribute to the weight of $w(S)$. By the inductive assumption, $w(S) < \alpha \cdot (|S| - 1)$ held right before time $t$. As $w(S)$ is not affected by CREP actions at step $t$, the inequality holds also right after time $t$.

2. CREP does not delete $c_A$ and $c_A \in S$. Let $X = S \setminus \{c_A\}$. Let $w(A, X)$ denote the total weight of all edges with one endpoint in $A$ and another in $X$. As CREP merged component-set $A$ and did not merge component-set $A \uplus X$, $A$ was mergeable ($w(A) \geq \alpha \cdot (|A| - 1)$),

while $A \uplus X$ was not, i.e., $w(A) + w(A, X) + w(X) = w(A \uplus X) < \alpha \cdot (|A| + |X| - 1)$. Therefore, $w(A, X) + w(X) < \alpha \cdot |X|$ right after weight $w_{x,y}$ is incremented at time $t$. Observe that when component-set $A$ is merged and all intra-$A$ edges have their weights reset to zero, neither $w(A, X)$ nor $w(X)$ is affected. Therefore after Crep merges $A$ into $c_A$, $w(S) = w(A, X) + w(X) < \alpha \cdot |X| = \alpha \cdot (|S| - 1)$.

3. Crep deletes $c_A$ creating singleton components $d_1, d_2, \ldots, d_r$ and some of these components belong to set $S$. This time, we define $X$ to be the set $S$ without these components ($X$ might be also an empty set). In the same way as in the previous case, we may show that $w(A, X) + w(X) < \alpha \cdot |X|$ after Crep performs merge and delete operations. Hence, at this time $w(S) \leq w(A, X) + w(X) < \alpha \cdot |X| \leq \alpha \cdot (|S| - 1)$. The last inequality follows as $S$ has strictly more components than $X$.

Since only one request is given at a time, and since all weights and $\alpha$ are integers, 11 immediately implies the following result.

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

### 3.3.3 Analysis: Lower Bound on OPT

For estimating the cost of Opt, we pick any input sequence $\sigma$ and we execute Crep on it. Then, we execute Opt on $\sigma$ and we analyze its cost in terms of the number of merges and deletions performed by Crep. We split 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 $S(c)$. For a singleton component, $\tau(c)$ is the time when the component that previously contained the sole node of $c$ was deleted; $\tau(c) = 0$ if $c$ existed at the beginning of input $\sigma$. We will use time 0 as an artificial time point that occurred before an actual input sequence.

For a non-singleton component $c$, we define $F(c)$ as the set of the following (node, time) pairs:

$$F(c) = \biguplus_{b \in S(c)} \{b\} \times \{\tau(b) + 1, \ldots, \tau(c)\} \quad .$$

Intuitively, $F(c)$ 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 $F(c_1)$ and $F(c_2)$ are disjoint. The union of all $F(c)$ (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 $(x, t) \in F(c)$ and $(y, t) \in F(c)$. Note that only the requests contained in $F(c)$ could contribute towards later creation of $c$ by Crep. In fact, by 1, the number of these requests that entailed an actual cost to Crep is exactly $(|S(c)| - 1) \cdot \alpha$.

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

**Lemma 12.** *Fix any component $c$ and partition $S(c)$ into a set of $g \geq 2$ disjoint component-sets $S_1, S_2, \ldots, S_g$. The number of communication requests in $F(c)$ that are between sets $S_i$ 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 $F(c)$ that were between sets $S_i$ and that *were paid* by CREP is $w(S(c)) - \sum_{i=1}^{g} w(S_i)$. It suffices to show that this amount is at least $(g-1)\cdot\alpha$. By 1, $w(S(c)) = (|S(c)|-1)\cdot\alpha$ and $w(S_i) \leq (|S_i|-1)\cdot\alpha$. Therefore, $w(S(c)) - \sum_{i=1}^{g} w(S_i) \geq (|S(c)|-1)\cdot\alpha - \sum_{i=1}^{g}(|S_i|-1)\cdot\alpha = (g-1)\cdot\alpha$. □

For any component $c$ maintained by CREP, let $Y_c$ denote 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 13.** *For any non-trivial component $c$, it holds that*

$$\textsc{Opt}(c) \geq (|Y_c|-1)\cdot\alpha - \sum_{b\in S(c)} (|Y_b|-1)\cdot\alpha \ .$$

*Proof.* Fix a component $b \in S(c)$ and any node $x \in b$. Let $\textsc{opt-mig}(x)$ 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: $\textsc{opt-mig}(b) = \sum_{x\in b} \textsc{opt-mig}(x)$ and $Y'_b = \bigcup_{x\in b} Y'_x$. Observe that $|Y'_b| \leq |Y_b| + \textsc{opt-mig}(b)$.

We aggregate components of $S(c)$ into component-sets called *bundles*, so that any two bundles have their nodes always in disjoint clusters. To this end, we construct a hypergraph, whose nodes correspond to clusters from $\bigcup_{b\in S(c)} Y'_b$. Each component $b \in S(c)$ defines a hyperedge that connects all nodes (clusters) that are in $Y'_b$. Now we look at connected components of this hypergraph (called *hypergraph parts* to avoid ambiguity). As the hyperedge related to component $b$ connects $|Y'_b|$ nodes, there are

$$\begin{aligned} B &\geq |\textstyle\bigcup_{b\in S(c)} Y'_b| - \sum_{b\in S(c)}(|Y'_b|-1) \\ &\geq |Y_c| - \textstyle\sum_{b\in S(c)}(|Y_b|-1) - \sum_{b\in S(c)} \textsc{opt-mig}(b) \end{aligned}$$

hypergraph parts. Each hypergraph part corresponds to a bundle consisting of components contained in clusters belonging to this part, i.e., the number of bundles is also $B$.

By 12, the number of communication requests in $F(c)$ that are between different bundles is at least $(B-1)\cdot\alpha$. Each such request is paid by OPT because, by the definition of

bundles, it involves a communication between two nodes which OPT stored in different clusters. Additionally, $\text{OPT}(c)$ involves $\sum_{b \in S(c)} \text{OPT-MIG}(b)$ node migrations in $F(c)$, and therefore $\text{OPT}(c) \geq (B-1) \cdot \alpha + \sum_{b \in S(c)} \text{OPT-MIG}(b) \cdot \alpha \geq (|Y_c| - 1) \cdot \alpha - \sum_{b \in S(c)} (|Y_b| - 1) \cdot \alpha$. □

**Lemma 14.** *For any input $\sigma$, let $\text{DEL}(\sigma)$ be the set of components that were eventually deleted by* CREP. *Then $\text{OPT}(\sigma) \geq \sum_{c \in \text{DEL}(\sigma)} |c|/(2k) \cdot \alpha$.*

*Proof.* Fix any component $c \in \text{DEL}(\sigma)$. Consider a tree $\mathcal{T}(c)$ which describes how component $c$ 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 single time by merging its children.

We now sum $\text{OPT}(b)$ over all components $b$ from $\mathcal{T}(c)$, including the root $c$ and the leaves $L(\mathcal{T}(c))$. The lower bound given by 13 sums telescopically, i.e.,

$$\begin{aligned} \textstyle\sum_{b \in \mathcal{T}(c)} \text{OPT}(b) &\geq (|Y_c| - 1) \cdot \alpha - \textstyle\sum_{b \in L(\mathcal{T}(c))} (|Y_b| - 1) \cdot \alpha \\ &= (|Y_c| - 1) \cdot \alpha \ , \end{aligned}$$

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 $\sum_{b \in \mathcal{T}(c)} \text{OPT}(b) \geq (\lceil |c|/k \rceil - 1) \cdot \alpha \geq |c|/(2k) \cdot \alpha$, where the second inequality follows because $c \in \text{DEL}(\sigma)$ and thus $|c| > k$.

The proof is concluded by observing that, for any two deleted components $c_1$ and $c_2$, the corresponding trees $\mathcal{T}(c_1)$ and $\mathcal{T}(c_2)$ do not share common components, and therefore $\text{OPT}(\sigma) \geq \sum_{c \in \text{DEL}(\sigma)} \sum_{b \in \mathcal{T}(c)} \text{OPT}(b) \geq \sum_{c \in \text{DEL}(\sigma)} |c|/(2k)$. □

### 3.3.4 Analysis: Upper Bound on CREP

To bound the cost of CREP, we fix any input $\sigma$ 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 $\text{SIZE}(m)$ we denote the size of the smaller component that was merged. For an artificial merge action, we set $\text{SIZE}(m) = 0$.

Recall that $\text{DEL}(\sigma)$ denotes the set of all components that become eventually deleted by CREP. Let $\text{FINAL}(\sigma)$ be the set of all components that exist when CREP finishes sequence $\sigma$. Note that $w(\text{FINAL}(\sigma))$ is the total weight of all edges after processing $\sigma$.

We split $\text{CREP}(\sigma)$ into two parts: the cost of serving requests, $\text{CREP}^{\text{req}}(\sigma)$, and the cost of node migrations, $\text{CREP}^{\text{mig}}(\sigma)$.

**Lemma 15.** *For any input $\sigma$, $\text{CREP}^{\text{req}}(\sigma) = |M(\sigma)| \cdot \alpha + w(\text{FINAL}(\sigma))$.*

*Proof.* The proof follows by an induction on all requests of $\sigma$. 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 the sum of all edge weights decreases exactly by $(s-1) \cdot \alpha$. Then, the value of both sides remain unchanged. □

**Lemma 16.** *For any input $\sigma$, with $(2+\epsilon)$-augmentation,*

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

*Proof.* If CREP has more than $2k$ nodes in cluster $V_i$ (for $i\in\{1,\ldots,\ell\}$), then we call this excess *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 $\sigma$ by $\textsc{ovr}^{\sigma}(V_i)$. It is sufficient to show the following relation for any sequence $\sigma$:

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

As the second summand of (3.1) is always non-negative, (3.1) will imply the lemma.

The proof will follow by an induction on all requests in $\sigma$. 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 $\sigma$ and we show it for sequence $\sigma\cup\{r\}$, where $r$ is some request.

We may focus on request $r$ that triggers component(s) migration 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 deleted 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 will 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., $\text{OVR}^\sigma(V_y) \geq (\epsilon/2)\cdot k \geq (\epsilon/2)\cdot|c_y|$. As component $c_y$ was migrated out of $V_y$, the number of overflow nodes in $V_y$ changes by

$$\Delta\text{OVR}(V_y) = -\min\{\,\text{OVR}^\sigma(V_y),\, |c_y|\,\} \leq -(\epsilon/2)\cdot|c_y| \ .$$

Therefore, the second summand of (3.2) is at most $(4/\epsilon)\cdot\alpha\cdot\Delta\text{OVR}(V_y) \leq -(4/\epsilon)\cdot\alpha\cdot(\epsilon/2)\cdot|c_y| = -2\cdot|c_y|\cdot\alpha$ as desired. □

### 3.3.5 Analysis: Competitive Ratio

In the previous two subsections, we related $\text{OPT}(\sigma)$ to the total size of components that are deleted by CREP (cf. 14) and $\text{CREP}(\sigma)$ to $\sum_{m\in M(\sigma)}\text{SIZE}(m)$, where the latter amount is related to the merging actions performed by CREP (cf. 16). Now we will link these two amounts. Note that each delete action corresponds to preceding real merge actions that led to the creation of the eventually deleted component.

**Lemma 17.** *For any input $\sigma$, it holds that*

$$\sum_{m\in M(\sigma)}\text{SIZE}(m) \leq \sum_{c\in\text{DEL}(\sigma)}|c|\cdot\log k + \sum_{c\in\text{FINAL}(\sigma)}|c|\cdot\log|c| \ ,$$

*where all logarithms are binary.*

*Proof.* We prove the lemma by an induction on all requests of $\sigma$. 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 $\sigma$ 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. Let $c_x$ and $c_y$ be merged components, with sizes $p = |c_x|$ and $q = |c_y|$, where $p \leq q$ without loss of generality. Due to such action, the right hand side of the lemma inequality increases by

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

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 delete action) and let $c_1, c_2, \ldots, c_g$ denote components that were merged to create component $c$ that was immediately deleted. 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 7.** *With augmentation at least* $2+\epsilon$, CREP *is* $O((1+1/\epsilon)\cdot k\cdot \log k)$*-competitive.*

*Proof.* Fix any input sequence $\sigma$. By 15 and 16,

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

Regarding a bound for $|M(\sigma)|$, we observe the following. First, if CREP executes artificial merge actions, then they are immediately followed by a delete 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 $\sum_{c\in\textsc{del}(\sigma)}|c|$. Second, if CREP executes a real merge action $m$, then $\textsc{size}(m)\geq 1$. Combining these two bounds yields $|M(\sigma)|\leq\sum_{m\in M(\sigma)}\textsc{size}(m)+\sum_{c\in\textsc{del}(\sigma)}|c|$. We use this inequality and later apply 17 to bound $\sum_{m\in M(\sigma)}\textsc{size}(m)$ obtaining

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

By 14, $\sum_{c\in\textsc{del}(\sigma)}|c|\cdot\alpha\leq 2k\cdot\textsc{Opt}(\sigma)$. This yields

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

where

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

To bound $\beta$, observe that the component-set $\textsc{final}(\sigma)$ contains at most $k\cdot\ell$ components, and hence by 11, $w(\textsc{final}(\sigma))<k\cdot\ell\cdot\alpha$. Furthermore, the maximum of $\sum_{c\in\textsc{final}(\sigma)}|c|\cdot\log|c|$ is achieved when all nodes in a specific cluster constitute a single component. Thus, $\sum_{c\in\textsc{final}(\sigma)}|c|\cdot\log|c|\leq\ell\cdot((2+\epsilon)\cdot k)\cdot\log((2+\epsilon)\cdot k)=O(\ell\cdot k\cdot\log k)$. 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 $\sigma$, which concludes the proof. □

## 3.4 Online Rematching

Let us now consider the special case where clusters are of size two ($k=2$, arbitrary $\ell$). 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 $\lambda$. 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; afterwards 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 $M^{\mathrm{GR}}$ ($M^{\mathrm{OPT}}$) denote the set of all edges $e = (u, v)$, such that $u$ and $v$ are collocated by GREEDY (OPT). Note that $M^{\mathrm{GR}}$ and $M^{\mathrm{OPT}}$ are perfect matchings on the set of all nodes.

For the analysis, 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 $M^{\mathrm{GR}}$ and $M^{\mathrm{OPT}}$ constitutes a set of alternating cycles: an alternating cycle of length $2j$ (for some $j \geq 1$) consists of $2j$ nodes, $j$ edges from $M^{\mathrm{GR}}$ and $j$ edges from $M^{\mathrm{OPT}}$, 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. We define the cycle-potential as

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

where $C$ is the number of all 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 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, 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 will show that for an appropriate choice of $\lambda$, $f$ and $g$, for all three stages described above the following inequality holds:

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

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

**Lemma 18.** *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 will now upper-bound the change in the edge-potentials. Let $e_1^{\text{old}}$ and $e_2^{\text{old}}$ be the edges that were removed from $M^{\text{OPT}}$ by the swap and let $e_1^{\text{new}}$ and $e_2^{\text{new}}$ be the edges added to $M^{\text{OPT}}$. 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 19.** *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 $\Delta\textsc{Greedy} + \Delta\Phi(e_c) = 0 = \Delta\textsc{Opt}$.

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 $\Delta\textsc{Greedy} + \Delta\Phi(e_c) = (f+1)\cdot w(e_c) \leq f+1 = (f+1)\cdot\Delta\textsc{Opt}$.

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

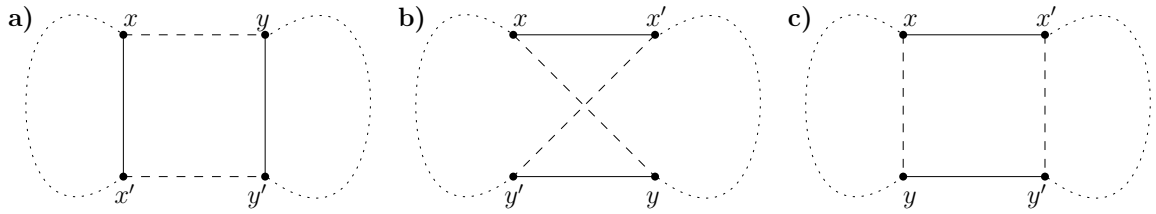


**Figure 3.1:** Three cases in the analysis of the third stage (a swap performed by Greedy). Solid edges denote edges that were removed from $M^{\mathrm{GR}}$ because of the swap, dashed ones denote the ones that were added to $M^{\mathrm{GR}}$. Dotted paths denote the remaining parts of the involved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

*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 $M^{\mathrm{GR}}$ before the swap and let $(x, y)$ and $(y, y')$ be the new edges in $M^{\mathrm{GR}}$ 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 $w^{\mathrm{S}}(e)$ and $\Phi^{\mathrm{S}}(e)$ denote the weight and the edge-potential of $e$ right before the swap. By the bounds above, $\Delta\textsc{Greedy} + \sum_{e \in E} \Delta\Phi(e) + \Delta\Psi = 2 \cdot \alpha - \Phi^{\mathrm{S}}(x, y) - \Phi^{\mathrm{S}}(x', y') + \Delta\Psi$, and hence to show (3.3) it suffices to show that the latter amount is at most $7 \cdot \Delta\textsc{Opt} = 0$. We consider three cases.

1. Assume that edges $(x, x')$ and $(y, y')$ were in different alternating cycles before the swap, see 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 $M^{\mathrm{OPT}}$. (It is possible that this edge is $(x, x')$; this occurs in the degenerate case when $C$ is of length 2.) As $M^{\mathrm{OPT}}$ 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, $\Delta\textsc{Greedy} + \sum_{e \in E} \Delta\Phi(e) + \Delta\Psi = (2 + g - f \cdot \lambda) \cdot \alpha \leq 0$.

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 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 $M^{\mathrm{OPT}}$, 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, $\Delta\textsc{Greedy} + \sum_{e \in E} \Delta\Phi(e) + \Delta\Psi = (2 - f \cdot \lambda) \cdot \alpha \leq (2 + g - f \cdot \lambda) \cdot \alpha \leq 0$.

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 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 $(x, y)$ and $(x', y')$ belong to $M^{\mathrm{OPT}}$. 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, $\Delta\textsc{Greedy} + \sum_{e \in E} \Delta\Phi(e) + \Delta\Psi = (2 - g + \lambda) \cdot \alpha \leq 0$. □

**Theorem 8.** *For $\lambda = 4/5$,* Greedy *is 7-competitive.*

*Proof.* We choose $f = 6$ and $g = 14/5$. The chosen values of $\lambda$, $f$ and $g$ satisfy the conditions of 19, 20 and 18. Summing these inequalities over all stages occurring while serving an input sequence $\sigma$ yields

$$\textsc{Greedy}(\sigma) + (\Psi_{\text{final}} - \Psi_{\text{initial}}) + \textstyle\sum_{e \in E} (\Phi_{\text{final}}(e) - \Phi_{\text{initial}}(e)) \leq 7 \cdot \textsc{Opt}(\sigma) \ ,$$

where $\Psi_{\text{final}}$ and $\Phi_{\text{final}}(e)$ denote the final values of the potentials and $\Psi_{\text{initial}}$ 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 $\sigma$. That is, $\Psi_{\text{final}} - \Psi_{\text{initial}} \geq -g \cdot \ell \cdot \alpha$ (as the number of alternating cycles is at most $\ell$) 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 paging, 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 Paging

**Theorem 9.** *Fix any $k$. If there exist a $\gamma$-competitive deterministic algorithm $B$ for BRP for two clusters, each of size $k$, then there exists a $\gamma$-competitive deterministic algorithm $P$ for the paging 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 paging problem, where $\sigma_t^P$ denotes the $t$-th accessed page. We show how to construct, in an online manner, an online algorithm $P$ for the paging problem that operates in the following way. It internally runs the algorithm $B$, starting on the initial assignment of nodes to clusters that will be defined below. For a requested page $\sigma_t^P$, it creates a subsequence of communication requests for the BRP problem, runs $B$ on them, and serves $\sigma_t^P$ on the basis of $B$'s responses.

We use the following $2k$ nodes for the BRP problem: paging 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 $k-1$ paging nodes, and the other cluster contains one paging node and all auxiliary nodes. There is a natural bijection between possible cache contents and well aligned configurations: the cache consists of the $k-1$ paging nodes that are in the same cluster as node $s$. (Without loss of generality, we may assume that the cache of any paging 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 paging node $p$, let $\textsc{comm}(p)$ be a subsequence of communication requests for the BRP problem, consisting ot the request $(p, s)$, followed by $\binom{k-1}{2}$ requests to all pairs of auxiliary nodes. Given an input sequence $\sigma^P$ for online paging, we construct the input sequence $\sigma^B$ for the BRP problem in the following way: For a request $\sigma_t^P$, we repeat a subsequence $\textsc{comm}(\sigma_t^P)$ till the node clustering maintained by $B$ becomes well aligned and $\sigma_t^P$ 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 paging request $\sigma_t^P$, the algorithm $P$ runs $B$ on $\textsc{comm}_t(\sigma_t^P)$. Right after processing $\textsc{comm}_t(\sigma_t^P)$, node configuration $c$ of $B$ is well aligned and $\sigma_t^P$ is collocated with $s$. Hence, $P$ may change its cache configuration to $\textsc{cache}(c)$: such a response is feasible because since $\sigma_t^P$ 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 node placement in clusters (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 $\sigma^P$ yields $2 \cdot \alpha \cdot P(\sigma^P) \leq B(\sigma^B)$.

Now we show that there exists an (offline) solution $\textsc{Off}$ to $\sigma^B$, whose cost is exactly $2 \cdot \alpha \cdot \textsc{Opt}(\sigma^P)$. Recall that, for a paging request $\sigma_t^P$, $\sigma^B$ 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 paging request $\sigma_t^P$. 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 $2\alpha$ (two paging 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 $\sigma^P$ yields $\textsc{Off}(\sigma^B) = 2 \cdot \alpha \cdot \textsc{Opt}(\sigma^P)$.

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

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

and therefore $P$ is $\gamma$-competitive. □

As any deterministic algorithm for the paging problem with cache size $k-1$ has a competitive ratio of at least $k-1$ [ST85a], we obtain the following result.

**Corollary 2.** *The competitive ratio of the BRP problem on two clusters is at least $k-1$.*

### 3.5.2 Additional Lower Bounds

**Theorem 10.** *No $\delta$-augmented deterministic online algorithm* 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 pair always exists. Such a request entails a cost of at least 1 for ONL. This way, we may define an input sequence $\sigma$ of an 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 that they will perform before the input sequence starts, the sum of their total costs on $\sigma$ 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 $\sigma$, the algorithm $\text{OFF}_j$ 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 $\sigma_t$, exactly one algorithm $\text{OFF}_j$ 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 offline algorithm $\text{OFF}_j$, 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 $\sigma$ grows. □

**Theorem 11.** *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 10, 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 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 will keep nodes $\{v_0, \ldots, v_3\}$ in the first two clusters and the remaining nodes in the remaining clusters (the remaining nodes will never change their clusters). More concretely, $\text{OFF}_1$ keeps nodes $\{v_0, \ldots, v_3\}$ always in configuration A and $\text{OFF}_2$ always in configuration B. Furthermore, we define the third algorithm $\text{OFF}_3$ 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, $\text{ONL}^{\text{req}}$, and the cost paid for its migrations, $\text{ONL}^{\text{mig}}$. Observe that, for any request $\sigma_t$, $\text{OFF}_1(\sigma_t) + \text{OFF}_2(\sigma_t) = \text{ONL}^{\text{req}}(\sigma_t)$. Moreover, as $\text{OFF}_3$ does not pay for any request and migrates at the same time as ONL does, $\text{OFF}_3(\sigma_t) = \text{ONL}^{\text{mig}}(\sigma_t)$. Summing up, $\sum_{j=1}^{3} \text{OFF}_j(\sigma_t) = \text{ONL}(\sigma_t)$ for any request $\sigma_t$. Taking into account the initial reconfiguration of nodes in $\text{OFF}_j$ solutions (which involves at most one swap of cost $2 \cdot \alpha$), we obtain that $\sum_{j=1}^{3} \text{OFF}(\sigma) \leq 2 \cdot \alpha + \text{ONL}(\sigma)$. 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 $\sigma$ grows. □

## 3.6 Conclusions

In this chapter we introduced a formal model for studying a placement of communicating virtual machines in a data center. For this model, we proposed and analysed an algorithm that uses minimal resource augmentation to structure groups of communicating virtual machines in a way that guarantees efficiency of moving the group to a different physical machine. This approach limits the total number of virtual machines in the data center in favor of reducing network usage by allowing efficient reconfiguration of interacting virtual machines. From the theoretical point of view, we designed the algorithm for the resource augmented scenario with competitive ratio $O(k \log k)$ (where $k$ is the capacity of a physical cluster), and we provided a lower-bound of $\Omega(k)$ for the competitive ratio of any deterministic algorithm that holds even in resource-augmented scenario. 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 a data center, in particular we neglect 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 arbitrary metric.

# Part II

# Managing Resources in Routers

# Chapter 4

# Caching of Routing Tables

In the second part of this thesis, we consider the problem of memory management in a single router. Modern routers consist of two logical components. Those two components use disjoint memory (often of distinct type) and disjoint processing units:

1. The forwarding plane, which contains forwarding table described in the previous section, and performs the actual packet transmission according to the forwarding table.

2. The control plane, which is responsible for reflecting the network topology in the forwarding table.

The forwarding component has finite memory, and in multiple scenarios the size of forwarding table exceeds the size of available memory. One of the solutions is to store only a part of a forwarding table on the forwarding device, that acts as a cache for the complete forwarding table that is stored in the control plane, see Figure 4.1. However, this might result in the situation, where the forwarding device do not posess sufficient information to proceed (*a cache miss*).

Even upon the cache miss, the packet still needs to reach its destination. The common way to handle the situation is to redirect the packet to the control plane component, which posseses the complete forwarding table. However, the packet cannot be forwarded by the control plane, hence it is supplemented with forwarding informations and sent back to the forwarding plane. The forwarding device overrides the usual procedure of forwarding table lookup, and instead it uses the supplemented information to forward the packet.

Upon the forwarding table cache miss, the control plane may decide to update the cached portion of the forwarding table that resides in the forwarding component. Standard caching-related problems arise, such as: which entry to evict to provide space to store the new entry. 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. 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).

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.

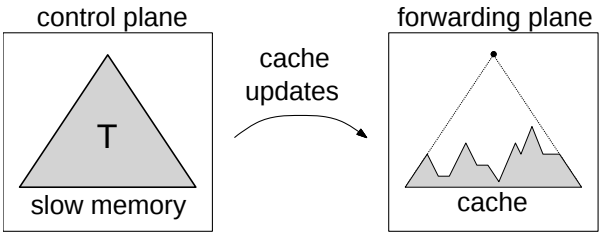


**Figure 4.1:** We propose a caching scenario, where the limited memory of a router stores a portion of the forwarding table. The complete forwarding table is stored at the controller.

Note that the technical feasibility of this solution heavily depends on the rule dependencies. In the most ubiquitous scenario, the rules are prefixes of IP addresses (they are bit strings). Whenever a packet arrives, the router follows a 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[1], then the tree is traversed from the root downwards, and the last found rule is used. This explains why we require the cached nodes to form a subforest: leaving a less specific rule on the router while evicting a more specific one (i.e., keeping a tree node in cache while evicting its descendant) will result in a situation where packets will be forwarded according to the less specific rule, and hence potentially exit through the wrong port. The LMP scheme also ensures that the described approach is implementable: one could simply add an artificial rule at the tree root in the router (matching an empty prefix). This ensures that when no actual matching rule is found in the router (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.

In contemporary networks, the control plane component can be physically separated from the forwarding plane, and to perform remote management over the network it uses protocols such as OpenFlow [MAB+08]. In such scenario, excessive cache updates cause not only delay in packet processing, but also cause congestion in the network between the forwarding component and the control component.

Our formal model is a novel variant of competitive paging, a classic online problem. Caching scenarios are best expressed and analysed in online settings. Similarly to Chapter 3, we use the competitive analysis to determine the performance of our strategy [BE98]. In this thesis, we consider the variant of caching with dependencies among cached elements motivated by the structure of forwarding table. In the framework of the competitive analysis, the paging problem was first analyzed by Sleator and Tarjan [ST85b], who presented $k$-competitive algorithms (where $k$ is the cache size) and a matching lower-bound. The simple paging problem was later generalized to allow different fetching costs (weighted paging) [CKPV91, You94] and additionally different item sizes (file caching) [You02], with the same competitive ratio. To the best of our knowledge, the variant of caching, where fetching items to the cache is not allowed unless some other items are cached (e.g., because of tree dependencies) was not considered previously in the framework of competitive analysis.

[1] We do not have to assume that they are actually stored in a real tree; this tree is implicit in the LMP scheme.

## 4.1 Problem Definition

In this chapter, we introduce a natural extension of an online paging problem, where items have inter-dependencies. In the classic online paging 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. The model variant, where fetching is optional (the requested item can be served without being in the cache, incurring some fixed cost) is called a *bypassing model*.

In our model, 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*.

Furthermore, we assume a bypassing model and 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. After serving the request, we may reorganize our cache arbitrarily, but the resulting cache has to still be a subforest of $T$. We pay $\alpha$ for fetching or evicting any single node, where $\alpha \geq 1$ is an integer and a parameter of the problem. Our goal is to minimize the overall cost of maintaining the cache and serving the requests.

Now, we are ready to formalize the model. We denote the height of $T$ by $h(T)$. For any node $v$, $T(v)$ denotes the subtree of $T$ rooted at $v$ (containing $v$ and all its descendants). 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 $B$.

We assume discrete time slotted into rounds, with round $t \geq 1$ corresponding to time interval $(t-1, t)$. In round $t$, the algorithm is given one (positive or negative) request to exactly one tree node and has to process it, i.e., pay associated costs (if any). 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 $\alpha$ for a single node fetch or eviction. 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 $\alpha$ 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.

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 ($k_{\text{ONL}}$ for an online algorithm and $k_{\text{OPT}}$ for the optimal offline one).

## 4.2 Algorithm

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 $\text{cnt}_t(A)$. At time $t$, TC verifies whether there exists a valid changeset $X$, such that

- *(saturation property)* $\text{cnt}_t(X) \geq |X| \cdot \alpha$ and
- *(maximality property)* $\text{cnt}_t(Y) < |Y| \cdot \alpha$ 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 $k_{\text{ONL}}$, 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 21 (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.

## 4.3 Analysis of TC

Throughout this chapter, 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 $\text{end}(P)$. A proof of the following technical lemma follows by induction and is presented in 4.4.

**Lemma 21.** *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. *$cnt_t(Y) < |Y| \cdot \alpha$ 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.*

In the following, we assume that no positive requests are given to nodes inside 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 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 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 4.3.2 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.

### 4.3.1 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 21.

**Observation 12.** *For any field $F$, $req(F) = size(F) \cdot \alpha$. 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 $F^\infty$. The set of all fields except $F^\infty$ is denoted by $\mathcal{F}$. Let $\text{size}(\mathcal{F}) = \sum_{F \in \mathcal{F}} \text{size}(F)$.

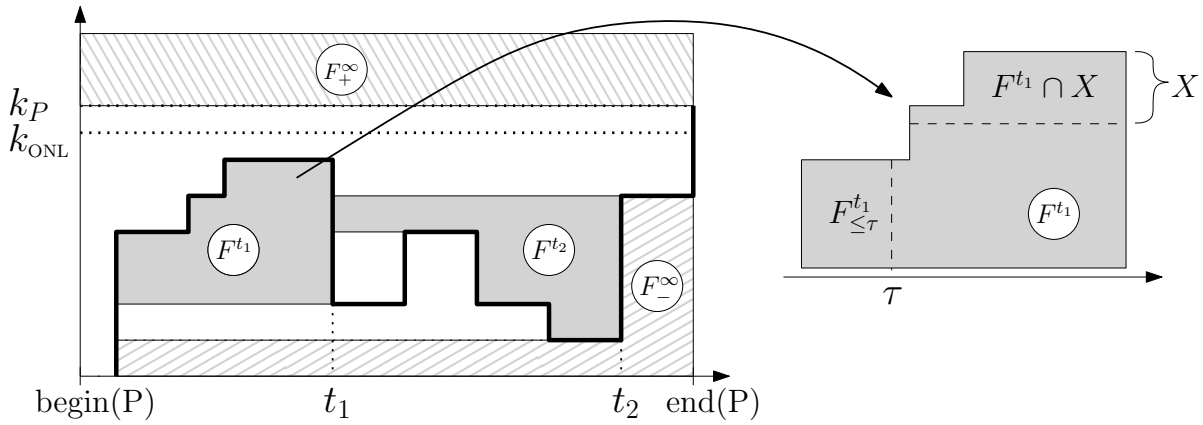


**Figure 4.2:** Partitioning of a single phase into fields for a line (a tree with no branches). The thick line represents cache contents. Possible final eviction at end($P$) is not depicted. $F^{t_1}$ is a negative field and $F^{t_2}$ is a positive one. In the particular depicted example, nodes are 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.

**Lemma 22.** *For any phase $P$ partitioned into a set of fields $\mathcal{F} \cup \{F^{\infty}\}$, it holds that* $\mathrm{TC}(P) \leq 2\alpha \cdot \mathit{size}(\mathcal{F}) + \mathit{req}(F^{\infty}) + k_P \cdot \alpha$.

*Proof.* By Observation 12, the cost associated with serving the requests from all fields from $\mathcal{F}$ is $\sum_{F \in \mathcal{F}} \alpha \cdot \mathrm{size}(F) = \alpha \cdot \mathrm{size}(\mathcal{F})$. The cost of the cache reorganization at the fields' ends is exactly the same. The term $\mathrm{req}(F^{\infty})$ represents the cost of serving the requests from $F^{\infty}$ and $k_P \cdot \alpha$ upper-bounds the cost of the final eviction (not present in an unfinished phase). □

### 4.3.2 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 $\mathrm{size}(F) \cdot \alpha$. Assume that these requests are evenly distributed among the nodes of $F$ (each node from $F$ receives $\alpha$ 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 $\alpha$ requests. Hence, for any two consecutive periods of a single node, Opt has to pay at least $\alpha$ (either $\alpha$ for positive requests or $\alpha$ for negative ones, or $\alpha$ 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 $\alpha \cdot \mathrm{size}(\mathcal{F})$.

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

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 $\textsc{Opt}(P')$ 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 appropriately large cost for $\textsc{Opt}$.

### 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 $\tau$, 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 $\tau$. 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 23.** *Fix any field $F^t$, the corresponding changeset $X_t$, and any time $\tau$.*

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 $\tau$, Lemma 21 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 $\tau$, by Lemma 21, $\mathrm{req}(F^t_{\leq \tau} \cap T') = \mathrm{cnt}_\tau(F^t_{\leq \tau} \cap T') \leq |F^t_{\leq \tau} \cap T'| \cdot \alpha$. □

By Lemma 23 applied at $\tau = t$ and Observation 12, we deduct the following corollary.

**Corollary 3.** *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 $req(F^t \cap D) \geq \alpha \cdot |D|$.*
2. *If $F^t$ is negative, then $req(F^t \cap (X_t \setminus D)) \geq \alpha \cdot |X_t \setminus D|$.*

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. $\text{req}(F^t \cap Y) = |Y| \cdot \alpha$ 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 23 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 $\alpha$ and the remaining part of $Y$ remains proper.

**Lemma 24.** *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 3 implies that any leaf of $Y$ was requested at least $\alpha$ 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 23, it would contain at most $\alpha$ 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 $\alpha$ less requests than $F^t_{\leq \tau} \cap (D \cup \{v\})$ had, it is not over-requested, either. □

**Corollary 4.** *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 $\alpha$ 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. While this result may be suboptimal, deriving a shifting strategy for a positive field that would have the same equal distribution guarantee as the one provided by Corollary 4 is not possible.

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 25.** *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 $\tau_j$ with the $((j-1)\cdot\alpha+1)$-th request to $v$. When this request arrives, 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 thus contradict the second property of Lemma 23. 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 26.** *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 $\text{size}(F^t)/(2h(T))$ 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 12, $\text{req}(F^t) = |X_t| \cdot \alpha$. 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 $\overline{\text{req}}(X)$ denote the number of grouped requests in the set $X$. Clearly, $\overline{\text{req}}(F^t) \geq |X_t| \cdot \alpha/2$, 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 25 for each node of $X_t^i$ separately. After such modification, at least $\lceil |X_t|/(2h(T)) \rceil \geq \text{size}(F_t)/(2h(T))$ nodes have at least $\alpha/2$ requests each. □

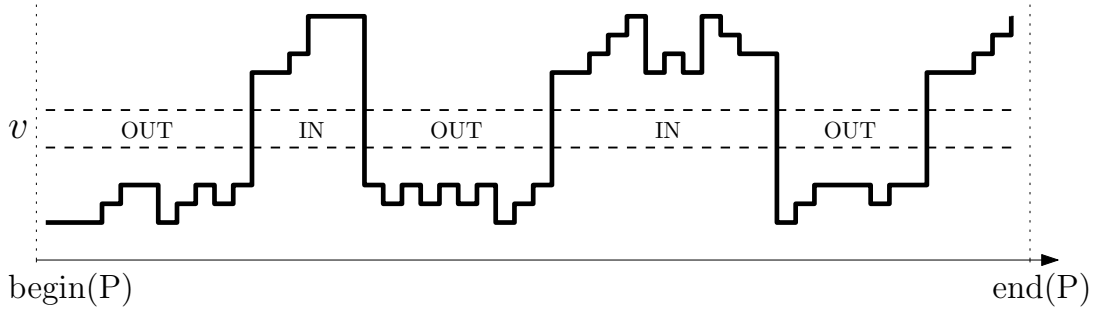


**Figure 4.3:** Partitioning of the phase into interleaving IN and OUT periods for node $v$. The thick line represents cache contents. The *leftover* OUT period (the last one) is present for node $v$ as it has finished phase $P$ inside TC's cache. The periods can be followed by requests contained in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

### Using Request Shifting for Bounding OPT

Finally, we may use our request shifting to relate $\text{size}(\mathcal{F}) = \sum_{F\in\mathcal{F}} \text{size}(F)$ 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 27.** *For any phase $P$, $\text{OPT}(P) \geq (\text{size}(\mathcal{F})/(4h(T)) - k_P) \cdot \alpha/2$.*

*Proof.* We transform $P$ using legal shifts that are described in Section 4.3.2 and Section 4.3.2. That is, we create a corresponding phase $P'$ that satisfies both Corollary 4 and Lemma 26. By Observation 13, it is sufficient to show that $\text{OPT}(P') \geq (\text{size}(\mathcal{F})/(4h(T)) - k_P) \cdot \alpha/2$.

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 $F^\infty$ 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 $p^{\text{IN}}$ and $p^{\text{OUT}}$ 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 $\alpha$ 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, $\text{OPT}(P') \geq (p^{\text{OUT}}/(2h(T)) - k_P) \cdot \alpha/2 \geq (\text{size}(\mathcal{F})/(4h(T)) - k_P) \cdot \alpha/2$. □

### 4.3.3 Competitive Ratio

To relate the cost of Opt to TC in a single phase $P$, we still need to upper-bound $\mathrm{req}(F^\infty)$ and relate $k_P \cdot \alpha$ to the cost of Opt (i.e., compare the bounds on TC and Opt provided by Lemma 22 and Lemma 27, respectively).

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

**Lemma 28.** *For any phase $P$, it holds that $\mathrm{req}(F^\infty) \leq 2 \cdot k_{\mathrm{ONL}} \cdot \alpha + 2 \cdot \textsc{Opt}(P)$.*

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

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

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

- Nodes from $F^\infty_-$ are in the cache $C_{\mathrm{end}(P)}$ and were not evicted from the cache. Thus, $\mathrm{req}(F^\infty_-) \leq |C_{\mathrm{end}(P)}| \cdot \alpha \leq k_{\mathrm{ONL}} \cdot \alpha$.
- All the requests from $V^{\mathrm{c}}_{\mathrm{OPT}}$ are paid by Opt, and hence $\mathrm{req}(F^\infty_+ \cap V^{\mathrm{c}}_{\mathrm{OPT}}) \leq \mathrm{req}(V^{\mathrm{c}}_{\mathrm{OPT}}) \leq \textsc{Opt}(P)$.
- $F^\infty_+$ is a valid changeset for cache $C_{\mathrm{end}(P)} \sqcup X_{\mathrm{end}(P)}$. As $V_{\mathrm{OPT}}$ 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 $\tau$ of $P$. We now observe the evolution of field $F^\infty$ from time $\tau$ till $\mathrm{end}(P)$. At time $\tau$, $\mathrm{req}(F^\infty) \leq 2 \cdot k_{\mathrm{ONL}} \cdot \alpha + 2 \cdot \textsc{Opt}(P)$. Furthermore, in subsequent times, it may only decrease: at any round $F^\infty$ gets an additional request, but on eviction $\mathrm{req}(F^\infty)$ decreases by $\alpha$ 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 22, Lemma 27 and Lemma 28, we immediately obtain the following corollary (holding for both finished and unfinished phases).

**Corollary 5.** *For any phase $P$, it holds that* $\mathrm{TC}(P) \leq O(h(T)) \cdot \mathrm{OPT}(P) + O(h(T) \cdot (k_P + k_{\mathrm{ONL}}) \cdot \alpha)$.

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

*Proof.* First, we compute the number of positive requests in $V_{\mathrm{OPT}}^c$. 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 $V_{\mathrm{OPT}}$ is a subforest of $T$, $X'_t$ is a tree cap of $X_t$. By Corollary 3, 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 $V_{\mathrm{OPT}}$. Hence the total number of positive requests outside of $V_{\mathrm{OPT}}$ 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 $\mathrm{OPT}(P)$ can be split into the cost associated with nodes from $V_{\mathrm{OPT}}$ and $V_{\mathrm{OPT}}^c$, 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 $V_{\mathrm{OPT}}$. Hence, $\mathrm{OPT}(P) \geq (|V_{\mathrm{OPT}}| - k_{\mathrm{OPT}}) \cdot \alpha + (k_P - |V_{\mathrm{OPT}}|) \cdot \alpha = (k_P - k_{\mathrm{OPT}}) \cdot \alpha$. Then, $k_P \cdot \alpha \leq k_P \cdot \mathrm{OPT}(P)/(k_P - k_{\mathrm{OPT}})$. As the phase is finished, $k_P \geq k_{\mathrm{ONL}} + 1$, and thus $k_P \cdot \alpha \leq (k_{\mathrm{ONL}} + 1) \cdot \mathrm{OPT}(P)/(k_{\mathrm{ONL}} + 1 - k_{\mathrm{OPT}})$. □

**Theorem 14.** *The algorithm* TC *is $O(h(T) \cdot k_{\mathrm{ONL}}/(k_{\mathrm{ONL}} - k_{\mathrm{OPT}} + 1))$-competitive.*

*Proof.* Let $R = h(T) \cdot k_{\mathrm{ONL}}/(k_{\mathrm{ONL}} - k_{\mathrm{OPT}} + 1)$. 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 $k_P > k_{\mathrm{ONL}}$, and hence Corollary 5 and Lemma 29 imply that $\mathrm{TC}(P) \leq O(R) \cdot \mathrm{OPT}(P)$. For an unfinished phase $k_P \leq k_{\mathrm{ONL}}$, and therefore, by Corollary 5, $\mathrm{TC}(P) \leq O(h(T)) \cdot \mathrm{OPT}(P) + O(h(T) \cdot k_{\mathrm{ONL}} \cdot \alpha)$. Summing over all phases of $I$ yields $\mathrm{TC}(I) \leq O(R) \cdot \mathrm{OPT}(I) + O(h(T) \cdot k_{\mathrm{ONL}} \cdot \alpha)$. □

## 4.4 No over-requested changesets

Before proving Lemma 21, we present the following technical claim.

**Claim 15.** *For any phase $P$, the following invariants hold for any time $t > \mathrm{begin}(P)$:*

1. *$\mathrm{cnt}_{t-1}(X) < |X| \cdot \alpha$ for a valid changeset $X$ for $C_t$,*
2. *$\mathrm{cnt}_t(X) \leq |X| \cdot \alpha$ for a valid changeset $X$ for $C_t$,*
3. *any changeset $X$ with property $\mathrm{cnt}_t(X) = |X| \cdot \alpha$ 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 $\mathrm{cnt}_t(X) \leq \mathrm{cnt}_{t-1}(X) + 1 \leq (|X| \cdot \alpha - 1) + 1 = |X| \cdot \alpha$, 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 $t > \mathrm{begin}(P)$. It is trivially true for $t = \mathrm{begin}(P) + 1$ as $\mathrm{cnt}_{t-1}(X) = 0$ 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 $\mathrm{cnt}_t(X) \geq |X| \cdot \alpha$ (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 $\mathrm{cnt}_t(X) \geq |X| \cdot \alpha$, 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 $\mathrm{cnt}_t(Y) = |Y| \cdot \alpha$. Then, $Y \sqcup X$ is a changeset for $C_t$, and $\mathrm{cnt}_t(Y \sqcup X) \geq |Y \sqcup X| \cdot \alpha$. 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 $\mathrm{cnt}_t(Y) = |Y| \cdot \alpha$, 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 $\mathrm{cnt}_t(X \setminus Y) = \mathrm{cnt}_t(X) \geq |X| \cdot \alpha \geq |X \setminus Y| \cdot \alpha$. By Invariant 2, $\mathrm{cnt}_t(X \setminus Y) \leq |X \setminus Y| \cdot \alpha$, and hence $\mathrm{cnt}_t(X \setminus Y) = |X \setminus Y| \cdot \alpha$. This contradicts Invariant 3 as $X \setminus Y$ cannot contain the node requested at round $t$ (because $Y$ contains this node). □

*Proof of Lemma 21.* The inequality $\mathrm{cnt}_t(X) \leq |X| \cdot \alpha$ is equivalent to Invariant 2 of Claim 15. Assume now that $X$ is applied at time $t$. By the definition of ALG, $\mathrm{cnt}_t(X) \geq |X| \cdot \alpha$, and thus $\mathrm{cnt}_t(X) = |X| \cdot \alpha$, i.e., Property 2 follows. Then, Invariant 3 of Claim 15 implies Property 1. Finally, Invariant 1 of Claim 15 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, $|X| \cdot \alpha = \mathrm{cnt}_t(X) = \mathrm{cnt}_t(X_1) + \mathrm{cnt}_t(X_2)$. If $\mathrm{cnt}_t(X_1) < |X_1| \cdot \alpha$ or $\mathrm{cnt}_t(X_2) < |X_2| \cdot \alpha$, then $\mathrm{cnt}_t(X_1) + \mathrm{cnt}_t(X_2) < (|X_1| + |X_2|) \cdot \alpha = |X| \cdot \alpha$, a contradiction. Therefore, $\mathrm{cnt}_t(X_1) = |X_1| \cdot \alpha$ and $\mathrm{cnt}_t(X_2) = |X_2| \cdot \alpha$. But then Invariant 3 of Claim 15 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 $C_t$. □

## 4.5 Implementation of TC

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 4.2). Here, we show that this operation can be performed efficiently. In particular, in the following two subsections, we will prove the following theorem.

**Theorem 16.** 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 21 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 4.5.1, using Lemma 21, 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 21 there are still exponentially many valid negative changesets to consider. In Section 4.5.2, 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.

### 4.5.1 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 21, 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 $O(h(T))$.

### 4.5.2 Negative Requests and Evictions

Handling evictions is more complex. If the request to node $v_t$ at round $t$ was negative, Lemma 21 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, $\mathrm{val}_t$ 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

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

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 $\mathrm{val}_t$,

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

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 21,

$\mathrm{cnt}_t(H^u) \leq |H^u| \cdot \alpha$, 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 21, $Y$ 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 $O(h(T) + \deg(T) \cdot |X_t|)$.

## 4.6 Cache Updates with Fixed Cost

In this section, 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 updates with fixed cost $\alpha$.

Namely, we take any input $I$ for the latter problem and create, in online fashion, an input $I_T$ for the tree caching problem. 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. Recall that an update of a rule stored at node $v$ in $I$ is mapped to a *chunk* of $\alpha$ negative requests to $v$ in $I_T$. 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 $\alpha$ 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.

## 4.7 Lower Bound on the Competitive Ratio

**Theorem 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 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 pages in the paging problem. The rest of the tree will be irrelevant.

For any input sequence $I$ for the paging problem, we may create a sequence $I_{\mathrm{T}}$ for tree caching, where a request to a page is replaced by $\alpha$ requests to the corresponding leaf. Now, we claim that any solution $A$ for $I$ of cost $c$ can be transformed, in online manner, into a solution $A_{\mathrm{T}}$ for $I_{\mathrm{T}}$ 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 pages, then $A_{\mathrm{T}}$ bypasses $\alpha$ corresponding requests to leaf $r$, fetches $r$ afterwards and evicts the corresponding leaves, paying $O(\alpha)$ times the cost of $A$. By doing it iteratively, $A_{\mathrm{T}}$ ensures that its cache is equivalent to that of $A$. In particular, a request free for $A$ is also free for $A_{\mathrm{T}}$.

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

The bidirectional reduction described above preserves competitive ratios up to a constant factor. Hence, applying the adversarial strategy for the paging problem that enforces the competitive ratio $R = k_{\mathrm{ONL}}/(k_{\mathrm{ONL}} - k_{\mathrm{OPT}} + 1)$ [ST85b] immediately implies the lower bound of $\Omega(R)$ on the competitive ratio for the tree caching problem. □

## 4.8 Conclusions

In this chapter we define a novel variant of online paging 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.

[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 K. Ahuja, Thomas L. Magnanti, and James B. 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. Also appeared in *Proc. of the 1st SODA*, pages 291–300, 1990.

[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. *Proc. 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.

[DS12] Ran Duan and Hsin-Hao Su. A scaling algorithm for maximum weight matching in bipartite graphs. In *Proc. 23rd Annual ACM-SIAM Symposium on Discrete Algorithms (SODA)*, pages 1413–1424, 2012.

[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.

[EILN15] 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.

[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.

[Gab85] H.N. Gabow. A scaling algorithm for weighted matching on general graphs. *SFCS '85 Proc. of the 26th Annual Symposium on Foundations of Computer Science*, pages 90–100, 1985.

[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.

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

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

[Ira02] 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.

[KK12] Zoltan Kiraly and Peter Kovacs. Efficient implementations of minimum-cost flow algorithms. In *ArXiv Technical Report 1207.6381*, 2012.

[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 E. 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] T. Leighton, F. 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.

[MAB+08] Nick McKeown, Tom Anderson, Hari Balakrishnan, Guru Parulkar, Larry Peterson, Jennifer Rexford, Scott Shenker, and Jonathan Turner. Openflow: Enabling innovation in campus networks. *SIGCOMM Comput. Commun. Rev.*, 38(2):69–74, 2008. doi:http://www.openflowswitch.org.

[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. *SIGCOMM CCR*, 42(5):44–48, 2012.

[MS91] Lyle A. McGeoch and Daniel D. 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] K. Pagiamtis and A. 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.

[ST85a] Daniel D. Sleator and Robert E. Tarjan. Amortized efficiency of list update and paging rules. *Communications of the ACM*, 28(2):202–208, 1985.

[ST85b] Daniel D. Sleator and Robert E. Tarjan. Amortized efficiency of list update and paging rules. *Communications of the ACM*, 28(2):202–208, 1985.

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

[STT03] Ed Spitznagel, David E. Taylor, and Jonathan S. 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 V. 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 S. Xin, Josh Rosen, Matei Zaharia, Michael J. Franklin, Scott Shenker, and Ion Stoica. Shark: SQL and Rich Analytics at Scale. In *Proc. ACM SIGMOD*, pages 13–24, 2013.

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

[You02] Neal E. 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.