Caching in Networks:

Non-Uniform Algorithms and Memory Capacity Constraints

Dissertation
of
Matthias Westermann

Acknowledgments

First of all, I would like to thank my advisor Prof. Dr. Friedhelm Meyer auf der Heide for his great support. The atmosphere in his research group at Paderborn University was very creative. In particular, Friedhelm always left me the freedom to do the work in my own style and time. Furthermore, I would like to thank the other reviewers of my thesis, Prof. Dr. Susanne Albers from Dortmund University and Prof. Dr. Burkhard Monien from Paderborn University. I would also like to thank Prof. Dr. Johannes Blömer and Dr. Rainer Feldmann for being additional members of the examination board.

I would like to thank my "co-advisor" Berthold Vöcking for a very valuable collaboration. It was a lot of fun to master several problems in theoretical computer science and other major spheres of life with Berthold, such as solving a traveling salesperson problem given by the touristic highlights in California and changing several car tires with unconventional tools in the forests of Massachusetts during one night. Also, many thanks to Christian Sohler, Klaus Schröder, Harald Räcke, and Christof Krick for many great discussions about computer science and other significant aspects of society.

Matthias Westermann
Paderborn, Germany
November 2000

This research was supported by the DFG-Sonderforschungsbereich 376
"Massively Parallel Computing: Algorithms – Design Methods – Applications"

Contents

Chapter 1

Introduction

In recent years, large computer systems connected by networks have become part of our everyday live. A good example is the widespread use of the Internet and Internet-related applications such as the World Wide Web (WWW). A basic functionality in these systems is the interactive use of shared data objects that can be accessed from each computer in the system. Examples for these objects are files in distributed file systems for Ethernet-connected workstations, cache lines in virtual shared memory systems for massively parallel computers, or pages in the WWW.

The dramatic growth of computer systems necessitates more and more an intelligent management of shared data objects. The daily congestion in the Internet is a clear evidence that the network becomes more and more a bottleneck as the size of the system increases. The same effect can be observed in other distributed systems, like massively parallel processors (MPPs) or networks of workstations (NOWs). The traditional management of shared data objects based on centralized mega-servers with special hardware meets its technical and economical limits. Thus, it is mandatory to design new distributed management strategies that guarantee the free flow of the shared data objects in the systems.

In general, large computer systems connected by networks consist of a set of computers each having its own processor and memory module. These computers are usually connected by a relatively sparse network constructed out of *links*, i.e., point-to-point connections, or *buses*, i.e., connections between two or more processors. The performance of these systems depends on a number of parameters, including processor speed, memory capacity, network topology, bandwidths, and latencies. Usually, the links or buses are the bottleneck in these systems because improving communication bandwidth and latency is often more expensive or more difficult than increasing processor speed and memory capacity. But whereas several standard methods are known for hiding latency, e.g., pipelined routing (see, e.g., [CMS96, CMSV96]), redundant computation (see, e.g., [ALMZ96a, ALMZ96b, KLM$^+$97, Mey83, Mey86, MW89]) or slackness (see, e.g., [Val90]), the only way to bypass the bandwidth bottleneck is to reduce the communication load by exploiting locality.

The principle of locality is already known from sequential computation. Two kinds of locality are usually distinguished: temporal and spatial locality (see, e.g., [Goo94]). *Temporal locality* means that in the near future, a program is more likely to reference those data objects that have been referenced in the recent past. This locality can be due to instruction references in program loops, or data references in working stacks. *Spatial locality* means that in the near future, a program is more likely to reference those data objects that have addresses close to the recent references. This is caused, e.g., by the traversal of data structures such as arrays. A further kind of locality is specific to computer networks: topological locality. *Topological locality* means that processors that are close together with respect to the structure of the interconnection network are likely to be interested in the same data objects. This locality can be due to a communication sensitive mapping of processes to the nodes in the network.

Usually, traditional management of shared data objects uses either the concept of caching or hashing. *Caching* exploits the temporal and topological locality. The communication load for accessing the shared data objects can be reduced by storing copies of data objects in memory modules of some processors. This phenomenon is well known and often explored in practice, e.g., in the WWW, where pages are cached in order to reduce network congestion. When a user issues a read request for a page, the requested page is transferred from a computer holding a copy of the page through the network to the computer of the user. Usually, the page is *cached*, i.e., a copy of the accessed page is kept in the local memory module of the user or in another memory module lying on the routing path used by the page. In this way, subsequent requests addressed to this page can be served locally. Similar caching mechanisms are used to speed up parallel and distributed computations using virtual shared memory on massively parallel processors (MPPs) or networks of workstations (NOWs). The common idea of all of these approaches is to use memory resources in order to reduce the bandwidth utilization.

Caching is used for speeding up single-computers, too. Here a small, fast memory module, the *cache*, is added to the processor in order to reduce the communication on the bus connecting the processor with a much larger main memory module. Data is partitioned into small blocks, called *cache lines*. Whenever the processor accesses a data item that is not already stored in the cache, the whole respective cache line is moved from the main memory into the cache. The major problem is to decide which cache line is ejected, i.e., removed from the cache, when the cache is full. This problem is usually referred to as *paging* as it was investigated first in the context of virtual memory where the data is usually divided into so-called pages. Paging has been investigated intensively as well in practice (see, e.g., [Bel66, CK99, FW74, Spi77]) as in theory (see, e.g., [CK99, FKL$^+$91, Ira97, IKP96, MS91, RS94, ST85]) in the past. In all of these analyses it turned out that, although the idea of paging is very simple, its analysis is not an easy task. Developing and analyzing strategies for caching in networks is even more complicated than for single-computers because several caches interact. Sometimes, creating a copy of a data object induces additional communication instead of reducing it, as this copy may need to be updated or invalidated later.

While caching aims to reduce the communication load by exploiting locality, *hashing* tries, to distribute the communication load evenly among the network resources, in order to avoid bottlenecks and thus congestion in the network. This is done by a randomized distribution of the shared data objects to all memory modules. But, this randomized distribution usually destroys the locality and hence the caching becomes inefficient.

This thesis deals with distributed caching strategies that combine the two apparently opposed concepts of caching and hashing. For this purpose, we reduce the complex interconnection network to a simple structure, the so-called *access tree*. This logical structure represents the communication potential of the whole network, but it abstracts from single interconnections. In particular, the routing paths in this structure are unique. Hence, the following questions can be answered efficiently by a distributed caching strategy for access trees.

- How many copies of a shared data object should be created?
- On which memory modules should these copies be placed?
- Which requests should be served by which copies?
- How should the copies of a shared data object be located?

Due to the simple structure of access trees, the caching strategy can abandon on additional communication for the management and localization of the created copies. Now, the logical access trees are embedded into the network via hashing. This is done in a random but locality preserving fashion. With this concept, the so-called *access tree strategy*, it is possible to use the effects of caching and hashing simultaneously, i.e., on one hand to reduce the communication load via caching, and on the other hand to distribute the remaining load evenly, to avoid network congestion.

In order to provide efficient access to shared data objects, the communication overhead caused by the caching strategy should be as small as possible. However, simply reducing the *total communication load*, i.e., the sum, taken over all messages, of the size of the messages multiplied with the length of their routing paths, can result in bottlenecks. In addition, the load has to be distributed evenly among all network resources. This corresponds to minimizing the *congestion*, i.e., the maximum, taken over all links, of the amount of data transmitted by the link. Known results for store-and-forward routing [LMRR94, MV95, OR97, SV96] and wormhole routing [CMS96, CMSV96, SV96] show that reducing the congestion is most important in order to get a good network throughput. For this reason, we believe that minimizing the congestion is a promising approach in order to develop caching strategies that work efficiently in theory and practice.

1.1 Our contribution

This thesis is based on the access tree strategy that we have originally introduced in [MMVW97a]. The access tree strategy aims to minimize the congestion for trees, meshes, and Internet-like clustered networks. For example, we achieve competitive ratio 3 for trees and competitive ratio $O(d \cdot \log n)$, w.h.p., for d-dimensional meshes with n nodes. Further, we present an $\Omega(\log n/d)$ lower bound for the competitive ratio for on-line routing in meshes which implies that our upper bound on the competitive ratio for meshes of constant dimension is optimal. All strategies presented in [MMVW97a], however, work only in a uniform cost model in which migrating a data object has the same cost as accessing the object. Furthermore, memory capacity constraints are neglected. These assumptions simplify the problem significantly. In this thesis we extend the access tree strategy to

- a non-uniform cost model,
- memory capacity constraints, and
- other classes of networks.

Typically, words or other small blocks of data that can be accessed in a computer system are much smaller than the data objects. For example, large files usually consist of several small records and pages of virtual shared memory consist of cache lines that can be accessed individually. Thus, migrating a data object can be much more expensive than accessing only a small piece of data from the object. However, the data of an object should be kept together in order to reduce the bookkeeping overhead. We present the first deterministic and distributed caching strategy that achieves competitive ratio 3 for trees in this non-uniform cost model. This competitive ratio is optimal because of the lower bound shown by Black and Sleator [BS89]. Our strategy does not only minimize the congestion but minimizes simultaneously the load on each individual edge up to an optimal factor of 3. Strategies for trees are of special interest as they can be used as subroutines in strategies for other networks, e.g., in the access tree strategy. However, our strategy still neglects memory capacity constraints. This result has been previously published as joint work in [MVW99].

Furthermore, we present a general framework for the development of caching strategies for networks with memory capacity constraints. This framework is based on the access tree strategy. Our strategies aim to minimize the congestion. As in the model without memory capacity constraints, our framework yields a caching strategy for d-dimensional meshes with n nodes that is $O(d \cdot \log n)$-competitive, w.h.p. In addition, we give several examples of networks for which our framework yields efficient strategies, including fat-trees, complete networks, Cayley networks, edge symmetric networks, hypercubes, cube-connected-cycles, de Bruijn networks, shuffle-exchange networks, and butterflies. For example, our framework yields a caching strategy for complete networks that is $O(1)$-competitive, w.h.p, with respect to the congestion at

the memory modules due to remote accesses. Most of the results have been previously published as joint work in [MVW00].

We believe that the access tree strategy is well suited for the application in practice because the congestion is provably small. Besides, the strategy is simple and produces only very small overhead for bookkeeping. In order to illustrate the practical usability, we have implemented the dynamic access tree strategy on a massively parallel mesh-connected computer system and tested it for three standard applications of parallel computing in [KMR+99]. The experimental results show that execution time and congestion are closely related. The access tree strategy achieves execution times that are close to the times of hand-optimized message passing strategies using full knowledge of the access pattern of the application. Furthermore, all experiments show that the access tree strategy clearly outperforms a standard caching strategy in which a fixed home processor is assigned to each data object that keeps track of the object's copies. In particular, the larger the network is, the more superior the access tree strategy becomes.

1.2 Cost model

The computer system is modeled by an undirected graph $G = (V,E)$ with node set V and edge set E such that the nodes represent the processors with their memory modules, and the edges represent the links. The capacities of the memory modules are described by a function $m : V \longrightarrow \mathbb{N}$ and the bandwidths of the links are described by a function $b : E \longrightarrow \mathbb{N}$. An application consists of read and write requests that are issued by the nodes. Each of these requests is directed to a shared data object from a finite set X of shared data objects.

The shared data objects in X are stored in the local memory modules possibly with redundancy. At each time step, for each object $x \in X$, let the *residence set* $R(x)$ represent the set of nodes holding a copy of x. Initially, only a single copy of each data object is stored somewhere in the computer system. This initial configuration is known by all nodes in the system. During the execution of an application, copies may be deleted, copies may be migrated, or new copies may be created as long as each node obeys its memory capacity constraints. But always, at least one copy of each data object must be stored in some of the nodes, i.e., at each time step, for each object $x \in X$, $R(x) \neq \emptyset$.

A read request for a data object x requires access to a copy of x, a write request requires updating all copies of x. After a request is served, the multiple copies can be reallocated. Any communication that proceeds along an edge increases the *(communication) load* on that edge by some amount which depends on whether a read, write, or migration operation is performed. The increase is defined as follows.

- Read operation: A read request for x issued by a node v can be served by any node u holding a copy of x. A path has to be allocated through the network from v to u. The communication load on each edge e on this path increases by 1.

- Write operation: A write request for x issued by a node v requires to update all copies of x. A multicast tree connecting v with all nodes holding a copy of x, i.e., a Steiner tree, has to be allocated. The communication load on each edge e in the Steiner tree increases by 1.

- Object migration: The strategy can replicate a copy of x from one node to another along an arbitrary path. The communication load on each edge e on this path increases by D, where $D \geq 1$ is an arbitrary integer representing the ratio between the load induced by the migration of the complete data object x and the load induced by accessing only a relative small piece of x, e.g., a single byte.

Efficient strategies for distributed caching have to work in a distributed fashion. In particular, the nodes do not have knowledge about the global state of the system, that is, each node notices only the read and write accesses and the copy migrations that pass the node. In order to accumulate additional knowledge a node has to communicate with other nodes, which also increases the load on the involved edges.

- Information exchange: Information about the global state of the system, e.g. the actual residence set, can be exchanged by sending messages along a path from one node to another. It is assumed that the messages have small size, e.g., they include an ident-number of a data object and a tag for an action that should be performed on the receiving node. The communication load on each edge on this path increases by 1.

For a given application, let the *relative load* on an edge be its load divided by its bandwidth. Finally, define the *congestion* to be the maximum over the relative loads of all edges in the network.

We use caching strategies for *single-computer systems* as a subroutine. This problem is modeled by a graph of two nodes u and v connected by a single edge e. The memory module of node v can hold all data objects simultaneously. The capacity of the memory module of u, however, is limited. Only u is allowed to issue requests. Note that in this case there is no significant difference between read and write requests. The cost considered in the single-computer case is simply the load on edge e.

1.3 Competitive analysis

Caching strategies can be viewed as on-line strategies that are typically evaluated in a competitive analysis. In this kind of analysis which was introduced by Sleator and Tarjan [ST85] the costs of the on-line strategies are compared with the costs of an optimal off-line strategy.

In order to obtain a simple, unambiguous model, we assume that an adversary initiates an application, i.e., a sequence $\sigma = \sigma_1\sigma_2\sigma_3\cdots$ of read and write requests each of which is issued by one of the nodes in the network. The strategies have to serve

these requests one after the other, that is, it is assumed that σ_{i+1} is not issued before the service of σ_i is finished. This assumption, however, does not mean that we want to model sequential applications. The assumption of a sequence is the simplest way to determine that the on-line and the off-line strategy have to serve the requests in the same order. Note that allowing the off-line strategy to use a different order than the on-line strategy would enable it to reduce its congestion almost arbitrarily.

First, we investigate caching strategies for networks without memory capacity constraints. For a given sequence σ, let $C_{\text{opt}}(\sigma)$ denote the minimum congestion produced by an optimal off-line strategy. An on-line strategy is said to be *c-competitive* if it has congestion at most $c \cdot C_{\text{opt}}(\sigma) + a$, for each sequence σ, where a is a term that does not depend on σ. The value c is also called the *competitive ratio* of the on-line strategy.

It is reasonable, to pay attention to the additive term a. This term must be independent on the sequence σ but it can depend, e.g., on the number of nodes in the network. Thus, the additive term a can dominate the expression $c \cdot C_{\text{opt}}(\sigma) + a$ for a long time, until the sequence σ reaches the necessary length. Therefore, it is interesting to have *strictly c-competitive* on-line strategies i.e., strategies having congestion at most $c \cdot C_{\text{opt}}(\sigma)$, i.e., $a = 0$, for each sequence σ. In this case, the value c is also called the *strict competitive ratio* of the on-line strategy.

In the model without memory capacity constraints we will show that all presented strategies are also able to handle parallel and overlapping requests, too. We restrict the class of allowed applications specified by the adversary to be *data-race free programs*, i.e., a write request to an object is not allowed to overlap with other requests to the same object, and there is some order among the requests to the same object such that, for each read and write request, there is a unique least recent write. Note that this still allows arbitrary concurrent requests to different objects and concurrent read requests to the same object. Without the restriction to data-race free programs, the optimal off-line strategy could, e.g., defer all write requests to an object x to the end of the execution and keep a valid copy of x at each node that reads x at some time. This "bad trick" would give load 1 on each edge for serving all read requests to x. The example illustrates that the restriction to the class of data-race free programs is necessary in order to allow a fair comparison of on-line against optimal off-line strategies.

When the on-line strategy serves a request, it does not know future requests. In contrast, the optimal off-line strategy is assumed to have full knowledge about the whole sequence including future requests. This assumption gives much more power to the optimal off-line strategy than to the on-line strategy. In fact, it turns out that this advantage is too large in the case of bounded memory size: Sleator and Tarjan show in [ST85] that the worst-case competitive ratio between the cost of a deterministic paging strategy and an optimal off-line strategy grows linearly with the memory size. In the case of randomized strategies this ratio grows logarithmically with the memory size [FKL$^+$91]. In order to compensate the advantage of knowing the future Sleator and Tarjan suggest to restrict the optimal off-line strategy by reducing its memory capacity. They show that, if the memory capacity of the optimal off-line strategy is reduced by a factor of two, then a constant competitive ratio of two can be achieved, e.g., for the

LRU (least recently used) paging strategy, which is the most used paging strategy in practice.

We use a similar approach for the evaluation of caching strategies for networks with memory capacity constraints. We compare an on-line strategy whose individual memory capacities have been increased by a factor of m, for some fixed m, with an optimal off-line strategy obeying the given memory capacities. For simplicity in notation, we increase the capacities of the on-line strategy rather than decreasing the capacities of the optimal off-line strategy. For a given sequence σ, let $C_{\text{opt}}(\sigma)$ denote the minimum congestion produced by an optimal off-line strategy, obeying the memory bound $m(v)$, for each node v. An on-line strategy is said to be (m,c)-*competitive* if it obeys the memory bound $m \cdot m(v)$, for each node v, and if it has congestion at most $c \cdot C_{\text{opt}}(\sigma) + a$, for each sequence σ, where a is a term that does not depend on σ. The value m is also called the *memory ratio* and the value c is also called the *congestion ratio* of the strategy.

In the case with memory capacity constraints, it is again interesting to have *strictly* (m,c)*-competitive* on-line strategies, i.e., strategies having congestion at most $c \cdot C_{\text{opt}}(\sigma)$, i.e., $a = 0$, for each sequence σ, while obeying the memory bound $m \cdot m(v)$, for each node v. In this case, the value m is also called the *strict memory ratio* and the value c is also called the *strict congestion ratio* of the on-line strategy.

If the on-line strategy uses randomization we have to describe the power given to the adversary more precisely. This is studied intensively in [BDBK$^+$90]. The adversary specifying the sequence is assumed to be *oblivious*, i.e., the requests do not depend on the decisions of the on-line strategies. A randomized strategy has to satisfy the congestion bound *with high probability (w.h.p.)*, that is, the probability that the congestion exceeds $\alpha \cdot (c \cdot C_{\text{opt}}(\sigma) + a)$ is at most $n^{-\alpha}$, for every $\alpha \geq 1$, where n denotes the number of nodes in the network. Note that this bound implies that the expected congestion is $O(c \cdot C_{\text{opt}}(\sigma) + a)$.

1.4 Previous work on the access tree strategy

This thesis is built upon joint work with Christof Krick, Bruce Maggs, Friedhelm Meyer auf der Heide, Harald Räcke, and Berthold Vöcking on data management in networks without memory capacity constraints in a uniform cost model. This work is described in [MMVW97a, MMVW97b, KMR$^+$99] and Berthold Vöcking's thesis [Vöc98]. In the following, these results will be sketched briefly.

We introduce new static and dynamic data management strategies for trees, meshes, and Internet-like clustered networks in [MMVW97a, MMVW97b]. Note that all presented strategies work only in a uniform cost model and that they do not obey memory capacity constraints. These assumptions simplify the problem significantly.

In the static model, we assume that we are given an application for which the the rates of read and write requests for all node-object pairs are known. The goal is to

calculate a static placement of the objects to the nodes in the network and to specify the routing such that the congestion is minimized. In the dynamic model, we assume no knowledge about the access pattern. An adversary specifies accesses at runtime. Here we present caching strategies that also aim to minimize the congestion by exploiting locality. These strategies are investigated in a competitive model.

We start by investigating static data management on tree-connected networks modeled by connected (hyper)graphs without cycles. We show the surprising result that a static placement exists that minimizes the communication load on all edges (or hyperedges) simultaneously. Obviously, this yields minimum congestion as well as minimum total communication load. We describe a deterministic strategy that calculates this placement. The sequential running time of this strategy is $O(n)$ for each object, where n denotes the size of the network. Moreover, the placement can be computed efficiently in a distributed fashion by the processors of the underlying tree network.

Besides we present the first deterministic dynamic strategy for tree-connected networks that works in a distributed fashion and achieves a competitive ratio of 3. This caching strategy is surprisingly simple, and the competitive ratio is optimal because of the lower bound shown by Black and Sleator [BS89].

As our results for static and dynamic data management on trees hold for trees with arbitrary link or bus connections having arbitrary bandwidths, our strategies are well suited in particular for Ethernet-connected NOWs. For example, the static strategy can be used for optimal static file allocation and the dynamic strategy for the implementation of efficient distributed shared memory systems in these networks.

The situation on meshes is much more complicated because there are several possible routing paths between every pair of nodes. In fact, we show by a reduction from PARTITION that the static problem is NP-hard already on a 3×3 mesh. The dynamic problem is even more complicated, since we have to solve an on-line routing problem and a data tracking problem in order to locate a suitable copy in case of a read access and all copies of an object in case of a write request.

We introduce a simulation approach that solves the static and the dynamic problem on meshes simultaneously. The strategy is based on a randomized but locality preserving embedding of access trees into the mesh, and it is called the access tree strategy. On the access trees, the static or dynamic strategy for trees is executed. Consider an arbitrary mesh of dimension d with n nodes. Here the access tree strategy yields an efficient strategy for static data placement on meshes with arbitrary side length achieving optimal congestion up to a factor of $O(d \cdot \log n)$, w.h.p. The placement of each object only takes time $O(n)$. In the dynamic model, the access tree strategy achieves competitive ratio $O(d \cdot \log n)$, w.h.p. We give a corresponding $\Omega(\log n/d)$ lower bound on the competitive ratio for on-line routing, which implies that the competitive ratio achieved by the dynamic access tree strategy is optimal for meshes of constant dimension.

Furthermore, we show how the access tree strategy can be adapted to Internet-like clustered networks. A *clustered network* is a network that consists of several small subnetworks, i.e., *clusters*, that are organized hierarchically. Communication between nodes of the same cluster is not as expensive as communication between

nodes of different clusters. The access trees are embedded into the clustered network in a preprocessing step. We show that this preprocessing can be done efficiently and locally for each participating cluster. The static variant of the access tree strategy allows to calculate a static placement with close-to-optimal congestion for clustered networks. The dynamic variant yields an efficient caching strategy.

We believe that the access tree strategy is well suited for the application in practice because the congestion is provably small. Besides the strategy is simple and produces only very small overhead for bookkeeping. In order to illustrate the practical usability, we have implemented variants of the dynamic access tree strategy in the *DIVA (Distributed Variables) library* [KRVW98] that provides direct access to global variables, i.e., shared data objects, from the individual nodes in the network. The current implementations are based on a massively parallel mesh-connected computer system.

We have evaluated the dynamic access tree strategies implemented in the DIVA library experimentally in [KMR+99]. We test several variations of this strategy on three different applications of parallel computing, which are matrix multiplication, bitonic sorting, and Barnes-Hut N-body simulation. The implemented algorithms for matrix multiplication and sorting are *oblivious*, i.e., their access or communication patterns do not depend on the input. The reason why we have decided to include these algorithms in our small benchmark suite is that they allow us to compare the dynamic data management strategies with hand-optimized message passing strategies. The third application, the Barnes-Hut N-body simulation, is non-oblivious. We believe that a communication mechanism that uses shared data objects is the best solution for this application, in contrast to the other two applications. However, we cannot construct a hand-optimized message passing strategy achieving minimum congestion for this application. Therefore, we concentrate on the comparison of different caching strategies.

Our experiments show that the access tree strategy produces much less congestion than the standard *fixed home strategy* (see, e.g., [Goo94]) in which a fixed home processor is assigned to each data object that keeps track of the object's copies. Furthermore, we observe that the execution time of the investigated application depends heavily on the congestion produced by the different data management strategies. Therefore, the access tree strategy outperforms the fixed home strategy clearly. In particular, the larger the network is, the more superior the access tree strategy becomes. While the fixed home strategy scales poorly, the access tree strategy comes reasonably close to the performance of the hand-optimized message passing strategies even in the case of large networks.

1.5 Other previous work

Competitive analysis of caching strategies in computer systems connected by networks begins with Karlin et al. [KMRS88] who analyze strategies for snoopy caching on buses. Bartal et al. [BFR92] introduce the *file allocation problem* in a competitive

model similar to our non-uniform model. The major difference to our model is that they consider a different cost measure than we do, that is, they consider the total communication load, i.e., the sum over the relative load on all edges, rather than the congestion, i.e., the maximum over the relative load on all edges.

The most comparative work is done by Awerbuch et al. [ABF93, ABF98] who consider the file allocation problem on arbitrary networks. In [ABF93] they present a centralized strategy that achieves an optimal competitive ratio $O(\log n)$ and a distributed strategy that achieves competitive ratio $O((\log n)^4)$ for the file allocation problem on an arbitrary n-node network without memory capacity constraints. In [ABF98], the distributed strategy is adapted to networks with memory capacity constraints resulting in an strategy that is $(\text{polylog } n, \text{polylog } n)$-competitive. All competitive ratios are with respect to the total communication load instead of the congestion. We believe that the congestion measure is more appropriate for the design of strategies than the total communication load as it prevents some of the links from becoming bottlenecks.

We give a brief description of the very intriguing distributed strategy of Awerbuch et al. because this will illustrate the influence of the cost measure on the design of caching strategies. Their strategy uses a hierarchical network decomposition, introduced in [AP90], that decomposes the network in a hierarchy of clusters with geometrically decreasing diameter. If a node accesses a file x that has no copy in a cluster of some hierarchy level, then the cluster leader is informed and increases a counter for the file. When the counter reaches D, the requesting node gets a copy of x, where D denotes the ratio between the cost for migrating and the cost for accessing a file. This strategy ensures that the total communication load is minimized up to a some logarithmic factors in the size of the network, which gives a good competitive factor with respect to the total communication load. The cluster leaders, however, can become bottlenecks. In particular, the edges incident to the leader of the top level clusters may become very congested. As a consequence, the competitive ratio in our congestion based cost model becomes bad.

Caching strategies for tree-like topologies are of special interest since many networks have such topology, e.g., Ethernet-connected NOWs. Besides, strategies for trees can be used as a subroutine for caching strategies on other networks. Therefore, much research deals with caching on trees. For example, Bartal et al. [BFR92] describe a randomized strategy for the file allocation problem on trees without memory capacity constraints that is 3-competitive with respect to the expected total load and a deterministic strategy that is 9-competitive with respect to the total load. The congestion is not considered, but it is easy to check that both strategies are $\omega(1)$-competitive with respect to the congestion.

Lund et al. [LRWY94] describe a 3-competitive deterministic but centralized strategy for the same problem. The advantage of their strategy is that it minimizes not only the total communication load but also the load on each edge and therefore also the congestion. Unfortunately, the strategy makes use of global knowledge about a work function that is influenced by each request issued in the network. In particular, read requests issued by a node can induce the invalidation of copies in a completely

different region of the network without sending invalidation messages. This illustrates that their strategy is inherently centralized, and hence yields no suitable solution in our distributed setting.

A lower bound that holds for each network including at least one edge is shown by Black and Sleator [BS89]. They give a lower bound of 3 on the competitive ratio of each deterministic caching strategy on two nodes connected by a single edge. Bartal et al. [BFR92] show that this bound holds also for randomized strategies.

Several recent papers deal with the distribution of pages in the World Wide Web. Plaxton and Rajaraman [PR96] show how to balance the pages among several caches by embedding a random cache tree for each page into the network. This balances the load well and ensures fast responses even for popular pages. However, the strategy uses a uniform embedding of the tree nodes onto the nodes in the Internet, which destroys topological locality. Karger et al. [KLL$^+$97] use a similar technique to relieve hot spots in the Internet but they pay attention to topological locality. In contrast to our assumption, they assume the latencies instead of the bandwidths to be the main problem for data transmission in the Internet. Further, they consider only read requests.

A difficult problem that has to be solved by each caching strategy is the *data tracking* problem, i.e., the problem of how to locate the copies of a particular object. To our knowledge, data tracking mechanisms that aim to minimize the congestion have not been investigated previously. Bartal et al. [BFR92] describe a data tracking mechanism for arbitrary networks that minimizes the total communication load. This mechanism is based on a distributed data structure that keeps track of all copies. Whenever a copy is created or deleted on a node, this data structure is updated. The mechanism enables each node to locate a copy of a particular object having approximately the distance of the closest copy. The total communication load induced by each of the create, delete, or locate operations are shown to be in $O(\ell \cdot \log^2 n)$, where ℓ denotes the length of the shortest path to a copy of the respective object and n is the size of the network. On each node, the memory overhead due to the distributed data structure is at most $O(|X|)$ with X denoting the set of shared objects. Plaxton et al. [PRR97] describe a similar mechanism which reduces the memory overhead for the data structure on each node to $O(k \cdot \log^2 n)$ with k denoting the number of copies on a node. However, this mechanism works only for networks that fulfill a low-expansion property, which is satisfied, e.g., by meshes of constant dimension.

1.6 Outline

The remainder of this thesis is organized as follows. In Chapter 2, caching strategies for networks without memory capacity constraints are investigated in a non-uniform cost model. First, the access tree strategy is introduced, and as an example the application of the access tree strategy to 2-dimensional meshes is sketched. Then, a deterministic and distributed caching strategy for trees is presented. This strategy is used as a subroutine in the access tree strategy. Thereafter, the access tree strategy is

applied to several classes of networks, including meshes, fat-trees, complete networks, Cayley networks, edge symmetric networks, hypercubic networks, and Internet-like clustered networks. Further, we present the universal caching strategy working on arbitrary networks for which an oblivious routing strategy is given. Finally, we will show that all presented strategies are also able to handle parallel and overlapping requests, too.

In Chapter 3, caching strategies for networks with memory capacity constraints are investigate in a non-uniform cost model. First, the general framework for the development of caching strategies with memory capacity constraints is introduced. This framework is based on the access tree strategy. Then, a caching strategy for trees with memory capacity constraints is presented. This strategy is used as a subroutine in the general framework. Thereafter, the general framework is applied to several classes of networks, including meshes, fat-trees, complete networks, Cayley networks, edge symmetric networks, and hypercubic networks. Further, we present the universal caching strategy working on arbitrary networks for which an oblivious routing strategy is given.

In Chapter 4, a detailed summary and discussion of the results of this thesis is presented. Further, some concluding remarks are given.

Chapter 2

Caching without Memory Capacity Constraints

In this chapter, we investigate caching strategies for networks without memory capacity constraints in a non-uniform cost model. The remainder of the chapter is organized as follows. In Section 2.1, the access tree strategy is introduced, and as an example the application of the access tree strategy to 2-dimensional meshes is sketched. In Section 2.2, a deterministic and distributed caching for trees is presented. This strategy is used as a subroutine in the access tree strategy. In Section 2.3, the access tree strategy is applied to several classes of networks, including meshes, fat-trees, complete networks, Cayley networks, edge symmetric networks, hypercubic networks, and Internet-like clustered networks. Further, we present the universal caching strategy working on arbitrary networks for which an oblivious routing strategy is given. Finally, in Section 2.4, we will show that all presented strategies are also able to handle parallel and overlapping requests, too.

2.1 The access tree strategy

The *access tree strategy* is based on a hierarchical decomposition of the network. This hierarchical decomposition allows to exploit topological locality in an efficient way. As an example, we describe recursively the hierarchical decomposition of a 2-dimensional mesh M with n nodes. Let m_1 and m_2 denote the side lengths of the mesh, i.e., $n = m_1 \cdot m_2$. We assume $m_1 \geq m_2$. If $m_1 = 1$ then we have reached the end of the recursion. Otherwise, we partition M into two non-overlapping submeshes of size $\lceil m_1/2 \rceil \times m_2$ and $\lfloor m_1/2 \rfloor \times m_2$. These submeshes are then decomposed recursively according to the same rules. Figure 2.1 gives an example for this decomposition.

The hierarchical decomposition has associated with it a *decomposition tree* $T(M)$, in which each node corresponds to one of the submeshes, i.e., the root of $T(M)$ corresponds to M itself, and the children of a node v in the tree correspond to the two submeshes into which the submesh corresponding to v is divided. In this way, the leaf

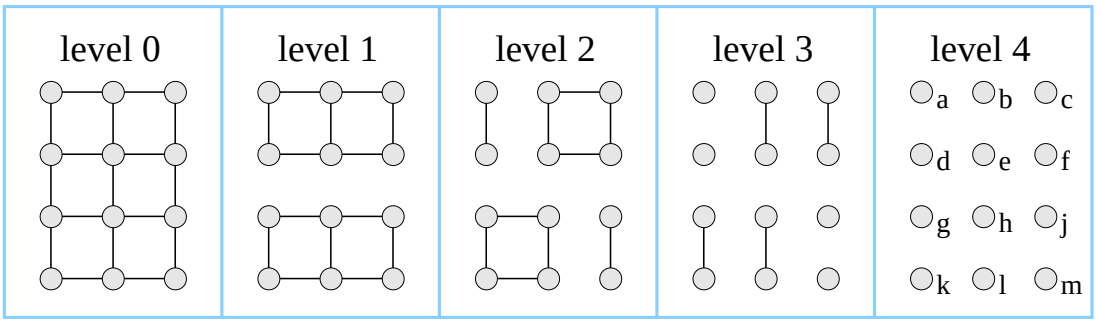

Figure 2.1: The hierarchical decomposition of $M(4,3)$.

nodes of $T(M)$ correspond to submeshes of size one and thus each leaf node represents a node of M.

We interpret $T(M)$ as a virtual network that we want to simulate on M. In order to compare the congestion in both networks we define bandwidths for the edges in $T(M)$, i.e., for a tree edge $e = (u,v)$, with u denoting the parent node, define the bandwidth of e to be the number of edges leaving the submesh corresponding to v. Note that this bandwidth definition is only required for the description of the theoretical results and does not affect the access tree strategy in any way. Figure 2.2 gives an example of a decomposition tree with bandwidths.

For each global variable, define an *access tree* to be a copy of the decomposition tree. We embed the access trees randomly into the mesh, i.e., for each variable, each node of the access tree is mapped at random to one of the nodes in the corresponding submesh. On each of the access trees, we execute a simple 3-competitive caching strategy. All messages that should be sent between neighboring nodes in the access trees are sent along the *dimension-by-dimension order path* between the associated nodes in the mesh, i.e., the unique shortest path between the two nodes using first edges of dimension 1 and then edges of dimension 2.

The analysis of the access tree strategy uses a bi-simulation between the mesh M and the decomposition tree $T(M)$. At first, it is shown that the decomposition tree $T(M)$ can simulate the mesh M. In this simulation, each leaf node of $T(M)$ issues the same read and write requests as its counterpart in M. It is shown that any application that produces congestion C when it is executed on M can be executed on $T(M)$ with congestion C, too. At second, it is shown that the mesh M can simulate the decomposition tree $T(M)$ in such a way that the congestion in M is only $O(\log n)$ times larger than the congestion in $T(M)$.

From the results on the congestion of these two simulations we can obtain the competitive ratio of the access tree strategy. The simulation of M by $T(M)$ looses a factor of at most $c_1 = 1$; the on-line strategy looses a factor of $c_2 = 3$ against the off-line strategy on $T(M)$; finally, the simulation of $T(M)$ by M looses a factor of at most $c_3 = O(\log n)$. Altogether, we achieve a competitive ratio of $c_1 \cdot c_2 \cdot c_3 = O(\log n)$.

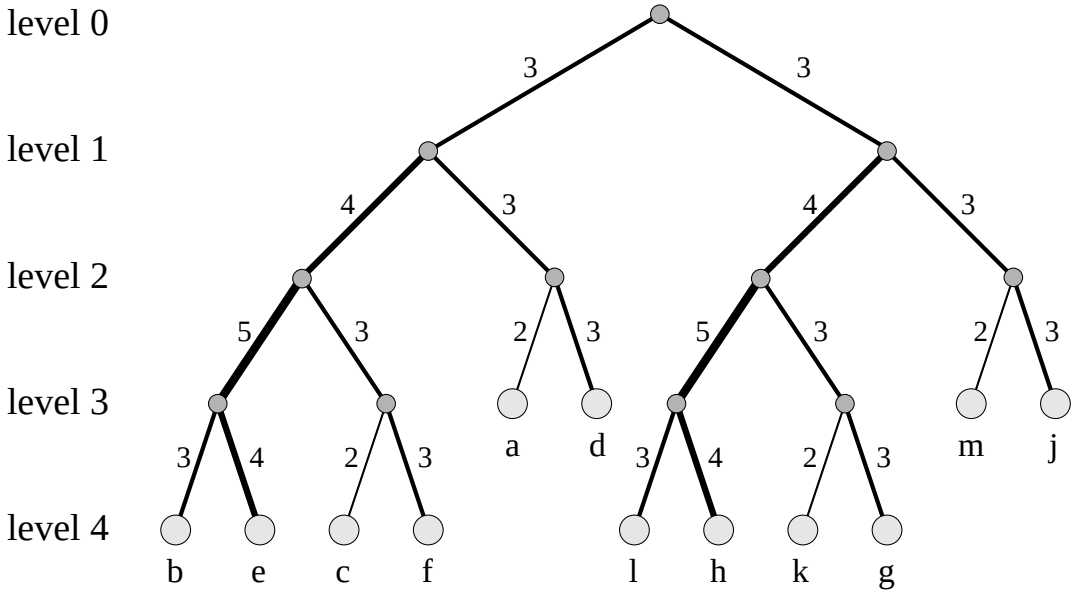

Figure 2.2: The decomposition tree $T(M(4,3))$. The node labels correspond to the labels in Figure 2.1. The edge labels indicate the bandwidths of the respective edges.

2.2 Caching on trees

In this section, deterministic caching strategies for a tree T are presented. The advantage of trees is that there is only one simple path between any pair of nodes. Thus, the placement of the objects automatically defines the routing paths from the accessing node to the respective copies. In particular, this means that the congestion is fixed as soon as the placement is specified, which makes the analysis for trees much easier than the one for networks including cycles. The edges in the tree T are allowed to have arbitrary bandwidth.

First, we present a very simple caching strategy for trees in a uniform cost model, i.e. $D = 1$. It is shown that this deterministic strategy is 3-competitive. Then, we generalize this strategy to a non-uniform cost model. Again, it is proved that this deterministic strategy is still 3-competitive. This competitive ratio is optimal because of the lower bound shown by Black and Sleator [BS89].

2.2.1 Uniform model

In the uniform cost model, i.e. $D = 1$, we assume that each object fits into one routing packet such that each migration of a copy along an edge increases the load on this edge by 1. Recall that each read or write request increases the load of each edge involved in this request by 1, and that each information message increases the load of each traversed edge by 1.

- Node v issues a read request for object x:

  The read request is served by u, the node holding a copy of x that is nearest to v.

  A new copy of x is created on each node incident to an edge on the path from u to v.

- Node v issues a write request for object x:

  v sends the update information, i.e., the value that should be written, to u, the node holding a copy of x that is nearest to v.

  u starts an invalidation multicast to all other nodes holding a copy of x, and modifies its own copy of x.

  A new copy of x is created on each node incident to an edge on the path from u to v.

Figure 2.3: The tree strategy in the uniform model for object x.

Our caching strategy manages all objects independently from each other. For each object x, the nodes that hold a copy of x always build a connected component. In Figure 2.3 the *tree strategy in the uniform model* is described for object x. Note that in case of a write request the invalidation multicast can be viewed as following: First, the copies are updated, and after this update the copies are deleted.

It remains to describe how the nodes locate the copies. As the nodes holding the copies of x always build a connected component, the data tracking problem can be solved very easily. Each node is attached a signpost pointing to the node that has issued the least recent write request. Initially, this signpost points to the only copy of x. Whenever x is updated the signposts are redirected to the node that has issued the corresponding write request. Note that this mechanism does not require extra communication, because only the signposts on nodes involved in the invalidation multicast have to be redirected. The number of signposts can be reduced by defining a root of the tree and omitting all those signposts that are directed to the root. Hence, we need signposts only to mark the trail from the root to the node that issued the least recent write request such that diameter of T signposts for each object are sufficient.

Besides, we attach markers to the copies that indicate the boundaries of the connected component built by the nodes that hold a copy. These markers are needed for the invalidation multicast. They are updated whenever the connected component changes. Also this mechanism does not require extra communication.

The following theorem shows that the tree strategy in the uniform model is strictly 3-competitive on trees with edges of arbitrary bandwidth.

Theorem 2.1 *The tree strategy in the uniform model minimizes the load on any edge up to a factor of 3.*

Proof. Fix an object $x \in X$, and a sequence $\sigma = \sigma_1\sigma_2\cdots$ of read and write requests for x. If σ starts with a read request, then we add a write request issued by the node holding the initial copy of x at the beginning of the sequence. Obviously, this causes no extra communication for an optimal strategy. We divide the sequence of requests into *phases* such that the first request in each phase is a write and each phase includes only one write.

For a phase t, let v_t denote the node issuing the write request at the beginning of t, and let U_t denote the connected subgraph induced by v_t and the nodes issuing the read requests in this phase. Further, for $t \geq 1$, let p_t denote the unique shortest path leading from U_{t-1} to v_t.

Each caching strategy has to send at least one message along each edge in U_t in phase t because all of the nodes that issue read requests have to receive the value written by v_t. Besides, for $t \geq 1$, a message has to be sent along p_t because the new value written by v_t has to be "merged" with the value written by v_{t-1}. Consequently, any strategy has to sent one message along each edge in $U_t \cup p_t$ for each phase t.

Now, we consider the load induced by our strategy. All messages induced by read and write requests of phase t except for the invalidation messages are routed along edges in $U_t \cup p_t$. In particular, the load of each of the edges in $U_t \cup p_t$ is increased by exactly two: by the service of a read or write request and the corresponding migration of a copy of x.

Finally, we consider the load induced by the invalidation multicast. The invalidation multicast for the write in phase t sends exactly one message along each edge in $U_{t-1} \cup p_{t-1}$. Consequently, our strategy sends at most three times as many messages along any edge as any other strategy does. ■

2.2.2 Non-uniform model

First, we describe a deterministic caching strategy for two nodes connected by a single edge. This strategy is 3-competitive, and it can be computed in a distributed fashion by the two connected nodes. The key feature of this strategy is that it can be extended to work on trees just by simulating it on any edge in the tree. The result of this approach is a simple, deterministic, and distributed caching strategy for trees which minimizes the load on any edge up to a factor of 3. Obviously, the bound on the load of any edge induces that our tree strategy is 3-competitive with respect to the total communication load and the congestion simultaneously.

Caching on a single edge

We describe a deterministic and distributed caching strategy for a single edge e connecting two nodes a and b. The strategy uses a simple counting mechanism which

- Node a issues a read request for object x:

  If $c_a < D$ then $c_a := c_a + 1$;
  If $c_a = D$ and a holds no copy of x then
  Move a new copy of x onto a;
  If $c_b = 0$ then delete the copy of x on b;

- Node a issues a write request for object x:

  If $c_b > 0$ then
  $c_b := c_b - 1$;
  else
  If $c_a < D$ then $c_a := c_a + 1$;
  If $c_a = D$ and a holds no copy of x then move a new copy of x onto a;
  If $c_b = 0$ and a holds a copy of x then delete the copy of x on b;

Figure 2.4: The edge strategy for requests issued by node a.

records read and write requests that are issued by the two nodes. Later on we will use this strategy for building our tree strategy. Then the nodes a and b correspond to the two connected components in which the tree is divided if e is removed.

Initially, we do not care about how the nodes a and b exchange information about the residence set, and how the counters are distributed among them. We assume that node a always knows whether or not node b holds a copy and vice versa. Afterwards we show how our strategy can be adapted to the distributed setting.

Each object x is handled independently from the other objects. Let us fix an object x. Concerning this object there are two counters c_a and c_b. Informally, these counters represent saving accounts for costs referring to x.

Initially, one copy of x is placed on one of the two nodes. W.l.o.g., assume that it is placed on a. Then c_a is set to D, and c_b is set to 0. In Figure 2.4 the *edge strategy* is described for requests issued by node a. The strategy works analogously for requests issued by node b. Note that the edge strategy always keeps one copy of x, since it only deletes a copy on a node if the other node also holds a copy.

The following lemma shows that the centralized edge strategy is strictly 3-competitive.

Lemma 2.2 *The centralized edge strategy minimizes the load up to a factor of 3.*

Proof. We use a potential function argument (cf. [ST85]). First, let us fix an optimal off-line strategy, which is denoted the *optimal strategy* in the following. W.l.o.g., the optimal strategy fulfills the following properties.

- If a node v issues a read request for an object x, then the optimal strategy does not delete a copy, that is, the only possible change of the residence set is that a new copy of x is moved to v.
- If a node v issues a write request for an object x, then the only possible changes of the residence set are that a new copy of x is moved to v and/or a copy of x is deleted on the neighbor of v.

Fix an object $x \in X$, and a sequence $\sigma = \sigma_1\sigma_2\cdots$ of read and write requests for x. Let $L_e(t)$ and $L_{opt}(t)$ denote the load of the edge strategy and the optimal strategy, respectively, after serving σ_t, and let $\Phi(t)$ denote the value of a potential function after serving σ_t, which is defined in detail later. In order to prove the lemma, we show the following invariant.

(a) $L_e(t) + \Phi(t) \leq 3 \cdot L_{opt}(t)$ and

(b) $\Phi(t) \geq 0$.

Let $c_a(t)$ and $c_b(t)$ denote the value of the counters c_a and c_b, respectively, after serving σ_t, and let $R_e(t)$ and $R_{opt}(t)$ denote the residence set of the edge strategy and the optimal strategy, respectively, after serving σ_t. We define

$$\Phi(t) = \Phi_a(t) + \Phi_b(t) - D \ ,$$

where, for $v \in \{a,b\}$,

$$\Phi_v(t) = \begin{cases} 2c_v(t), & \text{if } v \notin R_e(t) \text{ and } v \notin R_{opt}(t), \\ 3D - c_v(t), & \text{if } v \notin R_e(t) \text{ and } v \in R_{opt}(t), \\ 3D - 2c_v(t), & \text{if } v \in R_e(t) \text{ and } v \in R_{opt}(t), \\ c_v(t), & \text{if } v \in R_e(t) \text{ and } v \notin R_{opt}(t). \end{cases}$$

First, we prove invariant (b). The optimal strategy always holds a copy of x on a node. Hence, $v \in R_{opt}(t)$, for some node $v \in \{a,b\}$, and consequently $\Phi_v(t) \geq D$, which implies $\Phi_a(t) + \Phi_b(t) \geq D$. Hence, invariant (b) is shown.

Now we prove invariant (a) by induction on the length of σ. Obviously, (a) holds for the initial setting. For the induction step suppose that $L_e(t) + \Phi(t) \leq 3 \cdot L_{opt}(t)$. Let $\Delta L_e = L_e(t+1) - L_e(t)$, $\Delta L_{opt} = L_{opt}(t+1) - L_{opt}(t)$, $\Delta\Phi_a = \Phi_a(t+1) - \Phi_a(t)$, and $\Delta\Phi_b = \Phi_b(t+1) - \Phi_b(t)$. In order to prove the induction step, we show that

$$\Delta L_e + \Delta\Phi_a + \Delta\Phi_b \leq 3 \cdot \Delta L_{opt}. \tag{2.1}$$

We distinguish between read and write requests.

- Suppose σ_{t+1} is a read request issued by node a. In this case, Equation (2.1) can be checked with Table 2.1 containing all possible changes of configuration. Note that, if a issues a read request, the only possible changes of the residence sets are that one of the strategies moves a copy to a, or $c_b = 0$ and the edge strategy deletes the copy on b. In both cases, $\Delta\Phi_b = 0$.

$a \in R_e(t)$	$a \in R_{opt}(t)$	$a \in R_e(t+1)$	$a \in R_{opt}(t+1)$	ΔL_e	$\Delta\Phi_a \leq$	ΔL_{opt}
no	no	no	no	1	2	1
no	no	no	yes	1	$3D-1$	D
no	yes	no	yes	1	-1	0
no	no	yes	no	$1+D$	$2-D$	1
no	no	yes	yes	$1+D$	$2-D$	D
no	yes	yes	yes	$1+D$	$-1-D$	0
yes	no	yes	no	0	1	1
yes	no	yes	yes	0	$3D-2$	D
yes	yes	yes	yes	0	0	0

Table 2.1: Possible changes of configuration if node a issues a read request.

- Suppose σ_{t+1} is a write request issued by node a. We distinguish between the cases $c_b(t) > 0$ and $c_b(t) = 0$. Note that, if a issues a write request, the only possible changes of the residence set of the optimal strategy is that a new copy of x is moved to a and/or a copy of x is deleted on b.
  - Suppose $c_b(t) > 0$. In this case, Equation (2.1) can be checked with Table 2.2 containing all possible changes of configuration. Note that, if a issues a write request and $c_b(t) > 0$, the only possible transitions of the edge strategy are from $\{a\}$ to $\{a\}$, from $\{b\}$ to $\{b\}$, or from $\{a,b\}$ to $\{a,b\}$ or $\{a\}$.
  - Suppose $c_b(t) = 0$. In this case, Equation (2.1) can be checked with Table 2.3 containing all possible changes of configuration. Note that, if a issues a write request and $c_b(t) = 0$, the only possible transitions of the edge strategy are from $\{a\}$ to $\{a\}$ or from $\{b\}$ to $\{b\}$ or $\{a\}$.

This completes the proof of Lemma 2.2. ■

Next, we describe how the edge strategy can be adapted from the centralized to the distributed setting. Each node keeps always the current value of both counters c_a and c_b. It is obvious that this assignment makes it possible for the nodes to make the right decisions according to the edge strategy. Now we specify how each node keeps track of both counters c_a and c_b. W.l.o.g., consider node a.

- A read request for x is issued by b: If b holds no copy of x then a is able to update its counters. In the other case, b sends an information message along e if and only if b has increased its counter c_b.
- A write request for x is issued by b: If a holds a copy of x then a is able to update its counters. In the other case, b sends an information message along e if and only if b has decreased its counter c_a or increased its counter c_b.

$R_e(t)$	$R_{opt}(t)$	$R_e(t+1)$	$R_{opt}(t+1)$	ΔL_e	$\Delta\Phi_a \leq$	$\Delta\Phi_b \leq$	ΔL_{opt}
a	a	a	a	0	0	-2	0
a	a,b	a	a,b	0	0	1	1
a	a,b	a	a	0	0	-2	0
a	b	a	b	0	0	1	1
a	b	a	a,b	0	$3D$	1	$1+D$
a	b	a	a	0	$3D$	-2	D
b	a	b	a	1	0	-1	0
b	a,b	b	a,b	1	0	2	1
b	a,b	b	a	1	0	-1	0
b	b	b	b	1	0	2	1
b	b	b	a,b	1	$3D$	2	$1+D$
b	b	b	a	1	$3D$	-1	D
a,b	a	a,b	a	1	0	-1	0
a,b	a,b	a,b	a,b	1	0	2	1
a,b	a,b	a,b	a	1	0	-1	0
a,b	b	a,b	b	1	0	2	1
a,b	b	a,b	a,b	1	$3D$	2	$1+D$
a,b	b	a,b	a	1	$3D$	-1	D
a,b	a	a	a	1	0	-1	0
a,b	a,b	a	a,b	1	0	2	1
a,b	a,b	a	a	1	0	$2-3D$	0
a,b	b	a	b	1	0	2	1
a,b	b	a	a,b	1	$3D$	2	$1+D$
a,b	b	a	a	1	$3D$	$2-3D$	D

Table 2.2: Possible changes of configuration if node a issues a write request and $c_b(t) > 0$.

In this way, each node is able to keep both of its counters up-to-date. The following lemma shows that the additional information messages are sent very rarely.

Lemma 2.3 *The distributed edge strategy minimizes the load up to a factor of 3.*

Proof. We adopt the notations and definitions of Lemma 2.2. An information message is only sent if a request changes the value of a counter. Thus, the tables used for proving Lemma 2.2 change only slightly.

- Suppose σ_{t+1} is a read request issued by node a. Then, an additional message is sent only, if $a \in R_e(t)$. In this case, ΔL_e equals 1 rather than 0. Further, $\Delta\Phi_a$ equals -2 rather than 0, if $a \in R_{opt}(t)$ and $c_a(t) < D$. It is easy to check that Equation (2.1) is still satisfied by applying these changes to Table 2.1.

$R_e(t)$	$R_{opt}(t)$	$R_e(t+1)$	$R_{opt}(t+1)$	ΔL_e	$\Delta\Phi_a \leq$	$\Delta\Phi_b \leq$	ΔL_{opt}
a	a	a	a	0	0	0	0
a	a,b	a	a,b	0	0	0	1
a	a,b	a	a	0	0	$-3D$	0
a	b	a	b	0	1	0	1
a	b	a	a,b	0	1	0	$1+D$
a	b	a	a	0	1	$-3D$	D
b	a	b	a	1	-1	0	0
b	a,b	b	a,b	1	-1	0	1
b	a,b	b	a	1	-1	$-3D$	0
b	b	b	b	1	2	0	1
b	b	b	a,b	1	$3D-1$	0	$1+D$
b	b	b	a	1	$3D-1$	$-3D$	D
b	a	a	a	$1+D$	$-1-D$	0	0
b	a,b	a	a,b	$1+D$	$-1-D$	0	1
b	a,b	a	a	$1+D$	$-1-D$	$-3D$	0
b	b	a	b	$1+D$	$2-D$	0	1
b	b	a	a,b	$1+D$	$2-D$	0	$1+D$
b	b	a	a	$1+D$	$2-D$	$-3D$	D

Table 2.3: Possible changes of configuration if node a issues a write request and $c_b(t) = 0$.

- Suppose σ_{t+1} is a write request issued by node a. Then, an additional message is sent only, if $b \notin R_e(t)$. In this case, ΔL_e equals 1 rather than 0. Further, $\Delta\Phi_a$ equals -2 rather than 0, if $a \in R_{opt}(t)$ and $c_a(t) < D$. It is easy to check that Equation (2.1) is still satisfied by applying these changes to the Tables 2.2 and 2.3.

Hence, Lemma 2.3 is proved. ■

Caching on trees

The distributed edge strategy can be extended to a tree $T = (V,E)$. The edges in the tree T are allowed to have arbitrary bandwidths.

The *tree strategy* is composed out of $|E|$ individual edge strategies. The idea is to simulate the distributed edge strategy on each edge. Consider an arbitrary edge $e = (a,b)$. The removal of e divides T into two subtrees T_a and T_b, containing a and b, respectively. The two nodes a and b execute the algorithm described in Figure 2.4. The phrases "if a holds a (no) copy of x" and "node a issues a read (write) request for object x" are just replaced by "if a node in T_a holds a (no) copy of x" and "a node in T_a issues a read (write) request for object x", respectively.

The simulation works properly as long as the nodes in the residence set $R(x)$ build a connected component in the tree. A key feature of our edge strategy is that it fulfills this condition. This is shown in the following lemma.

Lemma 2.4 *The graph induced by the residence set $R(x)$ is always a connected component.*

Proof. Via induction on the length of the sequence of requests it can be shown that those counters on any simple path in the tree that are responsible for moving a copy along an edge towards the first node of the path are non-decreasing from the first to the last node on the path. Hence, all these counters "agree" about the distribution of copies. This ensures that all copies stay in a connected component.

Fix an object x, and a sequence $\sigma = \sigma_1\sigma_2\cdots$ of read and write requests for x. Fix a node v and two different edges $x = (u,v)$ and $y = (v,w)$ that are incident on v. The removal of x and y divides T into three subtrees T_u, T_v, and T_w, containing u, v and w, respectively. For each edge (a,b), there are the two counters $c_a^{(a,b)}$ and $c_b^{(a,b)}$ in the distributed edge strategy. Then, let $c_u^x(t)$ and $c_v^y(t)$ denote the values of the counters c_u^x and c_v^y, respectively, after serving σ_t.

In order to show the lemma, we prove by induction that $c_u^x(t) \le c_v^y(t)$. Obviously this is true for the initial setting. For the induction step suppose that $c_u^x(t) \le c_v^y(t)$. In the following we have to consider the three cases that σ_{t+1} is a request from a node in T_u, T_v, or T_w.

- Suppose σ_{t+1} is a request from a node in T_u. Then, $c_u^x(t+1) \in \{c_u^x(t), c_u^x(t)+1\}$ and $c_v^y(t+1) \in \{c_v^y(t), c_v^y(t)+1\}$. If $c_u^x(t+1) = c_u^x(t)+1 \le D$ then $c_v^y(t+1) \in \{c_v^y(t)+1, D\}$. Thus, $c_u^x(t+1) \le c_v^y(t+1)$.
- Suppose σ_{t+1} is a request from a node in T_v. Then, $c_u^x(t+1) \in \{c_u^x(t), c_u^x(t)-1\}$ and $c_v^y(t+1) \in \{c_v^y(t), c_v^y(t)+1\}$. Thus, $c_u^x(t+1) \le c_v^y(t+1)$.
- Suppose σ_{t+1} is a request from a node in T_w. Then, $c_u^x(t+1) \in \{c_u^x(t), c_u^x(t)-1\}$ and $c_v^y(t+1) \in \{c_v^y(t), c_v^y(t)-1\}$. If $c_v^y(t+1) = c_u^x(t)-1 \ge 0$ then $c_u^x(t+1) \in \{c_v^y(t)-1, 0\}$. Thus, $c_u^x(t+1) \le c_v^y(t+1)$.

This completes the proof of Lemma 2.4. ■

Note that the tree strategy also does not need any additional information exchange apart from the one done by the distributed edge strategy. Therefore, the following theorem follows immediately from Lemma 2.3. It shows that the tree strategy is strictly 3-competitive for trees with edges of arbitrary bandwidth.

Theorem 2.5 *The tree strategy minimizes the load on any edge up to a factor of 3.*

2.3 Applications of the access tree strategy

The access tree strategy can be applied to several classes of networks. Given any network G, the decomposition tree $T(G)$ can be constructed by a hierarchical decomposition of G. The quality of the resulting caching algorithm depends on some properties of this decompositions. Although it is not clear which properties can be obtained for general networks, applying the access tree strategy to almost any standard network yields interesting results. In this section, we give some examples, including meshes, fat-trees, complete networks, Cayley networks, edge symmetric networks, hypercubic networks, and Internet-like clustered networks. Further, we present the universal caching strategy working on arbitrary networks for which an oblivious routing strategy is given.

2.3.1 Meshes

In this section, we consider caching strategies for the mesh $M = M(m_1,\ldots,m_d)$, i.e., the d-dimensional mesh-connected network with side length $m_i \geq 2$ in dimension i. The number of nodes is denoted by n, i.e., $n = m_1 \cdots m_d$, the number of edges is $\sum_{i=1}^{d} m_1 \cdots m_{i-1} \cdot (m_i - 1) \cdot m_{i+1} \cdots m_d = \Theta(d \cdot n)$. Each edge has bandwidth 1. Thus, the relative and absolute load of an edge are identical.

We show that the access tree strategy is strictly $O(d \cdot \log n)$-competitive, w.h.p., for M. In order to prove this result, we give an upper bound on the maximum expected load in M due to the access tree strategy. To complete the proof, we show with a Chernoff bound (see, e.g., [HR90]) that the maximum load over all edges does not deviate to much from the expected load of an arbitrary edge. In order to apply a Chernoff bound reasonably, we have to make the access tree strategy more fine granularly, i.e, the access tree nodes have to be remapped dynamically in M when too many messages that simulate messages of the tree strategy traverse a node. The remapping is done by the general remapping scheme. We present this scheme in a more general fashion such that it can be used for all networks, not only for meshes. Finally, we give a lower bound for on-line routing on meshes showing that the above competitive ratio is optimal up to a factor $\Theta(d^2)$.

Upper bound

The access tree strategy uses a locality preserving embedding of access trees. It is based on a hierarchical decomposition of M, which we describe recursively. Let i be the smallest index such that $m_i = \max\{m_1,\ldots,m_d\}$. If $m_i = 1$ then we have reached the end of the recursion. Otherwise, we partition M into two non-overlapping submeshes $M(m_1,\ldots,\lceil m_i/2\rceil,\ldots,m_d)$ and $M(m_1,\ldots,\lfloor m_i/2\rfloor,\ldots,m_d)$. These submeshes are then decomposed recursively according to the same rules. Figure 2.1 on Page 16 gives an example for this decomposition.

The hierarchical decomposition has associated with it a decomposition tree $T(M)$, in which each node corresponds to one of the submeshes, i.e., the root of $T(M)$ corresponds to M itself, and the children of a node v in the tree correspond to the two submeshes into which the submesh corresponding to v is divided. Thus, $T(M)$ is a binary tree of height $O(\log n)$ in which the leaves correspond to submeshes of size one, i.e., to the nodes of M. We define the root to be on level 0 of this tree, and all nodes whose parents are on level i are defined to be on level $i+1$. For each node v in $T(M)$, let $M(v)$ denote the corresponding submesh. Furthermore, each edge e of $T(M)$ connecting a level i node u with a level $i+1$ node v is defined to be on level $i+1$ and $M(e) = M(v)$.

We interpret $T(M)$ as a virtual network that we want to simulate on M. In order to compare the congestion in both networks we define bandwidths for the edges in $T(M)$, i.e., for an edge e of $T(M)$, define the bandwidth of e to be the number of edges leaving submesh $M(e)$. Figure 2.2 on Page 17 gives an example of a decomposition tree with bandwidths.

For each object $x \in X$, define an access tree $T_x(M)$ to be a copy of the decomposition tree $T(M)$. We embed the access trees randomly into M, i.e., for each $x \in X$, each interior node v of $T_x(M)$ is mapped by a random hash function $h(x,v)$ to one of the processors in $M(v)$, and each leaf v of $T_x(M)$ is mapped onto the only processor in $M(v)$. For simplicity, we assume that the hash functions map in a truly random fashion, i.e., uniformly and independently.

The remaining description of our data management strategy is very simple: For object $x \in X$, we execute the tree strategy on the access tree $T_x(M)$. All messages that should be sent between neighboring nodes in the access trees are sent along the *dimension-by-dimension order path* between the associated nodes in the mesh, i.e., the unique shortest path between the two nodes using first edges of dimension 1, then edges of dimension 2, and so on.

The following lemma gives an upper bound on the maximum expected load in M due to the access tree strategy.

Theorem 2.6 *For each application on the d-dimensional mesh M with n nodes, the access tree strategy achieves maximum expected load $O(d \cdot \log n \cdot C_{\text{opt}}(M))$, where $C_{\text{opt}}(M)$ denotes the optimal congestion for the application.*

Proof. In order to prove the above result, we require a lower bound on the congestion of an optimal strategy and an upper bound on the expected load of the access tree strategy. Define $C_{\text{opt}}(T(M))$ to be the optimal congestion for the application when it is executed on the binary tree $T(M)$, under the assumption that each processor of M is simulated by its counterpart in $T(M)$, which is one of the leaf nodes. Note that $T(M)$ is used only as a tool in the proof; in our caching strategy, we actually use a separate tree $T_x(M)$ for each data object x. We give a lower bound that relates the optimal congestion on the mesh to $C_{\text{opt}}(T(M))$, and an upper bound that relates the expected load of the access tree strategy to $C_{\text{opt}}(T(M))$. We start with the lower bound.

Lemma 2.7 $C_{\text{opt}}(T(M)) \leq C_{\text{opt}}(M)$.

Proof. For a given strategy on M with congestion C we have to describe a strategy on $T(M)$ with congestion at most C.

We simulate the strategy for M on $T(M)$, except for the routing. Instead, for the routing paths in the mesh we use the unique shortest paths between the respective nodes, that is, whenever a message is to be routed between two mesh nodes, we instead route the same message along the unique shortest path in the tree between the leaf nodes corresponding to these mesh nodes. Let C' denote the congestion for a given application with the above strategy on $T(M)$. Let e denote an edge of $T(M)$ with relative load C'. Then the absolute load of e is $C' \cdot b(e)$.

Now consider the same application on M. Any message that crosses e in $T(M)$ has either to leave or to enter the submesh $M(e)$ in M. The number of edges leaving $M(e)$ is $b(e)$. Thus, the load on one of these edges is at least $C' \cdot b(e)/b(e) = C'$, and hence, $C \geq C'$. ∎

The following lemma gives the upper bound on the expected load of the access tree strategy.

Lemma 2.8 *The maximum expected load in M due to the access tree strategy is* $O(\log n \cdot d \cdot C_{\text{opt}}(T(M)))$.

Proof. Fix an arbitrary edge e in M. Let h denote the height of $T(M)$, and let $L_\ell(e)$ denote the load on e due to the simulation of edges on level ℓ of $T(M)$, for $1 \leq \ell \leq h$. We show that $E[L_\ell(e)] = O(d \cdot C_{\text{opt}}(T(M)))$, for $1 \leq \ell \leq h$, which yields the lemma as $h = O(\log n)$.

Fix a level ℓ. Let v be a node of $T(M)$ on level $\ell - 1$ such that $M(v)$ includes the edge e. If such a node does not exist then $E[L_\ell(e)] = 0$. Let v' be one of the two children of v. We bound the expected load on e due to the simulation of the tree edge $e_T = \{v, v'\}$ on level ℓ of $T(M)$.

First, we give a bound on the probability $P(e)$ that e is traversed by a dimension-by-dimension order path connecting the mesh nodes that simulate v and v'. The mesh $M(v)$ is partitioned by the hierarchical decomposition into two submeshes, one of which is $M(v')$. Let q_i denote the side length of $M(v')$ in dimension i. Let k denote the dimension of edge e. The nodes v and v' are embedded randomly into $M(v)$ and $M(v')$. $P(v)$ is bounded by the probability that the dimension-by-dimension order path between the host of v and the host of v' traverses *the row of e*, i.e., the maximum set of edges of dimension k that build a linear array that includes e. Regardless of whether the path starts at v or v', the number of reachable rows in dimension k with respect to the random embedding is at least $\prod_{i=1, i\neq k}^{d} q_i$. As each of the these rows is equally likely to be traversed,

$$P(e) \leq \frac{1}{\prod_{i=1, i\neq k}^{d} q_i} \leq \frac{q_k}{\prod_{i=1}^{d} q_i} \leq \frac{q_{\max}}{\prod_{i=1}^{d} q_i}$$

with $q_{\max} = \max_{1 \leq i \leq d}\{q_i\}$.

Next we relate this probability to the bandwidth $b(e_T)$ of the tree edge e_T. This bandwidth is defined to be the number of edges leaving $M(v')$. As a consequence,

$$\begin{aligned} b(e_T) &\leq \sum_{\substack{j=1 \\ q_j \geq \lfloor q_{\max}/2 \rfloor}}^{d} 2 \cdot \frac{\prod_{i=1}^{d} q_i}{q_j} \\ &\leq \sum_{\substack{j=1 \\ q_j \geq \lfloor q_{\max}/2 \rfloor}}^{d} 2 \cdot \frac{\prod_{i=1}^{d} q_i}{\lfloor q_{\max}/2 \rfloor} \\ &\leq 6d \cdot \frac{\prod_{i=1}^{d} q_i}{q_{\max}} \\ &\leq \frac{6d}{P(e)} \ . \end{aligned}$$

Note that we are allowed to exclude any dimension j with $q_j < \lfloor q_{\max}/2 \rfloor$ from the above sum because the hierarchical mesh decomposition ensures that the mesh is not divided along these dimensions before the decomposition of $M(v)$ on level $\ell - 1$ such that $M(v')$, which results from this decomposition, has no outgoing edges in one of these dimensions.

The maximum number of messages that are transmitted along the tree edge e_T is $3 \cdot C_{\text{opt}}(T(M)) \cdot b(e_T)$, since the tree strategy is strictly 3-competitive. Consequently, the expected load on edge e for simulating e_T is at most

$$3 \cdot C_{\text{opt}}(T(M)) \cdot b(e_T) \cdot P(e) \leq 18d \cdot C_{\text{opt}}(T(M)) \ .$$

The same bound holds for the edge connecting v with its other child. Hence, $E[L_\ell(e)] = O(d \cdot C_{\text{opt}}(T(M)))$, which yields the lemma. ■

Applying Lemma 2.7 and Lemma 2.8 yields Theorem 2.6. ■

The general remapping scheme

In this section, we introduce the *general remapping scheme*. This scheme is presented in a more general fashion such that it can be used for all networks, not only for meshes.

Consider a network G and a hierarchical decomposition of G. The *associated decomposition tree* $T(G)$ is the tree in which each node corresponds to one of the subgraphs, i.e., the root corresponds to G itself, the children of a node v correspond to the subgraph into which the subgraph corresponding to v is divided, and the leaves correspond to subgraphs of size one, i.e., to the nodes of G. We define the root of $T(G)$ to be on level 0, and all nodes whose parents are on level i are defined to be on level $i+1$. For each node v in $T(G)$, let $G(v)$ denote the corresponding subgraph. Furthermore, each edge e in $T(G)$ connecting a level i node u with a level $i+1$ node v is defined to be on level $i+1$ and $G(e) = G(v)$. Finally, we define the bandwidths, for each edges e in $T(G)$, to be the number of edges leaving subgraph $G(e)$.

For each object $x \in X$, define an access tree $T_x(G)$ to be a copy of the decomposition tree $T(G)$. We embed the access trees randomly into G, i.e., for each $x \in X$, each interior node v of $T_x(G)$ is mapped uniformly at random to one of the nodes in $G(v)$, and each leaf v of $T_x(G)$ is mapped onto the only node in $G(v)$.

The remaining description of our caching strategy is very simple: For each object $x \in X$, we execute the tree strategy on the access tree $T_x(G)$. All messages that should be sent between neighboring nodes in the access trees are sent along a path in G.

The access tree nodes have to be remapped when too many *access messages*, i.e., messages that simulate messages of the tree strategy, traverse a node. The remapping is done as follows. For every object x, and every node v of the access tree $T_x(G)$ we add a counter $r(x,v)$. Initially, this counter is set to 0. Whenever an access message for object x traverses node v, starts at node v, or arrives at node v, the counter $r(x,v)$ is increased by D, if this message is a migration message, and the counter $r(x,v)$ is increased by 1, otherwise. When the counter $r(x,v)$ reaches K the node v is remapped randomly to another node in $G(v)$, where K is some integer with $K = \Theta(\deg(T(G)) \cdot \Delta(T(G)) \cdot D)$, where $\deg(T(G))$ denotes the degree of $T(G)$, and $\Delta(T(G))$ denotes the maximum factor by which the bandwidths of two edges from $T(G)$ incident on the same node deviate. Remapping v to a new host means that we have to send a *remapping message* that informs the new host about the migration and, if the old host holds a copy of x, moves the copy to the new host. Remapping messages reset the counter $r(x,v)$ to 0. Furthermore, we have to send *notification messages* including information about the new host to the nodes in G that hold the access tree neighbors of v. These notification messages also increase the counters at their destination nodes, i.e., the counter $r(x,v')$, for each neighbor v' of v in the access tree, by 1. The counter mechanism ensures that the load directed to a copy on a randomly selected host is $O(K)$ rather than $\Theta(\kappa)$, where κ denotes the maximum number of write requests directed to the same object. Note that this bound would fail if the notification messages would not increase the counters.

For a given application on G let $C_{\text{opt}}(G)$ denote the optimal congestion for the application. Further, let $C_{\text{opt}}(T(G))$ denote the optimal congestion for the application when it is executed on the tree $T(G)$, under the assumption that each node of G is simulated by its counterpart in $T(G)$, which is one of the leaf nodes. The decomposition tree $T(G)$ is called Ψ_G*-useful* if, for every application, the following two conditions are fulfilled.

- $C_{\text{opt}}(T(G)) \leq C_{\text{opt}}(G)$.
- The maximum expected relative load in G due to access messages is $\Psi_G \cdot C_{\text{opt}}(T(G))$.

The load on the edges in G is increased by the notification and remapping messages. However, the following lemma shows that the impact of these messages on the load on the edges is relatively small.

Lemma 2.9 *Consider a network G with n nodes, and a Ψ_G-useful decomposition tree $T(G)$ of G. Then, for each application, the access tree strategy achieves congestion $O(\Psi_G \cdot C_{\text{opt}}(G) + \min\{\deg(T(G)) \cdot \Delta(T(G)) \cdot D, \kappa\} \cdot \deg(T(G)) \cdot \log n)$, w.h.p., where κ denotes the maximum number of write requests directed to the same object.*

Before we prove Lemma 2.9, we apply it to the mesh M. We can conclude with Lemma 2.7 and Lemma 2.8 that the decomposition tree $T(M)$ is $O(d \cdot \log n)$-useful. In addition, note that $\deg(T(M)) = 3$ and $\Delta(T(M)) = 2$. Then, we can conclude with Lemma 2.9 that the access tree strategy achieves congestion $O(d \cdot \log n \cdot C_{\text{opt}}(M) + \min\{D, \kappa\} \cdot \log n)$, w.h.p., where κ denotes the maximum number of write requests directed to the same object.

To yield the following theorem we have to relate the optimal congestion $C_{\text{opt}}(M)$ and $\min\{D, \kappa\}$. Either an object x is migrated at least once or each request message to x is directed to the node where the initial copy of x is placed. As the nodes in the mesh have maximum degree $2d$, it follows $2d \cdot C_{\text{opt}}(M) \geq \min\{D, \kappa\}$. Thus, the access tree strategy achieves congestion $O(d \cdot \log n \cdot C_{\text{opt}}(M))$, w.h.p., i.e., it is strictly $O(d \cdot \log n)$-competitive, w.h.p., for the d-dimensional mesh M with n nodes.

Theorem 2.10 *For every application on the d-dimensional mesh M with n nodes, the access tree strategy achieves congestion $O(d \cdot \log n \cdot C_{\text{opt}}(M))$, w.h.p., where $C_{\text{opt}}(M)$ denotes the optimal congestion for the application.*

Proof (of Lemma 2.9). First, we consider the notification messages. We show that the congestion due to notification messages in $T(G)$ is not larger than the congestion due to access messages. Note that $T(G)$ is used as a tool in the proof and actually a separate tree $T_x(G)$ is used for each data object.

Let N denote the maximum number of notification messages passing a single edge of $T(G)$. Suppose e_T is an edge of $T(G)$ that transmits N notification messages. Each of the notification messages traversing e_T was caused by at least load K due to access or notification messages that pass one of the two nodes incident on e_T. Therefore, at least one of the nodes incident on e_T is touched by at least load $N \cdot K/2$ due to access or notification messages. Hence, the load on at least one of the at most $\deg(T(G))$ tree edges incident to this node is at least $N \cdot K/(2 \cdot \deg(T(G)))$. Let e'_T denote one of the incident edges fulfilling this property. For $K \geq 4 \cdot \deg(T(G)) \cdot \Delta(T(G))$, the load on e'_T due to access or notification messages is at least $N \cdot 2 \cdot \Delta(T(G))$. Then the load on e'_T due to access messages is at least $N \cdot \Delta(T(G))$ because the maximum number of notification messages that cross e'_T is N.

By definition is $b(e_T) \geq b(e'_T)/\Delta(T(G))$. Hence, $N/b(e_T) \leq N \cdot \Delta(T(G))/b(e'_T)$, i.e., the relative load on e_T due to notification messages is not larger than the relative load on e'_T due to access messages. As we execute the strictly 3-competitive tree strategy on each access tree, we can conclude that the expected relative load due to access and notification messages on every edge in $T(G)$ is in $O(C_{\text{opt}}(T(G)))$. Thus, the expected load due to access and notification messages on an arbitrary edge of G is bounded by $O(\Psi_G \cdot C_{\text{opt}}(G))$ since $T(G)$ is a Ψ_G-useful decomposition tree of G.

Until now, we have not considered the remapping messages. These messages have no direct counterpart in the decomposition tree $T(G)$. However, for each remapping message we can specify some related access or notification messages that are sent along the edges of the access trees: Consider a node v of the access tree $T_x(G)$ of an object $x \in X$. Every time the counter $r(v,x)$ reaches K the node v is remapped and a remapping message is sent from the old host to the new host of v. $r(v,x)$ is increased whenever an access or notification message traverses v. Hence, each remapping message is caused by at least load K touching node v.

Again, we merge together the access trees of all objects to form the virtual decomposition tree $T(G)$. Fix a node v from this tree. Let $H(v)$ denote the number of remapping messages sent for all access tree nodes corresponding to v. The remapping messages sent for v only influence the load on edges in $G(v)$. In detail, a remapping message for v is sent between two randomly chosen nodes in the subgraph $G(v)$. Hence, for the expected load on an edge e of $G(v)$ this has the same effect as sending a remapping message along an edge in $T(G)$ leading from v to an uniformly at random chosen child of v. As we are primarily interested in the expected load on the edges in G, the remapping messages for v can be related to additional load on the edges in $T(G)$ leading from v to the children of v. Let e_T denote one of these edges. Then the additional load on e_T due to remapping messages is at most $H(v) \cdot D$.

The total load due to access and notification messages that are sent along the at most $\deg(T(G))$ tree edges incident on v is at least $H(v) \cdot K$. Thus, one of these edges has at least load $H(v) \cdot K/\deg(T(G))$. Let e'_T denote one of these edges fulfilling this property. For $K \geq \deg(T(G)) \cdot \Delta(T(G)) \cdot D$, the load on e'_T due to access or notification messages is at least $H(v) \cdot \Delta(T(G)) \cdot D$.

By definition is $b(e_T) \geq b(e'_T)/\Delta(T(G))$. Hence, $H(v) \cdot D/b(e_T) \leq H(v) \cdot D \cdot \Delta(T(G))/b(e'_T)$, i.e., the relative load on e_T due to remapping messages is not larger than the relative load on e'_T due to access and notification messages. As we execute the strictly 3-competitive tree strategy on each access tree, we can conclude that the expected relative load due to access, notification and remapping messages on every edge in $T(G)$ is in $O(C_{\text{opt}}(T(G)))$. Thus, the expected load due to access, notification, and remapping messages on an arbitrary edge of G is bounded by $O(\Psi_G \cdot C_{\text{opt}}(G))$ since $T(G)$ is a Ψ_G-useful decomposition tree of G.

Finally, let $L(e)$ denote the load on an arbitrary edge e of G due to all kind of messages. We have shown that

$$E[L(e)] = O(\Psi_G \cdot C_{\text{opt}}(M)) \ .$$

In order to complete the proof of this lemma, we have to show that the maximum load over all edges does not deviate too much from the expected load of an arbitrary edge.

We show that $L(e)$ can be decomposed into $\deg(T(G))$ parts such that $L(e) = \sum_{i=1}^{\deg(T(G))} L_i$, and each L_i is a sum of independent random variables. Then, we can apply a Chernoff bound to each L_i. We color the edges of the access trees with the colors $\{1,\ldots,\deg(T(G))\}$ such that all edges incident on the same node have different colors.

Fix a color $j \in \{1,\ldots,\deg(T(G))\}$. Let E_x be the set of tree edges of $T_x(G)$ with this color. For $e_T \in E_x$, let $A(e_T,x,i)$ be a random variable that is $1/b(e)$ if the path that simulates e_T includes e when the ith message corresponding to object x traverses e_T. Here a migration or remapping message of an object x is interpreted as D small messages each of which having unit size. Now define

$$L_j := \sum_{x\in X} \sum_{e_T\in E_x} \sum_i A(e_T,x,i) \quad .$$

Obviously, all of the random variables $A(e_T,x,i)$ for different x are stochastically independent. The coloring yields that all random variables for different e_T are independent, too. Further, the remapping ensures that the set of all random variables with fixed e_T and x can be partitioned into subsets of maximum size W, with $W = O(\min\{K,\kappa\})$, such that the values of all variables in the same subset are equal and any collection of variables from different subsets includes only independent variables. Recall that κ denotes the maximum number of write requests directed to the same object.

Hence, the term L_j can be viewed as a sum of independent random variables of maximum weight W. Applying a Chernoff bound (see, e.g., [HR90]) to this sum yields that $L_j = O(E(L_j) + W \cdot \log n)$, w.h.p., and therefore,

$$\begin{aligned} L(e) &= \sum_{i=1}^{\deg(T(G))} L_i \\ &= O(E(L) + \deg(T(G)) \cdot W \cdot \log n) \\ &= O(\Psi_G \cdot C_{\text{opt}}(M) + \deg(T(G)) \cdot W \cdot \log n) \quad , \end{aligned}$$

w.h.p. This completes the proof of Lemma 2.9. ■

Lower bound

The theorem in the previous section shows that the access tree strategy is strictly $O(d \cdot \log n)$-competitive. In the following, we give a lower bound for on-line routing which shows that this factor is nearly optimal.

We consider the following *on-line routing problem*: An adversary specifies a sequence of Δ routing requests, i.e., pairs $r_t = (s_t, d_t)$ of source and destination nodes, for $1 \le t \le \Delta$. An on-line routing algorithm must assign a routing path connecting s_t and d_t, for $1 \le t \le \Delta$, without knowing future requests, i.e., requests $r_{t'}$ with $t' > t$. The goal is to minimize the congestion.

The following theorem gives a lower bound on the competitive ratio for on-line routing on meshes. The bound holds for each deterministic or randomized on-line algorithm. A similar bound, but only for two-dimensional meshes, has been derived independently also by Bartal and Leonardi in the context of routing in "all-optical networks" [BL97].

Theorem 2.11 *Any on-line routing algorithm for the mesh of dimension $d \ge 2$ and side length m has competitive ratio $\Omega(\log m)$.*

Caching in networks includes the problem of on-line routing, which can be shown as follows. For every message in the routing problem, we define a corresponding object x_t that is placed initially on node s_t. At time t, the node s_t writes object x_t; immediately afterwards, node d_t reads object x_t. Then some data must be routed from s_t to d_t. From this observation, we can conclude the following corollary.

Corollary 2.12 *Any caching strategy for the mesh of dimension $d \geq 2$ and side length m has competitive ratio $\Omega(\log m)$.*

Proof (of Theorem 2.11). We show that for each C, $d \geq 2$, and $m \geq 4$ being a power of 2, there is a random routing problem $P_d(m,C)$ for which the minimum congestion is C whereas the expected congestion achieved by any on-line routing algorithm is $\Omega(C \cdot \log m)$.

Let $M_d(m)$ denote the d-dimensional mesh of side length m. We start by proving a lower bound for $M_2(m)$. First, we describe a random on-line routing problem $P_2(m,C)$ on $M_2(m)$, for which the minimum off-line congestion is C. Then we show that the expected congestion of any on-line strategy is $\Omega(C \cdot \log m)$.

$P_2(m,C)$ is defined as follows. Let (k,ℓ) denote the k-th node in the ℓ-th row of $M_2(m)$, for $1 \leq k,\ell \leq m$. The adversary starts by specifying $m/2$ pairs of source and destination nodes each of which should be connected by C routing paths. The pairs are $((m/2,\ell),(m/2,m/2+\ell))$, for $1 \leq \ell \leq m/2$. Further requests are described recursively: The mesh $M_2(m)$ can be partitioned into four $m/2 \times m/2$ submeshes. If $m/2 \geq 4$, then the adversary selects one from these submeshes at random and specifies routing requests in this submesh according to $P_2(m/2,C)$.

Altogether, this gives $\log m - 1$ batches of routing requests of size $C \cdot m/2, C \cdot m/4, \ldots, C \cdot 2$. For $1 \leq i \leq \log m - 1$, the routing batch specified in stage i is denoted by R_i and the submesh considered in this stage is denoted by S_i such that $S_1 = M_2(m)$.

It is easy to check that, for $1 \leq i \leq \log m - 2$, there exists an off-line schedule that routes the requests of batch R_i with congestion C through mesh S_i without using any edge of mesh S_{i+1}. Thus, all requests can be routed off-line with congestion C.

It remains to be shown that the expected congestion of any on-line strategy is $\Omega(C \cdot \log m)$. The mesh $M_2(m)$ consists of $m-1$ rows of vertical edges and $m-1$ columns of horizontal edges, each containing m edges. We number the rows and columns from 1 to $m-1$, respectively. In the following, we only consider the edges in odd rows and odd columns. These edges are called odd edges. Each routing path connecting a source and a destination node of a request in batch R_i has to traverse at least $m/2^{i+1}$ odd edges of submesh S_i. Note that this bound holds even if a path leaves the submesh S_i. Hence, if one chooses randomly and uniformly an edge from the odd edges in S_i, then the expected number of paths connecting two nodes of batch R_i using this edge is at least

$$\frac{|R_i| \cdot m/2^{i+1}}{|E_{\text{odd}}(S_i)|} = \frac{(C \cdot m/2^i) \cdot m/2^{i+1}}{m^2/2^{2i-2}} = \frac{C}{8}$$

with $E_{\text{odd}}(S_i)$ denoting the set of odd edges in S_i, for $1 \leq i \leq \log m - 1$. Choosing a random odd edge from $S_{\log m-1}$ rather than from S_i yields the same bound, because

the selection of the submeshes by the adversary corresponds to random selections of subsets of odd edges. Note that this holds only for the odd edges because if $m/2$ is even, then all of the odd edges in an $m \times m$ mesh will be contained in the $m/2 \times m/2$ submeshes, while all of the edges going between different submeshes are even. Hence, the expected congestion in $S_{\log m-1}$ due to requests from batch R_i is $C/8$, for $1 \leq i \leq \log m - 1$. Summing over all batches, yields that the expected congestion of $P_2(m,C)$ is $C \cdot (\log m - 1)/8$, which completes the proof for the 2-dimensional case.

We now describe the on-line routing problem $P_d(m,C)$ for the mesh $M_d(m)$ with $d \geq 3$. $M_d(m)$ can be partitioned into $J = m^{d-2}$ two-dimensional $m \times m$ submeshes $M_1, \ldots, M_J$, each of which consists only of edges of dimension 1 and 2. $P_d(m,C)$ is defined as follows. The adversary specifies the routing requests in each submesh M_j according to $P_2(m,C)$, for $1 \leq j \leq J$. In each submesh M_j it uses the same random bits, which means that it specifies exactly the same routing problem in each M_j, for $1 \leq j \leq J$. We have already seen that all requests can be routed off-line with congestion C inside the respective 2-dimensional mesh M_j. Hence, it remains to be shown that the expected congestion of any on-line strategy on this routing problem is $\Omega(C \cdot \log m)$. This we do by contradiction.

Suppose an on-line routing strategy exists for which the expected congestion on $P_d(m,C)$ is smaller than $C \cdot (\log m - 1)/8$. Then this routing strategy can be simulated on the $m \times m$ mesh $M_2(m)$ for $P_2(m, m^{d-2} \cdot C)$. In this simulation, the nodes and the edges of $M_2(m)$ simulate their respective counterparts in the meshes $M_1, \ldots, M_J$. The simulation yields congestion smaller than $m^{d-2} \cdot C \cdot (\log m - 1)/8$, since each edge of $M_2(m)$ has to simulate only m^{d-2} edges of $M_d(m)$. This contradicts the above result for $P_2(m, m^{d-2} \cdot C)$. Consequently, the expected congestion of any on-line strategy on $P_d(m,C)$ is at least $C \cdot (\log m - 1)/8$. ■

2.3.2 Fat-trees

In this section, we consider caching strategies for the fat-tree F of height H with n nodes. F has the topology of a symmetric tree, that is, for each inner-node, the subtrees rooted at the children of the node are isomorph. The fat-tree represents an indirect network, that is, only the leaf nodes are processors with memory modules, the inner nodes are only routing switches. We define the root of F to be on level 0, and all nodes whose parents are on level i are defined to be on level $i+1$. Furthermore, each edge e of F connecting a level i node with a level $i+1$ node is defined to be on level $i+1$. Thus, $F = F((d_1,b_1), \ldots, (d_H,b_H))$, i.e., for every level i in the tree F, each node on level i has d_{i+1} children and each edge on level i has bandwidth b_i. We make the reasonable assumption, for every level i, $b_i \geq b_{i+1}$.

The following lemma shows that the bandwidths of a fat-tree can be restricted without increasing the congestion.

Lemma 2.13 *W.l.o.g. we can assume, for every level i, $b_i \leq d_{i+1} \cdot b_{i+1}$.*

Proof. We have to show that a given strategy with congestion C on the fat-tree F has on the fat-tree $F' = F((d_1,b_1),\ldots,(d_\ell,\min\{b_\ell,d_{\ell+1}\cdot b_{\ell+1}\}),\ldots,(d_H,b_H))$ at most congestion C.

Fix an edge e' on level ℓ in F'. Let $e_1,\ldots,e_{d_{\ell+1}}$ denote the edge on level $\ell+1$ in F' that are incident to e, and, for an edge e, let $L(e)$ denote the load on e.

Suppose that $b_\ell = \min\{b_\ell, d_{\ell+1}\cdot b_{\ell+1}\}$. Then the lemma holds obviously. Now, suppose that $d_{\ell+1}\cdot b_{\ell+1} = \min\{b_\ell, d_{\ell+1}\cdot b_{\ell+1}\}$. Any message that crosses e' has to cross one of the edges $e_1,\ldots,e_{d_{\ell+1}}$, too. Thus, $L(e') \leq \sum_{i=1}^{d_{\ell+1}} L(e_i)$. The relative load of an edge is defined to be the load divided by the bandwidth of the edge. Hence, the relative load on e' is

$$\begin{aligned}\frac{L(e')}{d_{\ell+1}\cdot b_{\ell+1}} &\leq \frac{\sum_{i=1}^{d_{\ell+1}} L(e_i)}{d_{\ell+1}\cdot b_{\ell+1}} \\ &\leq \frac{\max\{L(e_1),\ldots,L(e_{d_{\ell+1}})\}}{b_{\ell+1}} \\ &\leq C\ ,\end{aligned}$$

which yields the lemma. ∎

The fat-tree F is denoted *realistic* if the bandwidths of the levels decrease geometrically in direction to the root, i.e., if, for every level i, $b_i \leq \alpha\cdot d_{i+1}\cdot b_{i+1}$, for some constant $\alpha < 1$.

The access tree strategy uses a locality preserving embedding of access trees. It is based on a hierarchical decomposition of F, which we describe recursively. If F has size one then we have reached the end of the recursion. Otherwise, we partition F into d_1 non-overlapping subtrees $F_1,\ldots,F_{d_1}$ with $F_i = F((d_2,b_2),\ldots,(d_H,b_H))$. These subtrees are then decomposed recursively according to the same rules.

The associated decomposition tree $T(F)$ is isomorph to F. Since we obtain, for fat-trees, a tradeoff between the height of the decomposition tree and the achieved congestion, we need a decomposition tree of arbitrary height $1 \leq h \leq H$. Thus, some levels of $T(F)$ have to be deleted possibly. This is done with the following *deletion algorithm*. While $\text{height}(T(F)) > h$, delete a level $0 < \ell < \text{height}(T(F))$ in $T(F)$ whose nodes have minimum degree. Deleting level ℓ means that, first, all nodes and edges on level ℓ are deleted, then the edges on level $\ell+1$ are connected to the respective nodes on level $\ell-1$, and finally the numbering of the levels is adjusted.

Now, the decomposition tree $T(F)$ has height h. The root of $T(F)$ corresponds still to F itself, and the children of a node v in the tree correspond to the subtrees into which the subtree corresponding to v is divided, and the leaves correspond to subtrees of size one, i.e., to the leaf nodes of F. Note that $T(F)$ is still isomorph to a fat-tree.

The remaining description of our caching strategy is analogous to the caching strategy for meshes in Section 2.3.1. Note that all access tree nodes are mapped only to leaf nodes of the fat-tree, since the inner nodes are only routing switches, and that all messages that should be sent between neighboring nodes in the access trees are sent along the unique shortest path between the associated nodes in the fat-tree.

The following theorem shows that we obtain, for $h = \min\{H, \log n / \log\log n\}$, an access tree strategy that is $O(\min\{H, \log n / \log\log n\})$-competitive, w.h.p., for fat-trees of height H with n nodes. For realistic fat-trees this result improves to $O(1)$-competitive, w.h.p. Further, we obtain a caching strategy that is strictly $O(\deg(F)^3 \cdot \log n)$-competitive, w.h.p. To yield the latter result we have to relate the optimal congestion $C_{\text{opt}}(F)$ and $\min\{D, \kappa\}$. Either an object x is migrated at least once or each request message to x is directed to the node where the initial copy of x is placed. As the nodes in the fat-tree have maximum degree $\deg(F)$, it follows $C_{\text{opt}}(F) \geq \min\{D, \kappa/\deg(F)\}$.

Theorem 2.14 *Consider an application on the fat-tree F of height H with n nodes. Let $C_{\text{opt}}(F)$ denote the optimal congestion for the application, and let κ denote the maximum number of write requests directed to the same object.*

- *For each $1 \leq h \leq H$, we obtain an access tree strategy with congestion $O((h + n^{1/h}) \cdot C_{\text{opt}}(F) + R)$, w.h.p., and, if $h = H$, with congestion $O(H \cdot C_{\text{opt}}(F) + R)$, w.h.p., with $R = \min\{(\deg(F) + n^{1/h})^2 \cdot D, \kappa\} \cdot (\deg(F) + n^{1/h}) \cdot \log n$.*
- *For realistic fat-trees this improves to $O(n^{1/h} \cdot C_{\text{opt}}(F) + R)$, w.h.p., and, if $h = H$, to $O(C_{\text{opt}}(F) + R)$, w.h.p.*

Proof. In order to prove the above result we apply Lemma 2.9. Fix an $1 \leq h \leq H$. Define $C_{\text{opt}}(T(F))$ to be the optimal congestion for the application when it is executed on the tree $T(F)$, under the assumption that each leaf node of F is simulated by its counterpart in $T(F)$, which is one of the leaf nodes. Note that $T(F)$ is used only as a tool in the proof and that we actually use a separate tree for each data object. According to Lemma 2.13 we can assume w.l.o.g., for each level i, $b_i \leq \alpha \cdot d_{i+1} \cdot b_{i+1}$, for some constant $\alpha \leq 1$.

To apply effectively Lemma 2.9, we have to show that the decomposition tree $T(F)$ is Ψ_F-useful for a certain Ψ_F, i.e., we have to prove the following two conditions.

- $C_{\text{opt}}(T(F)) \leq C_{\text{opt}}(F)$.
- The maximum expected relative load in F due to access messages is $O((\sum_{i=1}^{h} \alpha^i + n^{1/h}) \cdot C_{\text{opt}}(T(F)))$, and, if $h = H$, $O(\sum_{i=1}^{H} \alpha^i \cdot C_{\text{opt}}(T(F)))$.

In addition, note that $\Delta(T(F)) = \deg(T(F)) = \max\{\deg(F), n^{1/h}\}$. Then, Theorem 2.14 can be yield with Lemma 2.9.

The next lemma shows the lower bound for the optimal strategy.

Lemma 2.15 $C_{\text{opt}}(T(F)) \leq C_{\text{opt}}(F)$.

Proof. For a given strategy on F with congestion C we have to describe a strategy on $T(F)$ with congestion at most C.

We simulate the strategy for F on $T(F)$. For the routing paths in $T(F)$ we use the unique shortest paths between the respective nodes, that is, whenever a message is to

be routed between two fat-tree nodes, we instead route the same message along the unique shortest path in the tree between the leaf nodes corresponding to these fat-tree nodes. Let C' denote the congestion for a given application with the above strategy on $T(F)$. Let e denote an edge of $T(F)$ with relative load C'. Then the absolute load of e is $C' \cdot b(e)$.

Now consider the same application on F. Any message that crosses e in $T(F)$ has either to leave or to enter the subtree $F(e)$. The bandwidth of the edge leaving $F(e)$ is $b(e)$. Thus, the load on this edge is at least $C' \cdot b(e)/b(e) = C'$, and hence, $C \geq C'$. ■

The following lemma gives the upper bound on the expected load of the access tree strategy.

Lemma 2.16 *The maximum expected relative load in F due to access messages is $O((\sum_{i=1}^{h} \alpha^i + n^{1/h}) \cdot C_{\text{opt}}(T(F)))$, and, if $h = H$, $O(\sum_{i=1}^{H} \alpha^i \cdot C_{\text{opt}}(T(F)))$.*

Proof. Fix an arbitrary edge e in F and a level ℓ in $T(F)$. Let $L_\ell(e)$ denote the relative load on e due to the simulation of access messages passing edges on level ℓ of $T(F)$. Let v be a node of $T(F)$ on level $\ell - 1$ such that $F(v)$ includes the edge e. If such a node does not exist then $E[L_\ell(e)] = 0$. For each node u in $T(F)$, let the root of subtree $F(v)$ in F denote $F_r(v)$. Let ℓ_v denote the level of $F_r(v)$ in F. For all the children $v_1, \ldots, v_d$ of v in $T(F)$, the nodes $F_r(v_1), \ldots, F_r(v_d)$ are on the same level in F. Thus, let ℓ_c denote this level. Further, Let ℓ_e denote the level of e in F. We show that $E[L_\ell(e)] = O(\alpha^{H-\ell} \cdot C_{\text{opt}}(T(F)))$, if $\ell_e \geq \ell_c$, and $E[L_\ell(e)] = O(n^{1/h} \cdot C_{\text{opt}}(T(F)))$, if $\ell_c > \ell_e > \ell_v$, which yields the lemma, since the latter case can only occur for at most one level in $T(F)$.

First, suppose that $\ell_e \geq \ell_c$. Then, a child v' of v exists such that $F(v')$ includes the edge e. e is traversed by paths connecting the fat-tree nodes that simulate v and its children $v_1, \ldots, v_d$, if the host of v or the host of v' is mapped on a leaf node in $F(e)$, the respective subtree of the original decomposition tree of height H. The nodes v and v' are mapped randomly to one of the leaf nodes in $F(v)$ and $F(v')$. Thus, for the probability $P(v)$ that the host of v is mapped on a leaf node in $F(e)$ is

$$P(v) \leq \frac{\prod_{i=\ell_e+1}^{H} d_i}{\prod_{i=\ell_v+1}^{H} d_i} \leq \frac{1}{\prod_{i=\ell_v+1}^{\ell_e} d_i} \quad ,$$

and for the probability $P(v')$ that the host of v' is mapped on a leaf node in $F(e)$ is

$$P(v') \leq \frac{\prod_{i=\ell_e+1}^{H} d_i}{\prod_{i=\ell_c+1}^{H} d_i} \leq \frac{1}{\prod_{i=\ell_c+1}^{\ell_e} d_i} \quad .$$

Next, we relate the bandwidth $b(e_T)$ of an edge e_T connecting the node v with a child in $T(F)$ to the bandwidth $b(e) = b_{\ell_e}$. According to Lemma 2.13 we can assume w.l.o.g.,

for each level i, $b_i \leq \alpha \cdot d_{i+1} \cdot b_{i+1}$, for some constant $\alpha \leq 1$. As a consequence,

$$b(e_T) = b_{\ell_c} \leq b_{\ell_e} \cdot \alpha^{\ell_e - \ell_c} \cdot \prod_{i=\ell_c+1}^{\ell_e} d_i \ .$$

The maximum load on the edge e_T is $C_{\text{opt}}(T(F)) \cdot b(e_T)$. Consequently, the expected load on edge e for simulating edges on level ℓ of $T(F)$ is at most

$$\begin{aligned} \frac{C_{\text{opt}}(T(F)) \cdot b(e_T) \cdot (d \cdot P(v) + P(v'))}{b_{\ell_e}} &\leq C_{\text{opt}}(T(F)) \cdot 2\alpha^{\ell_e - \ell_c} \\ &\leq C_{\text{opt}}(T(F)) \cdot 2\alpha^{H-\ell} \ , \end{aligned}$$

with $d = \prod_{i=\ell_c+1}^{\ell_v} d_i$.

Finally, suppose that $\ell_c > \ell_e > \ell_v$. Then, the levels $\ell_c - 1, \ldots, \ell_v + 1$ of F must have been deleted in $T(F)$. We assume that e is traversed by all paths connecting the fat-tree nodes that simulate v and its children $v_1, \ldots, v_d$. Now, we relate the bandwidth $b(e_T)$ of an edge e_T connecting the node v with a child in $T(F)$ to the bandwidth $b(e) = b_{\ell_e}$. According to the definition of the fat-trees, for each level i, $b_i \geq b_{i+1}$. As a consequence,

$$b(e_T) = b_{\ell_c} \leq b_{\ell_e} \ .$$

The maximum load on the edge e_T is $C_{\text{opt}}(T(F)) \cdot b(e_T)$. Consequently, the expected load on edge e for simulating edges on level ℓ of $T(F)$ is at most

$$\frac{C_{\text{opt}}(T(F)) \cdot b(e_T) \cdot d}{b_{\ell_e}} \leq C_{\text{opt}}(T(F)) \cdot d \leq C_{\text{opt}}(T(F)) \cdot 2n^{1/h} \ .$$

Note that the deletion algorithm ensures that $d \leq 2n^{1/h}$ because this algorithm deletes always a level in $T(F)$ whose nodes have minimum degree. ■

This completes the proof of Theorem 2.14. ■

2.3.3 Complete networks

Some massively parallel computers, e.g., Cray T3E and T3D or Intel Paragon, have a network with very high bandwidth, so that not the network is the bottleneck but the individual memory modules. In particular, remote accesses to these modules are expensive, as local accesses are supported by additional local caches. These systems are well modeled by a complete network G_n with n nodes. We aim to minimize the congestion at the nodes due to remote accesses, that is, remote accesses increase the load at any node whereas local accesses are free. Each node has bandwidth 1. Thus, the relative and absolute load of a node are identical.

The access tree strategy uses a locality preserving embedding of access trees. It is based on a hierarchical decomposition of G_n, which we describe recursively. If G_n

has size one, i.e., $n = 1$, then we have reached the end of the recursion. Otherwise, we partition G_n into 2 non-overlapping complete networks $G_{\lceil n/2 \rceil}$ and $G_{\lfloor n/2 \rfloor}$. These complete networks are then decomposed recursively according to the same rules.

The associated decomposition tree $T(G_n)$ is a binary tree of height $O(\log n)$. Note that the bandwidth of a node v in $T(G_n)$ is the number of the nodes in $G_n(v)$. As for fat-trees, we need a decomposition tree of arbitrary height $1 \leq h \leq \log n$. Thus, some levels of $T(G_n)$ have to be deleted possibly. This is done with the deletion algorithm.

Now, the decomposition tree $T(G_n)$ has height h and degree $O(n^{1/h})$. Note that the root of $T(G_n)$ corresponds still to G_n itself, the children of a node v in the tree correspond to the complete networks into which the complete network corresponding to v is divided, and the leaves correspond still to complete networks of size one, i.e., to the nodes of G_n.

The remaining description of our caching strategy is analogous to the caching strategy for meshes in Section 2.3.1. Note that all messages that should be sent between neighboring nodes in the access trees are sent along the edge between the associated nodes in the complete network.

The following theorem shows that we obtain, for $h = 1$, an access tree strategy for complete networks with n nodes that is $O(1)$-competitive, w.h.p., with respect to the congestion at the nodes. Further, we obtain, for $h = \log n$, an access tree strategy that is strictly $O(\log n)$-competitive, w.h.p., with respect to the congestion at the nodes. To yield the latter result we have to relate the optimal congestion $C_{\text{opt}}(G_n)$ and $\min\{D, \kappa\}$. Either an object x is migrated at least once or each request message to x is directed to the node where the initial copy of x is placed. Thus, $C_{\text{opt}}(G_n) \geq \min\{D, \kappa\}$.

Theorem 2.17 *Consider an application on the complete network G_n with n nodes. For each $1 \leq h \leq \log n$, we obtain an access tree strategy with congestion $O(h \cdot C_{\text{opt}}(G_n) + \min\{n^{2/h} \cdot D, \kappa\} \cdot n^{1/h} \cdot \log n)$ at the nodes, w.h.p., where $C_{\text{opt}}(G_n)$ denotes the optimal congestion at the nodes for the application, and κ denotes the maximum number of write requests directed to the same object.*

Proof. In order to prove the above result we apply Lemma 2.9 which holds analogously for the congestion at the nodes. Fix an $1 \leq h \leq \log n$. Define $C_{\text{opt}}(T(G_n))$ to be the optimal congestion at the nodes for the application when it is executed on the tree $T(G_n)$, under the assumption that each node of G_n is simulated by its counterpart in $T(G_n)$, which is one of the leaf nodes. Note that $T(G_n)$ is used only as a tool in the proof and that we actually use a separate tree for each data object.

To apply effectively Lemma 2.9, we have to show that the decomposition tree $T(G_n)$ is Ψ_{G_n}-useful for a certain Ψ_{G_n}, i.e., we have to prove the following two conditions.

- $C_{\text{opt}}(T(G_n)) \leq C_{\text{opt}}(G_n)$.
- The maximum expected relative load at the nodes in G_n due to access messages is $O(h \cdot C_{\text{opt}}(T(G_n)))$.

In addition, note that $\Delta(T(G_n)) = \deg(T(G_n)) = O(n^{1/h})$. Then, Theorem 2.17 can be yield with Lemma 2.9.

The next lemma shows the lower bound for the optimal strategy.

Lemma 2.18 $C_{\text{opt}}(T(G_n)) \leq C_{\text{opt}}(G_n)$.

Proof. For a given strategy on G_n with congestion C at the nodes we have to describe a strategy on $T(G_n)$ with congestion at most C at the nodes.

We simulate the strategy for G_n on $T(G_n)$. For the routing paths in $T(G_n)$ we use the unique shortest paths between the respective nodes, that is, whenever a message is to be routed between two nodes in G_n, we instead route the same message along the unique shortest path in the tree between the leaf nodes corresponding to these nodes. Let C' denote the congestion at the nodes for a given application with the above strategy on $T(G_n)$. Let v denote a node of $T(G_n)$ with relative load C'. Then the absolute load of v is $C' \cdot b(v)$.

Now consider the same application on G_n. Any message that crosses v in $T(G_n)$ has either to leave or to enter the complete network $G_n(v)$. The number of nodes in $G_n(v)$ is $b(v)$. Thus, the load on one of these nodes is at least $C' \cdot b(e)/b(e) = C'$, and hence, $C \geq C'$. ■

The following lemma gives the upper bound on the expected load of the access tree strategy.

Lemma 2.19 *The maximum expected relative load at the nodes in G_n due to access messages is $O(h \cdot C_{\text{opt}}(T(G_n)))$.*

Proof. Fix an arbitrary node v in G_n and a level ℓ in $T(G_n)$. Let $L_\ell(v)$ denote the load on v due to the simulation of access messages passing nodes on level ℓ of $T(G_n)$. Let v_T be a node of $T(G_n)$ on level ℓ such that $G_n(v_T)$ includes the node v. If such a node does not exist then $E[L_\ell(v)] = 0$. We show that $E[L_\ell(v)] = O(C_{\text{opt}}(T(G_n)))$, which yields the lemma.

First, we give a bound for the probability $P(v)$ that the host of v_T is mapped onto v. Let $|G_n(v_T)|$ denote the number of processors in $G_n(v_T)$. As v_T is mapped uniformly at random to one of the nodes in $G_n(v)$,

$$P(v) \leq \frac{1}{|G_n(v_T)|} \quad .$$

Next, we relate this probability to the bandwidth $b(v_T)$ of v_T. This bandwidth is defined to be the number of the nodes in $G_n(v_T)$. As a consequence,

$$b(v_T) = |G_n(v_T)| \leq \frac{1}{P(v)} \quad .$$

The maximum load of v_T is $C_{\text{opt}}(T(G_n) \cdot b(e_T))$. Consequently, the expected load on node v for simulating v_T is at most

$$C_{\text{opt}}(T(G_n)) \cdot b(v_T) \cdot P(v) \leq C_{\text{opt}}(T(G_n)) \quad .$$

Hence, $E[L_\ell(v)] = O(C_{\text{opt}}(T(G_n)))$, which yields the lemma. ■

This completes the proof of Theorem 2.17. ■

2.3.4 The universal caching strategy

In this section, we present the *universal caching strategy* working on arbitrary networks for which an oblivious routing strategy is given. A routing strategy is called *oblivious* if the path traveled by each packet depends only on the origin and destination of the packet and not on the origins and destinations of the other packets nor on congestion encountered during the routing. Note that each oblivious routing strategy is also an on-line routing strategy. Finally, we apply the universal caching strategy to several classes of networks.

Suppose we are given an arbitrary network G with n nodes and an oblivious routing strategy for G. Each edge is assumed to have bandwidth 1. Thus, the relative and absolute load of an edge are identical. The universal caching strategy for G is a simulation of the access tree strategy for the complete network G_n on G. Note that both G and G_n have n nodes such that for each node in G a counterpart in G_n can be fixed. All messages that are sent between adjacent nodes in G_n are sent along the paths, determined by the oblivious routing strategy, between the associated nodes in G.

The following theorem shows that, for a network G in which a permutation can be routed obliviously with maximum expected load $L_\pi(G)$, the universal caching strategy is $O(L_\pi(G) \cdot \deg(G))$-competitive, w.h.p., and strictly $O(L_\pi(G) \cdot \deg(G) \cdot \log n)$-competitive, w.h.p. To yield the latter result we have to relate the optimal congestion $C_{\text{opt}}(M)$ and $\min\{D, \kappa\}$. Either an object x is migrated at least once or each request message to x is directed to the node where the initial copy of x is placed. As the nodes in G have maximum degree $\deg(G)$, it follows $\deg(G) \cdot C_{\text{opt}}(G) \geq \min\{D, \kappa\}$.

Theorem 2.20 *Consider an application on a network G with n nodes in which a permutation can be routed obliviously with maximum expected load $L_\pi(G)$. For each $1 \leq h \leq \log n$, we obtain a caching strategy with congestion $O(L_\pi(G) \cdot \deg(G) \cdot h \cdot C_{\text{opt}}(G) + \min\{n^{2/h} \cdot D, \kappa\} \cdot n^{1/h} \cdot \log n)$, w.h.p., where $\deg(G)$ denotes the degree of G, $C_{\text{opt}}(G)$ denotes the optimal congestion for the application, and κ denotes the maximum number of write requests directed to the same object.*

Proof. In order to prove the above result we require a lower bound on the congestion of an optimal strategy and an upper bound on the congestion of the universal caching strategy. Define $C_{\text{opt}}(G_n)$ to be the optimal congestion at the nodes for the application when it is executed on the complete network G_n, under the assumption that each node of G is simulated by its counterpart in G_n. Fix an $1 \leq h \leq \log n$. Define $T(G_n)$ to be the decomposition tree of G_n with height h and degree $O(n^{1/h})$, and define $C_{\text{opt}}(T(G_n))$ to be the optimal congestion at the nodes for the application when it is executed on the tree $T(G_n)$, under the assumption that each node of G_n is simulated by its counterpart in $T(G_n)$, which is one of the leaf nodes.

We give a lower bound that relates $C_{\text{opt}}(G)$ to $C_{\text{opt}}(T(G_n))$, and an upper bound that relates the congestion of the universal caching strategy to $C_{\text{opt}}(T(G_n))$. We start with the lower bound.

Lemma 2.21 $C_{\text{opt}}(T(G_n)) \leq \deg(G) \cdot C_{\text{opt}}(G)$.

Proof. Lemma 2.18 shows that $C_{\text{opt}}(T(G_n)) \leq C_{\text{opt}}(G_n)$. Hence, it remains to prove $C_{\text{opt}}(G_n) \leq \deg(G) \cdot C_{\text{opt}}(G)$.

For a given strategy on G with congestion C we have to describe a strategy on G_n with congestion at most $\deg(G) \cdot C$ at the nodes. We simulate the strategy for G on G_n, except for the routing. Instead, for a routing path in G, we use the edge between the respective nodes in G_n.

Each edge incident on a node v_G in G has at most load C, since each edge in G has bandwidth 1. Thus, the node v_{G_n} in G_n being the counterpart of v_G in G has at most relative load $\deg(G) \cdot C$, since each node in G_n has bandwidth 1. ■

The following lemma gives an upper bound on the congestion of the universal caching strategy.

Lemma 2.22 *The maximum expected relative load in G due to access messages is $O(L_\pi(G) \cdot h \cdot C_{\text{opt}}(T(G_n)))$.*

Proof. Lemma 2.19 shows that the maximum expected relative load at the nodes in G_n due to access messages is $O(h \cdot C_{\text{opt}}(T(G_n)))$. Hence, it remains to prove that, for a given strategy with maximum expected relative load L at the nodes in G_n, the simulation of this strategy has at most maximum expected relative load $O(L_\pi(G) \cdot L)$ at the edges of G.

Recall that, in the simulation of the strategy for G_n on G, all messages that are sent between adjacent nodes in G_n are sent along the paths, determined by the oblivious routing strategy, between the associated nodes in G. Obviously, all these routing requests can be partitioned into L partial permutations. Since a permutation can be routed in G with maximum expected (relative) load $L_\pi(G)$, the simulation has at most maximum expected relative load $O(L_\pi(G) \cdot L)$. Note that the routing strategy is oblivious, i.e., the path traveled by each packet depends only on the origin and destination of the packet. ■

Applying Lemma 2.21 and Lemma 2.22 yields that the maximum expected relative load in G due to access messages is $O(L_\pi(G) \cdot \deg(G) \cdot h \cdot C_{\text{opt}}(G))$.

Finally, let $L(e)$ denote the load on an arbitrary edge e of G due to all kind of messages. Analogously to the proof of Lemma 2.9, we can conclude that $E[L(e)] = O(L_\pi(G) \cdot \deg(G) \cdot h \cdot C_{\text{opt}}(G))$. In order to complete the proof of the lemma, we have to show that the maximum load over all edges does not deviate too much from the expected load of an arbitrary edge.

Analogously to the proof of Lemma 2.9, we decompose $L(e)$ into $\deg(T(G_n))$ parts such that $L(E) = \sum_{i=1}^{\deg(T(G_n))} L_i$, and each L_i is a sum of independent random variables

of maximum weight $W = O(\min\{K, \kappa\})$ with $K = \Theta(n^{2/h} \cdot D)$ and κ denoting the maximum number of write requests directed to the same object. Applying a Chernoff bound (see, e.g., [HR90]) to the sums yields that $L_j = O(E(L_j) + W \cdot \log n)$, w.h.p., and therefore,

$$\begin{aligned} L(e) &= \sum_{i=1}^{\deg(T(G_n))} L_i \\ &= O(E(L) + \deg(T(G_n)) \cdot W \cdot \log n) \\ &= O(L_\pi(G) \cdot \deg(G) \cdot h \cdot C_{\text{opt}}(M) + \min\{n^{2/h} \cdot D, \kappa\} \cdot n^{1/h} \cdot \log n) \quad , \end{aligned}$$

w.h.p. This completes the proof of Lemma 2.20. ■

Cayley networks

Meyer auf der Heide and Vöcking have presented in [MV95] an oblivious routing strategy for Cayley networks that achieves maximum expected load $O(p \cdot \text{diam}(G))$ for routing p permutations in a Cayley network G with n nodes, where $\text{diam}(G)$ denotes the diameter of G. Cayley networks are defined as follows. Let Γ be a finite algebraic group with identity 1, and suppose Σ is a set generator of Γ with $1 \notin \Sigma$. Then the Cayley network $G_{\Gamma,\Sigma} = (V, E)$ is defined by $V = \Gamma$ and $E = \{(a, b) \mid a^{-1}b \in \Sigma\}$. Note that many standard networks belong to this class, e.g., tori, cube-connected-cycles, and wrapped butterfly networks. Then, the following corollary can be concluded with Theorem 2.20. This corollary shows that, for a Cayley network G with n nodes, the universal caching strategy is $O(\text{diam}(G) \cdot \deg(G))$-competitive, w.h.p., and strictly $O(\text{diam}(G) \cdot \deg(G) \cdot \log n)$-competitive, w.h.p.

Corollary 2.23 *Consider an application on a Cayley network G with n nodes. For each $1 \leq h \leq \log n$, we obtain a caching strategy with congestion $O(\text{diam}(G) \cdot \deg(G) \cdot h \cdot C_{\text{opt}}(G) + \min\{n^{2/h} \cdot D, \kappa\} \cdot n^{1/h} \cdot \log n)$, w.h.p., where $C_{\text{opt}}(G)$ denotes the optimal congestion for the application, and κ denotes the maximum number of write requests directed to the same object.*

Edge symmetric networks

Further, Meyer auf der Heide and Vöcking have given in [MV99] an oblivious routing strategy for edge symmetric networks achieving maximum expected load $O(p \cdot \text{diam}(G)/\deg(G))$ for routing p permutations in an edge symmetric network G with n nodes. Note that, e.g., all equal-sided tori belong to this class. Then, the following corollary can be concluded with Theorem 2.20. This corollary shows that, for an edge symmetric network G with n nodes, the universal caching strategy is $O(\text{diam}(G))$-competitive, w.h.p., and strictly $O(\text{diam}(G) \cdot \log n)$-competitive, w.h.p.

Corollary 2.24 *Consider an application on an edge symmetric network G with n nodes. For each $1 \leq h \leq \log n$, we obtain a caching strategy with congestion*

$O(\mathrm{diam}(G) \cdot h \cdot C_{\mathrm{opt}}(G) + \min\{n^{2/h} \cdot D, \kappa\} \cdot n^{1/h} \cdot \log n)$, *w.h.p., where* $C_{\mathrm{opt}}(G)$ *denotes the optimal congestion for the application, and* κ *denotes the maximum number of write requests directed to the same object.*

Hypercubic networks

In addition, Meyer auf der Heide and Vöcking have introduced in [MV99] an oblivious routing strategy for de Bruijn networks that achieves maximum expected load $O(p \cdot \log n)$ for routing p permutations in the de Bruijn network with n nodes. It is obvious that a bi-simulation between a de Bruijn network and the shuffle-exchange network of the same size can be done with congestion 2. Then, the following corollary can be concluded with Theorem 2.20. This corollary shows that, for the wrapped butterfly, cube-connected-cycles, hypercube, de Bruijn, and shuffle-exchange network with n nodes, the universal caching strategy is $O(\log n)$-competitive, w.h.p., and strictly $O((\log n)^2)$-competitive, w.h.p.

Corollary 2.25 *Consider an application on the wrapped butterfly, cube-connected-cycles, hypercube, de Bruijn, or shuffle-exchange network G with n nodes. For each* $1 \leq h \leq \log n$, *we obtain a caching strategy with congestion* $O(\log n \cdot h \cdot C_{\mathrm{opt}}(G) + \min\{n^{2/h} \cdot D, \kappa\} \cdot n^{1/h} \cdot \log n)$, *w.h.p., where* $C_{\mathrm{opt}}(G)$ *denotes the optimal congestion for the application, and* κ *denotes the maximum number of write requests directed to the same object.*

Leighton has presented in [Lei92] an oblivious routing strategy for indirect butterfly networks that achieves maximum expected load $O(p)$ for routing p permutations in the indirect butterfly network with n nodes. Hence, the following corollary can be concluded with Theorem 2.20. This corollary shows that, for the indirect butterfly network with n nodes, the universal caching strategy is $O(1)$-competitive, w.h.p., and strictly $O(\log n)$-competitive, w.h.p.

Corollary 2.26 *Consider an application on the indirect butterfly network G with n nodes. For each* $1 \leq h \leq \log n$, *we obtain a caching strategy with congestion* $O(h \cdot C_{\mathrm{opt}}(G) + \min\{n^{2/h} \cdot D, \kappa\} \cdot n^{1/h} \cdot \log n)$, *w.h.p., where* $C_{\mathrm{opt}}(G)$ *denotes the optimal congestion for the application, and* κ *denotes the maximum number of write requests directed to the same object.*

2.3.5 Clustered Networks

A *clustered network* $G = (V, E)$ is a network that consists of several small subnetworks, i.e., *clusters*, that are arranged hierarchically. The *cluster-tree* $T(G)$ describes this hierarchical structure. The internal nodes of $T(G)$ correspond to the clusters of G, and the leaves correspond to the *terminal user nodes*. In the following, we interpret the terminal user nodes as clusters of size 1. Any two clusters that are connected by one

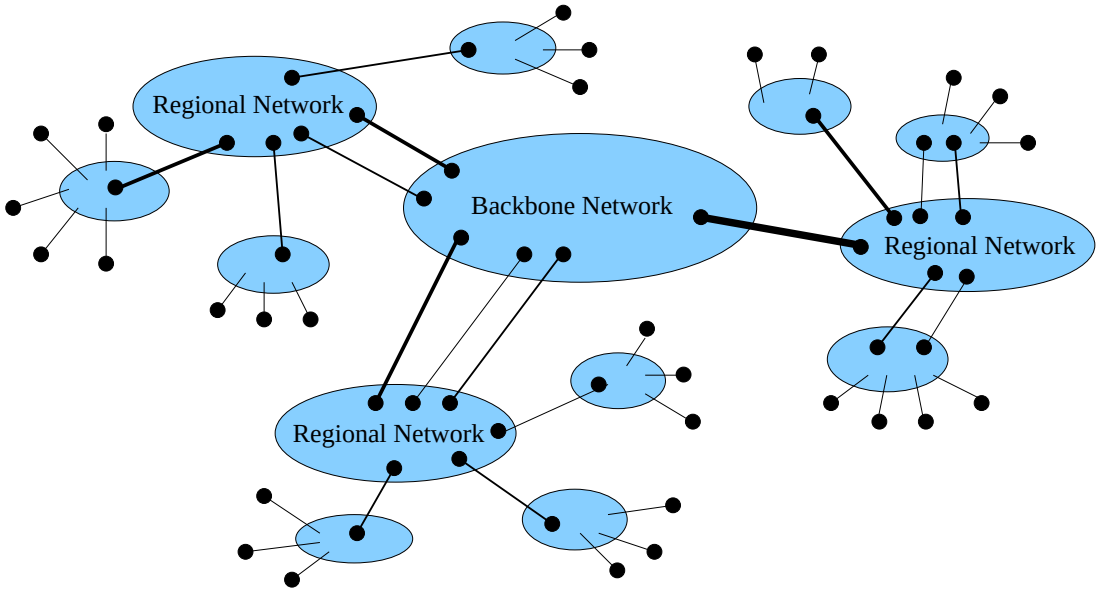

Figure 2.5: The topology of a wide area NOW.

or more edges in G are also connected by an edge in $T(G)$. The bandwidth of an edge in $T(G)$ is the sum of the bandwidths of the corresponding edges in G.

For instance, networks of workstations (NOWs) are usually organized as clustered networks. Figure 2.5 depicts a possible topology for a wide-area NOW. Note that our definition of clustered networks is slightly more general than the depicted network as it allows that user terminal nodes are attached also to the internal clusters.

The nodes inside the clusters are connected in an arbitrary fashion. However, we assume that communication between nodes in the same cluster is less expensive than communication between nodes in different clusters, which is just the basic idea behind any kind of clustering. This property can be formalized as follows. Consider a cluster K. Let V_K denote the set of nodes in the cluster. Define the weight $w(v)$ of a node $v \in V_K$ to be the sum of the bandwidths of the edges incident on v that leave K, and the weight $w(U)$ of $U \subseteq V_K$ by $w(U) = \sum_{v \in U} w(v)$. A *cut* U for $U \subseteq V_K$ is a partition of K into two subgraphs induced by U and $V_K \setminus U$. The *capacity* of a cut U is the sum of the bandwidths of the edges connecting the nodes in U with the nodes in $V_K \setminus U$. We define the *cross flux* by

$$\begin{aligned} \alpha(K) &= \min_{U \subseteq V_K} \frac{\text{capacity}(U) \cdot w(V_K)}{w(U) \cdot w(V_K \setminus U)} \\ &\approx \min_{U \subseteq V_K} \frac{\text{capacity}(U)}{\min\{w(U), w(V_K \setminus U)\}} \quad . \end{aligned}$$

We assume that $\alpha(K) = \Omega(1)$, for each cluster K.

The following example illustrates that the cross flux is a good measure for the bandwidth bottleneck of a cluster. We define the *fully loaded random routing problem*

as follows. Suppose $b(e)/2$ messages arrive along each edge e that connects cluster K with another cluster, where $b(e)$ denotes the bandwidth of e. For simplicity, we assume that $b(e)/2$ is an integer. For each of the arriving messages, we choose a departure edge e at random such that the message leaves the cluster along edge e with probability $b(e)/w(V_k)$. Note that the expected number of messages arriving or leaving the cluster via an edge e is $b(v)$, which corresponds to the maximum number of messages that can pass the edge e in one time step.

Define the *minimum cut ratio* S of the fully loaded random routing problem to be the minimum, over all cuts U, of the capacity of the cut divided by the expected number of messages that have to pass the cut. If S is smaller than 1 than there is at least one cut U such that the expected number of messages cannot pass this cut in one time step. The cross flux $\alpha(K)$ is equal to the minimum cut ratio S of the fully loaded random routing problem. Therefore, if $\alpha(K) < 1$ then there is a bandwidth bottleneck inside the cluster K.

We consider a distributed application or network computation in which the user terminal nodes issue read and write requests. The shared objects can be placed on any node of the network, but only the terminal user nodes are allowed to issue requests to the shared objects. Network computations in which also the internal nodes issue requests to the shared objects can be modeled by attaching a user terminal node to each of the internal nodes. The two nodes are connected by an edge, which virtual bandwidth corresponds to the maximum injection rate for requests issued by the original node.

The access tree strategy for clustered networks works as follows. For each object $x \in X$, define the *access tree* $T_x(G)$ to be a copy of the cluster-tree $T(G)$. Each interior node i of $T_x(G)$ is mapped at random to one of the nodes in the associated cluster K. In particular, i is mapped with probability $w(v)/w(V_K)$ onto node $v \in V_K$. On each access tree we execute the tree strategy.

The caching strategy requires a cluster initialization for the selection of the routing paths inside the clusters. For this path selection, we solve a multicommodity flow problem for each cluster K. The multicommodity flow problem is solved locally for each cluster K, e.g., with the randomized approximation scheme introduced by Leighton et al. in [LMP$^+$95]. Let $n(K)$ denote the number of nodes in the cluster and $m(K)$ the number of edges connecting these nodes. Then the initialization takes time $O(n(K)^3 \cdot m(K) \cdot \log^4 n(K))$, for each cluster K. Alternatively, the multicommodity flow problem can be calculated deterministically and exactly by an algorithm based on linear programming, which also can be done in time polynomial in the size of the clusters. Note that, independent from the number of shared objects, the initialization of a cluster needs to be done only once. A detailed description of the solution to the routing problem is given in the proof for the following theorem.

Theorem 2.27 *Consider an application running on the terminal user nodes of a clustered network G of size n. Suppose the cross flux of each cluster is in $\Omega(1)$. Let δ denote the maximum number of nodes in a cluster that are adjacent to nodes in*

other clusters, and let γ denote the maximum degree in the cluster tree $T(G)$, i.e., γ is the maximum number of tree nodes adjacent to any single tree node. Then the access tree strategy achieves congestion $O(\log\delta \cdot C_{\text{opt}}(G) + \gamma \cdot \kappa \cdot \log n)$, w.h.p., where $C_{\text{opt}}(G)$ denotes the optimal congestion for the application, and κ denotes the maximum number of write requests directed to the same object. If the clusters in G can be represented by planar graphs or by constant genus graphs, then this result improves to $O(C_{\text{opt}}(G) + \gamma \cdot \kappa \cdot \log n)$, w.h.p.

Proof. Consider a cluster K. Let V_K denote the set of nodes in the cluster. Let $\alpha(K)$ denote the cross flux of K. Let $\delta(K)$ denote the number of nodes in K that are adjacent to nodes in other clusters. Let $\gamma(K)$ denote the degree of K in the cluster-tree. We define the congestion of K to be the maximum relative load of all edges adjacent to nodes in K including edges connecting nodes in K with nodes in other clusters. Let $C_{\text{opt}}(K)$ denote the optimal congestion of K. We show, for cluster K, that the access tree strategy achieves congestion $O(\log\delta(K) \cdot C_{\text{opt}}(K)/\max\{1,\alpha(K)\} + \gamma(K) \cdot \kappa \cdot \log n)$, w.h.p. If K is planar or of constant genus, then this result improves to $O(C_{\text{opt}}(K)/\max\{1,\alpha(K)\} + \gamma(K) \cdot \kappa \cdot \log n)$, w.h.p. This yields the above theorem. Furthermore, it shows that the influence of a "bad" cross flux is only small and local to the respective cluster.

In the following, we describe how the routing paths that simulate the edges of the access trees are selected. According to this description, any message that is sent between two nodes simulating adjacent access tree nodes chooses a routing path at random. We assume that the path selection for each message is independent from the selection of other messages.

First, we fix the course of the routing paths along the edges that connect different clusters. Suppose K' is a cluster neighboring to K. Let u and u' be the cluster-tree nodes that represent the access tree nodes corresponding to K and K', respectively. Let e_T be the edge that connects the two clusters in the cluster-tree $T(G)$, and let $e_1,\ldots,e_k$ denote the corresponding edges that connect K and K' in G. We send a message that traverses e_T in $T(G)$ along edge e_i with probability $p(e_i) = b(e_i)/b(e_T)$, where $b(e_i)$ is the bandwidth of edge e_i in G, and $b(e_T) = \sum_{i=1}^{k} b(e_i)$ is the bandwidth of the edge e_T in $T(G)$. The following lemma gives an upper bound on the expected relative load on the edges between different clusters.

Lemma 2.28 *Let e be an edge of G that connects a node in cluster K with a node in one of the neighboring clusters. Then the expected load on e is $O(C_{\text{opt}}(K))$.*

Proof. Let e_T denote the edge in $T(G)$ that connects the nodes representing the two adjacent clusters. Our strategy executes the strictly 3-competitive tree strategy on the access trees. Thus, the relative load on e_T is $O(C_{\text{opt}}(K))$, and hence the absolute load is $O(C_{\text{opt}}(K) \cdot b(e_T))$. As a consequence, the expected absolute load for edge e is $O(C_{\text{opt}}(K) \cdot b(e_T) \cdot p(e)) = O(C_{\text{opt}}(K) \cdot b(e))$. Therefore, the expected relative load on e is at most $O(C_{\text{opt}}(K))$. ∎

Next we describe the path selection strategy for the routing inside the cluster K. Consider the following multicommodity flow problem. Let $V_K^* = \{v_1,\ldots,v_{\delta(K)}\} \subseteq V_K$

denote the set of nodes that are adjacent to nodes outside the cluster. We define $\delta(K)^2$ commodities $\ell_{i,j}$ with $1 \leq i, j \leq \delta(K)$. The source of commodity $\ell_{i,j}$ is v_i, its sink is v_j, and its demand is $w(v_i) \cdot w(v_j)/w(V_K^*)$. We solve the commodity flow problem on K while respecting the capacities, i.e., the bandwidths of the edges in K.

We are going to use the solution to the multicommodity flow problem to help guide us in selecting the routing paths inside the clusters. For a solution of the multicommodity problem, define the *throughput fraction* q as the minimum, over all commodities, of the fraction of the commodity's demand that is met, that is, for each commodity $\ell_{i,j}$, there is a flow of size $q \cdot w(v_i) \cdot w(v_j)/w(V_K^*)$ from v_i to v_j. An optimal solution maximizes the value of q. The following lemma relates the result of an optimal solution to the cross flux $\alpha(K)$.

Lemma 2.29 *In the general case, there is a solution to the multicommodity with* $q = \Omega(\alpha(K)/\log\delta(K))$. *If K is a planar graph or a constant genus graph then there is a solution with* $q = \Omega(\alpha(K))$.

Proof. First we consider the general case. Define the *minimum cut ratio* of a multicommodity flow problem to be the minimum, over all cuts, of the capacity of the cut divided by the flow demand across the cut. In [AR98] it is shown that any multicommodity flow problem can be satisfied up to a factor $q = \Omega(S/\log k)$ with S denoting the minimum cut ratio and k denoting the number of commodities. The minimum cut ratio of our multicommodity flow problem is $\alpha(K)/2$. Note that the "load" induced by the multicommodity flow problem on an edge is twice the load of the fully loaded random routing problem, which has minimum cut ratio $\alpha(K)$. As the number of commodities in our problem is $\delta(K)^2$, the maximum flow can be satisfied up to a factor $q = \Omega(\alpha(K)/\log(\delta(K)))$, which yields the result for general networks.

A better bound is known for the optimal throughput fraction of uniform multicommodity flow problems on planar graphs or graphs of constant genus. In a *uniform multicommodity flow problem* there is a unit-demand commodity between every pair of nodes. In this case, the optimal throughput fraction is shown to be in $\Omega(S')$, where S' denotes the cut ratio of the uniform problem (see [KPR93]). We transform our multicommodity flow problem into a uniform problem such that the cut ratio S' of the uniform problem is in $\Omega(S)$ and the optimal throughput fraction q' for the uniform problem is in $O(q)$, where S and q denote the cut ratio and the optimal throughput fraction of the original problem, respectively. This yields the desired result as $q = \Omega(q') = \Omega(S') = \Omega(S)$.

Interestingly, the bound on the optimal throughput fraction for uniform multicommodity problems is independent from the "size" of the respective problem. Therefore, we can arbitrarily blow up our multicommodity flow problem in the transformation without decreasing the optimal throughput fraction. Note that we do not aim to solve the resulting multicommodity flow problem but only aim to show the existence of a "good" throughput fraction.

A first transformation increases all weights such that each node, including the nodes not in V_K^*, have an integral weight not smaller than 1. This is done as follows.

We multiply all weights and capacities with a suitable large factor $F(e_T)$ such that each of the weights of the nodes in V_K^* and each of the capacities of the edges are not smaller than n^2, where n denotes the number of nodes in the cluster. Afterwards, the weights of all nodes are rounded up to the next integer, and the weights of all nodes with weight 0 in the original problem, i.e., the nodes not in V_K^*, are defined to have weight 1 in the new problem. Let $w'(v)$ denote the new weight of a node. We define n^2 commodities $\ell'(u,v)$ with $u,v \in V_T$. The source of commodity $\ell'(u,v)$ is u, its sink is v, and its demand is $w'(u) \cdot w'(v)/W$ with $W = w'(V_K) = \sum_{v \in V_K} w'(v)$.

Let S' denote the minimum cut ratio and q' the optimal throughput fraction of the multicommodity flow problem after the transformation. As all weights and capacities are multiplied with the same factor $F(e_T)$ and the sum of the added demands is smaller than n, which is a fraction of $1/n$ of the smallest capacity, $S' = \Theta(S)$ and $q' = \Theta(q)$.

A second transformation yields a uniform problem in which each commodity has demand $1/W$. For each node v with $w'(v) > 1$, we add $w'(v) - 1$ nodes, each of which is connected by an edge of unbounded capacity with v. Next, the weights of all nodes are defined to be 1. The resulting graph is still planar or of constant genus and has W nodes. We define W^2 commodities, one for each tuple of nodes. The demand of each commodity is defined to be $1/W$.

The multicommodity problem derived by the second transformation corresponds to the previous problem in the following way: Consider two nodes u and v from the original cluster graph. u and v are represented by a set of nodes $V(u)$ and $V(v)$ in the new graph. The demand of the commodities that should be routed from $V(u)$ to $V(v)$ is

$$\sum_{u' \in V(u)} \sum_{v' \in V(v)} \frac{1}{W} = \frac{|V(u)| \cdot |V(v)|}{W} = \frac{w'(u) \cdot w'(v)}{W} \quad ,$$

which corresponds to the demand that should be routed between u and v in the previous problem.

Therefore, the second transformation does not change the minimum cut ratio nor the optimal throughput fraction. Hence, we have shown that the original problem can be transformed into a uniform problem with minimum cut ratio $S' = \Theta(S)$ and optimal throughput fraction $q' = \Theta(q)$, which completes the proof of Lemma 2.29. ■

In the cluster initialization phase, we solve the maximum flow problem for the cluster K. For each node u, for each edge e incident on u, and each commodity $\ell_{i,j}$, let $f(u,e,i,j)$ denote the size of the flow of commodity $\ell_{i,j}$ that leaves node u across edge e according to the solution of the multicommodity flow problem. As a result of the initialization, each node u holds a table including its respective $f(u,e,i,j)$ values.

We use the solution of the multicommodity flow problem to choose the routing paths in the cluster. Note that none of the request messages has its source and its destination in the same cluster because messages are routed only between nodes that simulate neighboring access tree nodes, and each cluster hosts at most one node of each access tree. The hosts of the access tree nodes are chosen randomly from $V_K^* = \{v_1, \ldots, v_{\delta(K)}\}$, which is the set of nodes adjacent to nodes in other clusters. Hence, any

message coming from outside of K arrives at a node from V_K^* and has to be forwarded to a node also from V_K^*. Also a message that leaves the cluster starts at a node from V_K^* and has to be routed through the cluster to a node from V_K^*, i.e., to that node which is incident to the edge along which the message aims to leave the cluster. Hence, we have to describe only how to choose the routing paths between pairs of nodes from V_K^*.

Consider a message that traverses or starts at a node $u \in V_K$. Let $v_i \in V_K^*$ and $v_j \in V_K^* \setminus \{u\}$ denote the local source and the local destination of the message, respectively. Let $E(u)$ denote the set of edges that are incident to u and do not leave the cluster. Then u chooses an edge from $E(u)$ at random according to the following distribution: e is selected with probability

$$\frac{f(u,e,i,j)}{\sum_{e' \in E(u)} f(u,e',i,j)} \ .$$

The message is forwarded along the randomly selected edge. Intuitively, the message follows the suggestions of the multicommodity flow.

The following lemma relates the expected relative load of the edges inside cluster K to the optimal congestion $C_{\text{opt}}(K)$, which is defined to be the maximum relative load over all edges incident on the nodes in V_K, and hence includes also the relative load on edges that leave the cluster. Note that the bound on the expected relative load given in the lemma becomes $o(C_{\text{opt}}(K))$ if q is in $\omega(1)$, which means that the bandwidth inside the cluster is much higher than the bandwidth of the external links. In this case the congestion is dominated by the load on the edges connecting different clusters.

Lemma 2.30 *For each edge connecting two nodes of the cluster, the expected relative load on e is at most $O(C_{\text{opt}}(K)/q)$, where q denotes the throughput fraction computed by the multicommodity flow program.*

Proof. The only difference between arriving and leaving messages is the direction in which the messages travel along the randomly selected routing paths. As this is irrelevant for the expected load on any edge, we only consider arriving messages.

For a data object x, let $v(x) \in V_K^*$ denote the node that simulates the access tree node of x. Any message for x that arrives at cluster K is directed to node $v(x)$. Recall that $v(x)$ is chosen randomly from $V_K^* = \{v_1, \ldots, v_{\delta(K)}\}$. The probability that $v(x) = v_j$, for $1 \leq j \leq \delta(K)$, is $w(v_j)/w(V_K) = w(v_j)/w(V_K^*)$.

Suppose each node v_i injects an expected number of $q \cdot w(v_i)$ messages into the cluster and each message is directed to the host $v(x)$ of some data object x. Then the expected number of messages sent from node v_i to node v_j is $q \cdot w(v_i) \cdot w(v_j)/w(V_K^*)$, which corresponds to the amount of flow sent from v_i to v_j in the solution for the multicommodity flow problem. The random path selection inside the cluster is guided by the solution for the multicommodity flow in such a way that the expected number of messages traversing an edge is equivalent to the amount of flow passing the edge. Hence, in our example, the expected load on an edge e is not larger than the capacity of the edge, which is defined to be $b(e)$.

Now, consider the messages arriving during the execution of the application. By Lemma 2.28 the expected number of messages that arrive via an external edge e' connecting a node v_i from the cluster K with a node from a neighboring cluster K' is $O(C_{\text{opt}}(K) \cdot b(e'))$. As $w(v_i)$ is equivalent to the sum of the bandwidths of the external edges e' incident on v_i, the expected number of messages injected via node v_i into the cluster is $O(C_{\text{opt}}(K) \cdot w(v_i))$. As a consequence, the expected relative load on each edge is at most $O(C_{\text{opt}}(K)/q)$. ■

Combining the results of the Lemmata 2.28, 2.29, and 2.30 yields that the expected load for each edge in K is $O(\log \delta(K) \cdot C_{\text{opt}}(K)/\min\{1, \alpha(K)\})$ in the general case, and $O(C_{\text{opt}}(K)/\min\{1, \alpha(K)\})$, if K is planar or of constant genus. In order to complete the proof of Theorem 2.27, we have to show that the maximum relative load over all edges in K does not deviate too much from the expected load.

Consider edge e of the cluster K. Let $L(e)$ denote the load on e. Then we have to show that $L(e) = O(E[L(e)] + \gamma(K) \cdot \kappa \cdot \log n)$, w.h.p. Let $L_x(e)$ denote the load on e due to the accesses to data object x. Our tree strategy ensures that the load on each edge of the access tree due to request messages for x is $O(\kappa)$. Further, each edge e is involved in the simulation of at most $\gamma(K)$ different access tree edges. Therefore, $L_x(e) = O(\gamma(K) \cdot \kappa)$. This means that $L_x(e)$ is the sum of $O(\gamma(K) \cdot \kappa)$ not necessarily independent 0-1-random variables, each of which representing the load due to one request message. We add some dummy variables so that $L_x(e)$ is the sum of exactly $\tau = O(\gamma(K) \cdot \kappa)$ random variables $A_1(x), \ldots, A_\tau(x)$, for every $x \in X$. Then

$$L(e) = \sum_{x \in X} \sum_{i=1}^{\tau} A_i(x) = \sum_{i=1}^{\tau} \sum_{x \in X} A_i(x) \quad .$$

The variables $A_i(x)$ in the sum $S_i = \sum_{x \in X} A_i(x)$ are independent, for $1 \leq i \leq \tau$. Applying a Chernoff bound (see, e.g., [HR90]) to this sum yields that its value deviates by at most an additive term of $O(\log n)$ from $O(E[S_i])$, w.h.p., for $1 \leq i \leq \tau$. Since $L(e) = \sum_{i=1}^{\tau} S_i$, it follows $L(e) = O(E[L(e)] + \tau \cdot \log n) = O(E[L(e)] + \gamma(K) \cdot \kappa \cdot \log n)$, w.h.p., which completes the proof of Theorem 2.27. ■

2.4 Extending the results to data-race free applications

An important class of applications that allows parallel and overlapping requests is the class of *data-race free applications*, which is defined as follows. We assume that an adversary specifies a parallel application running on the nodes of the network, i.e., the adversary initiates read and write requests at the nodes of the network. A write request for an object is not allowed to overlap with other requests to the same object, and there is some order among the requests to the same object such that, for each read and write request, there is a unique least recent write. Note that this still allows arbitrary concurrent requests to different objects and concurrent read requests to the same object. An execution using a caching strategy is called *consistent* if it ensures

that a read request directed to an object always returns the value of the most recent write request to the same object.

We have to describe how parallel requests are handled by the presented strategy such that the execution is consistent. On trees this works as follows. Since we consider only data-race free applications, write requests do not overlap with other requests. Overlapping read requests are handled in the following way. Consider a request message m arriving on a node u that does not hold a copy of x. Let e denote the next edge on the path to the nearest copy. Suppose another request message m' directed to x has been sent already along e but a data message has not yet been sent back. Then the request message m is blocked on node u until the data message corresponding to m' passes e. When this message arrives, either a new copy is created on node u, and u serves the request message m, or m continues its path to the connected component of copies. As the other networks execute the tree strategy on access trees that are embedded in the network, they can follow the same approach.

In data-race free applications, the following problem can occur during the remapping of copies: remapping or notification messages overlap with access messages or other remapping or notification messages. For example, an access message arrives at an old host of an access tree node but the tree node has been remapped while the access message was in transit. We solve this problem as follows. All messages are acknowledged such that we can ensure that at most two messages are pending between two access tree nodes. If a message arrives at an abandoned host the sender is informed about this and resends the message to the new host. We define that neither the acknowledgment messages nor the overlapping messages influence the counters. Obviously, these messages have no influence on our asymptotic bounds as each of them corresponds to another message that we take into account.

We can conclude that all results given in this chapter hold also for data-race free applications, which indicates that the introduced strategies are well suited for practical usage.

Chapter 3

Caching with Memory Capacity Constraints

In this chapter, we investigate caching strategies for networks with memory capacity constraints in a non-uniform cost model. The remainder of the chapter is organized as follows. In Section 3.1, the general framework for the development of caching strategies with memory capacity constraints is introduced. This framework is based on the access tree strategy. In Section 3.2, a caching strategy for trees with memory capacity constraints is presented. This strategy is used as a subroutine in the general framework. In Section 3.3, the general framework is applied to several classes of networks, including meshes, fat-trees, complete networks, Cayley networks, edge symmetric networks, and hypercubic networks. Further, we present the universal caching strategy working on arbitrary networks for which an oblivious routing strategy is given.

3.1 The general framework

In chapter 2, we have presented the access tree strategy that is a caching strategy neglecting memory capacity constraints. The *general framework* is an extension of the access tree strategy to memory capacity constraints.

Basically, the access tree strategy simulates a simple 3-competitive caching strategy for a tree $T(G)$ on the original network G. This tree $T(G)$ is a tree whose leaf nodes correspond to the nodes of the network G. The analysis of the access tree strategy uses a bi-simulation between the network G and the tree $T(G)$. At first, it is shown that the tree $T(G)$ can simulate the network G. In this simulation, each leaf node of $T(G)$ issues the same read and write requests as its counterpart in G. It is shown that any application that produces congestion C when it is executed on G can be executed on $T(G)$ with congestion C, too. At second, it is shown that the network G can simulate the tree $T(G)$ in such a way that the congestion in G is only c_3 times larger than the congestion in the tree $T(G)$.

From the results on the congestion of these two simulations we can obtain the competitive ratio of the access tree strategy. The simulation of G by $T(G)$ looses a factor of at most $c_1 = 1$; the on-line strategy looses a factor of $c_2 = 3$ against the optimal off-line strategy on $T(G)$; finally, the simulation of $T(G)$ by G looses a factor of at most c_3. Altogether, we achieve a competitive ratio of $c_1 \cdot c_2 \cdot c_3 = O(c_3)$.

The problem with the adaption of this approach to the case of memory capacity constraints is the lack of an efficient caching strategy for trees with memory capacity constraints. It is also not clear how the access trees should look like. One possibility is to allow that only the leaf nodes, which correspond to the nodes in G, have memory modules. Another possibility is that all nodes have memory modules which then have to be simulated by the nodes of the network G. For neither of these alternatives, however, we know an efficient caching strategy, and we will see that we do not need such a strategy.

Define the *bandwidth tree* $T_b(G)$ to be a tree of maximum height h, which leaf nodes correspond to the nodes in G. Each of the leaf nodes in $T_b(G)$ has a memory module of the same size as the respective node in G. The inner nodes do not have memory modules. Define the *memory tree* $T_m(G)$ to be a tree that is isomorph to $T_b(G)$. Each of the leaf nodes in $T_m(G)$ has a memory module that is k times larger than the one of its counterpart in $T_b(G)$. Each non-leaf node in $T_m(G)$, except for the root, has a memory module, which capacity is equal to the sum of the capacities of its children. The memory capacity of the root is equal to the sum of the capacities of all nodes in G. Thus, it is guaranteed that the root node can hold all data objects simultaneously. The exact topology of these trees depends on the network G and will be defined later.

Suppose the bandwidth tree $T_b(G)$ can simulate the optimal off-line strategy running on G while increasing the congestion only by a factor of c_1; suppose the on-line strategy on $T_m(G)$ produces only a congestion that is a factor of c_2 larger than the congestion of the optimal off-line strategy on $T_b(G)$; and suppose the on-line strategy on $T_m(G)$ can be simulated on G so that the congestion in G is only a factor of c_3 larger than the congestion in $T_m(G)$ while each node in G requires a memory capacity that is h times the memory capacity of the corresponding leaf node in $T_m(G)$. Then the resulting strategy is $(k \cdot h + 1, c_1 \cdot c_2 \cdot c_3)$-competitive.

The major challenge in the construction of the general framework is the analysis of the factor between the on-line strategy running on $T_m(G)$ and the optimal off-line strategy running on $T_b(G)$. We use a strategy for single-computer caching as a subroutine in the on-line strategy on $T_m(G)$. We will show that the factor between the on-line and the optimal off-line strategy depends on the competitive ratio of the single-computer caching strategy.

Such a result as itself does not seem to be very interesting since it considers the congestion on two different networks. However, we will illuminate its significance in several examples that describe how to apply the general framework to several classes of networks.

3.2 The caching strategy for the memory tree

In this section, we present a caching strategy for the memory tree T_m. This strategy is used as a subroutine in the general framework.

The memory tree T_m is a virtual network whose topology is an arbitrary rooted tree. Each node in T_m has a memory module. The capacity of the modules at the leaf nodes is arbitrary. The capacity of a module at an inner node of the tree is defined to be the sum of the capacities of its children. The edges are allowed to have arbitrary bandwidths. Initially, each data object is stored in one of the leaf nodes.

First, we present a caching strategy for the memory tree in a uniform cost model, i.e. $D = 1$. Then, we generalize this strategy to a non-uniform cost model.

3.2.1 Uniform model

In the uniform cost model, i.e. $D = 1$, we assume that each object fits into one routing packet such that each migration of a copy along an edge increases the load of this edge by 1. Recall that each read or write request increases the load of each edge involved in this request by 1, and that each information message increases the load of each traversed edge by 1.

The caching strategy

An adversary specifies read and write requests that occur solely at the leaf nodes. These requests have to be served by the caching strategy. This strategy may create copies on all nodes and may delete these copies afterwards in order to reduce the congestion. Essentially, many copies mean cheap read requests but expensive write requests. Even in case of only read requests, however, the considered caching problem is not trivial as the capacities of the memory modules usually do not allow to give a copy of every object to every node.

The caching strategy for the memory tree consists of the following sub-strategies.

- *Basic strategy*: This strategy manages the shared objects as if the memory modules in the tree would have unlimited capacities. It decides on which nodes to place copies so that the costs for read and write requests are balanced. It simply neglects the memory capacity constraints.

- *Local strategy*: This strategy corresponds to a caching strategy for single-computer systems. It runs on every node of the tree except for the root, which is sufficiently large to hold copies of all objects simultaneously, by definition. The local strategy decides which object in a memory module is ejected when the basic strategy suggests to insert a new object.

- *Rescue-and-help strategy*: This strategy preserves at least one copy of a data object at any time. Besides it helps the basic strategy when it wants to access a copy that has been ejected by the local strategy.

The coordination between these three strategies is not an easy task. For this purpose, our caching strategy uses *dummy copies*, that is, in addition to the copies created by the basic strategy, the algorithm creates copies that are just placeholders so that they cannot be accessed in case of a read request and need not to be updated in case of a write request. Before we explain this mechanism in detail, we turn to the basic and the local strategy.

Nearly any known strategy for trees without memory capacity constraints can be used as basic strategy. The only restriction is that the strategy has to ensure that, for any object x, the nodes holding a copy of x build a connected component in the memory tree T_m. The overall competitive ratio of our algorithm depends on the competitive ratio of the basic strategy used.

In order to keep the description and the analysis of the overall strategy as simple as possible, we refer to a concrete example of a basic strategy in the following. We use the tree strategy in the uniform model introduced in Section 2.2.1: The nodes holding a copy of an object x build a connected component. Suppose node u issues a read or write request to object x. Let v denote the node of the connected component that is closest to u. Reads are served by establishing the unique shortest path from u to v. Writes are served by establishing the minimal multicast tree from u to all nodes in the connected component. The connected component is changed in the following way. In case of a write all copies in the connected component, except the one on node u, are deleted. In both the read and the write case, we create new copies on the path from u to v. In Theorem 2.1 it is shown that this simple caching strategy does not only minimize the congestion, it minimizes the load on any individual edge in the tree up to a factor of 3. Note that this result is optimal because of the lower bound shown by Black and Sleator [BS89].

When the basic strategy wants to store a copy of an object at a node, it calls the local strategy that decides which copy of another object is ejected eventually. For the local strategy, we can choose any caching strategy for single-computers, e.g., LRU or RANDOM (see, e.g., [FKL$^+$91, ST85]). The competitiveness of the overall strategy depends on the competitiveness of the caching strategy used. In the following we assume that the local strategy is (k,ℓ)-competitive. For example, LRU is known to be $(2,2)$-competitive [ST85].

The description so far leaves us with the following two problems.

- How to preserve at least one copy for each object?
- How to realize accesses to *virtual copies*, i.e., copies that have been ejected by the local strategy?

The rescue-and-help strategy is an extension to the basic strategy that solves these problems by using dummy copies. The dummy copies of an object x are used as placeholders in order to ensure that always one copy of x can be preserved without having to eject any other copy or dummy copy of another object for that purpose. Whenever the basic strategy invalidates copies in case of a write, we do not delete

them but change these copies to dummy copies instead. We assume that the local strategy does not take notice of this change. It treats these copies as usual copies, and may eject them later. As described before, each data object x has only one initial copy, and this copy is stored at a leaf node $v(x)$. We add dummy copies of x to all inner nodes on the path leading from $v(x)$ to the root. Note that the memory capacities of all inner nodes are large enough to hold all these initial dummy copies. Adding these initial dummy copies does not induce any cost since the inner nodes know the initial placement of the copies.

The rescue-and-help mechanism works as follows. At any time during the execution, let $V(x)$ denote the set of nodes that are included in the connected component of nodes that should hold a copy of x according to the basic strategy. Sometimes, some of the nodes in $V(x)$ may not hold a copy of x because it was ejected by the local strategy. Let $\text{top}(x)$ denote the unique node in $V(x)$ that is closest to the root. The path from $\text{top}(x)$ to the root is called the *rescue path* of x. Let $c(x)$ denote the (virtual) copy of x which the basic strategy assumes to be stored on $\text{top}(x)$. The rescue-and-help strategy always keeps one copy $c^*(x)$ of x. Initially, $c^*(x) = c(x)$ is stored on $\text{top}(x)$. Whenever $c^*(x)$ is ejected because of the insertion of another object, $c^*(x)$ is migrated upwards along the rescue path until it reaches a node that holds a dummy copy of x. Here it can be stored without having to eject any other object. At any time during the execution, the following invariant is maintained: $c^*(x)$ is stored on a node $\text{top}^*(x)$ on the rescue path.

The existence of $c^*(x)$ allows us to define an *assistance node* $a(x,v)$ for every node $v \in V(x)$. The assistance node $a(x,v)$ is that node on the path from v to the root that is closest to v and holds a copy of x. Whenever the basic strategy wants to access a virtual copy of x on node v the rescue-and-help strategy uses the copy of the assistance node $a(x,v)$ instead. That is, any path or multicast tree established by the basic strategy between a node u that issued a read or write request and the respective node(s) in $V(x)$ is redirected such that it connects u with the corresponding assistance node(s) instead of the node(s) in $V(x)$.

The migration of copies is handled in the following way. Let v denote the node from $V(x)$ that is closest to the node u. When the basic strategy establishes a path from v to u and migrates copies along that path, the rescue-and-help strategy simply establishes a path from $a(x,v)$ to u, and migrates copies from $a(x,v)$ to u. We distinguish three cases.

1. Suppose $a(x,v) \neq \text{top}^*(x)$. Figure 3.1 illustrates this situation. Then, the rescue-and-help strategy creates a copy on each node on the path from $a(x,v)$ to u. Note that in this case all migrations follow a path that leads *downwards*, i.e., in direction to the leaves.

2. Suppose $a(x,v) = \text{top}^*(x)$, and the path from $\text{top}^*(x)$ to u is going downwards only. Figure 3.1 illustrates this situation. In this case, the path includes $\text{top}(x)$. The rescue-and-help strategy migrates $c^*(x)$ to $\text{top}(x)$, and creates a dummy copy on each node on the migration path from $\text{top}^*(x)$ to $\text{top}(x)$, excluding

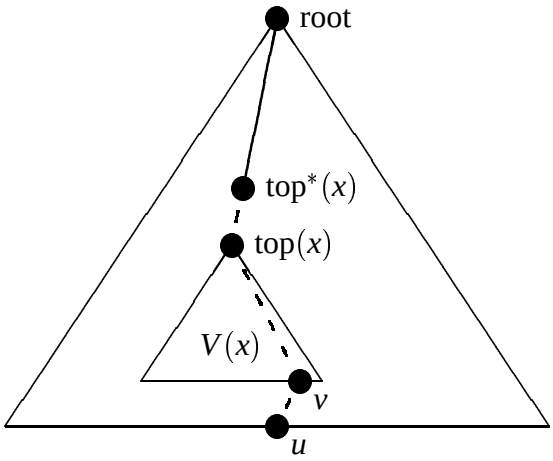

Figure 3.1: Case 1 and 2: $a(x,v)$ lies on the path from v to $\text{top}^*(x)$.

$\text{top}(x)$. Then a copy on each node on the path from $\text{top}(x)$ to u is created. The migration of $c^*(x)$ to $\text{top}(x)$ effectively re-unions $c^*(x)$ and $c(x)$.

3. Suppose $a(x,v) = \text{top}^*(x)$, and the path from $\text{top}^*(x)$ to u is going first upwards to a node w and then downwards to u. Figure 3.2 illustrates this situation. In this case, the rescue-and-help strategy first migrates $c^*(x)$ to w, thereby replacing all dummy copies on the path from $\text{top}^*(x)$ to w by real copies, excluding w. Starting from node w, which now holds $c^*(x)$, a copy on each node on the path from w to u is created, which re-unions $c^*(x)$ and $c(x)$ as w is the new $\text{top}(x)$ node after serving the request.

We assume that almost all accesses to copies and all insertions of copies are done by using the local strategy on the respective node. There is only one exception: In case 3 the copy on the node $a(x,v) = \text{top}^*(x)$ is accessed directly, that is, the access does not influence the local strategy. When using a *memory-less* local strategy, i.e., a strategy that ejects copies independent of previous accesses, like RANDOM, we do not have to care about this topic. In this way, the local strategy on each node v is influenced only by requests issued in the subtree rooted at v.

It remains to describe how our caching strategy on the memory tree can be executed in the distributed setting in which all decisions are based solely on local information. The tree strategy in the uniform model introduced in Section 2.2.1 is used as basic strategy. This tree strategy can be executed in a distributed fashion. Then, it is obvious that the rescue-and-help strategy can also be executed in the distributed setting. This completes the description of the caching strategy on the memory tree T_m.

Let $e = (u,v)$ denote an arbitrary edge in the tree so that v is closer to the root than u. We can observe, that the load on edge e corresponds to the load induced by the basic strategy plus some load $L_1(e)$ induced by serving requests that are redirected to assistant nodes plus some load $L_2(e)$ induced by upward migrations of c^*-copies. Note that

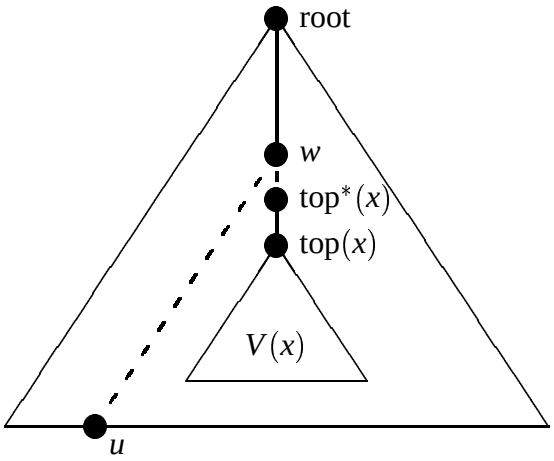

Figure 3.2: Case 3: $a(x,v) = \text{top}^*(x)$.

$L_1(e)$ is increased by 2 due to serving a request that is redirected to an assistant node, since first the request is served and thereafter the residence set is modified. Whenever $L_1(e)$ or $L_2(e)$ are increased due to an object x, then some copy or dummy copy of x has been ejected from node u recently, i.e., after the last time when these values have been increased due to object x. Let eject(e) denote the number of ejections from node u. Then we can conclude the following observation.

Observation 3.1 *For each edge e of T_m, $L_1(e) \leq 2 \cdot \text{eject}(e)$ and $L_2(e) \leq \text{eject}(e)$.*

The analysis of the caching strategy

We compare the load of the caching strategy on the memory tree T_m to the optimal load on the *corresponding bandwidth tree* T_b. This tree T_b has the same topology as T_m, the same bandwidths on all edges, but different memory capacities. The inner nodes on T_b do not have any memory modules, i.e., memory capacity null, and the memory modules at the leaves are smaller than the corresponding memory modules of T_m. These capacities are chosen in dependence on the choice for the local strategy on T_m: Suppose the single-computer caching strategy used as local strategy is (k,ℓ)-competitive. Then we define that the capacity of a leaf node in T_m is k times the capacity of the corresponding leaf node in T_b. We assume that the capacities of T_b are large enough so that all objects can be stored in the network. This assumption allows us to decrease the capacity of the root of the memory tree T_m by a factor of k as it is sufficient if the root can store all data objects simultaneously. Hence, the ratio between the total memory capacity of T_m and the total memory capacity of T_b is $k \cdot h + 1$, where h denotes the height of T_m and T_b.

The following lemma compares the load induced by our caching strategy running on T_m with the load of an optimal off-line strategy running on T_b, both of them applied to the same, arbitrary sequence of requests issued by the leaf nodes of the cor-

responding tree. The single-computer caching strategy used as local strategy may be randomized. In this case, the competitive ratio ℓ gives only an upper bound on the expected load of the single-computer caching strategy, and the following lemma gives only an upper bound on the expected load of the strategy on T_m. The lemma shows that our caching strategy minimizes the load on all edges simultaneously. Hence, the congestion produced by the on-line strategy executed on T_m is at most $3+3\ell$ times the congestion produced by an optimal off-line strategy executed on T_b.

Lemma 3.2 *Let e_b and e_m denote a pair of corresponding edges in T_b and T_m, respectively. Then the (expected) load on e_m is at most $3+3\ell$ times the optimal load on e_b, where ℓ denotes the competitive ratio of the local strategy.*

Proof. Unfortunately, the behavior of an optimal off-line strategy on the bandwidth tree T_b is difficult to understand. In order to get a more obvious behavior, we allow the optimal strategy to place copies on internal nodes, too. Let T_b' denote a tree that is isomorph to T_b, whose leaf nodes have the same memory capacity as the corresponding leaf nodes in T_b, and whose inner nodes have a memory capacity that is equal to the sum of the capacities of their respective children. We restrict, however, the placement of copies on T_b' in the following way: For a node v, let $T_b'(v)$ denote the subtree rooted at node v. Anytime, for each node v, the set of objects that are stored in a memory module in the subtree $T_b'(v)$, must be a subset of the objects that have a copy or dummy copy in the memory module of v. Recall that dummy copies are placeholders that cannot be used to serve a read request and need not to be updated in case of a write request. We assume that these copies can be created from scratch, that is, they do not require any kind of migration. The following lemma shows that, despite of this restrictions, T_b' is more powerful than T_b, so that any lower bound on the load on the edges of T_b' holds also for T_b.

Lemma 3.3 *The execution of any off-line caching strategy serving an arbitrary sequence of requests on T_b, can be simulated on T_b' without increasing the load on any edge.*

Proof. We simulate the strategy for T_b on T_b'. The management of the memory modules of the leaf nodes and the routing between these nodes, i.e., the selection of the paths and multicast trees, is done in T_b' exactly as in T_b. At any time during the simulation, an inner node v of T_b' holds a dummy copy of all those objects that are stored in the memory modules in $T_b'(v)$. In this way the restrictions on T_b' are satisfied. Since dummy copies can be created from scratch, this simulation does not require to increase the load on any edge. ∎

In contrast to the bandwidth tree T_b, the behavior of an optimal off-line strategy on T_b' is relatively clear. W.l.o.g., any optimal off-line strategy fulfills the following properties for each node v in T_b'.

- If a leaf node in the subtree $T_b'(v)$ issues a read or write request to object x, then the only possible modification of the memory module of v is that a new copy of x is stored in this module and that copies of other objects are ejected out of this module.
- If a leaf node outside the subtree $T_b'(v)$ issues a read or write request to object x, then the only possible modification of the memory module of v is that copies of objects are ejected out of this module.

Any optimal off-line strategy that does not possess these two properties can be easily changed so that the properties are fulfilled without increasing the load on any edge.

Now, fix a sequence $\sigma = \sigma_1\sigma_2\cdots$ of requests issued by the leaf nodes of the corresponding tree. Let $e_m = (u_m, v_m)$ denote an arbitrary edge of T_m and let $e_b' = (u_b', v_b')$ denote the corresponding edge of T_b'. We assume that v_m and v_b' are closer to the root than u_m and u_b', respectively. Further, let us fix an optimal off-line strategy on T_b', which is denoted the *optimal strategy* in the following. Let $L(e_b')$ denote the minimum load of the optimal strategy on the edge e_b'. The basic strategy on T_m produces at most load $3 \cdot L(e_b')$ on the edge e_m. Therefore, it remains to prove that the additional load due to the rescue-and-help strategy, i.e., $L_1(e_m) + L_2(e_m)$, is not larger than $3\ell \cdot L(e_b')$ (on expectation).

First, we consider the on-line strategy on T_m. From Observation 3.1 we can conclude that $L_1(e_m) + L_2(e_m) \leq 3 \cdot \text{eject}(e_m)$. Let ρ denote the subsequence of σ that includes all requests issued in $T(u_m)$ and that are noticed by the local strategy on u_m, i.e., ρ includes any request influencing the behavior of the local strategy on u_m. Let $\text{opt}(\rho)$ denote the minimum load of an optimal off-line single-computer caching strategy running on ρ and having a memory module whose capacity is k times smaller than the one of u_m. Thus, the memory capacity of the single-computer caching strategy corresponds to the capacity of u_b'. As the local strategy is (k,ℓ)-competitive, we can conclude

$$E[L_1(e_m) + L_2(e_m)] \leq E[3 \cdot \text{eject}(e_m)] \leq 3\ell \cdot \text{opt}(\rho) \ .$$

Next we consider the optimal strategy on T_b'. Any request in ρ issued to an object x increases the load on the edge e_b' if the memory module of u_b' does not include a copy of x. Hence, the load on e_b' is larger or equal to the load of an optimal off-line single-computer caching strategy running on ρ and having a memory module of the same capacity as u_b'. As a consequence,

$$L(e_b') \geq \text{opt}(\rho) \ .$$

These two bounds give us the required result, i.e., $E[L_1(e_m) + L_2(e_m)] \leq 3\ell \cdot L(e_b')$. Therefore, the expected load on e_m is at most $3 + 3\ell$ times the optimal load on e_b' and hence at most $3 + 3\ell$ times the optimal load on e_b, too. This completes the proof of Lemma 3.2. ■

3.2.2 Non-uniform model

In this section, we generalize the caching strategy for the memory tree to a non-uniform cost model. Now, we describe how the three sub-strategies, the basic strategy, the local strategy, and the rescue-and-help strategy, have to be adapted to the non-uniform model.

For the basic strategy, we can use nearly any known strategy for trees without memory capacity constraints in the non-uniform cost model. The only restriction is that the strategy has to ensure that, for any object x, the nodes holding a copy of x build a connected component in the memory tree. The overall competitive ratio of our algorithm depends on the competitive ratio of the used basic strategy. In order to keep the description and the analysis of the overall strategy as simple as possible, we refer to a concrete example of a basic strategy in the following. We use the tree strategy introduced in Section 2.2.2. In Theorem 2.5 it is shown that this caching strategy does not only minimize the congestion, it minimizes the load on any individual edge in the tree up to a factor of 3. Note that this result is optimal because of the lower bound shown by Black and Sleator [BS89].

For the local strategy, any caching strategy for single-computers, e.g., LRU or RANDOM (see, e.g., [FKL$^+$91, ST85]), can be used. The overall competitive ratio of our algorithm depends on the used caching strategy. In order to keep the description and the analysis of the overall strategy as simple as possible, we refer to a concrete example of a local strategy in the following. We use the RANDOM caching strategy for single-computers. This strategy works as follows. When a data object should be loaded into a memory module, one of the slots in the memory module is selected uniformly at random. The object that may be stored in that slot is ejected.

The rescue-and-help strategy presented in Section 3.2.1 has to be adapted as follows to the non-uniform cost model. Suppose a leaf node u had issued a read or write request for object x. Then, the migration of copies is handled in the following way. Let v denote the node from $V(x)$ that is closest to the node u. The basic strategy establishes a path p from v to u and may migrate copies along that path. Suppose that the basic strategy creates new copies on each node on the sub-path of p from v to v', excluding v. Note that, possibly, $v = v'$. Then the rescue-and-help strategy simply establishes a path from $a(x,v)$ to u, and migrates possibly copies along that path. We distinguish three cases.

1. Suppose $a(x,v) \neq \text{top}^*(x)$. Figure 3.1 on Page 60 illustrates this situation. Then, the rescue-and-help strategy creates with probability $1/D$ a copy on each node on the path from $a(x,v)$ to v'. Note that in this case all migrations follow a path that leads downwards, i.e., in direction of the leaves.

2. Suppose $a(x,v) = \text{top}^*(x)$, and the path from $\text{top}^*(x)$ to u is going downwards only. Figure 3.1 on Page 60 illustrates this situation. In this case, the path includes $\text{top}(x)$. With probability $1/D$ the rescue-and-help strategy does the following. $c^*(x)$ is migrated to $\text{top}(x)$, and dummy copies are added to each

node on the migration path from $\text{top}^*(x)$ to $\text{top}(x)$, excluding $\text{top}(x)$. Then a copy on each node on the path from $\text{top}(x)$ to v' is created. The migration of $c^*(x)$ to $\text{top}(x)$ effectively re-unions $c^*(x)$ and $c(x)$.

3. Suppose $a(x,v) = \text{top}^*(x)$, and the path from $\text{top}^*(x)$ to u is going first upwards to a node w and then downwards to u. Figure 3.2 on Page 61 illustrates this situation. In this case, the rescue-and-help strategy first migrates $c^*(x)$ to w, thereby replacing all dummy copies on the path from $\text{top}^*(x)$ to w by real copies, excluding w. Starting from node w, which now holds $c^*(x)$, a copy on each node on the path from w to v' is created, which re-unions $c^*(x)$ and $c(x)$ as w is the new $\text{top}(x)$ node after serving the request.

It remains to describe how our caching strategy on the memory tree can be executed in the distributed setting in which all decisions are based solely on local information. The tree strategy introduced in Section 2.2.2 is used as basic strategy. This tree strategy can be executed in a distributed fashion. Then it is obvious that the rescue-and-help strategy can also be executed in the distributed setting. This completes the description of the caching strategy on the memory tree T_m.

As in the previous section, we compare the load of the caching strategy on the memory tree T_m to the optimal load on the corresponding bandwidth tree T_b. Recall that the ratio between the total memory capacity of T_m and the total memory capacity of T_b is $k \cdot h + 1$, where h denotes the height of T_m and T_b. The following lemma compares the load induced by our caching strategy running on T_m with the load of an optimal off-line strategy running on T_b, both of them applied to the same, arbitrary sequence of requests issued by the leaf nodes of the corresponding tree. The lemma shows that our caching strategy minimizes the load on all edges simultaneously. Hence, the congestion produced by the on-line strategy executed on T_m is at most $3 + 3k/(k-1)$ times the congestion produced by an optimal off-line strategy executed on T_b.

Lemma 3.4 *Let e_b and e_m denote a pair of corresponding edges in T_b and T_m, respectively. Then the expected load on e_m is at most $3 + 3k/(k-1)$ times the optimal load on e_b.*

Proof. Unfortunately, the behavior of an optimal off-line strategy on the bandwidth tree T_b is difficult to understand. In order to get a more obvious behavior, we allow the optimal strategy to place copies on internal nodes, too. Let T_b' denote a tree that is isomorph to T_b, whose leaf nodes have the same memory capacity as the corresponding leaf nodes in T_b, and whose inner nodes have a memory capacity that is equal to the sum of the capacities of their respective children. We restrict, however, the placement of copies on T_b' in the following way: For a node v, let $T_b'(v)$ denote the subtree rooted at node v. Anytime, for each node v, the set of objects that are stored in a memory module in the subtree $T_b'(v)$, must be a subset of the objects that have a copy or dummy copy in the memory module of v. Recall that dummy copies are placeholders that cannot be used to serve a read request and need not to be updated in case of a write request.

We assume that these copies can be created from scratch, that is, they do not require any kind of migration. Lemma 3.3 shows that, despite of this restrictions, T_b' is more powerful than T_b, so that any lower bound on the load on the edges of T_b' holds also for T_b.

In contrast to the bandwidth tree T_b, the behavior of an optimal off-line strategy on T_b' is relatively clear. W.l.o.g., any optimal off-line strategy fulfills the following properties for each node v in T_b'.

- If a leaf node in the subtree $T_b'(v)$ issues a read or write request to object x, then the only possible modification of the memory module of v is that a new copy of x is stored in this module and that copies of other objects are ejected out of this module.
- If a leaf node outside the subtree $T_b'(v)$ issues a read or write request to object x, then the only possible modification of the memory module of v is that copies of objects are ejected out of this module.

Any optimal off-line strategy that does not possess these two properties can be easily changed so that the properties are fulfilled without increasing the load on any edge.

Now, fix a sequence $\sigma = \sigma_1\sigma_2\cdots$ of requests issued by the leaf nodes of the corresponding tree. Let $e_m = (u_m, v_m)$ denote an arbitrary edge of T_m and let $e_b' = (u_b', v_b')$ denote the corresponding edge of T_b'. We assume that v_m and v_b' are closer to the root than u_m and u_b', respectively. Further, let us fix an optimal off-line strategy on T_b', which is denoted the *optimal strategy* in the following. Let L_{opt} denote the minimum load of the optimal strategy on edge e_b'. The basic strategy on T_m produces at most load $3 \cdot L_{\text{opt}}$ on edge e_m. Therefore, it remains to prove that the additional load due to the rescue-and-help strategy on edge e_m is not larger than $3k/(k-1) \cdot L_{\text{opt}}$ on expectation.

We use a potential function argument (see, e.g., [ST85]). Let $L_{\text{r+h}}(t)$ denote the additional load due to the rescue-and-help strategy on edge e_m, and let $L_{\text{opt}}(t)$ denote the minimum load of the optimal strategy on edge e_b', after serving σ_t. Let $M_{\text{on}}(t)$ and $M_{\text{opt}}(t)$ denote the set of objects stored in the memory module of u_m and u_b', respectively, after serving σ_t. We define

$$\Phi(t) = 3k/(k-1) \cdot D \cdot |M_{\text{opt}}(t) \setminus M_{\text{on}}(t)| \ .$$

In order to prove the lemma, we show the invariant

$$E[L_{\text{r+h}}(t) + \Phi(t)] \leq 3k/(k-1) \cdot L_{\text{opt}}(t) \ .$$

This invariant can be shown by an induction on the length of σ. Obviously, this invariant holds initially. Assume that $E[L_{\text{r+h}}(t) + \Phi(t)] \leq 3k/(k-1) \cdot L_{\text{opt}}(t)$. We have to show that $E[L_{\text{r+h}}(t+1) + \Phi(t+1)] \leq 3k/(k-1) \cdot L_{\text{opt}}(t+1)$. Let $\Delta L_{\text{r+h}} = L_{\text{r+h}}(t+1) - L_{\text{r+h}}(t)$, $\Delta L_{\text{opt}} = L_{\text{opt}}(t+1) - L_{\text{opt}}(t)$, and $\Delta\Phi = \Phi(t+1) - \Phi(t)$.

We distinguish between requests issued inside or outside the subtree $T_m(u_m)$.

- Suppose σ_{t+1} is a read or write request for object x issued inside the subtree $T_m(u_m)$. In this case, Table 3.1 contains all possible changes of configuration.

$x \in M_{\text{on}}(t)$	$x \in M_{\text{opt}}(t)$	$x \in M_{\text{opt}}(t+1)$	$E[\Delta L_{\text{r+h}}] \leq$	$E[\Delta\Phi] \leq$	$\Delta L_{\text{opt}} \geq$
yes	no	no	0	0	1
yes	no	yes	0	0	D
yes	yes	yes	0	0	0
no	no	no	3	$1/D \cdot 1/k \cdot 3k/(k-1) \cdot D$	1
no	no	yes	3	$(1/D \cdot 1/k + (D-1)/D) \cdot 3k/(k-1) \cdot D$	D
no	yes	yes	3	$1/D \cdot (-1+1/k) \cdot 3k/(k-1) \cdot D$	0

Table 3.1: Possible changes of configuration if σ_{t+1} is a read or write request for object x issued inside the subtree $T_m(u_m)$.

Note that in this case $E[\Delta L_{\text{r+h}}] = 0$, if $x \in M_{\text{on}}(t)$, and $E[\Delta L_{\text{r+h}}] \leq 1 + 1/D \cdot 2D = 3$, otherwise. In addition, note that the probability that RANDOM ejects an object being in $M_{\text{opt}}(t)$ is

$$|M_{\text{opt}}(t)|/\text{capacity}(M_{\text{on}}(t)) \leq 1/k$$

because the capacity of RANDOM is k times the capacity of the optimal off-line algorithm. The table implies that $E[\Delta L_{\text{r+h}} + \Delta\Phi] \leq 3k/(k-1) \cdot \Delta L_{\text{opt}}$.

- Suppose σ_{t+1} is a read or write request for object x issued outside the subtree $T_m(u_m)$. In this case, $E[\Delta L_{\text{r+h}}] = 0$ and $E[\Delta\Phi] \leq 0$. Thus, $E[\Delta L_{\text{r+h}} + \Delta\Phi] \leq 3k/(k-1) \cdot \Delta L_{\text{opt}}$.

Hence, $E[L_{\text{r+h}}(t+1) + \Phi(t+1)] \leq 3k/(k-1) \cdot L_{\text{opt}}(t+1)$, which implies the lemma. ■

3.3 Applications of the general framework

The general framework can be applied to several classes of networks. Given any network G, the bandwidth tree $T_b(G)$ and the memory tree $T_m(G)$ can be constructed by a hierarchical decomposition of G. The quality of the resulting caching strategy depends on some properties of this decompositions and especially on the height of $T_b(G)$ and $T_m(G)$. Although it is not clear which properties can be obtained for general networks, applying the framework to almost any standard network yields interesting results. In this section we give some examples, including meshes, fat-trees, complete networks, Cayley networks, edge symmetric networks, and hypercubic networks. Further, we present the universal caching strategy working on arbitrary networks for which an oblivious routing strategy is given.

3.3.1 Meshes

In this section, we consider caching strategies for the mesh $M = M(m_1, \ldots, m_d)$, i.e., the d-dimensional mesh-connected network with side length $m_i \geq 2$ in dimension i. The number of nodes is denoted by n, i.e., $n = m_1 \cdots m_d$, the number of edges is $\sum_{i=1}^{d} m_1 \cdots m_{i-1} \cdot (m_i - 1) \cdot m_{i+1} \cdots m_d = \Theta(d \cdot n)$. Each node has a memory module of uniform capacity. Each edge has bandwidth 1. Thus, the relative and absolute load of an edge are identical.

The general framework is based on the access tree strategy. Thus, we adopt the hierarchical decomposition of M and the associated decomposition tree $T(M)$ of height $O(\log n)$ from Section 2.3.1.

Since we obtain, for meshes, a tradeoff between the height of the decomposition tree and the achieved congestion, we need a decomposition tree of arbitrary height $1 \leq h \leq \log n$. Thus, some levels of $T(M)$ have to be deleted possibly. This is done with the following deletion algorithm. While $\text{height}(T(M)) > h$, delete a level $0 < \ell < \text{height}(T(M))$ in $T(M)$ having minimum degree, where the degree of level ℓ is the maximum degree over all nodes on level ℓ. Deleting level ℓ means that, first, all nodes and edges on level ℓ are deleted, then the edges on level $\ell + 1$ are connected to the respective nodes on level $\ell - 1$, and finally the numbering of the levels is adjusted.

Now, the decomposition tree $T(M)$ has height h and degree $O(n^{1/h})$. Note that the root of $T(M)$ corresponds still to M itself, and the children of a node v in the tree correspond to the submeshes into which the submesh corresponding to v is divided, and the leaves correspond to subtrees of size one, i.e., to the nodes of M.

Define the bandwidth tree $T_b(M)$ and the memory tree $T_m(M)$ to be a copy of the decomposition tree $T(M)$. Each of the leaf nodes in $T_b(M)$ has a memory module of the same capacity as the respective node in M. The inner nodes in $T_b(M)$ do not have memory modules. Each of the leaf nodes in $T_m(M)$ has a memory module whose capacity is 2 times the capacity of the respective node in M. Furthermore, each non-leaf node in $T_m(M)$, except for the root, is assumed to have a memory module whose capacity is equal to the sum of the capacities of its children. The memory capacity of the root in $T_m(M)$ is equal to the sum of the capacities of all mesh nodes. Thus, it is guaranteed that the root node can hold all data objects simultaneously.

Our caching strategy for meshes simulates the strategy for the memory tree $T_m(M)$ on M. The virtual memory module of an inner node v of the memory tree $T_m(M)$ is simulated jointly by all the nodes in the submesh $M(v)$. The assignment of the slots in the memory module of v to the nodes in the submesh $M(v)$ is done in random fashion as follows. We permute uniformly at random all slots in the memory module of v. Then the memory module is partitioned into equally sized, contiguous parts, one for each of the nodes in $M(v)$. If v holds a copy of object x, let $s(x,v)$ denote the memory slot in the memory module of v holding the copy of x. Summing over all levels in the memory tree $T_m(M)$, we obtain that the on-line strategy needs, for each node of M, $2 \cdot h + 1$ times the memory capacities of the off-line strategy.

Now, we have to describe, how, in case of an access, a copy of an object is located in a submesh. For each object x, and each node v of the memory tree $T_m(M)$, we add a signpost $p(x,v)$. If a copy of x is stored in the memory module of v, $p(x,v)$ points to the mesh node in $M(v)$ holding the memory slot $s(x,v)$, and vice versa. In addition, $p(x,v)$ points to each signpost $p(x,u)$, with u is adjacent to v in the memory tree $T_m(M)$. We embed the signposts randomly into M, i.e., for each object x, and each node v of $T_m(M)$, $p(x,v)$ is mapped uniformly at random to one of the nodes in $M(v)$. Note that each signpost contains no data of an object. Thus, the memory requirements of the signposts can be neglected.

The remaining description of our caching strategy is simple: Suppose a message m, concerning an object x, that is sent from a node v to an adjacent node u in the memory tree $T_m(M)$. In addition, suppose that v and u hold a copy of x. In the mesh M this message m is sent along the dimension-by-dimension order path from the mesh node holding the memory slot $s(x,v)$ to the mesh node holding the signpost $p(x,v)$, then to the mesh node holding the signpost $p(x,u)$, and finally to the mesh node holding the slot $s(x,u)$. The dimension-by-dimension order path between two nodes is the unique shortest path between the two nodes using first edges of dimension 1, then edges of dimension 2, and so on. If v or u holds no copy of x, the path in the mesh M is appropriately shortened.

Finally, we require the general remapping scheme for frequently accessed memory slots and signposts. In case of the remapping of a memory slot $s(x,v)$, $s(x,v)$ is swapped with a uniform at random chosen memory slot in the submesh $M(v)$. For more details see Section 2.3.1.

Note that the distributed execution of the local strategy based on LRU is a major problem since the virtual memory modules of the memory tree may be distributed among several nodes in the mesh. Therefore, we prefer to use RANDOM instead of LRU.

The following theorem shows that, for each $1 \leq h \leq \log n$, we obtain a caching strategy for the d-dimensional mesh M with n nodes that is $(2h+1, O(d \cdot h \cdot n^{1/h}))$-competitive, w.h.p. Further, setting $h = \log n$, we obtain a caching strategy that is strictly $(2\log n + 1, O(d \cdot \log n))$-competitive, w.h.p. Recall for the latter result that $2d \cdot C_{\text{opt}}(M) \geq \min\{D, \kappa\}$.

Theorem 3.5 *Consider an application on the d-dimensional mesh M with n nodes. For each $1 \leq h \leq \log n$, we obtain a caching strategy with congestion $O(d \cdot h \cdot n^{1/h} \cdot C_{\text{opt}}(M) + \min\{n^{2/h} \cdot D, \kappa\} \cdot n^{1/h} \cdot \log n)$, w.h.p., where $C_{\text{opt}}(M)$ denotes the optimal congestion for the application, and κ denotes the maximum number of write requests directed to the same object.*

Proof. In order to prove the above result we apply Lemma 2.9 which holds analogously in this setting. Define $C_{\text{opt}}(T_b(M))$ to be the optimal congestion for the application when it is executed on the bandwidth tree $T_b(M)$, under the assumption that each node of M is simulated by its counterpart in $T_b(M)$, which is one of the leaf nodes.

To apply effectively Lemma 2.9, we have to show that the bandwidth tree $T_b(M)$ is Ψ_M-useful for a certain Ψ_M, i.e., we have to prove the following two conditions.

- $C_{\text{opt}}(T_b(M)) \leq C_{\text{opt}}(M)$.
- The maximum expected relative load in M due to access messages is $O(d \cdot h \cdot n^{1/h} \cdot C_{\text{opt}}(T_b(M)))$.

In addition, note that $\Delta(T_b(M)) = \deg(T_b(M)) = O(n^{1/h})$. Then Theorem 3.5 can be yield with Lemma 2.9.

The lower bound $C_{\text{opt}}(T_b(M)) \leq C_{\text{opt}}(M)$ for the optimal strategy can be yield with Lemma 2.7. The following lemma gives the upper bound on the expected load of our caching strategy.

Lemma 3.6 *The maximum expected relative load in M due to access messages is* $O(d \cdot h \cdot n^{1/h} \cdot C_{\text{opt}}(T_b(M)))$.

Proof. Let $L_\ell(e)$ denote the load on e due to the simulation of access messages passing edges on level ℓ of the memory tree $T_m(M)$, for $1 \leq \ell \leq h$. We show that $E[L_\ell(e)] = O(d \cdot n^{1/h} \cdot C_{\text{opt}}(T_b(M)))$, for $1 \leq \ell \leq h$, which yields the lemma.

Fix a level ℓ. Let v be a node of $T_m(M)$ on level $\ell - 1$ such that $M(v)$ includes the edge e. If such a node does not exist then $E[L_\ell(e)] = 0$. Let v' be one of the children of v. From the proof of Lemma 2.8 we can obtain that the expected load on e due to the simulation of the memory tree edge $e_T = \{v, v'\}$ on level ℓ of $T_m(M)$ is at most $O(d)$ times the relative load on the memory tree edge e_T. Thus, the expected load on e due to the simulation of the memory tree edge e_T is $O(d \cdot C_{\text{opt}}(T_b(M)))$, since Lemma 3.4 yields with $k = 2$ that the congestion produced by the on-line strategy executed on $T_m(M)$ is at most 9 times the congestion produced by an off-line strategy executed on $T_b(M)$. The same bound holds for the edges connecting v with its other $O(n^{1/h})$ children. Hence, $E[L_\ell(e)] = O(d \cdot n^{1/h} \cdot C_{\text{opt}}(T_b(M)))$, which yields the lemma. ■

This completes the proof of Theorem 3.5. ■

3.3.2 Fat-trees

In this section, we consider caching strategies for the fat-tree F of height H with n nodes. F has the topology of a symmetric tree, that is, for each inner-node, the subtrees rooted at the children of the node are isomorph. The fat-tree represents an indirect network, that is, only the leaf nodes are processors with memory modules of uniform capacity, the inner nodes are only routing switches. We define the root of F to be on level 0, and all nodes whose parents are on level i are defined to be on level $i+1$. Furthermore, each edge e of F connecting a level i node with a level $i+1$ node is defined to be on level $i+1$. Thus, $F = F((d_1, b_1), \ldots, (d_H, b_H))$, i.e., for every level i in the tree F each node on level i has d_{i+1} children and each edge on level i has bandwidth b_i. We make the reasonable assumption, for each level i, $b_i \geq b_{i+1}$.

The fat-tree F is called realistic if the bandwidths of the levels decrease geometrically in direction to the root, i.e., if, for each level i, $b_i \leq \alpha \cdot d_{i+1} \cdot b_{i+1}$, for some constant $\alpha < 1$.

The general framework is based on the access tree strategy. Thus, we adopt the hierarchical decomposition of F and the associated decomposition tree $T(F)$ of arbitrary height $1 \leq h \leq H$ from Section 2.3.2. The remaining description of our caching strategy is analogously to the caching strategy for meshes in Section 3.3.1.

The following theorem shows that, for each $1 \leq h \leq H$, we obtain a caching strategy for fat-trees of height H with n nodes that is $(2h+1, O(h+n^{1/h}))$-competitive, w.h.p., and, if $h = H$, $(2H+1, O(H))$-competitive, w.h.p. For realistic fat-trees this result improves to $(2h+1, O(n^{1/h}))$-competitive, w.h.p., and, if $h = H$, to $(2H+1, O(1))$-competitive, w.h.p. Further, we obtain, for each $1 \leq h \leq H$, a caching strategy that is strictly $(2h+1, O((\deg(F)+n^{1/h})^3 \cdot \log n))$-competitive, w.h.p. Recall for the latter result that $C_{\text{opt}}(F) \geq \min\{D, \kappa/\deg(F)\}$.

Theorem 3.7 *Consider an application on the fat-tree F of height H with n nodes. Let $C_{\text{opt}}(F)$ denote the optimal congestion for the application, and let κ denote the maximum number of write requests directed to the same object.*

- *For each $1 \leq h \leq H$, we obtain a caching strategy with congestion $O((h+n^{1/h}) \cdot C_{\text{opt}}(F) + R)$, w.h.p., and, if $h = H$, with congestion $O(H \cdot C_{\text{opt}}(F) + R)$, w.h.p., with $R = \min\{(\max\{\deg(F), n^{1/h}\})^2 \cdot D, \kappa\} \cdot \max\{\deg(F), n^{1/h}\} \cdot \log n$.*
- *For realistic fat-trees this improves to $O(n^{1/h} \cdot C_{\text{opt}}(F) + R)$, w.h.p., and, if $h = H$, to $O(C_{\text{opt}}(F) + R)$, w.h.p.*

Proof. This proof is analogously to the proof of Theorem 2.14 concerning caching on fat-trees without memory capacity constraints. ■

3.3.3 Complete networks

Some massively parallel computers, e.g., Cray T3E and T3D or Intel Paragon, have a network with very high bandwidth, so that not the network is the bottleneck but the individual memory modules. In particular, remote accesses to these modules are expensive, as local accesses are supported by additional local caches. These systems are well modeled by a complete network G_n with n nodes. We aim to minimize the congestion at the nodes due to remote accesses, that is, remote accesses increase the load at any node whereas local accesses are free. Each node has a memory module of uniform capacity and bandwidth 1. Thus, the relative and absolute load of a node are identical.

The general framework is based on the access tree strategy. Thus, we adopt the hierarchical decomposition of G_n and the associated decomposition tree $T(G_n)$ of arbitrary height $1 \leq h \leq \log n$ from Section 2.3.3. The remaining description of our caching strategy is analogously to the caching strategy for meshes in Section 3.3.1.

The following theorem shows that we obtain, for $h = 1$, a caching strategy for complete networks with n nodes that is $(3, O(1))$-competitive, w.h.p., with respect to the congestion at the nodes. Further, we obtain, for $h = \log n$, a caching strategy that is strictly $(2\log n + 1, O(\log n))$-competitive, w.h.p., with respect to the congestion at the nodes. Recall for the latter result that $C_{\text{opt}}(G_n) \geq \min\{D, \kappa\}$.

Theorem 3.8 *Consider an application on the complete network G_n with n nodes. for each $1 \leq h \leq \log n$, we obtain a caching strategy with congestion $O(h \cdot C_{\text{opt}}(G_n) + \min\{n^{2/h} \cdot D, \kappa\} \cdot n^{1/h} \cdot \log n)$ at the nodes, w.h.p., where $C_{\text{opt}}(G_n)$ denotes the optimal congestion at the nodes for the application, and κ denotes the maximum number of write requests directed to the same object.*

Proof. This proof is analogously to the proof of Theorem 2.17 concerning caching on complete networks without memory capacity constraints. ■

3.3.4 The universal caching strategy

In this section, we present the universal caching strategy working on arbitrary networks for which an oblivious routing strategy is given. A routing strategy is called oblivious if the path traveled by each packet depends only on the origin and destination of the packet and not on the origins and destinations of the other packets nor on congestion encountered during the routing. Note that each oblivious routing strategy is also an on-line routing strategy. Finally, we apply the universal caching strategy to several classes of networks.

Suppose we are given an arbitrary network G with n nodes and an oblivious routing strategy for G. Each node is assumed to have a memory module of uniform capacity, and each edge is assumed to have bandwidth 1. Thus, the relative and absolute load of an edge are identical. The universal caching strategy for G is a simulation of the access tree strategy for the complete network G_n on G. Note that both G and G_n have n nodes such that for each node in G a counterpart in G_n can be fixed. All messages that are sent between adjacent nodes in G_n are sent along the paths, determined by the oblivious routing strategy, between the associated nodes in G.

The following theorem shows that, for a network G in which a permutation can be routed obliviously with maximum expected load $L_\pi(G)$, the universal caching strategy is $(3, O(L_\pi(G) \cdot \deg(G)))$-competitive, w.h.p., and strictly $(2\log n + 1, O(L_\pi(G) \cdot \deg(G) \cdot \log n))$-competitive, w.h.p. Recall for the latter result that $\deg(G) \cdot C_{\text{opt}}(G) \geq \min\{\kappa, D\}$.

Theorem 3.9 *Consider an application on a network G with n nodes in which a permutation can be routed obliviously with maximum expected load $L_\pi(G)$. For each $1 \leq h \leq \log n$, we obtain a caching strategy with congestion $O(L_\pi(G) \cdot \deg(G) \cdot h \cdot C_{\text{opt}}(G) + \min\{n^{2/h} \cdot D, \kappa\} \cdot n^{1/h} \cdot \log n)$, w.h.p., where $\deg(G)$ denotes the degree of G, $C_{\text{opt}}(G)$ denotes the optimal congestion for the application, and κ denotes the maximum number of write requests directed to the same object.*

Proof. The proof is analogously to the proof of Theorem 2.20 concerning the universal caching strategy without memory capacity constraints. ■

Cayley networks

The following corollary can be concluded with Theorem 3.9 and the remarks in Section 2.3.4. This corollary shows that, for a Cayley network G with n nodes, the universal caching strategy is $(3, O(\text{diam}(G) \cdot \deg(G))$-competitive, w.h.p., and strictly $(2\log n + 1, O(\text{diam}(G) \cdot \deg(G) \cdot \log n))$-competitive, w.h.p.

Corollary 3.10 *Consider an application on a Cayley network G with n nodes. For each $1 \leq h \leq \log n$, we obtain a caching strategy with congestion $O(\text{diam}(G) \cdot \deg(G) \cdot h \cdot C_{\text{opt}}(G) + \min\{n^{2/h} \cdot D, \kappa\} \cdot n^{1/h} \cdot \log n)$, w.h.p., where $C_{\text{opt}}(G)$ denotes the optimal congestion for the application, and κ denotes the maximum number of write requests directed to the same object.*

Edge symmetric networks

The following corollary can be concluded with Theorem 3.9 and the remarks in Section 2.3.4. This corollary shows that, for an edge symmetric network G with n nodes, the universal caching strategy is $(3, O(\text{diam}(G))$-competitive, w.h.p., and strictly $(2\log n + 1, O(\text{diam}(G) \cdot \log n))$-competitive, w.h.p.

Corollary 3.11 *Consider an application on an edge symmetric network G with n nodes. For each $1 \leq h \leq \log n$, we obtain a caching strategy with congestion $O(\text{diam}(G) \cdot h \cdot C_{\text{opt}}(G) + \min\{n^{2/h} \cdot D, \kappa\} \cdot n^{1/h} \cdot \log n)$, w.h.p., where $C_{\text{opt}}(G)$ denotes the optimal congestion for the application, and κ denotes the maximum number of write requests directed to the same object.*

Hypercubic networks

The following corollary can be concluded with Theorem 3.9 and the remarks in Section 2.3.4. This corollary shows that, for the wrapped butterfly, cube-connected-cycles, hypercube, de Bruijn, and shuffle-exchange network with n nodes, the universal caching strategy is $(3, O(\log n))$-competitive, w.h.p., and strictly $(2\log n + 1, O((\log n)^2))$-competitive, w.h.p.

Corollary 3.12 *Consider an application on the wrapped butterfly, cube-connected-cycles, hypercube, de Bruijn, or shuffle-exchange network G with n nodes. For each $1 \leq h \leq \log n$, we obtain a caching strategy with congestion $O(\log n \cdot h \cdot C_{\text{opt}}(G) + \min\{n^{2/h} \cdot D, \kappa\} \cdot n^{1/h} \cdot \log n)$, w.h.p., where $C_{\text{opt}}(G)$ denotes the optimal congestion for the application, and κ denotes the maximum number of write requests directed to the same object.*

The following corollary can be concluded with Theorem 3.9 and the remarks in Section 2.3.4. This corollary shows that, for the indirect butterfly network with n nodes, the universal caching strategy is $(3, O(1))$-competitive, w.h.p., and strictly $(2\log n + 1, O(\log n))$-competitive, w.h.p.

Corollary 3.13 *Consider an application on the indirect butterfly network G with n nodes. For each $1 \leq h \leq \log n$, we obtain a caching strategy with congestion $O(h \cdot C_{\text{opt}}(G) + \min\{n^{2/h} \cdot D, \kappa\} \cdot n^{1/h} \cdot \log n)$, w.h.p., where $C_{\text{opt}}(G)$ denotes the optimal congestion for the application, and κ denotes the maximum number of write requests directed to the same object.*

Chapter 4

Summary and Discussion

We have presented and analyzed a general framework for the development of distributed caching strategies for networks with memory capacity constraints in a non-uniform cost model. The strategies aim to minimize the network congestion in order to minimize the communication overhead and avoid that some of the links become a bottleneck. We have shown that this framework yields efficient strategies that are able to exploit the locality included in an arbitrary application for several examples of networks, including meshes, fat-trees, complete networks, Cayley networks, edge symmetric networks, and hypercubic networks. In contrast to previously known strategies, our strategies give an integrated solution for the problem of data placement, data tracking, and routing. In the following we present a detailed summary of the results of this thesis. Table 4.1 and 4.2 give a survey over the results for caching without and with memory capacity constraints, respectively.

Trees. We present the first deterministic and distributed caching strategy that achieves competitive ratio 3 for trees in a non-uniform cost model. This competitive ratio is optimal because of the lower lower bound shown by Black and Sleator [BS89]. Our tree strategy minimizes not only the congestion but minimizes simultaneously the load on each individual edge up to a optimal factor of 3. Strategies for trees are of special interest as they can be used as subroutines in strategies for other networks, e.g., in the access tree strategy. However, our strategy neglects memory capacity constraints.

Meshes. The situation on mesh-connected networks is much more complicated than the one on trees because in meshes there are several possible routing paths between every pair of nodes. Let M denote the d-dimensional mesh with side length $n_i \geq 2$ in dimension i. This graph represents a network of $n = n_1 \cdots n_d$ processors, all of them with uniform memory capacity, and $\sum_{i=1}^{d} n_1 \cdots n_{i-1} \cdot (n_i - 1) \cdot n_{i+1} \cdots n_d$ communication links, all of them with uniform bandwidth.

Constructing bandwidth trees of different heights, we obtain a tradeoff between the bandwidth and memory utilization: For each $1 \leq h \leq \log n$, we obtain a caching

Caching without memory capacity constraints		
network	non-strict competitive ratio	strict competitive ratio
tree	3	3
d-dimensional mesh	$O(d \cdot \log n)$	$O(d \cdot \log n)$
fat-tree F of height H	$O(\min\{H, \log n / \log\log n)$	$O(\deg(F)^3 \cdot \log n)$
realistic fat-tree F of height H	$O(1)$	$O(\deg(F)^3 \cdot \log n)$
complete network (congestion at the nodes)	$O(1)$	$O(\log n)$
Cayley network G	$O(\text{diam}(G) \cdot \deg(G))$	$O(\text{diam}(G) \cdot \deg(G) \cdot \log n)$
edge symmetric network G	$O(\text{diam}(G))$	$O(\text{diam}(G) \cdot \log n)$
hypercube cube-connected-cycles de Bruijn shuffle-exchange wrapped butterfly	$O(\log n)$	$O((\log n)^2)$
indirect butterfly	$O(1)$	$O(\log n)$

Table 4.1: Survey over the results for caching without memory capacity constraints. Recall that n denotes the number of nodes in the networks and that all results hold w.h.p.

strategy for the d-dimensional mesh M with n nodes that is $(2h+1, O(d \cdot h \cdot n^{1/h}))$-competitive, w.h.p. Setting $h = \log n$, we obtain the strict congestion ratio $O(d \cdot \log n)$, w.h.p. Further, we present an $\Omega(\log n / d)$ lower bound for the competitive ratio for on-line routing in meshes, which implies that our upper bound on the congestion ratio for meshes of constant dimension is optimal.

Fat-trees. Define a fat-tree F of height H with n nodes to be a graph that has the topology of a symmetric tree, that is, for each inner-node, the subtrees rooted at the children of the node are isomorph. The fat-tree represents an indirect network, that is, only the leaf nodes are processors with memory modules of uniform capacity, the inner nodes are only routing switches.

Constructing bandwidth trees of different heights, we obtain a better trade-off between the bandwidth and memory utilization than for the mesh: For each $1 \leq h \leq H$, we obtain a caching strategy for a fat-tree F of height H with n nodes that is $(2h+1, O(h + n^{1/h}))$-competitive, w.h.p., and, if $h = H$, $(2H+1, O(H))$-

Caching with memory capacity constraints				
network	non-strict		strict	
	memory ratio	congestion ratio	memory ratio	congestion ratio
d-dimensional mesh	$2h+1$	$O(d \cdot h \cdot n^{1/h})$	$2\log n+1$	$O(d \cdot \log n)$
fat-tree F of height H	$2h+1$ $2H+1$	$O(h+n^{1/h})$ $O(H)$	$2h+1$	$O((\deg(F)+n^{1/h})^3 \cdot \log n)$
realistic fat-tree F of height H	$2h+1$ $2H+1$	$O(n^{1/h})$ $O(1)$	$2h+1$	$O((\deg(F)+n^{1/h})^3 \cdot \log n)$
complete network (congestion at the nodes)	3	$O(1)$	$2\log n+1$	$O(\log n)$
Cayley network G	3	$O(\mathrm{diam}(G) \cdot \deg(G))$	$2\log n+1$	$O(\mathrm{diam}(G) \cdot \deg(G) \cdot \log n)$
edge symmetric network G	3	$O(\mathrm{diam}(G))$	$2\log n+1$	$O(\mathrm{diam}(G) \cdot \log n)$
hypercube cube-connected-cycles de Bruijn shuffle-exchange wrapped butterfly	3	$O(\log n)$	$2\log n+1$	$O((\log n)^2)$
indirect butterfly	3	$O(1)$	$2\log n+1$	$O(\log n)$

Table 4.2: Survey over the results for caching with memory capacity constraints. Note that h is variable with $1 \leq h \leq \log n$ for the d-dimensional mesh and $1 \leq h \leq H$ for the (realistic) fat-tree of height H. Recall that n denotes the number of nodes in the networks and that all results hold w.h.p.

competitive, w.h.p. Further, we obtain, for each $1 \leq h \leq H$, a caching strategy that is strictly $(2h+1, O((\deg(F)+n^{1/h})^3 \cdot \log n))$-competitive, w.h.p. Setting $h = \min\{H, \log n/\log\log n\}$, we obtain the congestion ratio $O(\min\{H, \log n/\log\log n\})$, w.h.p., and the strict congestion ratio $O(\deg(F)^3 \cdot \log n)$, w.h.p.

Usually, it is assumed that the bandwidths of the levels decrease geometrically in direction to the root. Let B_i denote the sum of the bandwidths of all edges on level i with level H edges being incident on the leaf nodes. A fat-tree is called realistic, if, for each level i, $B_i \leq \alpha \cdot B_{i+1}$, for some constant $\alpha < 1$. For realistic fat-trees, we obtain, for each $1 \leq h \leq H$, a caching strategy that is $(2h+1, O(n^{1/h}))$-competitive, w.h.p., and, if $h = H$, $(2H+1, O(1))$-competitive, w.h.p.

Complete networks. Some massively parallel computers, e.g., Cray T3E and T3D or Intel Paragon, have a network with very high bandwidth, so that not the network is the bottleneck but the individual memory modules. In particular, remote accesses to these modules are expensive, as local accesses are supported by additional local caches. These systems are well modeled by a complete network of nodes with memory modules of uniform capacity. We aim to minimize the congestion at the nodes due to remote accesses, that is, remote accesses increase the load at any node whereas local accesses are free. Each node is assumed to have uniform bandwidth. We obtain a caching strategy for complete networks with n nodes that is $(3, O(1))$-competitive, w.h.p., and a caching strategy that is strictly $(2\log n+1, O(\log n))$-competitive, w.h.p., with respect to the congestion at the nodes.

The universal caching strategy. We present the universal caching strategy working on arbitrary networks for which an oblivious routing strategy is given. Each node is assumed to have a memory module of uniform capacity, and each edge is assumed to have bandwidth 1. For a network G in which a permutation can be routed obliviously with maximum expected load $L_\pi(G)$, the universal caching strategy is $(3, O(L_\pi(G) \cdot \deg(G)))$-competitive, w.h.p., and strictly $(2\log n+1, O(L_\pi(G) \cdot \deg(G) \cdot \log n))$-competitive, w.h.p. This means that the universal caching strategy is well suited for networks having a degree at most logarithmic in the network size and in which a permutation can be routed obliviously with maximum expected load at most logarithmic in the network size. We apply the universal caching strategy to several classes of networks.

- **Cayley networks.** For a Cayley network G with n nodes, the universal caching strategy is $(3, O(\mathrm{diam}(G) \cdot \deg(G))$-competitive, w.h.p., and strictly $(2\log n + 1, O(\mathrm{diam}(G) \cdot \deg(G) \cdot \log n))$-competitive, w.h.p.

- **Edge symmetric networks.** For an edge symmetric network G with n nodes, the universal caching strategy is $(3, O(\mathrm{diam}(G))$-competitive, w.h.p., and strictly $(2\log n+1, O(\mathrm{diam}(G) \cdot \log n))$-competitive, w.h.p.

- **Hypercubic networks.** For the wrapped butterfly, cube-connected-cycles, hybercube, de Bruijn, and shuffle-exchange network with n nodes, the universal caching strategy is $(3, O(\log n))$-competitive, w.h.p., and strictly $(2\log n + 1, O((\log n)^2))$-competitive, w.h.p. Further, for the indirect butterfly network with n nodes, the universal caching strategy is $(3, O(1))$-competitive, w.h.p., and strictly $(2\log n+1, O(\log n))$-competitive, w.h.p.

Clustered networks. We show how the access tree strategy can be applied to Internet-like clustered networks. A clustered network is a network that consists of several small subnetworks, i.e., clusters, that are organized hierarchically. Communication between nodes of the same cluster is not as expensive as communication

between nodes of different clusters. The access trees are embedded into the clustered network in a preprocessing step. We show that this preprocessing can be done efficiently and locally for each participating cluster. Thus, we yield an efficient caching strategy that is suitable, e.g., for managing WWW pages. In contrast to the caching strategies currently used in the Internet, our strategy keeps all copies of a data object consistent such that, e.g., modifications of WWW pages, are propagated to all copies of the page. However, our strategy neglects memory capacity constraints.

Previous models for caching in networks simply aim to minimize the total communication load, i.e., the sum, taken over all messages, of the size of the messages multiplied with the length of their routing paths. This can result in bottlenecks. We have improved upon such models as our model aims to minimize the congestion, i.e., the maximum, taken over all links, of the amount of data transmitted by the link, which corresponds to distributing the load evenly among all network resources. However, in contrast to other computational models for parallel computers, e.g., the BSP [Val90] and the LogP model [CKP+96], our congestion based model does not have any notion of time. We still simply sum up the load over all time steps. Thus, our models avoids bottlenecks in space but not in time. The BSP model, for example, assumes that a parallel algorithm proceeds in synchronized rounds. An adequate congestion measure for this model considers the sum, over all rounds, of the maximum edge load in the respective round. Unfortunately, investigating caching in networks in this model is much more complicated since shifting communication load on some of the edges from one round to another possibly has crucial effects. Therefore, we conclude that developing caching strategies in a cost model that incorporates some notion of time is a challenging task.

Further, our experimental results in [KMR+99] show that there is a further important cost factor in sending messages apart from the congestion. The sending of a message by a processor is called a *startup*. The overhead induced by the startup procedure, inclusive the overhead of the receiving processor, is called *startup cost*. Since we ignore the startup costs in our model it is an interesting question how to incorporate startup costs in an adequate way in our congestion based model.

We have shown that our general framework yields efficient caching strategies for several classes of networks. It can also be efficiently applied to other networks. All that is needed is a hierarchical network decomposition that possesses similar properties to the decompositions we have described. Basically, such a hierarchical network decomposition can be obtained by a recursively decomposition of the network. In order to achieve efficient caching strategies, in each decomposition step the following two rules have to be regarded.

- Decompose the network almost along the bisection cut into two subnetworks, such that communication between nodes of the same subnetwork is not as expensive as communication between nodes of different subnetworks.

- Decompose the network into two almost equally sized subnetworks, such that the associated decomposition tree is balanced.

Unfortunately, there could be a tradeoff between the two rules in arbitrary networks. Thus, it is an interesting question of whether or not there is a variation of the recursive decomposition described above that yields a caching strategy achieving close to optimal congestion on arbitrary networks?

Bibliography

[ABF93] B. Awerbuch, Y. Bartal, and A. Fiat. Competitive distributed file allocation. In *Proceedings of the 25th ACM Symposium on Theory of Computing (STOC)*, pages 164–173, 1993.

[ABF98] B. Awerbuch, Y. Bartal, and A. Fiat. Distributed paging for general networks. *Journal of Algorithms*, 28:67–104, 1998.

[ALMZ96a] M. Andrews, F. T. Leighton, P. T. Metaxas, and L. Zhang. Automatic methods for hiding latency in high bandwidth networks. In *Proceedings of the 28th ACM Symposium on Theory of Computing (STOC)*, pages 257–265, 1996.

[ALMZ96b] M. Andrews, F. T. Leighton, P. T. Metaxas, and L. Zhang. Improved methods for hiding latency in high bandwidth networks. In *Proceedings of the 8th ACM Symposium on Parallel Algorithms and Architectures (SPAA)*, pages 52–61, 1996.

[AP90] B. Awerbuch and D. Peleg. Sparse partitions. In *Proceedings of the 31th IEEE Symposium on Foundations of Computer Science (FOCS)*, pages 503–513, 1990.

[AR98] Y. Aumann and R. Rabani. An $O(\log k)$ approximate min-cut max-flow theorem and approximation algorithm. *SIAM Journal on Computing*, 27(1):291–301, 1998.

[BDBK+90] S. Ben-David, A. Borodin, R. M. Karp, G. Tardos, and A. Widgerson. On the power of randomization in online algorithms. In *Proceedings of the 22th ACM Symposium on Theory of Computing (STOC)*, pages 386–379, 1990.

[Bel66] L. A. Belady. A study of replacement algorithms. *IBM Systems Journal*, 5:78–101, 1966.

[BFR92] Y. Bartal, A. Fiat, and Y. Rabani. Competitive algorithms for distributed data management. In *Proceedings of the 24th ACM Symposium on Theory of Computing (STOC)*, pages 39–50, 1992.

[BL97] Y. Bartal and S. Leonardi. On-line routing in all-optical networks. In *Proceedings of the 24th International Colloquium on Automata, Languages and Programming (ICALP)*, pages 516–526, 1997.

[BS89] D. L. Black and D. D. Sleator. Competitive algorithms for replication and migration problems. Technical Report CMU-CS-89-201, Department of Computer Science, Carnegie-Mellon University, 1989.

[CK99] E. Cohen and H. Kaplan. Exploiting regularities in web traffic patterns for cache replacement. In *Proceedings of the 31th ACM Symposium on Theory of Computing (STOC)*, 1999.

[CKP+96] D. E. Culler, R. M Karp, D. Patterson, A. Sahay, K. E. Schauser, E. E. Santos, R. Subramonian, and T. von Eicken. LogP: A practical model of parallel computation. *Communications of the ACM*, 39(11):78–85, 1996.

[CMS96] R. J. Cole, B. M. Maggs, and R. K. Sitaraman. On the benefit of supporting virtual channels in wormhole routers. In *Proceedings of the 8th ACM Symposium on Parallel Algorithms and Architectures (SPAA)*, pages 131–141, 1996.

[CMSV96] R. Cypher, F. Meyer auf der Heide, C. Scheideler, and B. Vöcking. Universal algorithms for store-and-forward and wormhole routing. In *Proceedings of the 28th ACM Symposium on Theory of Computing (STOC)*, pages 356–365, 1996.

[FKL+91] A. Fiat, R. M. Karp., M. Luby, L. A. McGeoch, D. D. Sleator, and N. E. Young. Competitive paging algorithms. *Journal of Algorithms*, 12(2):685–699, 1991.

[FW74] P. A. Franaszek and T. J. Wagner. Some distribution-free aspects of paging performace. *Journal of the ACM*, 21:31–39, 1974.

[Goo94] A. J. van de Goor. *Computer Architecture and Design*. Addison-Wesley, 1994.

[HR90] T. Hagerup and C. Rüb. A guided tour of Chernoff bounds. *Information Processing Letters*, 33:305–308, 1990.

[IKP96] S. Irani, A. R. Karlin, and S. Phillips. Strongly competitive algorithms for paging with locality of reference. *SIAM Journal on Computing*, 25(3):477–497, 1996.

[Ira97] S. Irani. Page replacement with multi-size pages and applications to web caching. In *Proceedings of the 29th ACM Symposium on Theory of Computing (STOC)*, pages 701–710, 1997.

[KLL+97] D. Karger, E. Lehman, T. Leighton, M. Levine, D. Lewin, and R. Panigrahy. Consistent hashing and random trees: Distributed caching protocols for relieving hot spots on the World Wide Web. In *Proceedings of the 29th ACM Symposium on Theory of Computing (STOC)*, pages 654–655, 1997.

[KLM+97] R. R. Koch, F. T. Leighton, B. M. Maggs, S. B. Rao, A. L. Rosenberg, and E. J. Schwabe. Work-preserving emulations of fixed-connection networks. *Journal of the ACM*, 44(1):104–147, 1997.

[KMR+99] C. Krick, F. Meyer auf der Heide, H. Räcke, B. Vöcking, and M. Westermann. Data management in networks: Experimental evaluation of a provably good strategy. In *Proceedings of the 11th ACM Symposium on Parallel Algorithms and Architectures (SPAA)*, pages 165–174, 1999.

[KMRS88] A. R. Karlin, M. S. Manasse, L. Rudolph, and D. D. Sleator. Competitive snoopy caching. *Algorithmica*, 3(1):79–119, 1988.

[KPR93] P. Klein, S. A. Plotkin, and S. Rao. Excluded minors, network decomposition, and multicommodity flow. In *Proceedings of the 25th ACM Symposium on Theory of Computing (STOC)*, pages 682–690, 1993.

[KRVW98] C. Krick, H. Räcke, B. Vöcking, and M. Westermann. The DIVA (distributed variables) library. www.uni-paderborn.de/sfb376/a2/diva.html, Paderborn University, 1998.

[Lei92] F. T. Leighton. *Introduction to Parallel Algorithms and Architectures: Arrays • Trees • Hypercubes*. Morgan Kaufmann, San Mateo, CA, 1992.

[LMP+95] F. T. Leighton, F. Makedon, S. Plotkin, C. Stein, E. Tardos, and S. Tragoudas. Fast approximation algorithms for multicommodity flow problems. *Journal of Computer and System Science*, 50:228–243, 1995.

[LMRR94] F. T. Leighton, B. M. Maggs, A. G. Ranade, and S. B. Rao. Randomized routing and sorting on fixed–connection networks. *Journal of Algorithms*, 17:157–205, 1994.

[LRWY94] C. Lund, N. Reingold, J. Westbrook, and D. Yan. On-line distributed data management. In *Proceedings of the 2nd European Symposium on Algorithms (ESA)*, 1994.

[Mey83] F. Meyer auf der Heide. Efficiency of universal parallel computers. *Acta Informatica*, 19:269–296, 1983.

[Mey86] F. Meyer auf der Heide. Efficient simulations among several models of parallel computers. *SIAM Journal on Computing*, 15(1):106–119, 1986.

[MMVW97a] B. M. Maggs, F. Meyer auf der Heide, B. Vöcking, and M. Westermann. Exploiting locality for networks of limited bandwidth. In *Proceedings of the 38th IEEE Symposium on Foundations of Computer Science (FOCS)*, pages 284–293, 1997.

[MMVW97b] B. M. Maggs, F. Meyer auf der Heide, B. Vöcking, and M. Westermann. Exploiting locality for networks of limited bandwidth. Technical Report tr-rsfb-97-042, Paderborn University, 1997.

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

[MV95] F. Meyer auf der Heide and B. Vöcking. A packet routing protocol for arbitrary networks. In *Proceedings of the 12th Symposium on Theoretical Aspects of Computer Science (STACS)*, pages 291–302, 1995.

[MV99] F. Meyer auf der Heide and B. Vöcking. Shortest paths routing in arbitrary networks. *Journal of Algorithms*, 31(1):105–131, 1999.

[MVW99] F. Meyer auf der Heide, B. Vöcking, and M. Westermann. Provably good and practical strategies for non-uniform data management in networks. In *Proceedings of the 7th European Symposium on Algorithms (ESA)*, pages 89–100, 1999.

[MVW00] F. Meyer auf der Heide, B. Vöcking, and M. Westermann. Caching in networks. In *Proceedings of the 11th ACM-SIAM Symposium on Discrete Algorithms (SODA)*, pages 430–439, 2000.

[MW89] F. Meyer auf der Heide and R. Wanka. Time-optimal simulations of networks by universal parallel computers. In *Proceedings of the 6th Symposium on Theoretical Aspects of Computer Science (STACS)*, pages 120–131, 1989.

[OR97] R. Ostrovsky and Y. Rabani. Universal $O(\text{congestion} + \text{dilation} + \log^{1+\varepsilon} n)$ local control packet switching algorithms. In *Proceedings of the 29th ACM Symposium on Theory of Computing (STOC)*, pages 644–653, 1997.

[PR96] C. G. Plaxton and R. Rajaraman. Fast fault-tolerant concurrent access to shared objects. In *Proceedings of the 37th IEEE Symposium on Foundations of Computer Science (FOCS)*, pages 570–579, 1996.

[PRR97] C. G. Plaxton, R. Rajaraman, and A. W. Richa. Accessing nearby copies of replicated objects in a distributed environment. In *Proceedings of the 9th ACM Symposium on Parallel Algorithms and Architectures (SPAA)*, pages 311–320, 1997.

[RS94] P. Raghavan and M. Snir. Memory versus randomization in on-line algorithms. *IBM Journal of Research and Development*, 38(6):683–707, 1994.

[Spi77] J. R. Spirn. *Program Behavior: Models and Measurements*. Elsevier Computer Science Library. Elsevier, Amsterdam, 1977.

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

[SV96] C. Scheideler and B. Vöcking. Universal continuous routing strategies. In *Proceedings of the 8th ACM Symposium on Parallel Algorithms and Architectures (SPAA)*, pages 142–151, 1996.

[Val90] L. G. Valiant. A bridging model for parallel computation. *Communications of the ACM*, 33, 1990.

[Vöc98] B. Vöcking. *Static and Dynamic Data Management in Networks*. PhD thesis, Paderborn University, Germany, December 1998.