Static and Dynamic Data Management in Networks

Dissertation von Berthold Vöcking

Reviewer:

- Prof. Dr. Friedhelm Meyer auf der Heide, University of Paderborn, Germany
- Prof. Dr. Burkhard Monien, University of Paderborn, Germany
- Prof. Bruce Maggs, School of Computer Science, Carnegie Mellon University, Pittsburgh, USA

Acknowledgments

First of all, I would like to thank my advisor Professor Friedhelm Meyer auf der Heide for his great support and many helpful discussions. He showed me in which direction my research could go but always left me the freedom to do the routing and to choose my way. Besides, I would like to thank my "co-advisor" Professor Bruce Maggs who lead me to several spectacular attractions including an embedding of the AKS- into the Multibutterfly-network, a life-sized Diplodocus and many fancy restaurants in Pittsburgh.

I would like to thank Matthias Westermann and Klaus Schröder for a very productive collaboration and many valuable discussions, not only about distributed data management but also about maybe even more important aspects of life. Many thanks also to Christian Scheideler and Rolf Wanka with whom I shared an office during most of the last three and a half years. I enjoyed listening to many private lessons Christian and Rolf gave me, e.g., in probability theory, sorting algorithms, and history.

In particular, I have to thank the students who did most of the implementations for the experimental results presented in my thesis. Harald Räcke implemented the newly invented data management strategies in the DIVA (Distributed Variables) library; Christof Krick implemented a benchmark of applications for evaluating these strategies; and Mark Meierjohann did most of the simulations for the multimedia data server. All of them were very engaged and did a very good job.

This research was supported by the DFG-Sonderforschungsbereich 376 "Massive Parallelität".

Contents

Introduction

Computer networks and distributed data management have become part of our everyday live. A good example is the widespread use of the Internet application World Wide Web (WWW). The Internet consists of many independently acting computers that are connected by a network of buses and links. A user can read WWW pages on his own computer. These pages are stored in storage devices attached to some of the computers in the network. When a user issues a read request for a WWW page then the requested page is transferred through the network to the user's computer. Possibly, the accessed page is cached, that is, a copy of the page is kept in the user's local storage device or in another storage device lying on the path traversed by the data for several days or weeks. Because of the caching subsequent requests addressed to this page need not to be routed to the storage device holding the original copy but can be served locally.

Ideally, a distributed data management service is completely transparent to the user, this means, the user does not have to take care about how accesses to shared data objects are carried out. For example, a WWW user does not have to specify from which storage device a request should be served, and in particular, he does not need to determine the path through the network along which the data should be transmitted. However, the caching mechanism provided by the WWW is not completely transparent since cached copies will not be updated when the corresponding WWW page is modified. Hence, copies can become inconsistent. In general, the time a copy stays in a network cache is determined by the provider of the cache. Besides, the publisher of the page can specify an expiration time. A user wanting to read the original page rather than a cached copy that may be out of date has to send off an explicit "read through" command in order to get the actual data.

This thesis deals with fully transparent, distributed data management for large parallel and distributed systems in which the processors and the memory modules are connected by a relatively sparse network as, e.g., the Internet. The processors represent arbitrary computing units that can issue read and write requests for shared data objects, and the memory modules represent arbitrary storage devices in which the shared objects are stored. The Internet is only one of many examples that will be considered. Other examples are distributed file systems for Ethernet connected workstations, virtual shared memory for massively parallel computer systems, and a scalable multimedia data server architecture supporting an arbitrary number of users.

Efficient data management provides transparent and fast access to the shared data objects from any processor in the network. A data management strategy describes how copies of the data objects are distributed in the network. In particular, it has to answer the following questions:

- How many copies of a data object should be created?
- On which memory modules should these copies be placed?
- How should consistency among the copies be maintained?

Furthermore, the data management strategy has to describe how read and write accesses are served, e.g., which requests are satisfied by which copies.

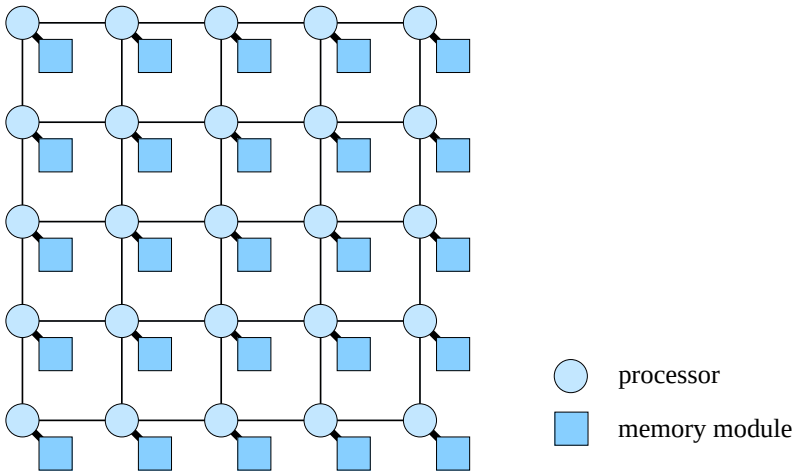


Figure 1: *Example of a direct network with mesh topology.*

We distinguish between static and dynamic data management strategies. Whereas static strategies are characterized by not changing the placement of the objects during the execution of an application, dynamic strategies possibly create, migrate, and invalidate copies at execution time. On the one hand, dynamic strategies are much more powerful than their static counterparts as, in principle, they are able to accommodate themselves to changes in the communication pattern of an application. On the other hand, new difficulties arise with the possibility to dynamically change the distribution of the copies, e.g., the problem of data tracking: a processor issuing a read or write request for a data object must be able to locate at least one copy of this object. This can become quite tricky if the copies move through the network dynamically.

All requests for reading or writing the shared data objects have to be served in a distributed fashion only using local information. Read or write accesses to remote copies of an object require the sending of messages between the accessing node and the nodes that hold the respective copies. Thus, in addition to the data placement, the routing paths from the accessing processors to the accessed memory modules need to be specified. In order to provide efficient access to shared data objects, the communication overhead caused by remote accesses should be as small as possible. However, simply reducing the total communication load, i.e., the sum, taken over all messages, the size of the message multiplied with the length of the routing path traversed by the message, 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 [48, 57, 60, 70] and wormhole routing [20, 23, 70] 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 data management strategies that work efficiently in theory and practice.

We will develop static and dynamic data management strategies for two different kinds of system scenarios, one of which assumes a direct interconnection network whereas the other assumes an indirect network.

Scenario 1 – Exploiting locality in networks of limited bandwidth. We investigate data management in computer systems in which several computers or processors are connected by a relatively sparse network. The network is direct, i.e., each node of the network corresponds to a processor with its own memory module of sufficiently large capacity and bandwidth such that the communication links between different processors are the major bottleneck of the system. Figure 1 gives an example of a direct network.

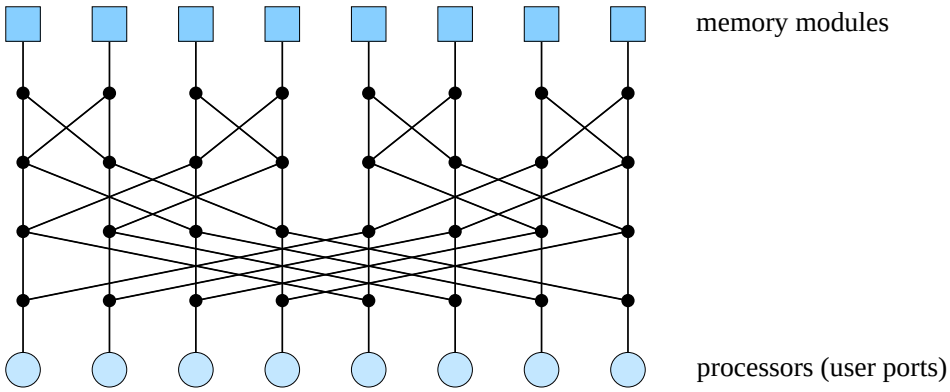


Figure 2: *Example of an indirect network with butterfly topology.*

Direct networks are the typical interconnection in large parallel and distributed systems such as massively parallel processors (MPPs) and networks of workstations (NOWs). We will consider the problem of placing and accessing a set of shared objects that are requested by the nodes in the network. The nodes issue read and write requests that should be served using a distributed data management strategy. The shared objects are, e.g., global variables in a parallel program, pages or cache lines in a virtual shared memory system, or pages in the WWW.

The most challenging task is to develop data management strategies that are able to exploit the locality in the access pattern of a given application as well as the locality in the topology of the underlying network. We will present new static and dynamic data management strategies for tree-connected networks, for meshes of arbitrary dimension, and for Internet-like clustered networks. We prove optimal or close-to-optimal bounds on the congestion produced by each of these strategies.

Probably the most interesting results are the dynamic data management strategies for meshes and for Internet-like clustered networks. Both of the strategies follow the same approach, which we call the "access tree strategy": Each data object possesses its own access tree, which is a data structure corresponding to a tree graph. For each object, the access tree is embedded in a random but locality-preserving fashion into the network. A simple dynamic caching strategy for tree networks is performed on each of these access trees. This approach simultaneously solves the dynamic data placement problem, the on-line problem of determining the routing paths, and the difficult data tracking problem. Apart from inevitable read request messages, the access tree strategy does not require the sending of any additional control or bookkeeping messages. We will prove the efficiency of the access tree strategy in theoretical analyses as well as in practical experiments.

Most of the results of Scenario 1 have been previously published as joint work in [52]. A detailed summary of our results is given in Section 1.3.

Scenario 2 – Contention resolution on a scalable multimedia data server. This scenario deals with a system consisting of a set of processors connected to a set of memory modules by an indirect networks. The bandwidth provided by an indirect network usually is much larger than the one of a direct network. Hence, the memory modules often become the major bottleneck. Figure 2 gives an example of the topology considered in this scenario.

Indirect networks are typical, e.g., for multimedia data servers in which the data is stored on several relatively slow storage devices such as hard disks. We focus on data management strategies for this concrete example. We assume that the access patterns of multimedia applications cannot be predicted in

advance as these applications are becoming more and more interactive. However, the real-time constraints of multimedia applications require that each individual request is guaranteed to be served very quickly.

In order to guarantee a short response time for every request the load of serving the requests has to be evenly distributed among the memory modules and the links of the network. Redundancy can help to solve this load balancing problem because it yields alternative resources, i.e., memory modules and network links, for serving individual requests. In this way, the load can be balanced adaptively to the access pattern. When too many user requests aim at the same resource, a contention resolution mechanism decides which of the requests gets the respective resource.

We will present a new proposal for a scalable data server architecture that has a butterfly-like interconnection network and places each copy on two memory modules. For this architecture, we will introduce and analyze contention resolution protocols that are able to handle batches of parallel requests as well as sequences of requests that arrive one by one over a possibly infinite period of time. These protocols achieve nearly optimal congestion at the links in the network and simultaneously minimize the maximum load at the memory modules. The small bounds on the network and module congestion guarantee that each individual request is served almost immediately.

A more detailed summary of our results is given in Section 2.3. Some of these results have been previously published as joint work in [18].

Scenario 1:

Exploiting Locality in Computer Networks of Limited Bandwidth

Large parallel and distributed systems, such as massively parallel processors (MPPs) and networks of workstations (NOWs), consist of a set of nodes each having its own local memory module. These nodes 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. In this scenario, we consider the problem of placing and accessing shared objects that are read and written by the nodes in the network. The objects are, e.g., global variables in a parallel program, pages or cache lines in a virtual shared memory system, or pages in the WWW.

The performance of MPPs and NOWs depends on a number of parameters, including processor speed, memory capacity, network topology, bandwidths, and latencies. Usually, the buses or links 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., [20, 23]), redundant computation (see, e.g., [1, 2, 41, 54, 55, 58]) or slackness (see, e.g., [80]), 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., [31]). *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 last reference. This is caused, e.g., by the traversal of data structures such as arrays. A further kind of locality is specific to parallel and distributed systems: 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.

We investigate data management strategies that exploit temporal and topological locality in order to use the network as efficiently as possible, that is, the strategies should serve all of the accesses to the data objects while using as little of the network resources as possible. A data management strategy consists of a placement and an access strategy. The placement strategy specifies the distribution of the data objects among the nodes, possibly with redundancy. The access strategy specifies how read and write accesses

are served, that is, it specifies which write access updates which of the object copies and which of the read accesses gets the requested data from which copy. Further, it specifies the routing paths from the accessing nodes that hold the copies that are to be accessed.

1.1 Formal description of the problem

The network is modeled by a (hyper)graph $G = (V, E)$ with node set V and edge (or hyperedge) set E such that the edges (or hyperedges) represent the links (or buses). The bandwidths of the links (or buses) are described by a function $b : E \longrightarrow \mathbb{N}$. The set of shared objects is denoted by X.

Our aim is to develop strategies for placing and accessing shared objects such that the locality inherent in an application is exploited in order to minimize the congestion that is defined as follows. For a given application, let the *(absolute) load* of an edge (or hyperedge) be the amount of data that is transferred by the corresponding link (or bus) during the application's execution. Let the *relative load* of an edge (or hyperedge) be its load divided by its bandwidth. Finally, define the *congestion* to be the maximum over the relative loads of all edges (or hyperedges) in the network.

We use two models to formalize information about the pattern of accesses to the shared objects of an application.

1.1.1 Description of the static model

In the static placement model, we are given a distributed application in which the processors communicate by reading and writing the shared objects. The model has two important features, which are:

1. The rates of the accesses from the processors to the shared objects are known in advance.
2. The locations of the copies of each object and the routing paths for the accesses must be determined in advance and cannot be changed during the execution of the application.

The given rates of accesses describe the total numbers of read and write accesses issued during the execution of the application or, alternatively, frequencies of read and write accesses measured in units of accesses/time steps. In general, the rates are specified by two functions $h_r : V \times X \longrightarrow \mathbb{R}$ and $h_w : V \times X \longrightarrow \mathbb{R}$ that describe the number of read and write accesses, respectively, from the nodes to the objects in some given period of time. We are interested in efficient algorithms that calculate a static placement of the objects and specify the routing paths such that the congestion is minimized.

The placement of the objects is allowed to be redundant, i.e., each object can be placed on several nodes. In order to maintain consistency among the different copies of an object, we assume that a write access to an object has to update all of its copies, whereas a read access to an object can be satisfied by one of its copies. Updates are allowed to be done by multicasts, i.e., the node that wants to update a set of copies sends out one message that is transmitted along the branches of a tree to all of the object's copies. The goal is to place the copies and to determine the routing paths (or trees in case of multicasts) in such a way that the congestion is minimized. Note that we do not demand all accesses from a particular node to a particular object to use the same path. Our algorithms, however, meet this condition whereas the lower bounds we give hold also for algorithms that do not fulfill the condition.

For simplicity, we initially assume that the load due to a read or write access on each edge (or hyperedge) belonging to the respective routing path or mulitcast tree is one. In practice, however, the load induced by read and write accesses may be different and dependent on the size of the respective data object or the size of the accessed part of the object. For example, read accesses may be realized by

first sending a request message to a node holding a copy, which then sends back a message including the requested data, whereas write requests only send an update or invalidation message to all nodes holding a copy. In Section 1.7.1, we introduce weighted variants of the functions h_r and h_w. These weights represent different costs for read and write accesses directed to data objects of arbitrary size. We will see that all results that we have obtained in the uniform cost model hold also in the non-uniform model.

The model described above is slightly restrictive because it does not include strategies that allow only a fraction of the copies to be updated in case of a write, which is implemented, e.g., in strategies using the majority trick introduced in [78]. However, all strategies using such techniques add time stamps to the copies. This requires that there is some definition of uniform time among different nodes. Since it is difficult to realize this in an asynchronous setting, we initially restrict ourselves to strategies that update all copies in case of a write. At the end of this chapter, in Section 1.7.2, a more general model will be considered, including, e.g., strategies using the majority trick.

1.1.2 Description of the dynamic model

In the dynamic model, there is no knowledge about the access pattern of an application in advance. 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 on the nodes of the network. These accesses must be served by a dynamic data management strategy. If this strategy is randomized, then the adversary is assumed to be oblivious, i.e., the accesses to the shared objects are not allowed to depend on the outcome of the random experiments of the dynamic data management strategy.

We restrict the class of allowed applications specified by the adversary to be data-race free programs, i.e., a write access to an object is not allowed to overlap with other accesses to the same object, and there is some order among the accesses to the same object such that, for each read and write access, there is a unique least recent write. Note that this still allows arbitrary concurrent accesses to different objects and concurrent read accesses to the same object.

A dynamic data management strategy is called *consistent* if it ensures that a read request directed to an object always returns the value of the most recent write access to the same object. Write accesses are assumed to be object alterations rather than overwrites. This has two important consequences: The first is that none of the write accesses can be ignored, not even in the case of immediately consecutive write accesses. The second is, that the writing node cannot build a new copy of the respective object from scratch, that is, the value to be written must be "merged" with an actual copy of the object.

We are interested in developing on-line distributed data management strategies that minimize the congestion. These strategies are allowed to migrate, create, and invalidate copies of an object during execution time. Migration means that a copy is moved along a path through the network; creation means that a copy is duplicated, and the new copy is migrated; and invalidation means that a copy is deleted. Initially, one copy of each object is placed somewhere in the network. Arbitrary communication between neighboring nodes in the network is allowed. However, each communication increases the load on the involved edge (or hyperedge). For simplicity, we assume that each object fits into one routing packet such that each migration of a copy along an edge (or hyperedge) increases the load of this edge (or hyperedge) by one. Also request, update, and invalidation messages are assumed to have size one. Non-uniform models assuming different cost for messages of different object sizes are discussed at the end of this chapter, in Section 1.7.1.

We use the competitive ratio as a measure of the efficiency of a dynamic data management strategy. For a given application $\mathcal{A}$, let $C_{\text{opt}}(\mathcal{A})$ denote the minimum congestion expended by an optimal dynamic strategy having full knowledge of the parallel program specified by the adversary, including knowledge of all requests even before they are issued. A dynamic strategy is said to be *k-competitive* if it produces

congestion at most $k \cdot C_{\text{opt}}(\mathcal{A})$, for any application $\mathcal{A}$. A randomized strategy has to satisfy this bound *with high probability (w.h.p.)*, that is, the probability that the congestion exceeds $\alpha \cdot k \cdot C_{\text{opt}}(\mathcal{A})$ is at most $n^{-\alpha}$, for every $\alpha \geq 1$, where n denotes the size of the network. Note that this bound implies that the expected congestion is in $O(k \cdot C_{\text{opt}}(\mathcal{A}))$.

Of course, the optimal strategy has an advantage over an on-line strategy because it has full knowledge of the dynamic access pattern. Without the restriction to data-race free programs, the optimal strategy could, e.g., defer all write accesses to an object x to the end of the execution and keep a valid copy of x at any node that reads x at some time. This "bad trick" would give load 1 on any edge (or hyperedge) for serving all read accesses to x. The example illustrates that the restriction to the class of data-race free programs is necessary in order to allow a more fair comparison of on-line against off-line strategies.

1.2 Previous and related work

The problem of distributing and accessing shared objects in networks has been considered in several previous studies. We give a brief overview among the most important work in this area.

The first attempts at solving the data management problem concentrated on developing heuristics for the static file assignment problem. This problem deals with assigning files or copies of files to the nodes of a network such that a cost function, which, e.g., models the total communication load or the storage cost, is minimized. Dowdy and Foster [26] give a survey about several different models for the file allocation problem. All models are described by mixed integer programs using different cost functions and constraints. Analogously to our static model, all models assume that write accesses have to update all copies of an object, whereas read accesses can be satisfied by any one copy. However, none of the described models assumes that copies are updated within a multicast tree.

Milo and Wolfson investigate static placement in a model that permits multicast trees for copy updates [85]. However, the multicast trees are allowed to branch out only on nodes holding a copy of the respective object, as opposed to our model which allows branchings at every node. Milo and Wolfson introduce placement algorithms that minimize the total communication load for rings, trees, and completely connected networks. All of their algorithms manage each data object independently from other objects. Let n denote the size of the network. Then placing the copies of an object on a ring takes time $O(n^5)$ whereas placing the copies of an object on a tree or on a completely connected network only takes time $O(n)$. It is easy to check that the placements computed by the algorithms for rings and trees also are optimal (with respect to the total load) if the multicast trees are allowed to branch out at every node. However, minimizing the total load and minimizing the congestion on a ring are obviously incompatible objectives for some applications. The question of whether or not a placement for trees exists that simultaneously satisfies both of these objectives is addressed by our work.

Much theoretical work deals with the emulation of PRAMs (Parallel Random Access Machines) on networks. All these emulations require some kind of data management as the PRAM model assumes shared memory. Each word of the shared memory can be viewed as a shared object in our model. The execution of a PRAM program, which accesses words of memory, corresponds to the execution of a network computation in our model. Serving a read or write request to a memory word in the PRAM corresponds to serving a read or write access to a shared object. In [36], Karlin and Upfal present a probabilistic emulation on an n-node butterfly. Their algorithm emulates one step of an n-processor Exclusive-Read-Exclusive-Write PRAM in time $\Theta(\log n)$, w.h.p. Ranade [66] showed how combining can be used to improve this result, i.e., he showed that a step of a Concurrent-Read-Concurrent-Write PRAM can be emulated in the same time. Both strategies use random hash functions to distribute the memory cells uniformly among the nodes. This scheme can be adapted to other networks, too. For instance, this

yields n-processor PRAM emulations for the $\sqrt{n} \times \sqrt{n}$ mesh with slowdown $\Theta(\sqrt{n})$. Although this bound cannot be improved for general PRAM emulations, it is not satisfactory for applications including locality.

Competitive analysis of dynamic data management algorithms begins with Karlin et al. who analyze algorithms for snoopy caching on buses [35]. Bartal et al. [10] introduce a dynamic file allocation problem in a competitive model similar to our dynamic model. The major differences to our model are that they assume different costs for migrating a shared data object and accessing it, and that the objective is to minimize the total communication load rather than the congestion.

Awerbuch et al. investigate the dynamic file allocation problem in arbitrary networks. Their work is based on the problem definition of Bartal et al. and, hence, aims to minimize the total communication load. In [5] they present a centralized algorithm that achieves optimal competitive ratio $O(\log n)$ and a distributed algorithm that achieves competitive ratio $O\left((\log n)^4\right)$ for the dynamic file allocation problem on an arbitrary network of size n. Both algorithms are deterministic. Furthermore, the distributed algorithm is adapted to systems with limited memory capacities [6].

We give a brief description of the very intriguing distributed algorithm of Awerbuch et al. because this will illustrate the influence of the total load model and the congestion model on the design of an efficient data management strategy. The algorithm uses a hierarchical network decomposition, introduced in [7], 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 is 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 load is minimized up to a polylogarithmic factor, which gives a good competitive factor in the total load model. However, the cluster leaders can become bottlenecks. In particular, the edges incident to the leader of the top level clusters can become very congested. As a consequence, the competitive ratio in our congestion model becomes bad.

Data management algorithms for trees are of special interest since many networks have a tree-like topology, e.g., Ethernet-connected NOWs. Furthermore, algorithms for trees can be used as a subroutine in data management strategies for other networks. Therefore, much research deals with data management on trees. For example, Bartal et al. [10] describe a randomized strategy for the dynamic file allocation problem on trees that achieves a competitive ratio of 3 with respect to the expected total load and a deterministic strategy that achieves a competitive ratio of 9 with respect to the total load. The congestion is not considered, but it is easy to check that both strategies have competitive ratio $\omega(1)$ with respect to the congestion.

Lund et al. [50] describe a 3-competitive deterministic but centralized strategy for the dynamic file allocation problem on trees. The advantage, of their algorithm is that it minimizes not only the total communication load but also the load on any edge and, therefore, also the congestion. Unfortunately, the algorithm makes use of global knowledge about a work function that is influenced by any 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 algorithm is inherently centralized, and, hence, yields no suitable solution in our distributed setting.

A lower bound for dynamic data management on trees is shown by Black and Sleator [16]. They give a lower bound of 3 on the competitive ratio for deterministic data migration algorithms for two processors connected by a single edge, where *data migration* means that only one node may hold a copy at any given time. If requests are limited to writes only, the dynamic data management problem collapses to the data migration problem, and, therefore, this lower bound holds also for our dynamic data management model. Bartal et al. [10] extend the lower bound to the class of randomized algorithms.

Several recent papers deal with the distribution of pages in the WWW. Plaxton and Rajaraman [64] 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. [34] 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 accesses to the data objects.

A difficult problem that has to be solved by any dynamic data management strategy is the *data tracking*, 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. describe a data tracking mechanism for arbitrary networks [10] 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 processor 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. 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 is introduced in [65]. However, this mechanism works only for networks that fulfill a low-expansion property, which is satisfied, e.g., by meshes of constant dimension.

1.3 Summary of new results

We introduce new static and dynamic data management strategies for tree-connected networks, for meshes of arbitrary dimension and side length, and for Internet-like clustered networks. The strategies for trees are deterministic, the others are randomized. All strategies aim to minimize the congestion by exploiting locality. To our knowledge this is the first analytic treatment of this problem.

We start by investigating static data management on tree-connected networks modelled 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 an efficient algorithm that calculates this placement. (For networks with point-to-point connections this algorithm is more or less equivalent to the algorithm of Wolfson and Milo [85], for which they were only able to prove optimality in the total load model.) The sequential running time of the algorithm 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 distributed dynamic caching strategy for tree-connected networks that achieves constant competitive rate with respect to the congestion. The dynamic strategy for trees is surprisingly simple, and we prove a competitive ratio of 3, which is the best possible ratio because of the lower lower bound shown by Black and Sleator [16].

As our results for static and dynamic data management on trees hold for trees with arbitrary link or bus connections having arbitrary bandwidths, our algorithms 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 static and dynamic data management strategies for trees will be presented in Section 1.4.

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. 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 distributed 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 simulated. Consider an arbitrary mesh of dimension d with n nodes. Here the access tree strategy yields an efficient algorithm 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 $O(d \cdot \log n)$. 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. The theoretical results for data management on meshes will be presented in Section 1.5.

Furthermore, we show how the access tree strategy can be adapted to Internet-like clustered NOWs. 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 dynamic data management service 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. Our characterization of the clustered networks and the corresponding results will be explained in more detail in Section 1.6.

In Section 1.7, we will show that most of the results for static data management hold even in more general models including, e.g., strategies sending access messages of different size or strategies using the majority trick. Furthermore, we show that the dynamic access tree strategy for meshes and clustered networks can be extended to handle non-uniform object sizes.

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. 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. Besides we compare the access tree strategy with a standard caching scheme in which a fixed home processor is assigned to each data object that keeps track of the object's copies. The access tree strategy clearly outperforms the fixed home strategy in all experiments. The practical studies of the dynamic access tree strategy for meshes will be presented in Section 1.8.

1.4 Data management on tree-connected networks

In this section, we describe static and dynamic data management strategies for tree-connected networks. The advantage of these networks is that there is only one simple path between any pair of nodes. Thus, the placement strategy automatically defines the routing paths from the accessing processor 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.

In the following, the network is modeled by a (hyper)graph $T = (V,E)$. The edges (or hyperedges) are allowed to have arbitrary bandwidth. Let $\mathrm{diam}(T)$ denote the diameter of T, $\mathrm{degree}(T)$ its maximum node degree, i.e., the maximum number of edges (or hyperedges) incident on the same node, and $\mathrm{rank}(T)$ the maximum hyperedge rank, i.e., the maximum number of nodes that are connected by the same hyperedge.

1.4.1 Static placement on trees

1.4.1.1 Trees with point-to-point connections

For simplicity, we initially assume that the tree T is a "normal" graph, i.e., $\mathrm{rank}(T) = 2$. We have to place shared data objects from a set X with respect to the functions $h_r : V \times X \longrightarrow \mathbb{R}$ and $h_w : V \times X \longrightarrow \mathbb{R}$ that describe the number of read and write accesses, respectively, from the nodes to the objects. Each object can be placed statically on one or more nodes. A read access to an object can be satisfied by any one of its copies, and a write access must update all copies.

Our static placement strategy, which we call *nibble strategy*, places each data object $x \in X$ independently from the other objects. We use the following notations and definitions concerning the access rates to a fixed object x. For a node $v \in V$, define $r(v) = h_r(v,x)$, $w(v) = h_w(v,x)$, and $h(v) = r(v) + w(v)$. Thus, $h(v)$ represents the total number of accesses to object x by node v. $h(v)$ is also denoted the *weight* of node v. For any subtree $T' = (V',E')$, i.e., a connected subgraph of T, we define $r(T') = \sum_{v \in V'} r(v)$, $w(T') = \sum_{v \in V'} w(v)$, and $h(T') = r(T') + w(T')$.

All copies of x are placed on some nodes that form a connected component including the *center of gravity* $g(T)$, which is defined as follows. Consider the set of nodes v such that the removal of v partitions T into subtrees, each of which contains at most 1/2 of the total weight. It is easy to check that this set is not empty. We choose an arbitrary node from this set to be the center of gravity $g(T)$, e.g., the one with the smallest index. Note that the gravity center depends on the access rates to object x such that the gravity center for another object possibly is a different node.

In the following, $g(T)$ is assumed to be the root of the tree T, which defines the parent and the children of each node. The subtree $T(v)$ rooted at $v \in V$ is defined to be the maximal subtree including v but not the parent of v. After we have fixed this notation, the rest of the description of the nibble strategy is very simple.

The nibble placement: A node v gets a copy of x if and only if $v = g(T)$ or $h(T(v)) > w(T)$.

The nibble placement can be calculated in time $O(|V|)$ for each object. If the function h_r and h_w are represented by an array including an entry for each node, then this bound corresponds to the input size. We only need to compute the total number of write accesses and, for each node v, the sum of the read and write accesses issued in the *subtrees incident on v*, i.e., the subtrees into which the tree is partitioned if v is removed from the tree. This can be done by a depth first search algorithm taking time $O(|E|) = O(|V|)$. Moreover, the placement can be calculated efficiently by the processors of the tree network in a distributed fashion. Here the total number of write accesses and the weight of all subtrees incident on the nodes can

be computed in $2 \cdot \mathrm{diam}(T)$ rounds each of which takes time $O(\mathrm{degree}(T))$. The computation for several objects can be pipelined, which gives time $O((|X| + \mathrm{diam}(T)) \cdot \mathrm{degree}(T))$ for the placement of all objects in X.

We point out that our nibble strategy is more or less equivalent to the algorithm of Wolfson and Milo [85]. However, we will show a stronger result than they do, that is, we prove that the nibble placement minimizes not only the total communication load but the load on all edges simultaneously, and hence, also the congestion, regardless of the bandwidths of the edges.

Theorem 1.4.1 *The nibble placement achieves minimum load on each edge of the tree.*

Proof. The nibble strategy describes a subset of nodes that hold a copy of x. This subset defines a mapping of copies to the nodes. We show that an arbitrary mapping can be transformed into the mapping of the nibble strategy without increasing the load on any edge. We describe the transformation in two steps. The following lemma gives the result of the first transformation.

Lemma 1.4.2 *Any mapping S for x can be transformed into a mapping S^* such that the nodes that hold a copy build a connected component including $g(T)$ without increasing the load on any edge.*

Proof. Consider the minimum Steiner tree connecting all copies of S. Let S' be the mapping in which every node of this Steiner tree holds a copy. Then transforming S to S' does not increase the load on any edge because each write access has to to update all nodes of S, which means that each write access crosses each edge in the Steiner tree. If S' includes $g(T)$ then we are finished. Otherwise, we have to transform S' into S^* as follows.

The nodes holding the copies of S' form a connected component not including $g(T)$. Thus, one of the subtrees $T(v_1), T(v_2), \ldots$ includes all of the copies, where $v_1, v_2, \ldots$ denote the nodes incident on $g(T)$. Let $T_A = (V_A, E_A)$ denote this subtree, and let $T_B = (V_B, E_B)$ denote the remaining subtree, i.e., the subtree of T induced by the set of nodes in $T \setminus T_A$. Further, let $T_C = (V_C, E_C)$ denote the subtree of T_A induced by the set of the nodes holding a copy of S', let u denote the node in V_C that is closest to $g(T)$, and let P denote the shortest path connecting u with $g(T)$. Figure 1.1 illustrates the situation.

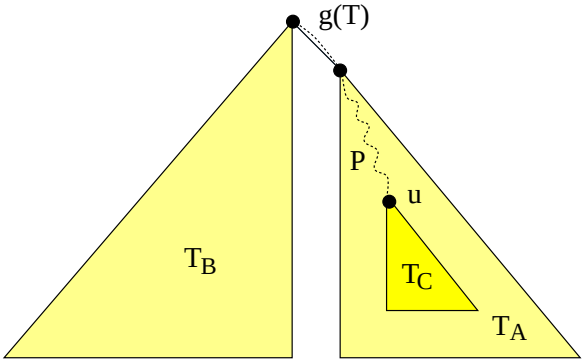


Figure 1.1: *The subtrees T_A, T_B, and T_C, the nodes $g(T)$ and u, and the path P.*

Suppose $r(T_B) > w(T_A)$. Placing additional copies onto all the nodes of the path P influences only the load on the edges of this path. On the one hand, each write access issued in T_A has to traverse this path now. On the other hand, the read accesses issued in T_B do not have to traverse this path anymore. Thus, the load on every edge on the path is increased by at most $w(T_A) - r(T_B) < 0$, which means that adding copies to all nodes on P yields the strategy we are looking for, S^*.

Now assume $r(T_B) \leq w(T_A)$. Because $g(T)$ is the center of gravity, $h(T_A) \leq h(T)/2$, which induces $h(T_B) = h(T) - h(T_A) \geq h(T)/2 \geq h(T_A)$. Combining both equations yields

$$w(T_B) = h(T_B) - r(T_B) \geq h(T_A) - w(T_A) = r(T_A) \quad .$$

Let S'' denote the mapping having only one copy that is placed on u. Then transforming S' to S'' only influences the load on edges in E_C. The load on these edges is increased by at most $r(T_A)$ and decreased by at least $w(T_B)$. This gives an increase of at most $r(T_A) - w(T_B) \leq 0$ on these edges. Therefore, changing S' into S'' does not increase the load on any edge.

Finally, we change S'' into S^* by moving the copy from u to $g(T)$. This effects only the load on the edges of P, which is increased by at most $h(T_A)$ due to accesses issued in T_A and decreased by $h(T_B) \geq h(T_A)$ due to accesses issued in T_B. As a consequence, S can be transformed via S' and S'' into S^* without increasing the load on any edge. ■

Now we describe the full transformation. First, we transform S into a mapping S^* such that the set of nodes U holding a copy build a connected component including the center of gravity $g(T)$. By Lemma 1.4.2 this transformation can be done without increasing the load on any edge. The nibble strategy places a copy on a node $v \in V$ if and only if $h(T(v)) > w(T)$ or $v = g(T)$. We show that S^* can be transformed into the mapping computed by the nibble strategy without increasing the load on any edge, that is, we show that

- a copy can be added to all nodes $v \in V \setminus U$ with $h(T(v)) > w(T)$, and
- a copy can be removed from all nodes $v \in U \setminus \{g(T)\}$ with $h(T(v)) \leq w(T)$.

without increasing the load on any edge.

Consider $v \in V \setminus U$ with $h(T(v)) > w(T)$. All nodes v' on the shortest path from v to U (including v) satisfy $h(T(v')) \geq h(T(v)) > w(T)$ because U includes $g(T)$, and at most half the weight of the tree is in that subtree incident on $g(T)$ which includes v. Adding a copy to all of these nodes influences only the load on the edges on this path. In particular, the load on these edges is increased by at most $w(T) - w(T(v))$ because of write accesses issued by nodes not in $T(v)$ that have to traverse these edges after adding the copies. However, the load on these edges is decreased by at least $r(T(v))$ because of read accesses issued by nodes in $T(v)$ which can be satisfied now by the copy on v. Combining both effects, the load is increased by at most $w(T) - w(T(v)) - r(T(v)) = w(T) - h(T(v)) < 0$. This means adding a copy to all nodes $v \in V \setminus U$ with $h(T(v)) > w(T)$ does not increase the load on any edge.

Now consider $v \in U \setminus \{g(T)\}$ with $h(T(v)) \leq w(T)$. Let T' denote the subtree of $T(v)$ induced by the nodes holding a copy. Then each node v' in the subtree T' satisfies $h(T(v')) \leq h(T(v)) \leq w(T)$. Removing all copies from all nodes in T' influences only the load on edges in T'. On the one hand, the load on these edges is increased by at most $r(T(v))$ because of reads issued by nodes in $T(v)$. On the other hand, the load is decreased by at least $w(T) - w(T(v))$ because of writes issued by nodes not in $T(v)$. Thus, the load on these edges is increased by at most $r(T(v)) - (w(T) - w(T(v))) = h(T(v)) - w(T) \leq 0$. Hence, removing the copies from all nodes $v \in U \setminus \{g(T)\}$ with $h(T(v)) \leq w(T)$ does not increase the load on any edge, which completes the proof of Theorem 1.4.1. ■

1.4.1.2 Trees with arbitrary bus connections

The nibble strategy can be adapted to tree-connected networks with arbitrary bus connections. In this case $T = (V, E)$ is a hypergraph rather then a "normal" graph. Each hyperedge represents a bus that connects all nodes incident to the hyperedge. The buses are allowed to have arbitrary bandwidths. However, in the

following we can neglect the bandwidths as we will see that the nibble strategy minimizes the load on any bus regardless of the bandwidths.

The strategy is adapted in the following way. First, the hypergraph T is transformed into a "normal" tree $T' = (V', E')$, i.e., each edge in T' is incident to at most 2 edges. Then the nibble strategy is applied to T'. Finally, the calculated optimal placement on T is transformed into an optimal placement on the hypergraph T.

The transformation from T to T' is done by local substitutions. Each hyperedge is simulated by a star, i.e., hyperedge e_H incident on the nodes $v_1, \ldots, v_k$ is replaced by a node $v(e_H)$ that is connected by k edges to the nodes $v_1, \ldots, v_k$. Further, $r(v(e_H))$ and $w(v(e_H))$ are set to 0.

After the nibble strategy is applied to T' the calculated placement must be transformed into a placement for the hypergraph T. This means that we have to remove the copies from the nodes in T' representing the hyperedges in T. These copies are replaced according to the following rule: Consider a hyperedge $e_H = \{v_1, \ldots, v_k\}$ from T represented by the node $v(e_H)$ in T'. Let $T_1, \ldots, T_k$ denote the subtrees resulting from the deletion of $v(e_H)$, and let m be the index satisfying $h(T_m) = \max\{h(T_i) \mid 1 \leq i \leq k\}$. If $h(T_m) > r(T)$ then the copy on $v(e_H)$ is removed and a copy is added on v_m (unless v_m does not hold a copy already). Otherwise, the copy is removed and a copy is added to each of the nodes $v_1, \ldots, v_k$ (unless the respective node does not hold a copy already).

The above strategy can be calculated sequentially in time $O(|X| \cdot |V|)$ for all objects. Further, the placement can be calculated in a distributed fashion by the nodes of T in time $O((|X| + \mathrm{diam}(T)) \cdot (\mathrm{degree}(T) + \mathrm{rank}(T)))$. The following theorem shows that the strategy yields minimal congestion regardless of the bandwidths of the buses.

Theorem 1.4.3 *The nibble placement achieves minimum load on each hyperedge of the tree.*

Proof. The transformation from T to T' consists of several local transformations, one for each hyperedge. Each of these transformations does not affect the minimum load on any edge not involved in the transformation. Also the transformation from the optimal placement for T' to a valid placement for T consists only of local transformations on the hyperedges that does not change the load on any edge not involved in the transformation. As a consequence, we can restrict our investigations to the effects of each of these local transformations independently. We have to show for each hyperedge e_H that the load on e_H is minimal after the local transformation under the assumption that the load on every edge of the respective star in T' is minimal.

Suppose that $v(e_H)$ gets no copy by applying the nibble strategy in T'. Then the final transformation does not add a copy to any of the nodes $v_1, \ldots, v_k$. In this case the load on the hyperedge is equivalent to the load of the edge leading to the connected component of the nodes holding the copies. Hence, the load of the hyperedge is not larger than the maximum load over all edges in the star. Because a star can simulate a hyperedge such that the maximum load over all edges in the star is not larger than the load of the hyperedge, it follows that the load of e_H is minimal.

Now suppose that $v(e_H)$ has got a copy and $h(T_m) > r(T)$, where m is defined as described above. Then we show that the nibble strategy on T' places no copy in subtree T_j, with $1 \leq j \leq k$ and $j \neq m$. If the nibble strategy places a copy in T_j then the edge $\{v(e_H), v_j\}$ has load

$$w(T_j) + w(T_m) = w(T_j) + h(T_m) - r(T_m) > w(T_j) + r(T) - r(T_m) \geq h(T_j)$$

because all writes issued in T_m and in T_j have to traverse this edge. Otherwise, the same edge has load at most $h(T_j)$, because only the accesses from T_j have to pass this edge. Consequently, the nibble strategy does not place a copy in T_j because it minimizes the load on any edge.

In the transformation to a valid placement for the hypergraph T we place a copy on v_m. Since there is no copy outside of T_m after the transformation, none of the accesses issued in T_m have to cross e_H. This yields load $h(T) - h(T_m) = r(T) + w(T) - h(T_m) < w(T)$ on the hyperedge. This is the optimal solution for the hyperedge, because placing a copy on more than one of the subtrees yields load $w(T)$, and placing only copies in one of the subtrees T_j with $j \neq m$ yields load at least $h(T) - h(T_j) \geq h(T) - h(T_m)$ on the hyperedge.

Finally, suppose that $v(e_H)$ has got a copy in T' and that $h(T_m) \leq r(T)$. Then placing a copy onto all nodes as done in the transformation yields load $w(T)$ on the hyperedge. Placing a copy onto at least two of the nodes also yields at least load $w(T)$. Further, placing a copy onto one of the nodes $v_1, \ldots, v_k$ or only copies in one of the subtrees yields load at least

$$h(T) - h(T_j) \geq h(T) - h(T_m) = r(T) + w(T) - h(T_m) \geq w(T) \quad .$$

Hence, the nibble strategy yields minimum load on e_H. ■

1.4.2 Dynamic data management on trees

In the dynamic setting, all placement decisions have to be made on-line, i.e., we assume that an adversary initiates the read and write accesses arbitrarily at execution time. A dynamic strategy is allowed to migrate, to create, and to invalidate, copies during the execution of the application. Initially, one copy of each object is placed somewhere in the network. Arbitrary communication between neighboring nodes in the network is allowed. However, sending a message along an edge (or hyperedge) e, e.g., for requesting a data object or moving a copy, increases the load on e by one.

We present a dynamic caching strategy that is suitable for tree-connected networks with arbitrary bus connections and non-uniform bandwidths. The 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. Read and write accesses from a node v to an object x are handled in the following way.

- v wants to read x: v sends a request message to the nearest node u holding a copy of x. u sends the requested value of x to v. A copy of x is created on each node incident to an edge (or hyperedge) on the path from u to v.
- v wants to write x: v sends a message including the new value, i.e., the value that should be written, to the nearest node u holding a copy of x. u starts an invalidation multicast to all other nodes holding a copy of x, then modifies its own copy, and sends the modified copy to v. This copy is stored on each node incident to an edge (or hyperedge) on the path from u to v.

Recall that write accesses are assumed to be object alterations rather than overwrites such that node v cannot build a new copy of x from scratch. Therefore, it sends the value to node u, which computes the new value of x, and creates copies of x on the path between the two nodes. On the first view, putting copies on the path between u and v seems to be unnecessary and not very helpful. However, we will show that this write policy leads to a 3-competitive algorithm. (An alternative write policy puts a new copy only on node v, which, however, also requires the sending of two messages between u and v. We point out that this policy leads to a worse competitive ratio, i.e., 5 rather than 3.)

Since we consider only data-race free applications, write accesses do not overlap with other accesses. Overlapping read accesses 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 (or hyperedge) 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

including a copy of x 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, a new copy is created on node u, and u serves the request message M.

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 $\mathrm{diam}(T)$ 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 dynamic caching strategy is 3-competitive for tree-connected networks with link or bus connections of arbitrary bandwidth.

Theorem 1.4.4 *The dynamic caching strategy minimizes the load on any edge (or hyperedge) up to a factor of 3.*

Proof. Fix an object $x \in X$. The accesses from the nodes to x can be ordered by their occurrence. (Concurrent reads are ordered arbitrarily, not necessarily in the order in which they are served by the dynamic caching strategy.) If the sequence of accesses to x starts with a read access, then we add a write access issued by the processor 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 accesses into *phases* such that the first access in each phase is a write and each phase includes only one write.

Let v_0 denote the node holding the initial value, and U_0 the subgraph of T including only this node. For a phase $t \geq 1$, 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, let p_t denotes the unique shortest path leading from U_{t-1} to v_t. Any consistent strategy has to send at least one message along each edge (or hyperedge) in U_t in phase t because all of the reading nodes have to receive the value written by v_t. Besides a message has to be sent along p_t either from U_{t-1} to v_t or from v_t to U_{t-1} because the value written by v_t has to be "merged" with the value written by v_{t-1}, if $t > 1$, or with the initial value placed on v_0, otherwise. Consequently, any consistent strategy has to sent one message along each edge (or hyperedge) 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 accesses of phase t except for the invalidation messages are routed along edges (or hyperedges) in $U_t \cup p_t$. In particular, each of the edges (or hyperedges) in $U_t \cup p_t$ is traversed by exactly two of these messages: either by the two messages for the write issued by v_t or by a read request message and the corresponding data message.

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

1.5 Data management on meshes

In this section, we consider data management 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 processors 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 is assumed to have bandwidth 1. Thus, the relative and absolute load of an edge are identical. We investigate static and dynamic strategies that aim to minimize the congestion.

1.5.1 Static placement on meshes

First, we prove that calculating an optimal placement with respect to minimizing the congestion is NP-hard on meshes. Thus, static data management for meshes is much harder than for trees, for which we are able to find an optimal placement in linear time. This observation leads to the idea of approximating an optimal solution for the mesh by embedding "access trees" into the mesh and simulating the optimal strategy on these trees. We show that this approach achieves minimum congestion up to a factor $O(d \cdot \log n)$, w.h.p.

1.5.1.1 NP-hardness of static placement

The integral static placement problem is defined as follows. We are given a graph, a set of shared objects, integral access rates from the nodes of the graph to the objects, and an integer k. Each object must be placed on one node, and for each access a routing path from the accessing node to the respective object has to be specified. The problem is to decide whether or not there is a solution such that the congestion of the routing paths is not larger than k.

The above problem is restricted to non-redundant placement. Note, however, that non-redundant placement is not harder than redundant placement. This follows from the observation that any optimal solution for the redundant placement problem places the objects without redundancy if all accesses are write accesses. Hence, NP-hardness of non-redundant placement induces NP-hardness also of redundant placement.

Theorem 1.5.1 *The static placement problem on the* 3×3 *mesh is NP-hard.*

Remark 1.5.2 *An alternative definition of the integral placement problem assumes that all accesses from a particular node to a particular object use the same routing path. The NP-hardness result given in Theorem 1.5.1 and the following proof hold without any change also for this more restrictive definition.*

Proof. We describe a reduction from PARTITION, which is known to be NP-complete (cf. [29]). The input of PARTITION are integers $k_1,\ldots,k_n$ and k with $\sum_{i=1}^{n} k_i = 2k$ and $k_i \leq k$, for each $i \in \{1,\ldots,n\}$. The problem is to decide whether there exists a subset $S \subset \{1,\ldots,n\}$ such that $\sum_{i\in S} k_i = k$.

We code an instance of PARTITION into a static placement problem on the 3×3 mesh. Figure 1.2 describes the labeling of the mesh used in the following. The shared objects in the placement problem are $x_1,\ldots,x_n$ and y. The access rates are defined as follows:

$$\forall j \in \{1,2,3\} \quad \forall i \in \{1,\ldots,n\}: \quad h(v_j,x_i) := k_i$$

$$h(v_2,y) := 4k+1, \quad h(h,y) := k, \quad h(h',y) := k.$$

with $h(v,x)$ denoting the number of accesses from node v to object x. All other rates are 0. We have to prove that the placement can be realized with congestion at most k if and only if there exists a subset $S \subset \{1,\ldots,n\}$ with $\sum_{i\in S} k_i = k$.

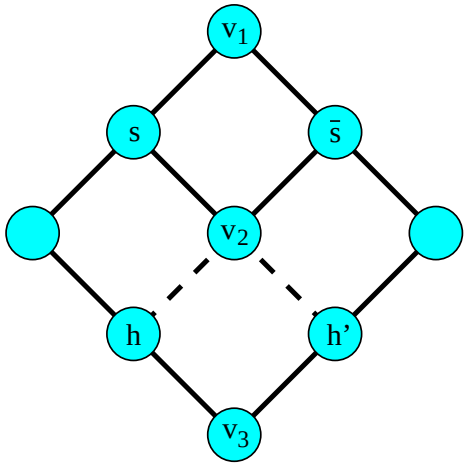


Figure 1.2: *The labeling of the* 3×3 *mesh.*

Suppose there exists a subset S fulfilling the PARTITION constraint. Then the following placement achieves congestion k. Each object x_i with $i \in S$ is placed onto node s. The routing paths from v_1, v_2, and v_3 to s are defined to be the unique shortest paths using only solid edges. Each object x_i with $i \notin S$ is placed onto node $\bar{s}$. The routing path from v_1, v_2, and v_3 to $\bar{s}$ are also defined to be the unique shortest paths using only solid edges. Object y is placed onto node v_2. For the routing path from h and h' to v_2 we choose the dashed edges. Counting the load for each edge yields that the congestion of this placement is k.

Now suppose there is no subset S fulfilling the PARTITION constraint. Assume for contradiction that there is a placement with congestion k or less. Then object y is placed on node v_2 according to this placement since otherwise one of the edges leaving v_2 has congestion larger than k. Depending on the selection of the routing paths from h and h' to object y on node v_2, we distinguish the following two cases:

- **Case 1:** Suppose each of the $2k$ accesses to y traverses one of the dashed edges. Then these edges are blocked for the other accesses, such that we can virtually remove them from the graph and consider only the placement of the x_i's.

  Consider one of the x_i's. Suppose it is placed on s (or $\bar{s}$). Then moving x_i from s to v_1 decreases the load on the edges between s (or $\bar{s}$) and v_1 for the accesses to x_j by v_1, but increases the load on the same two edges by the same amount for the accesses by the nodes v_2 and v_3. Similar arguments hold for movements to other nodes. As a consequence, all x_i's have to be placed either on s or $\bar{s}$ because otherwise the total load on the solid edges is larger than $10k$, which corresponds to an average load larger than k on these edges.

  Now, in order to achieve congestion k on the edges (s, v_1) and $(\bar{s}, v_1)$, the objects must be distributed among s and $\bar{s}$ in such a way that each of the two edges gets the same load. However, according to our assumption, an appropriate partition of the objects $x_1, \ldots, x_n$ does not exist.

- **Case 2:** Now suppose that $\ell \geq 1$ of the routing paths for the accesses to y do not traverse the dashed edges. Then the total load due to accesses to y sums up to $2k + 2\ell$ as we will have ℓ paths of length 3 and $2k - \ell$ paths of length 1. Further, only ℓ accesses to the x_i's can cross the dashed edges.

  The total load of accesses to the x_i's is minimized if the objects accessed via a dashed edge are placed on v_2. This is because an object x_i causes a load of $4k_i$ if it is placed on v_2 and the accesses

from v_3 to the object use a dashed edge, whereas in all other cases the accesses to x_i cause a load of $5k_i$ or larger.

As a consequence, the total load over all accesses to the x_i's is $4\ell + 5(2k-\ell) = 10k-\ell$ or larger, and, hence, the total load over all accesses to all objects is at least $10k-\ell+2k+2\ell = 12k+\ell$. Thus, the average load per edge over all 12 edges is larger than k, which contradicts the assumption that the considered placement has congestion k.

■

1.5.1.2 The static access tree strategy

Now we describe a static placement strategy for meshes that can be calculated efficiently and achieves minimal congestion up to a small factor, i.e., $O(d \cdot \log n)$ with d denoting the dimension and n the number of nodes. 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 $M_1 = M(m_1,\ldots,\lceil m_i/2\rceil,\ldots,m_d)$ and $M_2 = M(m_1,\ldots,\lfloor m_i/2\rfloor,\ldots,m_d)$. M_1 and M_2 are then decomposed recursively according to the same rules. Figure 1.3 gives an example for this decomposition.

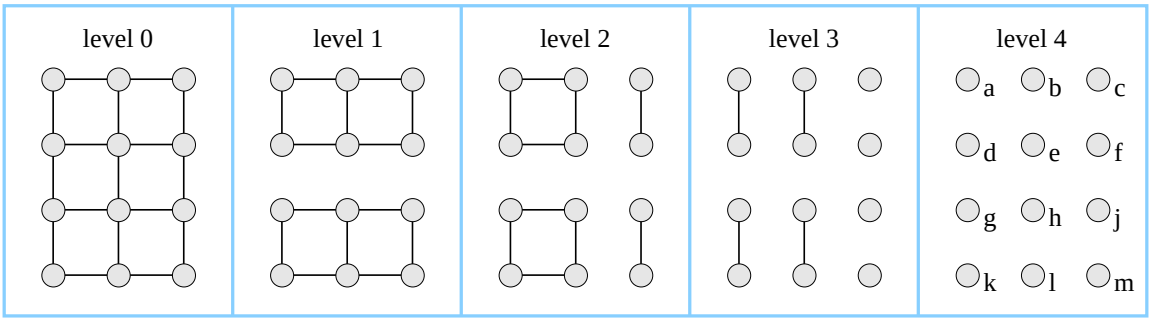


Figure 1.3: *The partitions of* $M(4,3)$.

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 processors 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 1.4 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

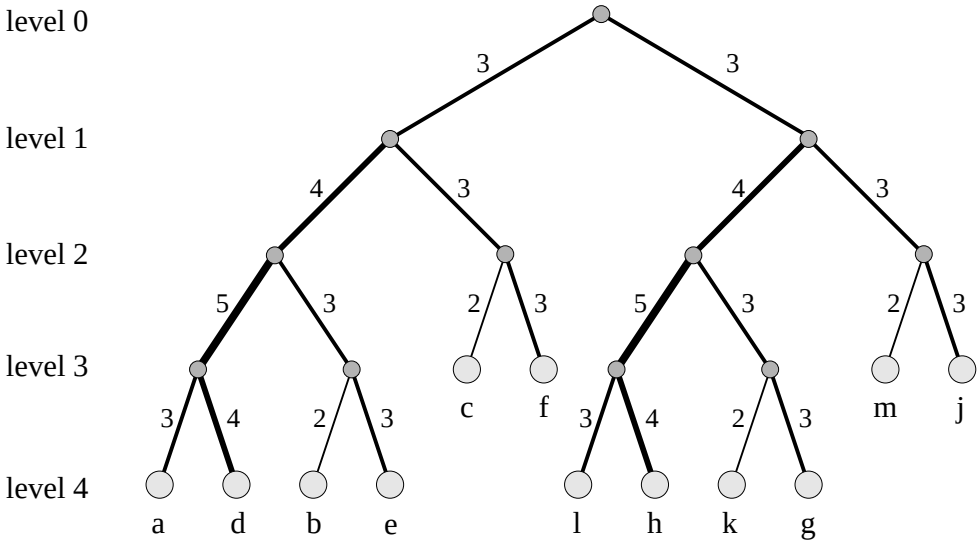


Figure 1.4: *The decomposition tree $T(M(4,3))$. The node labels correspond to the labels in Figure 1.3. The edge labels indicate the bandwidths of the respective edges.*

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 simulate the nibble strategy described in Section 1.4.1 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 paths* 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 static access tree strategy yields a static placement of the objects onto the nodes and specifies the routing paths. This placement can be calculated in time $O(|X| \cdot |V|)$, because, according to Section 1.4.1, the optimal static placement of an object on its access tree can be calculated in time linear in the number of nodes of the tree, and the number of nodes in each access tree is smaller than twice the number of nodes in the mesh. The following theorem shows that the static access tree strategy achieves small congestion.

Theorem 1.5.3 *For any application on the mesh M of dimension d with n nodes, the static access tree strategy achieves congestion $O(C_{\text{opt}}(M) \cdot d \cdot \log n)$, w.h.p., where $C_{\text{opt}}(M)$ denotes the optimal congestion for the application in the static model.*

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 access tree strategy. Define $C_{\text{opt}}(T(M))$ to be the minimum 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 static placement scheme, we actually use a separate tree $T_x(M)$ for each data object. 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 congestion of the access tree strategy to $C_{\text{opt}}(T(M))$. We start with the lower bound.

Lemma 1.5.4 $C_{\text{opt}}(T(M)) \leq C_{\text{opt}}(M)$.

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 $C' \cdot b(e)$. (Recall that $b(e)$ is the bandwidth of 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 an upper bound on the expected load of the access tree strategy. For an edge e of M, let $L(e)$ denote the load of e and $E[L(e)]$ the expectation of this value.

Lemma 1.5.5 *For any edge e of M, $E[L(e)] = O(\log n \cdot d \cdot C_{\text{opt}}(T(M)))$.*

Proof. Let h denote the height of $T(M)$, and let $L_\ell(e)$ denote the load of 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,

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

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 $C_{\text{opt}}(T(M)\cdot b(e_T))$. Consequently, the expected load on edge e for simulating e_T is at most

$$C_{\text{opt}}(T(M))\cdot b(e_T)\cdot P(e) \quad \leq \quad C_{\text{opt}}(T(M))\cdot 6d \ .$$

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. ■

Consider an arbitrary edge e of the mesh M. Let $L(e)$ denote the load of e. Applying Lemma 1.5.4 and Lemma 1.5.5 yields $E[L(e)] = O(\log n\cdot(d\cdot C_{\text{opt}}(M)+\kappa))$. In order to complete the proof of Theorem 1.5.3, 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 $L(e)$ is the sum of three terms L_1, L_2, and L_3, each of which being a sum of independent random variables, such that we can apply a Chernoff bound. We color the edges of the access trees with three colors $\{1,2,3\}$ such that all edges incident on the same node have different colors. For $1\leq j\leq 3$, let $E_{x,j}$ be the set of tree edges of $T_x(M)$ with color j. For $e_T\in E_{x,j}$, let $A(e_T)$ be a random variable that is 1 if the dimension-by-dimension order path that simulates e_T crosses e, and that is 0, otherwise. Further, for $e_T\in E_{x,j}$, let $K(e_T)$ denote the load of e_T due to the static placement scheme in the access tree $T_x(M)$. Then define

$$L_j := \sum_{x\in X}\sum_{e_T\in E_{x,j}} A(e_T)\cdot K(e_T) \ ,$$

for $1\leq j\leq 3$. Note that $L(e) = L_1+L_2+L_3$. The coloring yields that the $A(e_T)$'s are independent for all $e_T\in\bigcup_{x\in X}E_{x,j}$, for fixed $j\in\{1,2,3\}$, because no two tree edges represented by these variables share an endpoint. Therefore, L_j is a sum of weighted independent random variables. Let κ denote the write contention, i.e., the maximum number of write accesses to the same object. The nibble strategy guarantees that for every $e_T\in E_{x,j}$, $K(e_T) = O(\kappa)$. As a consequence, the maximum weight in the sum of random variables is at most $O(\kappa)$. Thus, applying a Chernoff bound (see, e.g., [32]) to the sum yields that $L_j = O(E(L_1)+\kappa\cdot\log n)$, w.h.p., and therefore,

$$L(e) \ = \ L_1+L_2+L_3 \ = \ O(E(L)+\kappa\cdot\log n) \ = \ O(\log n\cdot(d\cdot C_{\text{opt}}(M)+\kappa)) \ ,$$

w.h.p. The write contention κ and the optimal congestion $C_{\text{opt}}(M)$ are related as follows. Each write access has to update all copies and each copy is placed statically at a node with degree $2d$. Consequently, $C_{\text{opt}}(M)\geq\kappa/(2d)$, which yields $L(e) = O(\log n\cdot d\cdot C_{\text{opt}}(M))$, w.h.p. As this bound holds for any edge, we have obtained the desired upper bound on the congestion. This completes the proof of Theorem 1.5.3. ■

1.5.2 Dynamic data management on meshes

In this section, we show how the access tree strategy described above can be modified so that it achieves minimum congestion up to a factor of $O(d\cdot\log n)$ in the dynamic model. Note that this is the same bound that we proved for the static access tree strategy in Theorem 1.5.3. Further, we give a lower bound for on-line routing on meshes showing that the above competitive ratio is optimal up to a factor $\Theta(d^2)$.

1.5.2.1 The dynamic access tree strategy

The access tree strategy described for the static model can be easily adapted to the dynamic model by simulating the 3-competitive dynamic instead of the optimal static strategy on the access trees: Initially, a copy of each object is placed on one of the mesh nodes. This corresponds to an initial placement on a leaf node of the access tree. Whenever a copy is migrated according to the tree strategy it is sent along the dimension-by-dimension order path to the mesh node that simulates the respective access tree node.

We use the same definitions as for the static access tree strategy. In particular, $C_{\text{opt}}(M)$ and $C_{\text{opt}}(T(M))$ denote the optimal congestion for a given application on the mesh M and the decomposition tree $T(M)$, respectively. But here the term "optimal congestion" refers to the dynamic model. It is easy to check that the results of Lemma 1.5.4 and Lemma 1.5.5 that relate $C_{\text{opt}}(M)$ to $C_{\text{opt}}(T(M))$, and vice versa, hold without any change for the dynamic model.

Applying the two lemmas to the straightforward adaption of the static access tree strategy yields a bound of $O(C_{\text{opt}}(M) \cdot d \cdot \log n + \kappa \cdot \log n)$ on the congestion produced by this strategy. In the static model, the optimal congestion was at least $\kappa/(2d)$. From this we could deduce $O(d \cdot \log n)$-competitiveness. Unfortunately, in the dynamic model we cannot apply this bound. In fact, it is easy to construct counterexamples in which the optimal congestion is in $o(\kappa/d)$. In order to achieve $O(d \cdot \log n)$-competitiveness in the dynamic model the access tree nodes have to be remapped dynamically when too many *access messages*, i.e., messages that simulate messages of the 3-competitive tree strategy, traverse a node. The remapping is done as follows.

For every object x, and every node v of the access tree $T_x(M)$ we add a counter $\tau(x,v)$. Initially, this counter is set to 0, and the counter $\tau(x,v)$ is increased by one whenever an access message for object x traverses node v, starts at node v, or arrives at node v. When the counter $\tau(x,v)$ reaches K the node v is remapped randomly to another node in $M(v)$, where K is some integer of suitable size, e.g., $K = 18$. Remapping v to a new host means that we have to send a *migration 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. Migration messages reset the counter $\tau(x,v)$ to 0. Furthermore, we have to send *notification messages* including information about the new host to the mesh nodes that hold the access tree neighbors of v. These notification messages also increase the counters at their destination nodes, i.e., the counter $\tau(x,v')$ for any neighbor v' of v in the access tree. The counter mechanism ensures that the number of messages that are directed to a copy on a randomly selected host is $O(K)$ rather than $\Theta(\kappa)$. Note that this bound would fail if the notification messages would not increase the counters.

Possibly, migration or notification messages overlap with access messages or other migration 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. This mechanism guarantees that each randomly selected host gets at most three additional messages. We define that neither the acknowledgment messages nor the overlapping messages influence the counters. For simplicity, we will not consider any of these messages in the following analysis. Obviously, they have no influence on our asymptotic bounds as each of them corresponds to another message that we take into account.

The following theorem shows that the dynamic access tree strategy produces minimal congestion up to a factor of order $d \cdot \log n$. In the following section, we will see that this factor is very close to optimal.

Theorem 1.5.6 *The dynamic access tree strategy is $O(d \cdot \log n)$-competitive, for meshes of dimension d with n nodes.*

Proof. The load on the mesh edges is increased by the migration and the notification messages. However, the impact of these messages on the expected load of the edges is relatively small, which can be shown as follows.

First, we consider the notification messages. All notification messages for an object $x \in X$ are sent between nodes that simulate neighboring nodes in the access tree $T_x(M)$. We merge the access tree of all objects to form the virtual decomposition tree $T(M)$ as defined in the previous section. Recall that each node of $T(M)$ represents a submesh according to the decomposition hierarchy. The bandwidth of an edge connecting a tree node v with its parent in $T(M)$ is equal to the number of edges leaving the corresponding submesh $M(v)$. We show that, if K is chosen sufficiently large, then the congestion of the notification messages in $T(M)$ is not larger than the congestion of the access messages.

Let F denote the maximum number of notification messages passing a single edge of $T(M)$. Suppose e_T is an edge of $T(M)$ that transmits F notification messages. We show that one of the edges incident to e_T must have been traversed by many access messages. Each of the migration messages traversing e_T was caused by K other messages. These messages are either 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 $K \cdot F/2$ access or notification messages. Hence, the load on at least one of the three edges incident to this node is at least $K \cdot F/6$. Let e'_T denote one of the incident edges fulfilling this property. For $K \geq 18$, the load on e'_T due to access or notification messages is at least $3F$. Then the load on e'_T due to access messages is at least $2F$ because the maximum number of notification messages that cross e'_T is F.

The relative load of an edge is defined to be the (absolute) load divided by the bandwidth of the edge. The bandwidth of two edges from $T(M)$ incident on the same node deviates at most by a factor of 2. Hence, $b(e_T) \geq b(e'_T)/2$, which yields $F/b(e_T) \leq 2F/b(e'_T)$. Consequently, 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 simulate the 3-competitive tree strategy on each access tree, the expected relative load due to access messages on every edges is in $O(C_{\text{opt}}(T(M)))$. Hence, we can conclude that also the expected relative load due to notification messages is in $O(C_{\text{opt}}(T(M)))$. Now applying Lemma 1.5.4 and Lemma 1.5.5 yields that the expected load for access and notification messages on an arbitrary edge of the mesh is bounded by $O(\log n \cdot d \cdot C_{\text{opt}}(M))$.

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

Again we merge together the access trees of all objects to form the virtual decomposition tree $T(M)$. Fix a node v from this tree. Let $H(v)$ denote the number of migration messages sent for all access tree nodes corresponding to v. Then the total number of access or notification messages that are sent along the at most 3 tree edges incident on v is at least $H(v) \cdot K$. Thus, one of these edges has load $H(v) \cdot K/3$ or larger. Let e_T denote this edge. The migration messages transmitted along e_T only influence the load on edges in $M(v)$. We will relate the load due to migration messages on these edges to the relative load due to access and notification messages on e_T.

The migration messages for v are sent between two randomly chosen nodes from the submesh $M(v)$. Consider an arbitrary edge e from $M(v)$. The expected load on edge e due to these messages is at most $6d \cdot H(v)/b(v)$. (This bound follows analogously to the proof of Lemma 1.5.5.) The relative load $L(e_T)$ on the access tree edge e_T is at least the (absolute) load on this edge divided by $b(e_T)$, which yields $L(e_T) \geq H(v) \cdot K/(3 \cdot b(e_T))$. Combining both bounds yields that the expected load on edge e due to the migration messages is at most $d \cdot L(e_T)$, for $K \geq 18$. We have observed above that

$L(e_T) = O(C_{\text{opt}}(T(M)))$. Consequently, the expected load for the migration messages of v on edge e is bounded by $O(d \cdot C_{\text{opt}}(T(M)))$.

Each edge of the mesh is included in at most $O(\log n)$ submeshes of the decomposition hierarchy. Therefore, the expected load due to migration messages on an arbitrary edge of the mesh is bounded by $O(d \cdot \log n \cdot C_{\text{opt}}(T(M)))$. By Lemma 1.5.4, $C_{\text{opt}}(T(M)) \leq C_{\text{opt}}(M)$. Hence, the expected load due to migration messages is in $O(d \cdot \log n \cdot C_{\text{opt}}(M))$.

Finally, let $L(e)$ denote the load on an arbitrary mesh edge e due to all kind of messages. We have shown that $E[L(e)] = O(d \cdot \log n \cdot C_{\text{opt}}(M))$. Analogously, to the proof for the static model, we decompose $L(e)$ into three parts such that $L(e) = L_1 + L_2 + L_3$, and each L_j is the sum of independent random variables, one for every placement of every object x. Each of these variables indicates the number of messages for x that used e between remappings of x, and, hence, each has a maximum value of $O(K)$. Applying a Chernoff bound to the sums represented by the L_j's yields that $L(e) = O(E[L(e)] + K \cdot \log n) = O(E[L] + \log n)$, e.g., for $K = 18$. Hence, the congestion is $O(\log n \cdot d \cdot C_{\text{opt}}(M))$, w.h.p., which completes the proof of Theorem 1.5.6. ■

1.5.2.2 A lower bound for on-line routing and dynamic data management

The results in Theorem 1.5.6 compare the congestion of the dynamic access tree strategy with the congestion of an optimal off-line strategy. The theorem shows that the access tree strategy achieves minimum congestion up to a factor of $O(d \cdot \log n)$. In the following, we give a lower bound for on-line routing that 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 $1 \leq t \leq \Delta$. An on-line routing algorithm must assign a routing path connecting s_t and d_t, for $1 \leq t \leq \Delta$, 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 any deterministic or randomized on-line algorithm. The same 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" [11].

Theorem 1.5.7 *Any on-line routing algorithm for the mesh of dimension $d \geq 2$ and side length m has competitive ratio $\Omega(\log m)$.*

Dynamic data management 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 1.5.8 *Any dynamic data management strategy for the mesh of dimension $d \geq 2$ and side length m has competitive ratio $\Omega(\log m)$.*

Proof of Theorem 1.5.7. We show that for any C, $d \geq 2$, and $m \geq 4$ being a power of 2, there is a random routing problem $\mathcal{P}_d(m,C)$ 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 $\mathcal{P}_2(m,C)$ 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 $\Omega(C \cdot \log m)$.

$\mathcal{P}_2(m,C)$ 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 $\mathcal{P}_2(m/2,C)$.

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 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 $\mathcal{P}_2(m,C)$ is $C \cdot (\log m - 1)/8$, which completes the proof for the 2-dimensional case.

We now describe the on-line routing problem $\mathcal{P}_d(m,C)$ 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. $\mathcal{P}_d(m,C)$ is defined as follows. The adversary specifies the routing requests in each submesh M_j according to $\mathcal{P}_2(m,C)$, 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 $\mathcal{P}_d(m,C)$ 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 $\mathcal{P}_2(m, m^{d-2} \cdot C)$. 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 $\mathcal{P}_2(m, m^{d-2} \cdot C)$. Consequently, the expected congestion of any on-line strategy on $\mathcal{P}_d(m,C)$ is at least $C \cdot (\log m - 1)/8$. ■

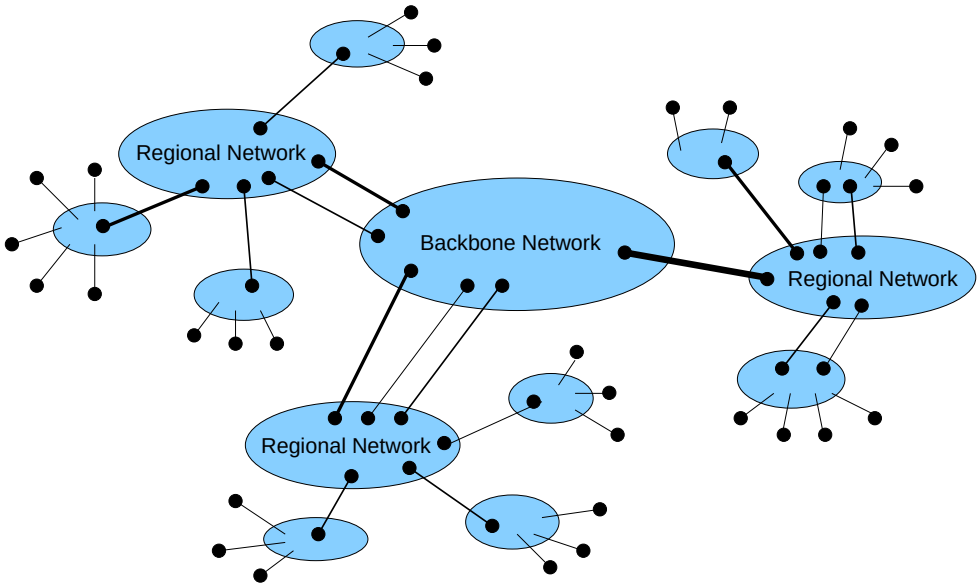


Figure 1.5: *The topology of a wide area NOW.*

1.6 Data management on 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 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 1.5 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 $V_K \backslash U$. The *capacity* of a cut U is the sum of the bandwidths of the edges connecting the nodes in U with the nodes in $V_K \backslash U$. This capacity is denoted by $\mathrm{cap}(U)$. We define the *cross flux* by

$$\alpha(K) = \min_{U \subseteq V_K} \frac{\mathrm{cap}(U) \cdot w(V_K)}{w(U) \cdot w(V_K \backslash U)} \approx \min_{U \subseteq V_K} \frac{\mathrm{cap}(U)}{\min\{w(U), w(V_K \backslash U)\}} \ .$$

We assume that $\alpha(K) = \Omega(1)$, for any 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 accesses to shared objects of a set X. 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 $v \in V_K$. For the static access tree strategy, we simulate the nibble strategy on these access trees. The placement for each object can be calculated in time linear in the total number of nodes in the network. For the dynamic access tree strategy, we simulate the 3-competitive dynamic caching strategy on each access tree.

The static and the dynamic strategies require 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 [49]. 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 1.6.1 *Consider a distributed 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 one tree node. Further, let κ denote the write contention, i.e., the maximum number of write accesses to the same object, and let $C_{\mathrm{opt}}(G)$ denote the optimal congestion for the application when it is executed on G. Then the access tree strategy achieves congestion*

$$O(\log \delta \cdot C_{\mathrm{opt}}(G) + \gamma \cdot \kappa \cdot \log n) \ ,$$

w.h.p. If the clusters in G can be represented by planar graphs or by constant genus graphs, then this result improves to

$$O(C_{\mathrm{opt}}(G) + \gamma \cdot \kappa \cdot \log n) \ ,$$

w.h.p. These bounds hold for static and dynamic data management.

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 in the static or dynamic model, respectively. 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. For the dynamic strategy, we assume that the path selection for each message is independent from the selection of other messages. For the static strategy, we assume that all messages that are directed to the same object use the same random bits such that all accesses to a particular object from a particular node use the same routing path.

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 1.6.2 *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 simulates the optimal static or the 3-competitive dynamic strategy, respectively, 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 $1\le i,j\le\delta(K)$. 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 1.6.3 *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 [4] 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 [38]). 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 denotes 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 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 we have multiplied all weights and capacities with the same factor F 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 1.6.3. ■

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 access 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 1.6.4 *For any 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 1.6.2 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 Lemmas 1.6.2, 1.6.3, and 1.6.4 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 it is $O(C_{\text{opt}}(K)/\min\{1, \alpha(K)\})$ if K is planar or of constant genus. In order to complete the proof of Theorem 1.6.1, 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 static and dynamic data management strategies for trees ensure that the load on each edge of the access tree due to access 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 $O(\gamma(K) \cdot \delta(K))$ not necessarily independent 0-1-random variables, each of which representing the load due to one access 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

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

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., [32]) 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 $L(e) = O(E[L(e)] + \tau \cdot \log n) = O(E[L(e)] + \gamma(K) \cdot \kappa(K) \cdot \log n)$, w.h.p., which completes the proof of Theorem 1.6.1. ■

1.7 More general and other models

1.7.1 Non-uniform object sizes and slice-wise accesses

For simplicity, we have assumed until now that all objects and also all access messages have unit size. Now we consider models that charge different costs for sending messages of different size.

Our static strategies, in particular the nibble strategy, can easily be adapted to arbitrary object and message sizes. This can be done by weighting the access rates h_r and h_w according to the costs of reads and writes, respectively. For instance, if a read access to an object requires the sending of a request message of size r_1 to the nearest node holding a copy, and this node sends back a data message of size r_2, then the access is weighted with $r_1 + r_2$. Analogously, if a write requires the transmission of an update message of size w along a multicast tree, then this access is weighted with w. (Note that it is not necessary for a write message to contain a header that lists all of the nodes that contain a copy of the object if we add signposts to the nodes as described for the dynamic tree strategy.) Applying the nibble strategy with respect to the weighted rates yields an optimal placement.

The dynamic strategies profit from large objects. In the analysis for our 3-competitive tree strategy, we showed that the strategy only sends messages across edges where the actual data has to be sent. For each of these links, we sent three messages, at most two of them were data messages. Let $|x|$ denote the size of an object and let us assume that the size of request and invalidation messages is 1. Then the competitive ratio for the dynamic tree strategy becomes $(2+1/|x|)$ rather than 3.

It is more complicated to adapt the dynamic strategies to slice-wise accesses, i.e., accesses in which only a relatively small part of an object is read or written. In this case, migrating an object becomes much more expensive than accessing it. Slice-wise accesses are typical for distributed file systems. For this scenario, Bartal et al. [10] describe a randomized and a deterministic algorithm for trees that achieve competitive ratio 3 and 9, respectively. However, these competitive ratios measure the total communication load rather than the congestion. Lund et al. [50] describe a deterministic but centralized strategy with competitive ratio 3. This ratio holds for the load of any edge. If the algorithm of Lund et al. is used in the access tree strategies instead of our distributed algorithm, then this yields centralized strategies for dynamic data management with slice-wise accesses on meshes and clustered networks fulfilling the same congestion bounds as the distributed access tree strategy for uniform object size. (To obtain the result for meshes, the dynamic access tree strategy must be changed slightly: Let D denote the factor that measures the difference between the cost for migrating and the cost for accessing an object. Then a node v in the access tree of an object x is migrated to a new host when $\tau(v,x)$ reaches $K \cdot D$ rather than K.)

1.7.2 Other update policies

The static model restricts the class of allowed update policies since it is assumed that a write has to update all copies. For instance, the majority trick, introduced in [78] for PRAM simulations, does not fit into this model. Here only more than half of the copies of an object are updated in case of a write and more than half of the copies are accessed in case of a read. This ensures that every read access gets at least one copy that has been updated by the least recent write. However, this technique requires that a time stamp is added to each copy in order to figure out which of the copies accessed by a read is the most recent one. Since it is difficult to implement this in an asynchronous setting we restricted ourselves until now to strategies that update all copies in the case of a write.

In the following, we consider a more general model in which reads and writes are treated totally symmetrically. Therefore, the model is called the *symmetric model*. We assume that all accesses are known in advance but their order is unknown. (This corresponds to knowing the number of read and write accesses from the nodes to the objects.) A static placement strategy has to specify for each read and write access which copies to read and which to update, respectively. Read and write accesses can be satisfied within a multicast. The placement strategy has to fulfill the following condition: Every read access has to touch at least one of the copies touched by the least recent write access. An access *touches a node* if the corresponding multicast tree includes the node, and an access *touches a copy* if it touches the node holding the copy.

The symmetry between read and write accesses results from the fact that the order of the accesses is not known when the static placement strategy is specified. The condition that every read access has to touch at least one of the copies touched by the least recent write access is equivalent to the following condition: The multicast tree of every read access to an object x must overlap with the multicast tree of every write access to x at an arbitrary node that holds a copy of x.

Note that the set of admissible strategies for the symmetric model includes all admissible strategies for our original model and also strategies using the majority trick. Also the *inverse nibble strategy*, described in the following, is an admissible strategy for the symmetric model. According to the inverse nibble strategy, a read access always touches all copies whereas a write touches only the copy closest to the

writing node. The copies are placed analogously to the nibble strategy, but the rates for reads and writes are exchanged, i.e., $w(v)$ is set to $h_r(v,x)$ and $r(v)$ is set to $h_w(v,x)$, for every node $v \in V$ and every object $x \in X$. The following theorem shows that a combination of the nibble and the inverse nibble strategy is optimal in the symmetric model.

Theorem 1.7.1 *Consider a static placement problem on a tree-connected network $T = (V,E)$ in the symmetric model. If the total number of reads R to an object x is greater than or equal to the total number of writes W then placing x according to the nibble strategy yields minimum load on any edge (or hyperedge). Otherwise, applying the inverse nibble strategy yields this result.*

Remark 1.7.2 *The above theorem has consequences for the static result on meshes. It shows that simulating the combination of nibble and inverse nibble strategy on meshes of dimension d and size n yields congestion $O(C_{\text{opt}} \cdot d \cdot \log n + \min\{\kappa_r, \kappa_w\} \cdot \log n)$ in the symmetric model. Here κ_r denotes the read contention, i.e., the maximum number of read accesses to the same object, and κ_w denotes the write contention, i.e., the maximum number of write accesses to the same object.*

Furthermore, it shows that the results for static placement on clustered networks given in Theorem 1.6.1 hold also in the symmetric model if the combination of nibble and inverse nibble strategy is simulated on the access trees. (The κ in the bounds given in the theorem can be substituted by $\min\{\kappa_r, \kappa_w\}$.)

Proof. Suppose $R \geq W$. We show that any strategy S that does not touch all copies of x in case of a write can be transformed into a strategy S' that touches all copies without increasing the load on any edge (or hyperedge). Then, by Theorem 1.4.1, S' can be transformed into the nibble strategy without increasing the load on any edge. Hence, the nibble strategy yields minimum load on any edge (or hyperedge).

Consider an arbitrary strategy S that does not touch all copies in case of a write. Then there exist two nodes u and v connected via an edge (or hyperedge) e such that some of the write accesses issued in $T(u)$ do not touch v although there is at least one copy in $T(v)$, where $T(u)$ and $T(v)$ denote the maximal subtrees of T including u and v, respectively, but not e. We distinguish two cases:

- **Case 1:** Any write access issued in $T(v)$ touches u. In this case we remove all copies in $T(v)$. This does not increase the load due to read accesses issued in $T(v)$, because in spite of these copies all of the read accesses issued in $T(v)$ have to cross e because some of the write accesses issued in $T(u)$ do not touch v.

- **Case 2:** Some of the write accesses issued in $T(v)$ do not touch u. In this case, we show that there is a subtree $T^* = (V^*, E^*)$ of T such that each node $w \in V^*$ is touched by any write access issued in $T'(w)$, but, if w is a leaf of T^*, then at least one write access issued in $T'(w)$ does not leave $T'(w)$, where $T'(w)$ denotes the maximal subtree including w but no edge (or hyperedge) from E^*. Picture 1.6 illustrates this definition.

  The subtree T^* can be constructed as follows. Initially, we set $E^* = \{e\}$, and $V^* = \{u,v\}$. Suppose T^* does not fulfill the property described above. Then a node $w \in V^*$ is not touched by every write access issued in $T'(w)$. Hence, there is a node z from $T'(w)$ that is adjacent to w such that some of the write accesses issued in $T'(z)$ do not leave $T'(z)$, where $T'(z)$ is the maximal subtree of T including z but not the edge (or hyperedge) e_z connecting z with w. We add e_z to E^* and z to V^*. Then we apply the argument recursively. The resulting tree $T^* = (V^*, E^*)$ fulfills the desired property.

  Now strategy S is transformed into S' as follows. We add a copy to each node included in V^* that does not hold a copy, and we define that every copy on the nodes in V^* is touched by every write access issued in T. This transformation increases only the load on the edges (or hyperedges)

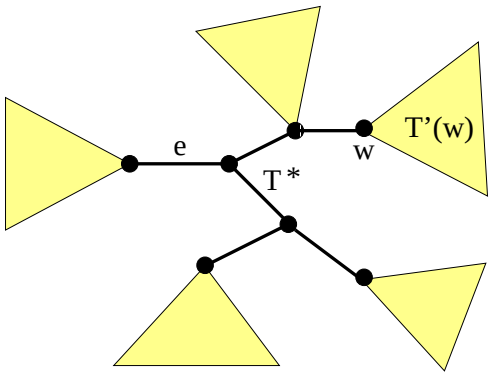


Figure 1.6: *The picture shows the subtree $T^* = (V^*, E^*)$. For a node $w \in V^*$, $T'(w)$ denotes the maximal subtree including w but no edge (or hyperedge) from E^*. For each node $w \in V^*$, any of the write accesses issued in $T'(w)$ touches w. If w is a leaf node in T^*, then at least one of the write accesses issued in $T'(w)$ does not leave the subtree $T'(w)$.*

included in E^* because, before the transformation, every write access touched one of the nodes in V^*. After the transformation, none of the read accesses has to traverse an edge (or hyperedge) of E^*, whereas, before the transformation, every read access issued in T had to traverse all of these edges because it had to touch every leaf node of T^* in order to overlap with the multicast tree of every write access. Consequently, our transformation saves load $R - W \geq 0$ on every edge (or hyperedge) from E^*.

In both cases the number of copies that are not touched by every write access is decreased at least by one. Thus, applying the above transformation repeatedly yields a strategy in which every write access touches all copies.

Now suppose $R < W$. We have to prove that the inverse nibble strategy minimizes the load on every edge in this case. For symmetry reasons, this can be shown analogously to the previous case by treating writes as reads and reads as writes. ■

1.7.3 An alternative model for dynamic data management

Recently, Wolfson et al. have introduced an alternative model for dynamic data management [84]. They assume that the access patterns do not vary too much with time, that is, "the pattern of a processor in a time period (of fixed length) is repeated for several time periods". An adaptive data replication (ADR) algorithm is called *convergent-optimal*, if starting with an arbitrary allocation of copies, the algorithm converges to an optimal allocation within some time periods, under the assumption that the access frequencies of the nodes do not vary within these periods. Here an *optimal allocation* refers to an allocation that yields the minimum total communication load for the respective access frequencies.

Wolfson et al. present a *convergent-optimal* ADR algorithm for trees, which is a variant of the static placement strategy of the algorithm in [85]. The algorithm converges within a number of periods that is bounded by the diameter of the network. In each period of time, the processors collect statistics about the read and write accesses passing the edges incident on the nodes. Once the ADR algorithm has transformed the initial allocation of copies for a particular object into an allocation in which the nodes holding a copy build a connected component, the algorithm maintains this property for the rest of the execution. Based on the local edge statistics the algorithm decides for each edge incident on a border node of the connected component whether or not the connected component grows, shrinks, or migrates along an edge.

Interestingly, the criteria applied in the ADR algorithm for deciding whether the connected component grows, shrinks, or migrates correspond to the criteria applied in our static nibble strategy for deciding whether a node belongs to the connected component of nodes holding a copy or not. In fact, it is easy to check that the allocation obtained by the ADR algorithm when it converges is equivalent to the allocation of our static nibble strategy. On the one hand, this equivalence shows that the ADR algorithm of Wolfson et al. does not only minimize the total communication load but also the congestion. On the other hand, it shows that our nibble strategy for trees and, hence, also the static access tree strategies for meshes and clustered networks can be adapted to work in a dynamic context, too, in which access patterns change slowly.

1.8 Experimental evaluation of the access tree strategy

In this section, the dynamic access tree strategy for two-dimensional meshes will be evaluated experimentally. We test several variations of the access tree strategy for three applications of parallel computing, which are matrix multiplication, bitonic sorting, and a Barnes-Hut N-body simulation. We compare the congestion and the execution time of these variations of the access tree strategy to a standard caching strategy that uses a fixed home for each data object. For the matrix multiplication and the sorting algorithm, we will additionally compare the dynamic data management strategies to hand-optimized message passing strategies that achieve minimal congestion. We will see that the congestion and the execution times are closely related. Besides we will see that the access tree strategy clearly outperforms the fixed home strategy. The larger the network is the more superior the access tree strategy is against the fixed home strategy.

The tested variants of the access tree strategy and the fixed home strategy have been implemented in the DIVA (Distributed Variables) library [42]. The library includes routines which allow to create, to read, and to write *global variables*, i.e., shared data objects, on a mesh-connected processor network with local memory modules. Furthermore, it provides synchronization mechanisms such as barriers and locks for which access trees are used, too. These mechanisms will be described later on.

The variations of the access tree strategy use trees of different heights and degrees. The idea behind varying the degree of the access trees is to reduce the overhead due to additional startups: Any intermediate stop on a processor simulating an internal node of the access tree requires that this processor receives, inspects, and sends out a message. 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*. Obviously, a more flat access tree reduces the number of intermediate stops and, therefore, the overall startup costs.

The implemented algorithms for matrix multiplication and sorting are *oblivious*, i.e., their access or communication patterns do not depend on the input. We cannot expect a dynamic on-line strategy to perform as well as a hand-optimized message passing strategy for oblivious algorithms. Often, using global variables is more convenient than using a message passing mechanism. However, the tested matrix multiplication and sorting algorithms are so simple that we would prefer to use message passing rather than global variables in order to get a more efficient implementation. The reason 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 a hand-optimized message passing strategy.

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. However, we cannot construct a hand-optimized message passing strategy achieving minimal congestion for this

application. Therefore, we concentrate on the comparison of different dynamic data management strategies.

In the theoretical analysis we have assumed that the memory resources are unbounded. In the experiments we will not investigate the effects of bounded memory capacities either. The strategies implemented in the DIVA library, however, are able to deal with this problem. If the local memory module is full then data objects will be replaced in least recently used fashion. However, in all our experiments there will be a sufficient amount of memory so that no data objects have to be replaced (unless otherwise stated).

The different variations of the access tree strategy. The access tree strategy described in Section 1.5 uses a 2-ary hierarchical mesh decomposition. Alternatively, we use 4-ary and 16-ary decompositions leading to 4-ary and 16-ary access trees, respectively. The 4-ary decomposition just skips the odd decomposition levels of the 2-ary decomposition, and the 16-ary decomposition then skips the odd decomposition levels of the 4-ary one.

Another way to get flatter access trees is to terminate the hierarchical mesh decomposition with submeshes of size k. An access tree node that represents a submesh of size $k' \leq k$ gets k' children, one for each of the nodes in the submesh. We define the *ℓ-k-ary access tree strategy*, for $\ell = \{2,4\}$ and $k \geq \ell$, to use an ℓ-ary hierarchical mesh decomposition that terminates at submeshes of size k.

All variations of the access tree strategy embed an access tree for each data object and run the 3-competitive tree strategy, as described in Section 1.4.2, on these access trees. If write requests are not acknowledged then it is difficult to figure out whether or not all accesses are completed at some point of time, which is a problem for the implementation of efficient synchronization mechanisms. Therefore, we add acknowledgements to the invalidation messages.

For simplicity, the access tree nodes are embedded statically (without remapping) into the respective submeshes via a random hash function. In order to shorten the routing paths we use a more regular embedding of the access trees than described in the theoretical analysis. We assume that the processors are numbered from 0 to $P-1$ in row major order and each data object has a unique ident-number smaller than ρ, where ρ denotes a sufficiently large prime number. Let h be a randomly selected polynomial of degree 2 over $\mathbb{Z}_\rho$. The root of the access tree for an object with ident-number x is mapped onto node $h(x) \bmod P$. The embedding of all access tree nodes below the root depends on the embedding of their respective parent node: Consider an access tree node v with parent node v'. Let M denote the submesh represented by v and M' the submesh represented by v', which includes M. Suppose v' is mapped to the node in the ith row and jth column of M'. Then v is mapped to the node in row $i \bmod m_1$ and column $j \bmod m_2$ of M, where m_1 and m_2 denote the number of rows and columns of M, respectively.

The modified embedding adds dependencies between the mappings of different nodes included in the same access tree such that the theoretical analysis does not hold anymore. However, we have not recognized any bad effects due to these dependencies in our experiments. The major advantage of the modified embedding is that it decreases the expected distances between the processors simulating neighbored access tree nodes. For example, consider the 4-ary mesh decomposition. Suppose the submesh M' represented by v' is of the size $2m \cdot 2m$. Then the size of M is $m \cdot m$. In this case, the expected distance between v and v' is $(4/3) \cdot m$ in case the original mapping is used, and it is m in case the modified mapping is used. For the 2-ary decomposition the difference becomes even larger: if the mesh is quadratic then the factor between the original and the modified mapping is about $8/3$ on even decomposition levels and about 2 on odd decomposition levels.

The fixed home strategy. The variants of the access tree strategy will be compared to a data management strategy following a standard approach in which each global variable is assigned a processor keeping track of the variable's copies. This processor is called the *home* of the variable. The home of each variable is chosen uniformly at random from the set of processors. In order to manage the copies of the data objects, we use the well known *ownership scheme* described, e.g., in [31]. Originally, the ownership scheme was developed for shared memory systems in which many processors with local caches are connected to a centralized main memory module by a bus. The scheme works as follows.

At any time either one of the processors or the main memory module can be viewed as the *owner* of a data object. Initially, the main memory module is the owner of the object. A write access issued by a processor that is not the owner of the data object assigns the ownership to this processor. A read access issued by another processor moves the ownership back to the main memory. Write accesses of the owner can be served locally, whereas all other write accesses have to invalidate all existing copies and create a new copy at the writing processor. Read accesses by processors that do not hold a copy of the requested data object move a copy from the owner to the main memory module (if the main memory module is not the current owner itself) and a copy to the reading processor. In this way, subsequent read accesses of that processor can be served locally.

In a network with distributed memory modules, the home processor plays the role of the main memory module. It keeps track of all actual copies and is responsible for their invalidation in case of a write access. In the original scheme, invalidation is done by a *snoopy cache controller*, that is, each processor monitors all of the data transfers on the bus. Of course, this mechanism does not work in a network. Here the home processor has to send an invalidation message to each of the nodes holding a copy.

If each write access of a processor to a data object is preceded by a read access of this processor to the same object then the fixed home strategy using the ownership scheme corresponds to a P-ary access tree strategy with P denoting the number of processors. Interestingly, this condition is met for every write access in the three applications that we will investigate. Therefore, the fixed home strategy is very well suited for comparisons with the access tree strategy.

Synchronization mechanisms. The DIVA library also includes routines for barrier synchronization and the locking of global variables. These routines are implementations of elegant algorithms that use access trees.

The implemented barrier synchronizations allow to synchronize arbitrary *groups*, i.e., subsets of the processors. The groups can be generated dynamically at run time. With each group, we associate a randomly embedded access tree. A call for a barrier synchronization sends a message upwards in the access tree. Each node of the access tree waits until it receives a synchronization message from each of its children that represent a submesh including at least one processor of the group. After receiving all these messages the access tree node either sends a synchronization message to its parent node, or, if the node represents the smallest submesh that includes all members of the group, it initiates a multicast downwards along the branches of the access tree notifying all group members about the completion of the barrier.

The degree of the access trees for the synchronization messages correspond to the one that is used for the access trees of the global variables with one exception. For the barrier synchronizations of the fixed home strategy, we use 4-ary access trees since P-ary trees obviously perform very badly for large meshes.

The implemented locks are attached directly to global variables. A call for a lock deletes all of the variable's copies and creates a new exclusive copy on the processor that has requested the lock. All subsequent read, write, or lock requests of other processors are blocked as soon as they reach the node holding the exclusive copy. If the lock is released then the work of the blocked requests is continued as usual. Note that in case the degree of the access tree is relatively small, the processor holding the locked

copy does not become a bottleneck even if all other processors try to access the locked variable because the data management strategy running on each access tree guarantees that at most one request for a variable is blocked at each endpoint of a tree edge.

For the fixed home strategy, the locking mechanism described above is implemented on P-ary access trees. Of course, here bottlenecks can occur if all processors try to lock the same variable at the same time. Nevertheless, here our locking mechanism is promising, too, because a request for locking a variable already creates an exclusive copy on the requesting node, which will probably issue a write request in one of the next steps.

The hardware platform. All of our experiments have been done on the Parsytec GCel. The advantage of this system is that it allows to scale up to 1024 processors. The GCel is a distributed memory computer based on the Transputer T805. Its nodes are connected by a 32×32 mesh network. The system can be partitioned into submeshes of arbitrary size. The used routing mechanism is a wormhole router that transmitts the messages along dimension-by-dimension order paths as we have assumed in our theoretical analysis.

The GCel has the following major characteristics. We have measured a maximum link bandwidth of about 1 Kbyte/sec.. This bandwidth can be achieved in both directions of a link almost independently of the data transfer in the other direction. However, fairly large messages of about 1 Kbyte have to be transmitted to achieve this high bandwidth. The processor speed is about 0.29 integer additions a micro sec., which we have measured by adding a constant term to all entries in a vector of 1000 integers ($\widehat{=}$ 4 bytes). According to these values, the ratio between the link and the processor speed is about 0.86.

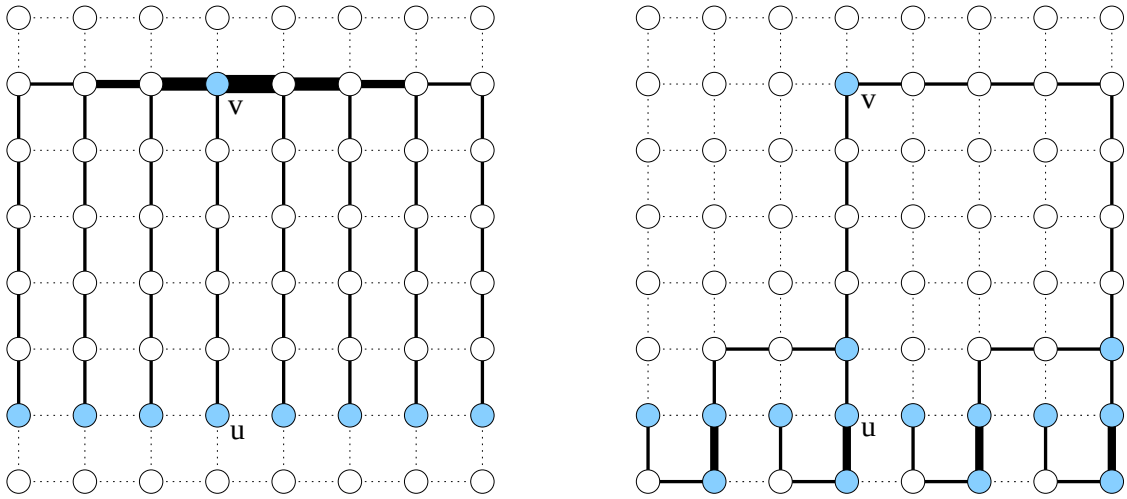
The results on the congestion of different data management strategies for different applications given in the following are, of course, independent from hardware characteristica like link bandwidth and processor speed. They hold for any mesh connected parallel computer of the same size which routes the messages along dimension-by-dimension order paths. The measured execution times heavily depend on the characteristics of the underlying hardware. However, we believe that the results regarding the efficiency of different data management strategies hold for other machines with similar ratios between link and processor speed, too.

1.8.1 Matrix multiplication

The first application is an algorithm for multiplying matrices. We have decided to implement the matrix square $A := A \cdot A$ rather than general matrix multiplication $C := A \cdot B$ because the matrix square requires the data management strategy to create and invalidate copies of the matrix entries whereas the general matrix multiplication does not require the invalidation of copies. Note that the invalidation makes the problem more difficult only for the dynamic data management strategies but not for a hand-optimized message passing strategy.

For simplicity, we assume that the size of the mesh is $\sqrt{P} \times \sqrt{P}$ and that the size of the matrix is $n \times n$, where n is a multiple of $\sqrt{P}$. Let $p_{i,j}$ denote the processor in row i and column j, for $0 \leq i, j < \sqrt{P}$. The matrix is partitioned into P equally sized blocks $A_{i,j}$ such that block $A_{i,j}$ includes all entries in row i' and column j' with $i \cdot n/\sqrt{P} \leq i' < (i+1) \cdot n/\sqrt{P}$ and $j \cdot n/\sqrt{P} \leq j' < (j+1) \cdot n/\sqrt{P}$, for $0 \leq i, j < \sqrt{P}$. The block size is $m = n^2/P$. Processor $P_{i,j}$ has to compute the value of "its" block $A_{i,j}$, i.e., $A_{i,j} := \sum_{k=0}^{\sqrt{P}-1} A_{i,k} \cdot A_{k,j}$.

Each block $A_{i,j}$ is represented by a global variable $A[i, j]$. We assume that $A[i, j]$ has been initialized by processor $p_{i,j}$ such that the only copy of this variable is already stored in the cache of this processor. At the end of the execution, the copies are left in the same configuration. Hence, we measure the matrix algorithm



(a) (b)

Figure 1.7: *The pictures show the data flow induced by distributing the copies for a single data block in a row of the mesh during the read phase. Picture (a) shows the data flow for the fixed home strategy. Picture (b) shows the data flow for the access tree strategy. The width of a line represents the number of copies transmitted along the respective edge. Node u represents the node that is responsible for computing the new value of the data block. Node v denotes the randomly selected fixed home or the randomly selected root of the access tree, respectively. At the beginning of the read phase, the node u holds the only copy of the data block. At the end of the read phase, each of the colored nodes holds a copy of the data block.*

as if it is applied repeatedly in order to compute a higher power of a matrix. The matrix multiplication algorithm works as follows.

The processors execute the following program in parallel. At the beginning, each processor $p_{i,j}$ initializes a local data block H to 0. Then the processors execute a "read phase" and a "write phase" that are separated by a barrier synchronization. The *read phase* consists of $\sqrt{P}$ steps: In step $0 \leq k' < \sqrt{P}$, processor $p_{i,j}$ reads $A[i,k]$ and $A[k,j]$, where $k = (k' + i + j) \bmod \sqrt{P}$, then computes $A_{i,k} \cdot A_{k,j}$, and adds this product to H. Note that the definition of k yields a *staggered* execution, i.e., at most two processors read the same block in the same time step. In the subsequent *write phase*, each processor $p_{i,j}$ writes its local data block H into the global variable $A[i,j]$.

Communication patterns of different data management strategies

The *hand-optimized* strategy works as follows. Each processor $p_{i,j}$ sends its block $A[i,j]$ along its row and its column, that is, the block is sent simultaneously along the four shortest paths from processor $p_{i,j}$ to the processors $p_{i,0}$, $p_{i,\sqrt{P}-1}$, $p_{0,j}$, and $p_{\sqrt{P}-1,j}$. Each processor which is passed by the block creates a local copy. The algorithm finishes when all copies of all blocks are distributed. Obviously, this strategy achieves minimal total communication load and minimal congestion. The congestion is $m \cdot \sqrt{P}$.

Next we consider the communication pattern of the 4-ary access tree and the fixed home strategy. In the read phase both strategies distribute copies of each block $A_{i,j}$ in row i and column j. In the write phase, both strategies send only small invalidation messages to the nodes that hold the copies that have been created in the read phase. Obviously, for large data blocks, the communication load produced in the read phase clearly dominates the communication load produced in the write phase.

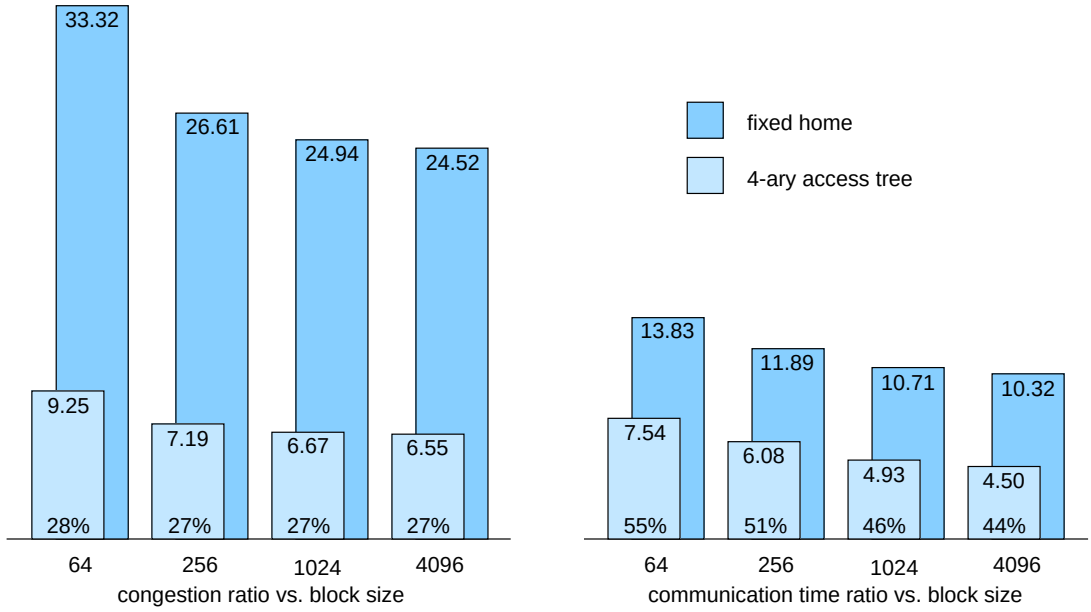


Figure 1.8: *The graphics represent measured congestion and execution time ratios for matrix multiplication on a* 16×16 *mesh.*

Let us consider the communication pattern of the read phase in more detail. Figure 1.7 depicts how the copies are distributed among the processors in a row. (The distribution of copies in a column is similar.) The fixed home strategy sends a copy of variable $A[i, j]$ from node $p_{i,j}$ to the randomly selected home of the variable and then $2\sqrt{P}-2$ copies from the fixed home to each node that is in the same row or column as node $p_{i,j}$. The expected total communication load produced by the fixed home strategy is $\Theta(m \cdot P)$ for the read accesses directed to a single block. The 4-ary access tree strategy distributes block $A_{i,j}$ along a 2-ary subtree of the access tree to all nodes in row i and to all nodes in column j, respectively. This yields an expected total communication load of $\Theta(m \cdot \sqrt{P} \cdot \log P)$ for the read accesses directed to a single block.

Summing up over all blocks yields that the expected total communication load produced by the fixed home strategy is $\Theta(m \cdot P^2)$ whereas the access tree strategy only produces a load of $\Theta(m \cdot P^{3/2} \cdot \log P)$. Under the assumption that the load is distributed relatively evenly among all edges we get that the congestion of the fixed home strategy is $\Theta(m \cdot P)$ whereas the congestion of the access tree strategy is $\Theta(m \cdot \sqrt{P} \cdot \log P)$. Thus, the congestion produced by the fixed home strategy deviates by a factor of $\Theta(\sqrt{P})$ from the optimal congestion whereas the congestion produced by the access tree strategy deviates only by a logarithmic factor from the optimum.

1.8.1.1 Experimental results

First, the different data management strategies are compared for a fixed numbers of processors. We execute the matrix multiplication for different block sizes ranging from 64 to 4096 integers on a 16×16 submesh of the GCel. Figure 1.8 depicts the results of a comparative study of the fixed home and the 4-ary access tree strategy, on the one hand, and the hand-optimized message passing strategy, on the other hand.

The *congestion ratio* of the fixed home and the access tree strategy are defined to be the congestion produced by the respective strategy divided by the congestion of the hand-optimized strategy. The hand-optimized strategy achieves minimal congestion growing linear in the block size. The congestion ratios

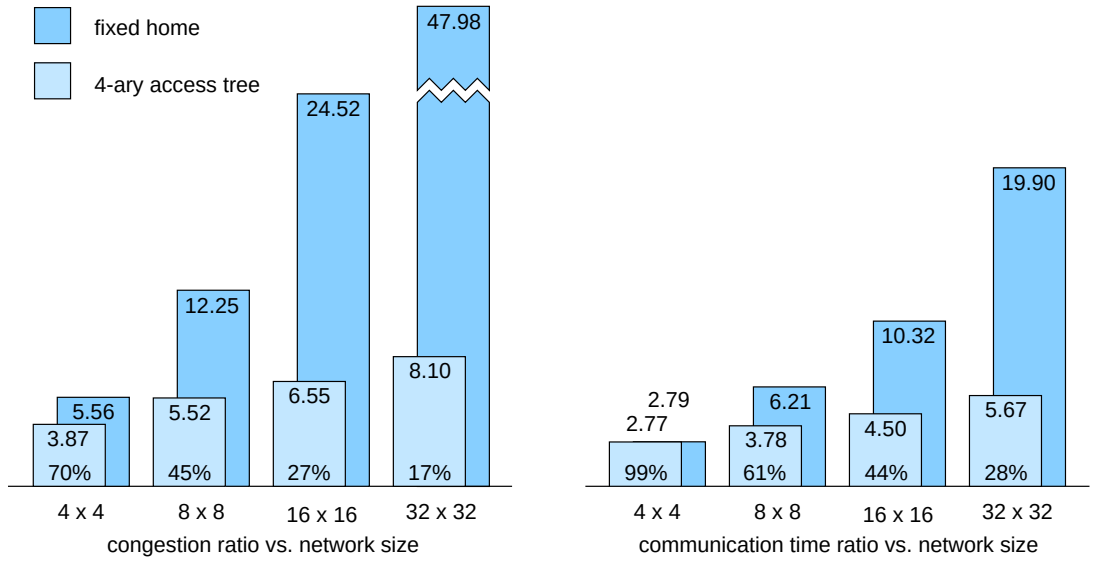


Figure 1.9: *The graphics represent measured congestion and communication time ratios for the matrix multiplication with a fixed block size of 4096 on meshes of different size.*

of the dynamic data management strategies decrease slightly with the block size because read request and invalidation messages become less important when the block size is increased.

The *communication time* is defined to be the time needed for serving all read and write requests and executing the barrier synchronizations. The *communication time ratio* relates the communication time taken by the fixed home and the access tree strategy to the communication time taken by the hand-optimized strategy. In order to measure the communication time, we have simply removed the code for local computations from the parallel program. Hence, only the read, write, and synchronization calls remain. The reason for measuring the communication time rather than the execution time is that the time taken for the multiplication of large matrices is clearly dominated by the time needed for local computations, especially, for computing the products of matrix blocks. For example, using the hand-optimized strategy, the fraction of local computations for matrices with blocks of 4096 integers is about 95 %.

The communication times of all tested strategies grow almost linearly in the block size. Interestingly, the time ratios are smaller than the congestion ratios. The reason for this phenomenon is that a large portion of the execution time of the hand-optimized strategy can be ascribed to the startup cost because this strategy only sends messages between neighbored nodes. The number of startups of the hand-optimized strategy is about $2 \cdot \sqrt{P}$ per node, where P denotes the number of processors. For the other two strategies, let us only consider the startups of those messages including the program data because their startup cost are a lot larger than the startup cost for small control messages. The average number of these startups of the fixed home strategy is about $2 \cdot \sqrt{P}$ per node, which corresponds to the hand-optimized strategy. The average number of startups of the access tree strategy, however, is about $4 \cdot \sqrt{P}$ per node because it sends the messages along 2-ary multicast trees. Therefore, the time ratio of the fixed home strategy improves more on the congestion ratio than the one of the access tree strategy. Nevertheless, the access tree strategy is about a factor of 2 faster than the fixed home strategy.

Next we scale over the network size. We run the matrix multiplication for a fixed block size on networks of size 4×4, 8×8, 16×16, and 32×32. Figure 1.8 illustrates the results. The congestion of the hand-optimized strategy grows linearly in the side length of the mesh. The analysis of the access pattern

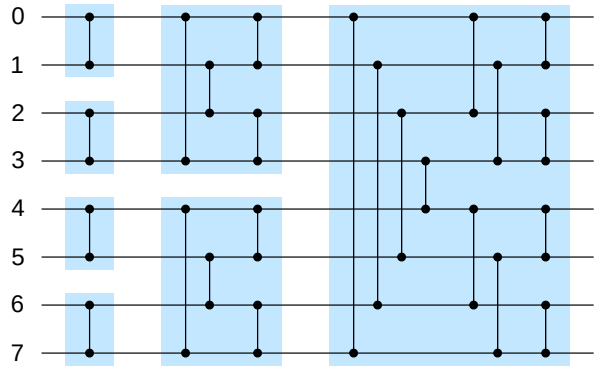


Figure 1.10: *The bitonic sorting circuit for $P = 8$.*

of the access tree and the fixed home strategy promised that the congestion ratios of the two strategies are of order $\log P$ and $\sqrt{P}$, respectively. The increase in the measured congestion ratios corresponds to this assertion. The communication times behave in a similar fashion. As a result, the advantage of the access tree strategy over the fixed home strategy increases with the network size. In the case of 1024 processors, the access tree strategy is more than 3 times faster than the fixed home strategy.

We have also measured the congestion for the 2-ary, the 2-4-ary, the 4-16-ary and the 16-ary access tree strategies. In general, the smaller the degree of the access tree, the smaller the congestion. However, the 4-ary access tree strategy achieves the best communication and execution times because it chooses the best compromise between minimizing the congestion and minimizing the number of startups.

1.8.2 Bitonic sorting

We have implemented a variant of Batcher's bitonic sorting algorithm [13]. Our implementation is based on a sorting circuit. Figure 1.10 gives an example of the sorting circuit for $P = 8$ processors. A processor simulates a single wire in each step. Each step consists of a simultaneous execution of a specified set of comparators. A comparator $[a : b]$ connects two wires a and b each holding a key and performs a compare-exchange operation, i.e., the maximum is sent to b and the minimum to a. The number of parallel steps is the *depth* of the circuit.

The bitonic sorting circuit described in Figure 1.10 is a slight variant of a circuit introduced by Knuth [39], which, in contrast to Knuth's proposal, only uses *standard comparators*, i.e., $a \leq b$ for each comparator $[a : b]$. In our implementations, each processor holds a set of m keys rather than only a single key, and we replace the original compare-exchange operation by a *merge&split operation*, i.e., the processor that has to receive the minimum gets the lower m keys, the other one gets the upper m keys. Several previous experimental studies have shown that bitonic sort is a well suited algorithm for a relatively small number of keys per processor [17, 21, 24, 82].

For simplicity, we assume that the size of the mesh is $\sqrt{P} \times \sqrt{P}$, and that $\sqrt{P} = 2^d$ with d denoting an arbitrary integer. For simplicity, we assume that P is a power of 2. A processor in row $x = x_{d-1} \ldots x_0$ and column $y = y_{d-1} \ldots y_0$ is assigned the unique ident-number $pid = x_{d-1}y_{d-1} \ldots x_2y_2x_1y_1x_0y_0$. This assignment numbers the leaves of each access tree from left to right.

Initially, each processor is assigned a global variable $M[pid]$, which includes "its" m keys. We assume that variable $M[pid]$ has been initialized by processor pid such that the only copy of this variable is in the cache of this processor. Note that the sorting algorithm leaves the copies in the same configuration, too.

Hence, we consider a situation in which the sorting algorithm is used repeatedly as a subroutine in another parallel program.

At the beginning of the algorithm each processor *pid* reads "its" keys from the global variable $M[pid]$ and sorts them locally with a fast algorithm, i.e., the quicksort routine from [71]. Then the sorted list of keys is written back to the variable $M[pid]$. Afterwards, the following program is executed by all processors in parallel.

```
for i := 1 to log(P) do
    δ := 2^i;
    buddy := (pid div δ + 1) · δ − pid mod δ − 1;
    if pid mod δ < δ/2 then
        merge&split(pid, buddy, low)                    /* phase i, step 1 */
    else
        merge&split(pid, buddy, high);                  /* phase i, step 1 */
    for j := 2 to i do
        κ := 2^(i−j);
        if pid mod (2 · κ) < κ then
            merge&split(pid, pid + κ, low)              /* phase i, step j */
        else
            merge&split(pid, pid − κ, high);            /* phase i, step j */
```

It remains to describe the *merge&split* routine. Suppose processor *pid* calls $merge\&split(pid_1, pid_2, pos)$ in step j of phase i. Then processor *pid* executes the following operations. At first, the processor issues a barrier synchronization call. Then processor *pid* reads the keys from $M[pid_1]$ and $M[pid_2]$ and issues another barrier synchronization call. Afterwards, the processor computes its new keys, that is, if $pos = low$ then it computes the m lower keys among the the keys from $M[pid_1]$ and $M[pid_2]$, and if $pos = high$ then it computes the m upper keys among these keys. Finally, processor *pid* writes these keys in a sorted order to $M[pid]$. After all processors have executed the above program, the keys can be found in a sorted order in the global variables $M[0], \ldots, M[P-1]$.

Locality in the algorithm

The algorithm consists of $\log P$ phases such that phase i, for $1 \leq i \leq \log P$ consists of i parallel *merge&split* steps. The *merge&split* steps of phase i implement $2^{\log P - i}$ parallel and independent *merging circuits* each of which has depth i and covers 2^i neighbored wires. The merging circuits are marked in grey in Figure 1.10. The arrangement of the mergers includes locality. Further the merging circuits include locality themselves. Both kinds of locality are exploited by the access tree strategy. This is explained in the following.

For a processor *pid* and an integer k with $0 \leq k \leq \log P$, we define the *k-neighborhood* of *pid* to include all processors whose highest $\log P - k$ bits of the ident-number are equivalent to the corresponding bits of *pid*. The neighborhood sets reflect the locality according to the access tree topology, that is, two processors that are included in the k-neighborhood of each other have distance smaller than or equal to k in the 4-ary access tree. All wires of a merging circuit of phase i are represented by processors that belong to the same i-neighborhood. In addition to the locality in the arrangement of the merging circuit,

locality can be found inside each merging circuit, that is, any pair of processors that communicate in step j of phase i are included in the same $(i-j+1)$-neighborhood. (In order to exploit this locality for the barrier synchronizations, too, we only synchronize subgroups of processors in the *merge&split* steps, that is, in step j of phase i a processor only synchronizes with the processors in its $(i-j+1)$-neighborhood.) We will show that the access tree strategy takes great advantage from the locality whereas the fixed home strategy does not take any advantage from this locality (apart from the local barrier synchronizations, which we also use for this strategy).

First, we consider the fixed home strategy. In each step, each processor *pid* reads a variable that is stored in its own cache and a variable that is stored in the cache of another processor. The first read access is served locally, whereas the second one induces the following communication overhead. Processor *pid* has to send a request message to the fixed home of the variable, the fixed home sends a request message to the processor holding the copy, this processor sends the requested data to the fixed home, and finally the fixed home sends the data to processor *pid*. The write access at the end of the step is served in a similar fashion but only requiress the sending of small invalidation messages. It is easy to check that the average load per edge induced by the fixed home strategy in a step as described above is $\Theta(m \cdot \sqrt{P})$. Thus, the average load over all steps is $\Theta(m \cdot \sqrt{P} \cdot \log^2 P)$, which deviates by a factor of $\Theta(\log^2 P)$ from the minimal congestion of $\Theta(m \cdot \sqrt{P})$ that can be obtained for the given embedding of the bitonic sorting circuit.

When the 2-ary access tree strategy is used then most parallel *merge&split* steps take place in small, independent subgrids. The communication of a processor in step j of phase i only uses edges in the smallest subgrid according to the decomposition hierarchy that includes the $(i-j+1)$-neighborhood of processor *pid*. The side lengths of this subgrid are about $2^{(i-j+1)/2}$ such that the length of the routing path from processor *pid* to its *merge&split* partner is in $O(2^{(i-j)/2})$. Hence, the average communication load per edge in step j of phase i is $O(m \cdot 2^{(i-j)/2})$. Summing up over all these steps yields that the average load over all steps is

$$\sum_{i=1}^{\log P} \sum_{j=1}^{i} O\left(m \cdot 2^{(i-j)/2}\right) \quad = \quad O\left(m \cdot \sqrt{P}\right) \quad . \tag{1.1}$$

If we assume that the 2-ary access tree strategy distributes the load relatively evenly among all edges, which is a reasonable assumption, then it achieves asymptotically optimal congestion. The same asymptotic results hold also for access trees with other constant degrees. However, the 2-ary access tree is the most promising as the 2-ary mesh decomposition corresponds exactly to the locality of the bitonic sorting circuit. The following experimental results will give an insight into how large the constant terms in the congestion bounds are and what the impact of the congestion on the execution time is.

Experimental results

We compare the access tree and the fixed home strategy to a hand-optimized message passing strategy. The hand-optimized strategy simply exchanges two messages between every pair of nodes jointly computing a *merge&split* operation. Each of these messages is sent along the dimension-by-dimension order path connecting the respective nodes. This simple message passing strategy achieves optimal congestion for the used embedding of the bitonic sorting circuit into the mesh.

Analogously to the experimental studies of the matrix multiplication investigated in the previous section, we use ratios that relate the congestion produced and the time taken by the dynamic data management strategies to the respective values of the hand-optimized strategy. However, we consider the execution time rather than the communication time as the time spent in local computations during the execution of the sorting algorithm is very limited. (If the number of keys is sufficiently large in

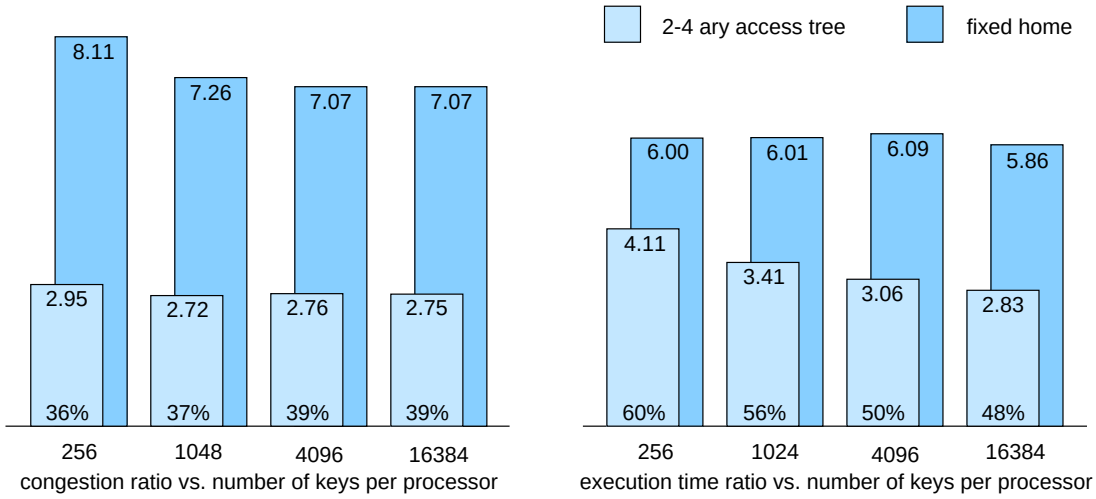


Figure 1.11: *The graphics represent measured congestion and execution time ratios for bitonic sorting on a* 16×16 *mesh.*

comparison to the number of processors then the time needed for the initial sorting of each processor's keys dominates the execution time. However, the parameter configurations we will investigate are far below this threshold.)

Figure 1.11 shows the congestion and the execution time ratios measured on a 16×16 mesh for different numbers of keys per processor. The congestion values of all strategies increase linearly in the number of keys. The congestion ratios of the fixed home and the access tree strategy are slightly decreasing because control messages, i.e., request, invalidation, and acknowledgment messages, become less important when the data messages become larger. The measured execution times of all strategies behave almost linearly, too. The measured ratios show that the execution time is closely related to the congestion. Especially, the execution time ratios for large numbers of keys are amazingly close to the congestion ratios.

In contrast to the results for the matrix multiplication, the 2-ary and the 2-4-ary access tree strategy perform slightly better than the 4-ary strategy. The improvements upon the 4-ary strategy are about 5 % and 8 %, respectively. An explanation for this phenomenon is that the topology of 2-ary access tree matches best the locality in the bitonic sorting circuit. The 2-4-ary access tree strategy improves additionally because of the smaller number of startups in comparison to the 2-ary tree. A further difference to the results for the matrix multiplication is that the time ratio for the access tree strategy is larger than the congestion ratio. The reason for the larger time ratio is that the number of startups of the access tree strategy is much larger than the one of the hand-optimized strategy.

Figure 1.12 illustrates the behavior of the strategies when scaling over the number of processors. The analysis of the locality in the bitonic sorting algorithm predicted that the congestion ratio of the fixed home strategy is of order $\log^2 P$ whereas the congestion ratio of the access tree strategy is in $O(1)$. The ratio of the fixed home strategy behaves as expected. On first view, the measured increase of the congestion ratio seems to be in contradiction to the results of the analysis. However, the measured increase just reflects the increase of the geometric sum given in Equation 1.1. Hence, we can expect that the congestion ratio of the access tree strategy converges against a constant term close to 3.

The measured ratios on the execution time follow the course of the congestion ratios. Therefore, we can conclude that that the execution time of the access tree strategy deviates by a constant factor of about 3

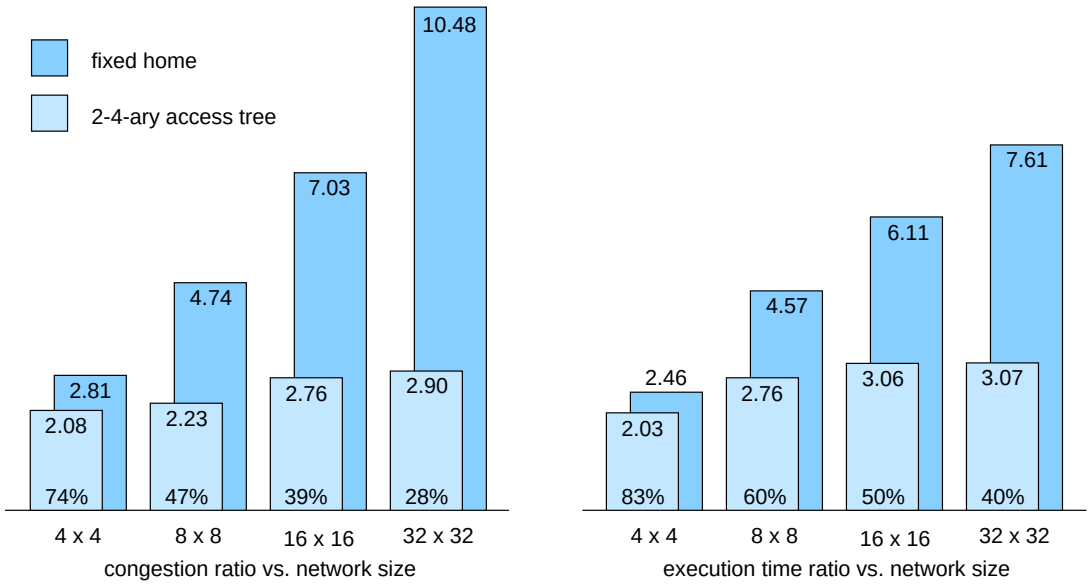


Figure 1.12: *The graphics represent measured congestion and communication time ratios for the bitonic sorting with 4096 keys per processor.*

from the hand-optimized strategy whereas the fixed home strategy loses more and more with an increasing number of processors.

1.8.3 Barnes-Hut N-body simulation

This application simulates the evolution of a system of bodies under the influence of gravitational forces. We have taken an implementation of the Barnes-Hut algorithm [9] from the SPLASH II benchmark [75, 86], and adapted the program code to our DIVA library.

The program implements a classical gravitational N-body simulation, in which every body is modeled as a point of mass exerting forces on all other bodies in the system. The simulation proceeds over time-steps, each step computing the force on every body and thereby updating that body's position and other attributes. By far the greatest fraction of the sequential execution time is spent in the force computation phase. If all pairwise forces are computed directly, this has a time complexity of $O(N^2)$. Since an $O(N^2)$ complexity makes simulating large systems impractical, hierarchical tree-based methods have been developed that reduce the complexity to $O(N \log N)$.

The Barnes-Hut algorithm is based on a hierarchical octree representation of space in three dimensions. The root of this tree represents a space cell containing all bodies in the system. The tree is built by adding particles into the initially empty root cell, and subdividing a cell into its eight children as soon as it contains more than a single body. The result is a tree whose internal nodes are cells and whose leaves are individual bodies. Empty cells resulting from a cell subdivision are ignored. The tree (and the Barnes-Hut algorithm) is therefore adaptive in that it extends to more levels in regions that have high particle densities.

The tree is traversed once per body to compute the force acting on that body. The force-calculation algorithm for a body starts at the root of the tree and conducts the following test recursively for every cell it visits. If the center of mass of the cell is far enough away from the body, the entire subtree under that cell is approximated by a single particle at the center of mass of the cell, and the force exerted by this center

of mass on the body is computed. If, however, the center of mass is not far enough away, the cell must be "opened" and each of its subcells visited. A cell is determined to be far enough away if the following condition is satisfied:

$$d \cdot \vartheta > \ell \ ,$$

where ℓ is the length of a side of the cell, d is the distance from the body to that cell's center, and ϑ is a user-defined accuracy parameter (ϑ is usually between 0.3 and 1.2, see [74]). In this way, a body traverses those parts of the tree deeper down which represent space that is physically close to it, and groups distant bodies at a hierarchy of length scales.

The main data structure in the application is the Barnes-Hut tree. Since the tree changes in every time-step, we use pointers such that the tree can be reconstructed in every time-step. Each cell and each body is represented by a global variable. The variables for the bodies and the cells have a size of 88 and 194 bytes, respectively. Among other information, every cell has pointers to its children.

Initially, each processor holds about an equal number of bodies, each of which having a fixed position and velocity. These values are updated over a fixed number of time steps representing a physical time period of length Δt. Each time step is computed within 6 phases:

1. load the bodies into the tree;
2. upward pass through the tree to find center-of-mass of cells;
3. partition the bodies among the processors;
4. compute the forces on all bodies;
5. advance the body positions and velocities;
6. compute the new size of space.

Each of the phases within a time-step is executed in parallel, and the phases are separated by barrier synchronizations. The tree building phase (Phase 1) and the center-of-mass computation phase (Phase 2) require further synchronization by locks since different processors might try to simultaneously modify the same part of the tree.

The parallelism in most of the phases is across the bodies, that is, each processor is assigned a subset of the bodies, and the processor is responsible for computing the new positions and velocities of these bodies. The force computation phase (Phase 4) needs almost all the sequential computation time (about 90 %, see [75]). Therefore, the load balancing is obtained by counting the work that has to be done for a body in the force computation phase within a time step, and using this work count as measure of the work associated to that body. Each processor is assigned about the same amount of work.

For the load balancing, we use a *costzones partitioning scheme*: The Barnes-Hut tree is conceptually laid out in a two-dimensional plane, with a cell's children laid out from left to right in increasing order of child number. The cost of every body is profiled and stored with it, it corresponds to the number of bodies and cells that had to be opened for the force calculation in the last time step. (This methods works well because the system evolves slowly and the body distribution does not change too much between successive time steps.) As a result of Phase 2, a cell stores the total work associated with all the bodies it contains. The total work in the system is divided among processors so that every processor has a contiguous, equal range or zone of work. For example, a total work of 1000 units would be split among 10 processors so that zone 1–100 units is assigned to the first processor, zone 101–200 units to the second, and so on. Which cost zone a body in the tree belongs to is determined by the total cost of an inorder traversal of the tree up to that body. In Phase 3, the processors traverse the tree in parallel and pick up the bodies that belong in

their costzone. Hence, each processor is responsible for a contiguous interval of leaves of the Barnes-Hut tree.

Locality in the application

The costzones partitioning scheme yields a very good load balance. Ideally, the resulting partitions correspond to physical regions that are spatially contiguous and equally sized in all directions. Such partitions maintain the "physical locality" and, therefore, minimize interprocessor communication and maximize data re-use. This is of particular interest for the dominating force computation phase in which each processor has to open an expanded path from the root to each of its bodies.

The partitions produced by the costzones partitioning scheme are contiguous in the tree, which, however, does not imply that the partitions are also spatial contiguous in the physical space. Sometimes a processor is assigned two or more physical regions. Besides these regions are not equally sized in all directions. However, if the number of bodies is much larger than the number of cells then the processors mostly pick up large cells from upper levels of the tree such that enough locality is obtained within each of these cells.

Furthermore, the costzones partitioning scheme is not only able to exploit the spatial locality but it also translates physical locality into topological locality. Analogously, to the sorting algorithm we use processor ident-numbers that correspond to a numbering of the leaves of the mesh-decomposition tree from left to right. (Recall that the mesh-decomposition tree corresponds to all superimposed access trees.) Thus, the costzone partitioning yields a locality preserving mapping of the leaves of the Barnes-Hut tree to the leaves of the mesh-decomposition tree. As a consequence, bodies that are close together in the physical space have a tendency to be computed by the same processor or by two processor that are close together with respect to the distances defined by the mesh-decomposition tree.

Experimental results

First, we consider a fixed number of processors $P = 256$ and scale over the number of bodies N from 10,000 to 60,000. The Barnes-Hut algorithm is executed on a 16×16 submesh of the GCel. We use the scaling rule proposed in [74]: If the number of bodies N is scaled by a factor of s, the time step duration Δt and the accuracy parameter ϑ must each be scaled by a factor of $1/\sqrt[4]{s}$. The idea behind these scaling rules is that all sources of error should be scaled so that their error contribution are about equal. We set

$$\Delta t \;=\; 0.25/\sqrt[4]{N} \;\text{ and }\; \vartheta \;=\; 7.2/\sqrt[4]{N}\;.$$

Hence, Δt ranges between 0.025 for 10,000 bodies and 0.016 for 60,000 bodies, ϑ ranges between 0.72 and 0.46. The decrease of Δt leads to an increase in the execution time as the number of time steps has to be increased in order to simulate a given period of time. As we will run simulations over a fixed number of time steps, the change of Δt does not have a great influence on our results. The decrease of ϑ, however, directly influences the execution time needed for each individual time step. Physicists usually simulate hundreds or thousand of time steps. Since the execution times for the individual time steps are already relatively stable after the simulation of the first two steps, we only simulate 7 time steps, from which only the last 5 are included in the measurement.

Figure 1.13 describes the congestion and the execution time of the measured rounds. The measured congestion corresponds to the maximum number of messages that have traversed the same edge during the execution of the 5 rounds, and the execution time corresponds to the total time needed for the 5 rounds in milliseconds. The slightly super-linear increase of the execution time is due to the additional scaling

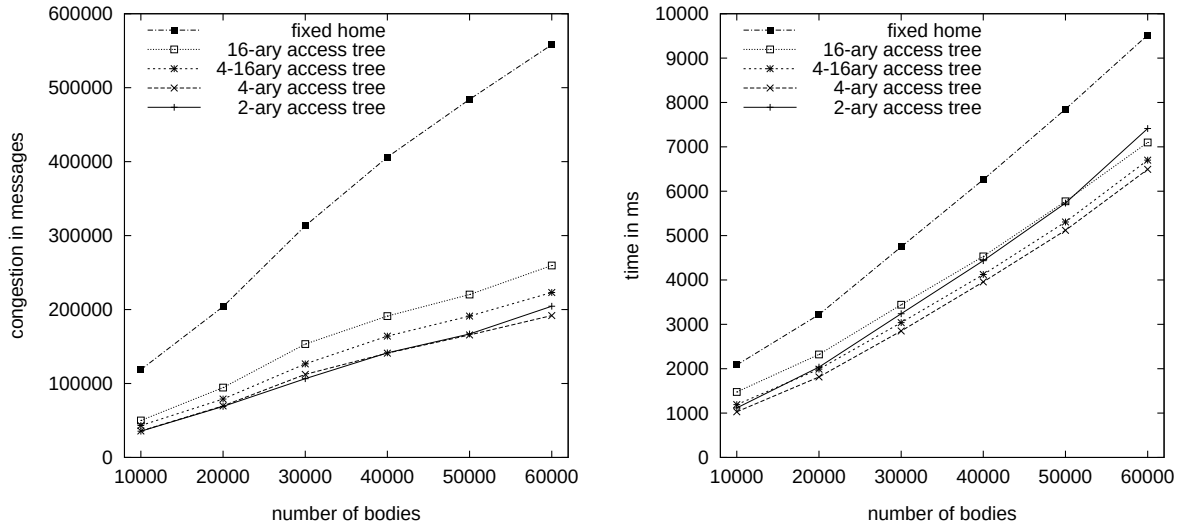


Figure 1.13: *Congestion and execution time for the Barnes-Hut N-body simulation.*

of the accuracy parameter ϑ as described above, which, however, mainly influences the local computation time, only.

A comparison between the access tree strategies and the fixed home strategy clearly shows that the access tree strategies are able to exploit the topological locality in the Barnes-Hut algorithm. In general, the higher the access tree is, the smaller is the congestion. (The increase of the congestion for the 2-ary access tree from 50,000 to 60,000 bodies is due to copy replacement, which starts at 60,000 bodies for the 2-ary access tree strategy. All other strategies do not have to replace copies as the storage capacities are sufficient for these strategies.) However, because of the large overhead due to additional startups, the 2-ary strategy is clearly slower than the 4-ary although it has a very small congestion. As a result, the 4-ary strategy performs best among all strategies.

The order of the different access tree strategies concerning the execution time changes slightly if we run the N-body simulation on meshes that cannot be partitioned properly by a 4-ary decomposition. For example, the decomposition of a 12×16 mesh yields 1×2 submeshes on the decomposition level before final whereas the 4-16-ary decomposition ends with submeshes of size 3×4. Since submeshes of size 1×2 produce more overhead for additional startups than saving for congestion, the 4-16-ary access tree strategy outperforms the 4-ary strategy slightly on the 12×16 mesh.

Most of the execution time of the N-body simulation is spent in the force computation phase (Phase 4), e.g., for 60,000 bodies the 4-ary access tree strategy on the 16×16 mesh spends about 78 % of the execution time in this phase. Another phase which is of special interest for a small number of bodies is the tree building phase (Phase 1). For 10,000 bodies, the fixed home strategy spends about 44 % of its time in this phase. We investigate the tree building and the force computation phases in further detail.

Figure 1.14 shows the congestion and the execution time of the tree building phase. The Barnes-Hut tree is rebuilt in each simulated time step. For each of its bodies, a processor follows a path in the tree leading downwards from the root to a node which does not have a child that represents the region of space to which the body belongs. If this node is a cell then the processor can add its body to this cell as a child. If this node is another body then this body is replaced by a cell and added to this cell as a child. Afterwards, the process of loading the processor's body into the tree is continued. Locks are used in order to avoid different processors simultaneously changing the data of the same body. We use the locking mechanism described in Section 1.8 and attach a lock to each body.

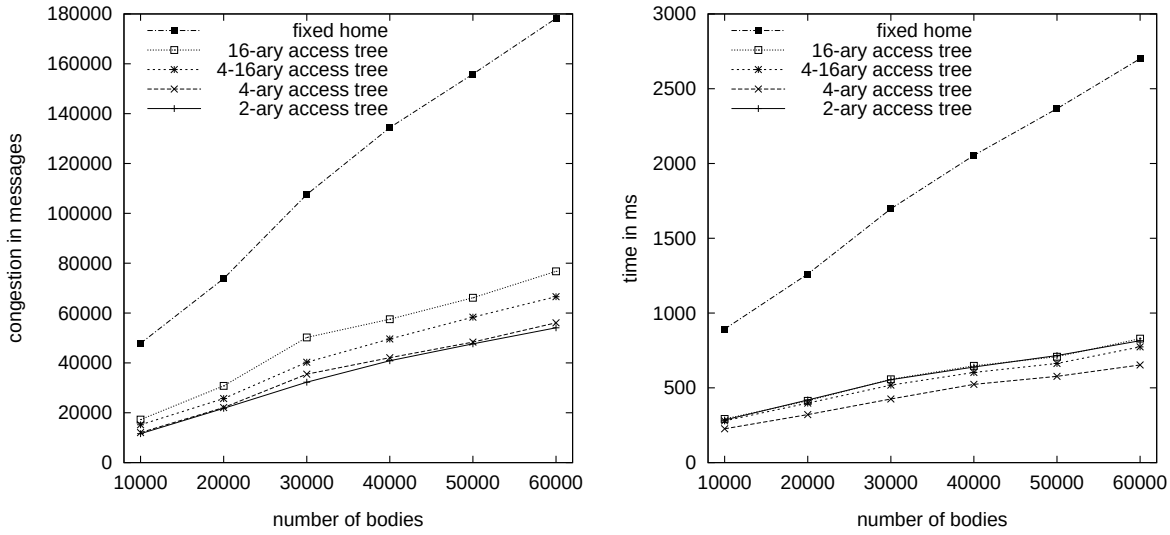


Figure 1.14: *Congestion and execution time of the tree building phase.*

Obviously, the root cell is the bottleneck of the algorithm for loading all the bodies into the Barnes-Hut tree. This cell has to be read once for every body, which, however, is only expensive in the beginning of the phase when several processors contend for locking the root as they want to add a body as a child of the empty root. Later on each processor holds a copy of the root such that reading the root is very cheap. The access tree strategies profit much from their capability to distribute the copy of the root cell very efficiently via a multicast tree, whereas, using the fixed home strategy, one processor, the home of the root, has to deliver a copy of the root to each processor one by one. Similar bottlenecks also occur for other nodes on the top level of the Barnes-Hut tree, resulting in a much higher offset for the congestion of the fixed home strategy. Obviously, this offset increases with the number of processors.

What, however, is the reason for the steeper slope of the fixed home strategy's congestion and execution time? – Interestingly, the difference in the slope does not occur in the simulation of the first time step in which the Barnes-Hut tree is built for the very first time. In the later time steps, each write access to a cell leads to the invalidation of all copies of the cell that has been created in the previous time step. The access tree strategy performs this invalidation more efficiently than the fixed home strategy since it uses multicast trees. A larger number of bodies leads to a larger number of cells whose copies need to be invalidated in the tree building phase, which explains the steeper slope of the fixed home strategy. Most of the copies to be invalidated are created in the force computation phase. This phase is considered next.

Figure 1.15 depicts the congestion and the execution time of the force computation phase added over 5 time steps. Besides the picture shows the time spent in local computations in this phase. For 60,000 bodies, the 4-ary access tree strategy only uses 25 % of the time for communication whereas the fixed-home strategy requires about 33 % for communication. As the force computation phase does not include write accesses to global variables but many read accesses, many copies are created during this phase. These copies have to be invalidated in consecutive phases like the tree building phase of the next step. (The time for the local computation has been measured by running the force computation phase twice, and measuring only the time needed for the second execution in which all required copies are already stored in the local caches.)

In the force computation phase, each processor traverses the Barnes-Hut tree once per body to compute the forces acting on the body. The calculation follows the path from the root to the body and opens several

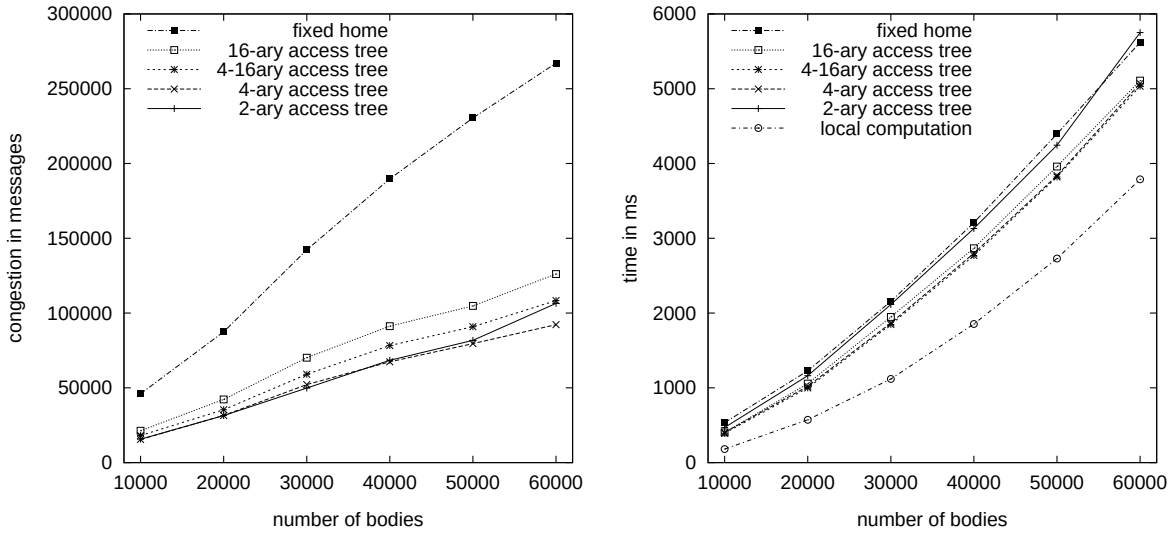


Figure 1.15: *Congestion and execution time of the force computation phase.*

cells and bodies in the neighborhood of this path, where the width of the neighborhood is defined by the parameter ϑ. Due to the locality of the algorithm, the neighborhoods of the paths for a set of bodies that are computed by the same processor overlap to a great amount. This leads to cache hit ratios of about 99 % even for a relatively small number of bodies.

Decreasing ϑ means that the neighborhood of a body's path grows. The curve for the local computation time is super-linear since we decrease ϑ with the number of bodies. The congestion, however, does not increase noticeable in a super-linear fashion as it only grows with the number of cache misses in this phase. This number is determined by the size of the union of the neighborhoods of the paths for all bodies computed by a processor. Interestingly, this union does not seem to grow in a super-linear fashion as the congestion does not either.

In the force computation phase, the variations of the access tree strategy win against the fixed home strategy because of their capability to efficiently distribute copies of a global variable to those submeshes where they are needed. Also in this phase, the 4-ary access tree strategy outperforms all other strategies.

Finally, let us investigate what happens if the number of processors is scaled from 64 to 512. We change the number of processors P and the number of bodies N simultaneously in such a way that $N = 200 \cdot P$. For simplicity, we use a naive scaling, i.e., we set $\vartheta = 2$, and $\Delta t = 0.025$, rather than decreasing this parameters by a factor of $\sqrt[4]{N}$.

Figure 1.16 shows how the congestion and the execution time behave if the number of processors is scaled. The congestion produced by the access tree and the fixed home strategy is mainly determined by the largest side length of the mesh rather than by the number of processors. Obviously, the congestion produced by the access tree strategy grows less than the congestion of the fixed home strategy. The superiority of the access tree strategy against the fixed home strategy increases with the number of processors. For the largest number of processors that we have tested, the ratio of the execution times taken by the access tree and the fixed home strategy is about 49 %. The results on the communication time, i.e., the execution time minus the time taken for local computations in the force computation phase, are even more impressive. The best ratio of the communication time taken by the access tree and the fixed home strategy is about 33 % for 512 processors. Hence, the access tree strategy improves by a factor of 3 on a standard caching strategy using a randomly selected fixed home for each variable.

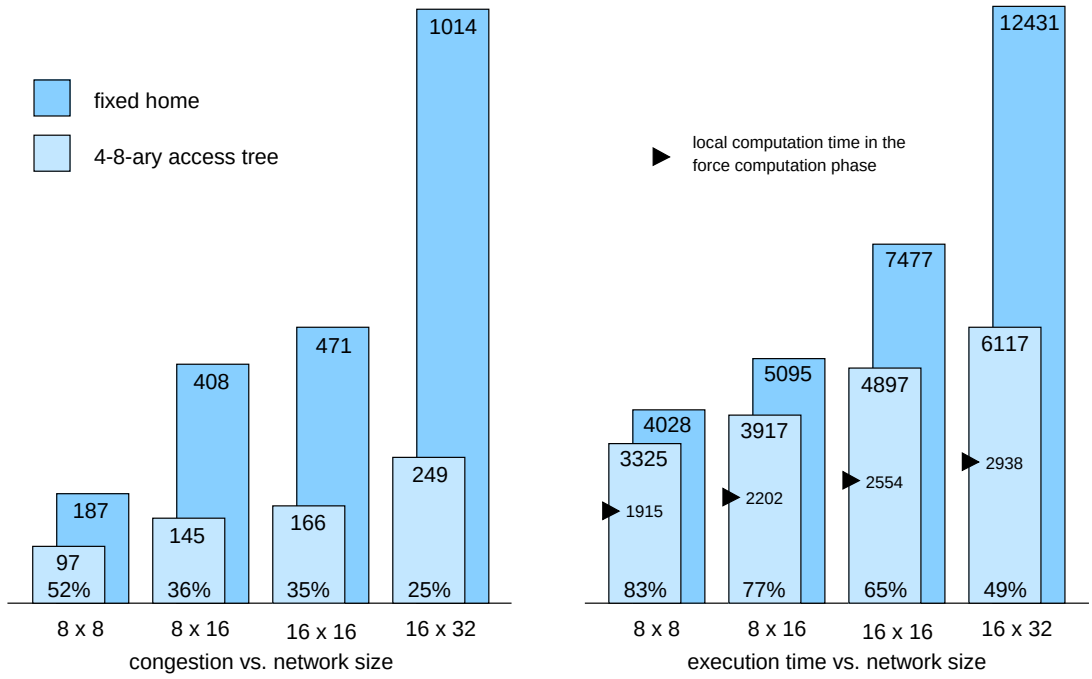


Figure 1.16: *The graphic represents measured values for the Barnes-Hut N-body simulation for different numbers of processors P. The number of bodies is increased with the number of processors, that is, $N = 200 \cdot P$. The congestion is measured in units of 1000 messages. The time is measured in milliseconds.*

1.9 Conclusions

We have presented and analyzed static and dynamic data management strategies for trees, meshes, and clustered networks. All of these strategies aim at minimizing the network congestion in order to minimize the communication overhead and avoid that some of the links or buses become a bottleneck. Our theoretical analyses show that the introduced strategies are able to exploit the locality included in an arbitrary application. We have proven that the congestion produced by the data management strategies is minimal or close to minimal. In experimental studies, we have seen that the execution time and the congestion of a parallel program are closely related so that the data management strategies that we have developed in the theoretical models are well suited for practical use.

The dynamic access tree strategies for meshes and clustered networks are probably of special interest. In contrast to previously known strategies, they give an integrated solution for the problem of data placement, data tracking, and routing. The major advantage of this approach is that it dispenses with the need for control messages except of the inevitable read request messages. A disadvantage is that it requires also large data messages to follow the routing paths described by the access trees. However, sending all messages along the branches of the access trees is not only needed to realize the dynamic distribution of copies and to update all signposts and markers but is also an important property that is needed to guarantee that the congestion of the routing paths is small. Thus, sending large data packages along shortest paths in the mesh in order to shortcut the longer routing paths determined by the access trees holds the danger of an increase of the congestion.

Our experimental results show that the startup cost are a further important cost factor apart from the congestion. We have not incorporated the startup cost in our theoretical model. However, in our practical studies we have reduced the number of startups by increasing the degree of the access trees such that they

become flatter. This decreases the number of startups required to serve the requests but possibly increases the length of some of the routing paths and also the number of startups to be done by individual nodes. On the GCel, 4-ary access trees seem to be most useful. Only the sorting application performs better when using a 2-ary access tree because the pattern of locality in the bitonic sorting circuit corresponds to the 2-ary mesh decomposition. It is an interesting question how to incorporate startup cost in an adequate way in our congestion-based theoretical model.

In contrast to other models for parallel computing, e.g., the BSP [80] and the LogP model [22], the simple congestion model in which we analyzed the data management strategies does not have any notion of time. We have improved on previous models for data management as our models aim to minimize the congestion rather than the total communication load. However, 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 static and dynamic data management 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 data management strategies in a cost model that incorporates some notion of time is a challenging task.

Furthermore, developing static or dynamic strategies that minimize the congestion while obeying memory capacity constraints is a difficult task. Theoretically founded results for this problem are not known. The dynamic access tree strategies for meshes implemented in the DIVA library apply a straightforward replacement mechanism. Its description will illustrate the practical problems arising with memory capacity restrictions. Each node of the mesh dedicates a fixed sized portion of its local memory for caching copies of the data objects. When the capacity of the assigned memory for caching is exceeded than the copies are replaced in least recently used fashion. The replacement mechanism does not distinguish between copies that virtually correspond to different access tree nodes. In order to preserve at least one copy of each data object, the copy which has been updated least recently is excluded from the replacement. The data management system holds with an error if the capacity is exceeded on a node holding only copies excluded from the replacement. Devising a simple, provably efficient data management strategy that uses a more clever replacement mechanism is an interesting open problem.

The access tree strategy can also be efficiently adapted to other networks than meshes and clustered networks. All that is needed is a hierarchical network decomposition that possesses similar properties to the one we have described for meshes. We believe that the following rules yield a good cooking recipe for devising efficient static and dynamic data management strategies for many computer networks:

- Split the network along the bisection cut into two almost equal subnetworks, and apply the same decomposition recursively to the resulting subnetworks. This yields a decomposition tree whose height is logarithmic in the size of the network.
- For each data object use an access tree whose topology corresponds to the decomposition tree. Map each access tree node uniformly at random to a node of the corresponding subnetwork.
- Simulate the 3-competitive tree strategy on each access tree. Choose the routing paths that simulate the access tree edges by a multicommodity flow program as described for the routing inside the clusters of the hierarchically clustered networks.

It is an interesting question what the quality of a data management strategy resulting from the application of the above recipe to an arbitrary network is? Of particular interest is the question of whether or not there is a variation of the bisection cut decomposition described above that yields a strategy achieving close to minimal congestion on arbitrary networks?

Scenario 2:

Contention Resolution on a Scalable Multimedia Data Server

With recent advances in computing and communication technology it has become feasible to provide on-line access to multimedia data such as images, video, audio, 3D graphics, etc. Most multimedia applications require large storage capacities and large data transfer rates such that the data has to be distributed among several storage devices of high capacity, e.g., magnetic or optical disks. Usually, the data of each application is partitioned into many relatively small pieces that are to be retrieved from the disks on demand, e.g., for the real-time playback of a movie or the animation of a virtual walk through a 3D scene.

The major difference between multimedia data and classical data, e.g., textual data, is that the access to multimedia data is subject to real-time constraints. A multimedia data server has to guarantee short response time for every particular request in contrast to a standard file server which usually aims to minimize the throughput or the average response time in consideration of some fairness aspects. The response time that a multimedia server has to guarantee heavily depends on the degree of interactivity of the applications that should be supported. The more interactive the applications supported by the multimedia server are, the stronger are the real-time constraints that the server has to fulfill. For example, the access pattern of highly interactive multimedia applications, such as a walk through a “virtual world”, are less predictable than the pattern of applications allowing only restricted interactivity as, e.g., simple video playback without features such as Pause, Fast Forward, or Rewind. In general, if the access pattern of an application is difficult to predict, then data cannot be retrieved for long time periods in advance so that the server must respond to every request very quickly. Therefore, a data server that supports interactive multimedia applications has to provide short response times and must be able to satisfy arbitrary requests without respect to a particular access pattern.

Usually, the storage devices are the main bottleneck of a data server because building fast devices of high capacity is difficult and expensive. The large storage requirements of multimedia applications do not allow to use expensive DRAMs or SRAMs so that cheaper and slower storage devices such as magnetic or optical disks have to be used. These disks are much too slow to support several users in parallel. Moreover, some multimedia applications consume data at such a high rate that a single disk is not even able to deliver the data for a single user. For example, the access rates of HDTV quality video are about 81 MBytes/sec. whereas current commercial disks can serve only access rates of about 12 MBytes/sec [30, 69]. One way to get higher service rates is to break the multimedia data into several relative small pieces that are

distributed among several disks such that these disks can serve several simultaneous requests or even a single request for a large piece of data in parallel.

A data server that includes several disks in order to support several users needs a high speed interconnection network between the disks and the user ports. If the number of disks and user ports is not too large then the network can consist of a single bus. However, a scalable server architecture requires that the bandwidth of the network grows with the number of disks, which means that a sparse interconnection network is needed. The major problem in the design of such a data server is to devise data management strategies that distribute the load for serving the requests evenly among the disks and links in order to avoid that a single disk or a single network link becomes the bottleneck of the system. One way to address this load balancing problem is to use redundancy, that is, to store each piece of data on several disks, rather than only on one. In case of redundancy k, each data piece is stored on k disks such that each read request can be served by k alternative disks. Write accesses occur usually considerably rarer than read accesses. For this reason, we focus on the read accesses and will consider the write accesses in a special treatment.

When too many requests aim at the same disk or the same link, then a contention resolution mechanism is needed deciding which of the requests gets the respective resource. A request that has lost must switch over to alternative resources, or, if no alternatives are available, the request is rejected. The larger the redundancy, the larger is the number of alternatives, and the less requests have to be rejected. On extreme, a sufficiently large set of disks storing all of the data is assigned to each of the user port. Then any user request can be satisfied regardless of requests of other users. Of course, this solution yields an optimal support for each user, but, because of its enormous storage requirement, the solution is not resource efficient.

What is needed is a data server that allows to satisfy the real-time constraints of multimedia applications with a limited number of resources. We aim to develop a theoretical concept for a scalable data server architecture that satisfies these requirements. We formalize this problem as follows.

2.1 Design objectives and constraints

The data server architecture should be scalable, that is, it should be able to support an arbitrary number of users. For every user, the data server has to provide a *user port* at which the user issues his request. The user ports are connected by a network to a set of memory modules. Each of the *memory modules* represents a disk or a small set of disks that are connected by a link or bus, respectively, to the rest of the network. Regardless whether a memory module represents one or more disks, we view it as a logical storage device that is able to serve only a limited number of requests in parallel. The *interconnection network* is modeled by a graph. Each node in the graph represents either a user port, a memory module, or an internal switch. The edges between the nodes represent the communication links in the network. For scalability reasons we demand that the network is *sparse*, i.e., the maximum node degree in the underlying family of graphs determining the network topology does not depend on the size of the server.

We assume that the data is divided into equally sized *data blocks*. The data blocks have to be placed statically onto the memory modules, possibly with redundancy. When a user issues a request at his user port then the data server has to *serve the request*, i.e., a *routing path* has to be established from the user port to one of the memory modules holding the requested data block. The number of routing paths that are allowed to use the same link at the same time is defined to be the *link bandwidth*. Analogously, the *module bandwidth* denotes the number of requests that can be served by a memory module in parallel, i.e., the number of routing paths that are allowed to end at the same module. In case of redundancy, a contention resolution mechanism decides which request is served by which memory module and which routing path is used. This mechanism is described in a *contention resolution protocol*.

The real-time capability of the data server is investigated in two models. The first model deals with requests that arrive batch-wise whereas the second model assumes that the requests arrive sequentially, one by one.

- A *batch of requests* is a set of requests that arrive nearly at the same time such that the contention resolution protocol can decide simultaneously about all requests in the batch. We assume that a batch includes at most one request from each user port. We distinguish between *batches of distinct requests*, in which each data block is addressed by at most one request, and *batches with data hot spots*, in which the same data block is possibly requested by several users. We allow requests which belong to the same batch and that are directed to the same data block to be *combined*, i.e., the routing paths of these requests are merged to form a *multicast tree* such that each module of bandwidth b can be the root of at most b multicast trees, and each edge of bandwidth B can be included in at most B multicast trees. (This approach is discussed comprehensively in Section 2.6.) We investigate how large the module and the link bandwidths of the proposed data server must be such that the data server is able to serve an arbitrary batch (of distinct or arbitrary requests) *with high probability*, i.e., with probability $1-n^{-\alpha}$, for any constant α.

- A *sequence of requests* models requests that arrive one by one, rather than in large batches. We investigate whether the data server is able to serve an infinite sequence of events $\sigma_1, \sigma_2, \ldots$, where σ_i is either an event requesting a data block or an event releasing a data block. *Serving a request* $\sigma_i = (\rho_i, x_i)$ means that a path between the port ρ_i and a memory module that holds the data block x_i has to be locked until the block is released in a later time step. We assume that the sequence is specified in advance, i.e., before the data blocks are distributed among the modules, such that the sequence does not depend on the behavior of the data server. We allow arbitrary *sequences of distinct requests*, i.e., sequences in which none of the ports and none of the data blocks is included in two requests that overlap in time. The challenging part of this problem is that we assume that the events in the sequence are revealed only one by one, that is, the data server gets the request σ_i at time i, and the contention resolution protocol has to specify the serving disk and the routing path for the request σ_i before the event σ_{i+1} is presented. We investigate how large the disk and the link bandwidths of the proposed data server must be such that the data server is able to serve any individual request of an arbitrary sequence, w.h.p.

We demand that the server architecture is *resource efficient*: Let n denote the number of user ports, and m denote the number of data blocks. The number of memory modules should scale with the number of ports, that is, the server should have $\Theta(n)$ memory modules. The size of the network should be $\Theta(n \cdot \log n)$, which is minimal for an indirect network connecting n ports with $\Theta(n)$ disks, and the size of each memory module should be $\Theta(m/n)$, which limits the redundancy. We are looking for a resource efficient architecture with a contention resolution protocol that can be executed efficiently in a distributed fashion. Our main focus lies on the question of how limited redundancy can help to reduce the module and link bandwidths.

In the next sections, we will describe and analyze a server architecture that fulfills all of these constraints using a redundancy of 2. We will show that the architecture requires only very small link and module bandwidths.

2.2 A new proposal for a scalable data server architecture

We introduce a scalable data server architecture using a "butterfly-like" interconnection network that connects n ports with $2n$ memory modules. For simplicity, we assume that n is a power of 2.

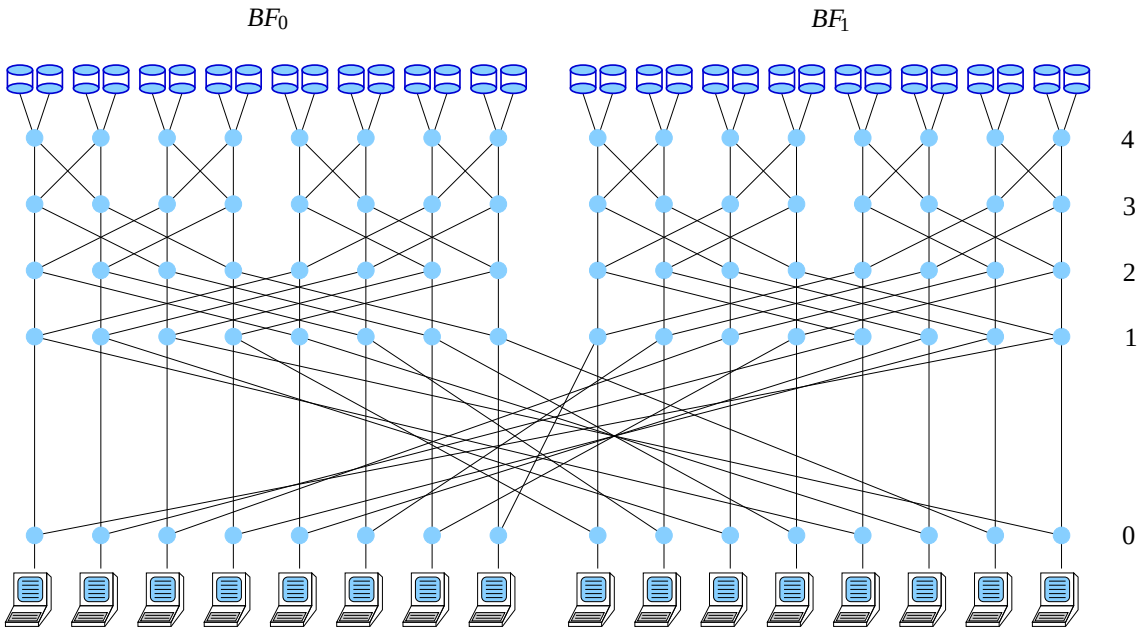


Figure 2.1: *Example for the RB_{16}-server.*

We define a $\log n$ dimensional *butterfly network* as follows. The network has $n(\log n+1)$ nodes arranged in $\log n+1$ *levels* of n nodes each. Each node has a distinct label $\langle \ell, v\rangle$, where ℓ is the level of the node ($0 \le \ell \le \log n$) and $v = v_1 v_2 \ldots v_{\log n}$ is a $\log n$-bit binary number that denotes the *column* of the node. All nodes of the form $\langle \ell, v\rangle$, $0 \le \ell \le \log n$, are said to belong to column v. Two nodes $\langle \ell, v\rangle$ and $\langle \ell', v', \rangle$ are linked by an edge if $\ell' = \ell + 1$ and either v and v' are identical or v and v' only differ in the bit in position ℓ', where the bit positions are numbered 1 through $\log n$. We call the first type of edge a *straight edge* and the second a *cross edge*. The nodes on level 0 are called the *input nodes*, and the nodes on level $\log n$ are called the *output nodes*.

We use a randomized variant of the butterfly network for the interconnection between the user ports and the memory modules. We define a *randomly-wired butterfly* RB_n as follows. Network RB_n has the same set of nodes and edges as the $\log n$-dimensional butterfly network, except that the cross edges incident on the input nodes of RB_n are permuted randomly according to the following rule. Each node $\langle 0, v_1 \ldots v_{\log n}\rangle$ of RB_n is connected to the node $\langle 1, v'_1 \ldots v'_{\log n}\rangle$ by a cross edge if and only if $v_1 \neq v'_1$ and $\pi_{v_1}(v_2 \ldots v_{\log n}) = v'_2 \ldots v'_{\log n}$, where π_0 and π_1 are random permutations of the set of $(\log n - 1)$-bit numbers.

The *RB_n-server* is defined as follows. The server consists of n ports and $2n$ memory modules which are connected by the randomly-wired butterfly RB_n. Each of the input nodes of RB_n corresponds to a user port, and each of the output nodes has two memory modules attached, each of which is connected to its respective output node by an edge. An example for the topology of the RB_n-server with $n = 16$ is shown in Figure 2.1. The set of edges connecting the nodes on level i with the nodes on level $i+1$ is denoted *edge level i*, for $1 \le i < \log n$. Edge level 0 is also called the *randomization level*. The set of edges connecting the memory modules to the nodes on level $\log n$ is defined to be *edge level* $\log n$. The bandwidths of the edges on this level are assumed to be equivalent to the respective module bandwidths.

The data blocks are distributed with redundancy 2 among the memory modules. The placement of the blocks onto the modules is defined as follows. The levels 1 to $\log n$ of RB_n include two disjoint subbutterflies of dimension $\log n - 1$, one of them consists of the nodes $\langle \ell, 0 \ldots\rangle$ and the other consists of

the nodes $\langle \ell, 1 \ldots \rangle$, for $1 \leq \ell \leq \log n$. The first of these two subbutterflies plus the n memory modules that are connected to its $n/2$ output nodes is called BF_0, and the other subbutterfly plus the n memory modules that are connected to its $n/2$ output nodes is called BF_1. Each data block is stored on two memory modules: the first module is chosen uniformly at random from the n modules of BF_0 while the second module is chosen uniformly at random from the n modules of BF_1.

Due to the redundant placement we have two alternative memory modules that can serve each request. The routing path between a user port and a memory module is always chosen to be the unique shortest path that connects the port with the module. Hence, we have also two alternative routing paths for each request, one of which leads through BF_0 whereas the other leads through BF_1. The prior one is called *BF_0-path*, the latter one is called *BF_1-path*. The random wiring of level 0 and the random placement of the blocks onto the disks ensures that the BF_0- and the BF_1-path for each request follow a random and independent course through BF_0 or BF_1, respectively. We investigate how the freedom of choice between these two independent alternatives can help to minimize the module and the link bandwidth.

2.3 Summary of new results

We investigate contention resolution protocols for the RB_n-server architecture. These protocols are natural generalizations of well-known "two-choice" algorithms for PRAM simulations on completely connected networks and balls-and-bins problems. To our knowledge, we present the first analysis of "two-choice" algorithms for the more complicated problem of data placement and routing in a sparse network. Moreover, we give the first analysis of a "two-choice" algorithm for infinite sequences with arbitrary insertions and deletions of distinct requests (or balls).

In Section 2.5, we analyze how much module and link bandwidth is needed in order to serve batches of distinct requests on the RB_n-server. We develop a generalization of the "collision protocol", which originally has been introduced by Dietzfelbinger and Meyer auf der Heide [25] for PRAM simulations on completely connected networks, see next section. Our collision protocol allows to choose different bandwidths for the edges on each level of the RB_n-server. In particular, it allows to choose different bandwidths for the memory modules, on the one hand, and the network links, on the other hand. We show that, e.g., a module bandwidth of $O(1)$ and a link bandwidth of $O(\log\log n/\log\log\log n)$ is sufficient to serve any batch of distinct requests, w.h.p. Further, we show that the collision protocol can be calculated efficiently in a distributed fashion by the nodes of the interconnection network, that is, the execution of the contention resolution protocol takes $O(\log\log n)$ rounds, each of which can be executed in $O(\log n)$ time. Note that the result on the link bandwidth improves exponentially on the well known result for any non-redundant placement strategy which requires a module and link bandwidth of $\Omega(\log n/\log\log n)$, regardless of the distribution of the data blocks among the memory modules.

In Section 2.6, we consider batches that include data hot spots. We describe how to modify the collision protocol in order to support the combining of requests. If the maximum hot spot size is not too large, i.e., the maximum number of requests directed to the same data block is at most $n^{1-\alpha}$ for an arbitrary constant $\alpha > 0$, then the modified collision protocol achieves the same asymptotic bounds on the bandwidth as the original protocol for batches of distinct requests. However, we give a counterexample for larger hot spots that requires a link bandwidth of $\Omega(\log n/\log\log n)$. Interestingly, this counterexample holds for any contention resolution protocol on the RB_n-server, which shows that the interconnection network has to be changed in order to be able to serve batches with arbitrary data hot spots. We solve this problem by adding further randomization to the network design. The final result is a variation of the RB_n-server with module bandwidth $O(1)$ and link bandwidth $O(\log\log n/\log\log\log n)$ that serves any batch of requests including arbitrary data hot spots, w.h.p.

In Section 2.7, we introduce a contention resolution protocol for serving infinite sequences of distinct requests. The protocol is called "minimum protocol". It is a generalization of a load balancing scheme by Azar et al. [8] for throwing n balls into n bins, see next section. We generalize their load balancing scheme in two aspects, which are, we select sequences of links, i.e., routing paths, rather than choosing single bins, and we consider infinite sequences of arbitrary insertions and deletions rather than only n insertions (or an infinite sequence of insertions and random deletions). The minimum protocol can be executed efficiently in a distributed fashion: only the BF_0- and the BF_1-path for each request have to be inspected in order to decide which of them is to be chosen. We show that the protocol requires only a module and link bandwidth of $O(\log\log n)$ in order to serve each individual request of an infinite sequence, w.h.p. A similar bound [8] has been shown previously only for simple balls-and-bins processes corresponding to a server architectures in which the ports and the modules are completely connected and the sequence of requests includes only random rather than adversarial deletions.

In all our analyses, we will consider only read accesses. In case write accesses occur less frequently than read accesses, we think, it is most useful to update always both copies of a data block. If the number of parallel write accesses is not larger than $n^{1-\alpha}$ then updating always both copies requires only a constant fraction of the bandwidth, w.h.p. Alternatively, read and write requests can be handled symmetrically by accessing two out of three copies for each request. This technique has been introduced in the context of PRAM simulations and is described in more detail in the next section. As two out of three access schemes can be simulated by one out of two schemes, see e.g. [51], all of our asymptotic results hold for the symmetrical access scheme as well.

We complement the asymptotic bounds for the investigated contention resolution protocols with numerical and simulation results. All of these results show that redundant placement in conjunction with clever contention resolution protocol is clearly superior to non-redundant placement even for data servers of a relatively small size. This is discussed in more detail in Section 2.8.

2.4 Previous and related results

There are several different sub-areas of practical and theoretical research that relate to our work. We provide a summary of the work that is most relevant or most comparable to our work.

PRAM simulations on completely connected networks

Contention resolution protocols for PRAM simulations were developed and analyzed in [25], [37], and [56]. An exclusive read, exclusive write PRAM consisting of n processors that have direct access to a shared memory is simulated on a *distributed memory machine* consisting of n processors that are connected by a completely connected network with n memory modules. In [25], Dietzfelbinger and Meyer auf der Heide introduce the following contention resolution protocol, which we denote as "collision protocol".

Each processor of the PRAM is simulated by a processor of the distributed memory machine. Each PRAM memory cell is represented by b copies each of which is placed at random on one of the memory modules. Each copy gets a time stamp which always shows the most recent step in which the copy has been updated due to a write access. Each step of the PRAM is simulated within a number of rounds. In a round, each processor sends a request to the memory modules holding copies of the memory cell that the processor wants to read or write. If a module gets c or less requests then it serves all of them, if a module receives more than c request then it does not serve any of the requests. The algorithm for the simulation of one step of the PRAM is terminated when more than $d > b/2$ of the requests of each processor have

been served. In case of a read access, the copy with the latest time stamp of the received copies includes the actual value of the memory cell.

The algorithm for the simulation of a PRAM step can be terminated within $O(\log\log n)$ rounds, w.h.p., even for constant c, b, and d [25, 56]. Our collision protocol for the RB_n-server is a variant of the collision protocol for PRAM simulations. As we only have to deal with read accesses b we can set $b = 2$ and $d = 1$, i.e., we use redundancy 2 and each request for a data block can be satisfied by one out of the two modules storing the block. The parameter c is replaced by a set of parameters describing the bandwidth of the memory modules and the links on different levels of the interconnection network. The challenging task in our work is to generalize the results for a complete interconnection network to a sparse one.

Balls-and-bins processes

Our approach to distributing the data blocks and serving the requests is closely related to the classical balls-and-bins problem. It is well known that if n balls are tossed randomly into n bins, the maximum number of balls in any bin will be $\Theta(\log n/\log\log n)$ w.h.p. The situation changes dramatically if each ball can be placed in one out of two randomly placed bins. For example, the collision strategy for PRAM simulations can be adapted to the parallel allocation of n balls to n bins as follows. Initially, each ball chooses two bins at random. The parallel allocation process proceeds in rounds. In a round, each ball that is not yet allocated asks its two bins if they accept them. Each bin that is asked by at most c balls accepts all of them. The balls that have not been accepted in a round try again in the next round. Interestingly, if this process terminates then it guarantees a maximum load of c independent from the number of rounds. It is shown in [25, 56] that the collision protocol allocates all balls in $O(\log\log n)$ rounds, w.h.p., even for small constant c, e.g., $c = 3$.

Azar et al. [8] consider the following sequential process for throwing n balls into n bins: for each ball pick two bins independently and uniformly at random, and place the ball in the bin with the smaller load at the time of placement. They show that after all the balls are placed in bins, the maximum load of any bin is $\log\log n + O(1)$, w.h.p. Our minimum algorithm is a variant of the load balancing scheme of Azar et al. Note that serving requests on the RB_n-server is a more complex situation. Thinking of each request as a ball and each network edge as a bin, we see that finding a path for each message corresponds to placing each ball in several dependent bins. These dependencies substantially increase the difficulty of the analysis. We point out that our analysis of the minimum protocol is completely different to the analysis in [8].

Azar et al. [8] also consider infinite sequences of insertions and random deletions, i.e.: Initially, there are n balls that are arbitrarily placed in n bins. In each step, a ball is chosen uniformly at random and deleted, and then a new ball is allocated according to the minimum rule. They showed that, independent of the initial distribution of the balls, after $O(n^3)$ steps the maximum load in any bin is $\log\log n + O(1)$.

Allowing arbitrary deletions instead of assuming random deletions yields a more difficult situation as the system cannot recover quickly (i.e., in a number of steps that is polynomial in n) from any bad distribution. A straightforward modification of our analysis for the minimum protocol (i.e., considering a completely connected network between the user ports and the memory modules rather than a bounded degree network) gives an interesting bound for infinite balls and bins processes with arbitrary deletions in which each ball is placed in the least loaded of two randomly chosen bins: If the number of inserted and not deleted balls does not exceed the number of bins at any point of time, then the maximum load at any fixed point of time is at most $O(\log\log n)$, w.h.p. Recently, Cole et al. in [19] have presented a slight improvement on this result. They obtain a maximum load of $\log\log n + O(1)$, w.h.p., but only for sequences of a length that is polynomial in n.

Routing in butterfly networks

There is a vast amount of literature on routing in butterfly networks [45, 46]. Much of the early work focuses on store-and-forward or packet routing [44, 48, 53, 62, 66, 76, 79]. More recently, there has been progress in analyzing wormhole routing algorithms [20, 23, 28, 67]. In the following, we focus only on the butterfly circuit-switching literature.

In two early studies, Beizer [14] and Beneš [15] showed that any static permutation routing problem can be routed with congestion 1 and dilation $2\log n$ on an n-input Beneš network, i.e., a network that consists of two copies of an n-input butterfly placed "back-to-back" such that each output node of the first copy is identified with the corresponding output node of the second copy. Subsequently, Waksman [83] provided an elegant algorithm that takes $O(n\log n)$ time to determine all the paths, but requires global knowledge of the source and destination of all the messages. Later, Nassimi and Sahni [59] showed how to implement Waksman's algorithm in parallel on the Beneš network in time $O(\log^4 n)$. However, their algorithm is complex and requires the Beneš network to emulate a completely connected network by executing a series of sorting routines.

The two-fold butterfly network BB_n is a network that consists of two copies of an n-input butterfly placed "back-to-front" (rather than "back-to-back") such that each output node of the first copy is identified with the input node in the same column of the second copy. Although the Beneš network and the BB_n have a closely related topology, it is a long-standing open problem, whether or not it is possible to route an arbitrary permutation routing problem in an off-line fashion with congestion 1 on the BB_n. In [18], we present a routing algorithm that routes any permutation with congestion $O(\log\log n/\log\log\log n)$, w.h.p. Besides we show how to route sequentially arriving requests for connections between the inputs and the outputs of BB_n. For this on-line routing problem we achieve congestion $O(\log\log n)$, w.h.p. The off-line and the on-line algorithm for routing in BB_n correspond to the collision and the minimum protocol, respectively, for the RB_n-server.

A complementary approach aims at maximizing the throughput. Previous work has studied the model in which each link can support at most q paths, and the goal is to maximize the number of established paths. Kruskal and Snir [43] showed that if each input in an n-input butterfly network sends a message to a randomly chosen output, and at most one message can use any edge of the network (i.e., $q = 1$), then the expected number of messages that succeed in establishing paths to their destinations is $\Theta(n/\log n)$. Koch [40] generalized the result of Kruskal and Snir by showing that if each edge can support q messages, $q \geq 1$, then the expected fraction of messages that succeed in establishing paths is $\Theta(n/\log^{1/q} n)$. Maggs and Sitaraman [53] generalized the previous two results by showing that, by making two passes through a butterfly, it is possible to route an $\Omega(n/\log^{1/q} n)$ fraction of any permutation (rather than only a random permutation), w.h.p.

The use of randomness to design multistage networks dates back to Ikeno[33] as well as Bassalygo and Pinsker [12]. Networks such as the RB_n are known to have good routing and fault tolerance properties [77, 47]. Recent results provide algorithms for routing circuits for any permutation routing problem with congestion 1 in multibutterfly and multi-Beneš networks with set-up time $O(\log n)$ [3, 63].

Explicit analyses of multimedia data server architectures

Many practical and theoretical studies of multimedia data servers focus on the data layout for video playback, e.g., [30, 61, 72, 73, 81]. The data for these video servers is usually "striped" across multiple disks, that is, each movie is divided into blocks of relative small size, and these data blocks are distributed regularly, e.g., in a round robin fashion, among the disks. Requests are served in synchronized cycles of fixed duration. In each cycle, for each video stream a data block from a particular disk is accessed, and

in the following cycle the following block from the next consecutive disk is accessed, and so on. Thus, each stream is assigned a slot of bandwidth. A new video stream only has to wait until a free slot cycles through the disks. The advantage of striping is that it utilizes the bandwidth and the storage capacities of all disks while avoiding contention. The disadvantage of striping is that it is not suitable for applications with less predictable access patterns than simple video playback.

The "RIO (Randomized I/O) Storage Server", introduced by Santos and Muntz in [69], aims to support multimedia applications with arbitrary access patterns. Therefore, the data blocks are distributed randomly rather than striped regularly among the disks. Interestingly, a fixed fraction, e.g., 25 %, of the blocks are replicated and placed with redundancy 2; the blocks that are to be replicated are chosen at random. The idea behind the partial replication scheme is to reduce the storage requirement in comparison to a full replication scheme in which every data block is placed randomly on two disks. Several experimental and simulation results for the partial replication scheme on servers with 4, 8, 16, and 64 disks are given. It is assumed that the requests arrive one by one, and each request has to be served within a fixed time period. When a new request arrives and the requested block is placed on two disks then the one that has the smallest number of requests is selected. The interconnection network is not taken into account. The simulation results show that the partial replication significantly reduces the fraction of requests that miss the deadline. Further results on the performance of the RIO server are given in [27]. For example, the partial replication is compared to a full replication scheme in which each block is placed with redundancy 2. The full replication scheme turns out to be superior as it allows to guarantee much shorter response times, e.g, for 64 disks, the deadlines can be reduced by a factor of at least 2 in comparison to the partial replication scheme.

Further simulation results for data server architectures with random data layout are given by Rottmann et al. [68]. They do not use redundancy but they take the communication network between the user ports and the memory modules into account. For example, they consider a server using a packet-switched butterfly network in which the communication links are by a constant of $t_a = 2$, 3, or 4 faster than the modules, that is, a disk takes t_a steps to read a data block whereas each link in the network can forward one data block in a step. In this model, the performance nearly scales linearly in $1/t_a$, which indicates that the bandwidth of the network is not a bottleneck it $t_a > 1$. Note that our simulations show that the behavior of a butterfly-like network using a circuit-switched routing mechanism is different. Here a higher link bandwidth helps to guarantee shorter response times.

2.5 Serving batches of distinct requests

In this section, we introduce a contention resolution protocol for serving batches of distinct requests on the RB_n-server. The protocol is called "collision protocol". It is a variation of a contention resolution mechanism developed for PRAM simulations on completely connected networks [25, 37, 56]. The challenging task is to show that the collision protocol is not only able to resolve the contention at the memory modules, which is the major problem to be solved in the PRAM simulations, but simultaneously is able to minimize the congestion in the network so that only a small link bandwidth is needed.

First, we will describe the collision protocol. Then we will give an asymptotical analysis for the protocol. We will show that, e.g., a module bandwidth of $O(1)$ and a link bandwidth of $O(\log\log n/\log\log\log n)$ is sufficient to serve any batch of distinct requests, w.h.p. Afterwards, we will calculate some explicit numerical bounds, and we will give simulation results for the protocol on small and medium-sized server architectures. These results show that also the constants in the asymptotic bounds are very small.

2.5.1 The collision protocol

.

The collision protocol has to choose one out of two routing paths for each request, either the request's BF_0- or its BF_1-path. The selected paths are not allowed to exceed the module or link bandwidths. Let $d = \log n$, and let c_i denote the bandwidths of the edges on level i, for $1 \leq i \leq d$, such that c_d corresponds to the module bandwidth. None of the edges on level i can support more than c_i routing paths.

The selection of the routing paths proceeds in rounds as follows. Initially all request and their paths are *active* and none of the paths is *selected*. The event that more than c_i active paths share an edge of level i is called a *collision*. A path p is eligible to be selected if none of the edges included in the path is involved in a collision, that is, for any $1 \leq i \leq d$, the number of active paths traversing edge e_i is at most c_i, where e_i denotes the edge of p on level i. Let the *buddy* of p denote the other path that belongs to the request of p. If p and its buddy are both eligible to be selected, only one is selected arbitrarily, e.g., the BF_0-path. A request and its paths cease to be active in a round if one of the paths is selected in that round.

The path selection is terminated after some number t of rounds, which will be specified later on. Note that the collision algorithm selects a path p in a round only if p does not share an edge on level i with more than $c_i - 1$ other active paths. This implies that any edge on level i that is included in at least one selected path is included in at most $c_i - 1$ other selected or active paths. As a consequence, the maximum number of selected paths that use an edge on level i is at most c_i. Hence, the selected paths fulfill the bandwidth constraints.

Each round of the collision algorithm can be implemented using a store-and-forward algorithm as a subroutine: in a first pass, for each active request, a packet is sent along the BF_0- and the BF_1-path from level 0 to level $d = \log n$. During this pass, for each edge, the number of packets traversing the edge is counted. In a second pass, all packets are routed backward along their respective paths from level d to level 0. During this pass, the number of active paths that traverse each edge on the path followed by a packet is compared to the bandwidth of the edge. If none of the edges on a path is overloaded then the path is eligible to be selected. We suggest to use the simple routing algorithm described in [53]. This algorithm is randomized and takes time $O(\log n)$, w.h.p., for each round.

2.5.2 Asymptotical analysis of the collision protocol

The collision algorithm as described above is not guaranteed to select a path for every request. However, in the following theorem, we show that the algorithm will do so, if the bandwidths are chosen sufficiently large. The theorem is rather technical as it allows to choose different values for the bandwidths on different levels. We give two examples.

- The first example assumes that the memory modules are the major bottleneck of the data server. We set $c_d = \Theta(1)$ and $c_1 = \cdots = c_{d-1} = \Theta(\log\log n / \log\log\log n)$. Recall that $c_1, \ldots, c_{d-1}$ denote the link bandwidths on the edge levels 1 to $d-1$, whereas c_d denotes the module bandwidth. For this configuration, the collision algorithm selects a path for every request in $t = O(\log\log n)$ rounds, with probability $1 - n^{-c_d+1+o(1)} = 1 - n^{-\Theta(1)}$.
- Another possibility is to choose the same bandwidths for all the links and the memory modules by setting $c_1 = \cdots c_d = \Theta(\log\log n / \log\log\log n)$. In this case the collision algorithm selects a path for every request in only $t = O(\log\log n / \log\log\log n)$ rounds, w.h.p. The probability for success, which is determined by the smallest bandwidth, is even larger than for the previous example.

Finally, before stating the theorem, we give some comments on the quality of the probability bound. The probability for the existence of $c+1$ packets that are addressed to the same memory module in BF_0

and to the same memory module in BF_1 is $\Omega(1/n^{c-1})$. If this happens, for $c = c_d$, the collision protocol fails, which shows that the probability bound given in the following theorem is exact apart from some logarithmic terms.

Theorem 2.5.1 *Let $\varepsilon > 0$ denote an arbitrary constant term. Suppose $c_1, \ldots, c_d$ are chosen sufficiently large such that $\sum_{i=1}^{d} 1/c_i! \leq 1/(1+\varepsilon)$ but $c_i \leq \log n$, for $1 \leq i \leq d$. Define $c = \min\{c_i \mid 1 \leq i \leq d\}$. Then the probability that the collision algorithm does not select a routing path for every request in $t = \log_c \log n + O(1)$ rounds is $(\log n)^{O(c)}/n^{c-1}$.*

Proof. First we show that if the algorithm has not selected a path for every request after t rounds, we can construct a "witness tree" of height t. Next we show how the witness tree can be pruned to avoid stochastic dependencies. Then we show by enumeration that the occurrence of a pruned witness tree is very unlikely. Initially, we prove a slightly weaker bound on the probability, that is, we show that the probability that the protocol fails to select a routing paths for every request is $(\log n)^{O(c)}/n^{c/2-1}$ rather than $(\log n)^{O(c)}/n^{c-1}$. Finally, we will describe how to modify the proof in order to achieve the stronger bound on the probability.

Constructing a witness tree. The witness tree is a tree graph whose nodes represents bad events, i.e., collisions, that cause the protocol to fail. The topology of the witness tree is defined as follows. The *inner tree* of the witness tree is a c-ary tree of height $t-1$, where $t = \log_c \log_{1+\varepsilon} n + 2$ corresponds to the number of executed rounds. The nodes of the inner tree are the internal nodes of the witness tree. To the inner tree, we add some nodes that are the leaves of the witness tree so that each internal node of the witness tree has C children, where $C = \max\{c_i \mid 1 \leq i \leq d\}$. In particular, each non-leaf node of the inner tree gets $C - c$ additional child nodes, and each leaf node of the inner tree gets C child nodes. Thus, the witness tree is C-ary tree of height t. We assume that the tree is ordered, and that the children of a node belonging to the inner tree are placed to the left of the other children. The number of nodes of the witness tree is

$$M = \frac{c^{t+1}}{(c-1)} + \frac{c^t \cdot (C-c)}{(c-1)} \leq C^3 \cdot \log_{1+\varepsilon} n \ .$$

In the following, we associate requests to the nodes of the witness tree such that the witness tree represents a set of collision events that cause the collision protocol to fail. Fix the randomization level and the mapping of the blocks in X to the modules. Further, fix the batch of n requests.

We assume that one of the requests is still active after round t. Let r denote this request. Then r must have had a collision with c_i other requests on level i of BF_0 in round t, for an $1 \leq i \leq d$. Let $r_1, \ldots, r_{c_i}$ denote these requests. We associate r to the root of the witness tree and $r_1, \ldots, r_{c_i}$ to the leftmost children of the root. Since $r_1, \ldots, r_c$ are still active in round t, they were not selected in round $t-1$. Applying the argument recursively to these requests yields an assignment of requests to the tree nodes such that each internal node v represents a collision of the requests associated to v and its c_i leftmost children at an edge on level i, for an $1 \leq i \leq d$.

This construction associates one request to each of the internal nodes but only to some of the leaf nodes of the witness tree. If the collision represented by a node takes place on level i of RB_n then only the c_i leftmost children have a request associated. With each tree level, we switch between the subbutterflies BF_0 and BF_1 such that collisions represented by nodes of even depth take place in BF_0 and collisions represented by nodes of odd depth take place in BF_1.

To give each witness tree a unique representation, we assume that the requests have unique IDs, and the children of a node are listed in increasing order of request ID from left to right. In order to simplify the following analysis we use *random naming*, that is, we assume that the IDs are randomly permuted such that the request ID does not give any evidence about the port at which a request is issued. This assumption

will significantly simplify the following analysis since the random naming ensures that any request with fixed ID has the same probability to pass a given edge.

Pruning the witness tree. The requests in a witness tree are not necessarily pairwise distinct. Hence, the collision events represented by a witness tree are not necessarily stochastically independent. Note that, if they were stochastically independent, it would be relatively straightforward to argue the theorem. The intuitive reason why the dependencies do not affect the final conclusion is that there are only few nodes in the witness tree. Hence, the dependencies are "rare".

In order to handle dependencies, we prune nodes from the witness tree as necessary. This pruning is done by a traversal through the tree visiting the internal nodes in breadth-first-search order starting at the root. Nodes which are pruned during the traversal or whose children are pruned during the traversal are excluded from the sequel traversal. For a node v visited during the traversal, let $B(v)$ denote the set of nodes visited before v. Furthermore, let $\Gamma(v)$ denote the set of nodes that are non-pruned children of the nodes in $B(v)$ but not in $B(v)$ themselves.

When a node v is visited we apply a pruning if and only if one of v's children has associated a request which is also associated to a node w in $B(v) \cup \Gamma(v)$:

a) If w is from $B(v)$ then the tree is pruned below v, that is, the children of v and all subtrees rooted at them are removed from the tree.

b) If w is from $\Gamma(v)$ then the tree is pruned below v and below w.

The node v is called *pruning node*, and w is called the *conflicting node* of v. If there is more than one choice for a conflicting node w we make the choice arbitrarily, so that each pruning node can be associated with exactly one conflicting node.

We continue the pruning process till either there are no more nodes to visit or there are at least $k = \lceil c/2 \rceil$ pruning nodes. In the latter case, we apply a final pruning. If v is the kth pruning node, we remove from the tree all nodes not included in $B(v) \cup \Gamma(v)$. This effectively stops the pruning process at the kth pruning node.

Each internal node u of the pruned witness tree represents a collision of the requests associated to u and its children. Therefore, the internal nodes are called *collision nodes*. Suppose u represents a collision on level i of BF_j, for some $1 \leq i \leq d$ and $j \in \{0,1\}$. Then the BF_j-path of the request associated to u collides with the c_i BF_j-paths of the requests associated to u's children. We define the *configuration* of u to be a description of all the colliding BF_j-paths.

Each pruning node v represents the event that the BF_0- or BF_1-paths of the requests associated to v and its conflicting node w share an edge. The subbutterfly BF_j-th, for $j \in \{0,1\}$, in which the paths overlap is determined by the depth of v in the tree. We define the *configuration* of v to describe the two overlapping BF_j-paths, that is, the BF_j-paths associated to v and w. The BF_j-path of v is called *pruning path*, the BF_j-path of w is called *conflicting path*.

The configurations possess the following properties.

> **Property 1:** Any request is associated to at most one node of the pruned witness tree, and each BF_0- or BF_1-path is included in the configuration of at most one collision node. This property follows from the pruning of the subtrees below the pruning nodes.
>
> **Property 2:** The pruning path of a pruning node v is not included in the configuration of any collision node or any pruning node visited before v. (Conflicting paths can be included in more than one configuration.) This property follows from the pruning of the subtrees below the conflicting nodes in $\Gamma(v)$.

Property 3: The number of collision nodes, m, and the number of pruning nodes, p, fulfill the equation $m \geq (k-p) \cdot \log_{1+\varepsilon} n$. This can be seen as follows. For each of the p pruning nodes, some nodes from at most two of the c subtrees rooted at the children of the root are removed. Thus, at least $c-2p \geq k-p$ of these subtrees remain untouched. Each of them includes $(c^{t-1}-1)/(c-1) \geq \log_{1+\varepsilon} n$ internal nodes, which yields the equation above.

Bounding the probability of the existence of a witness tree. We bound the probability of the existence of a pruned witness tree via enumeration. Define the *tree shape* to be a description of the topology of a pruned witness tree including a specification of the pruning and the conflicting nodes. Define an *admissible witness tree configuration* to be a description of a tree shape and all node configurations that eventually, i.e., for some choice of the randomization level and some mapping of the data blocks to the modules, match to the shape and the node configurations of a witness tree as constructed above. An admissible configuration is called a *partial configuration* if it includes only configurations for some of the nodes. An admissible configuration is said to be *active* if the outcome of the randomization level and the random mapping corresponds to the paths described in the configuration.

Let $\mathcal{T}$ denote the set of tree shapes corresponding to at least one admissible witness tree configuration, and let $\mathcal{K}_T$ denote the set of all admissible witness tree configurations with tree shape $T \in \mathcal{T}$. Then the probability that the collision algorithm does not select a path for every request can be bounded by

$$\sum_{T \in \mathcal{T}} \underbrace{\sum_{K \in \mathcal{K}_T} \text{Prob}(K \text{ is active})}_{=: E(T)} .$$

We aim to give an upper bound on $E(T)$, for a fixed tree shape $T \in \mathcal{T}$. $E(T)$ is equal to the expected number of active witness tree configurations with tree shape T. Note that the tree shape T only restricts the number of admissible configurations, that is, it defines the set $\mathcal{K}_T$, but does not influence the probability for a given configuration $K \in \mathcal{K}_T$ to become active.

Instead of summing over all admissible configurations in $\mathcal{K}_T$ and multiplying each individual configuration with its probability, we consider the nodes of the witness tree one by one and calculate an upper bound on the expected number of configurations for each individual node. In particular, we consider first all the internal tree nodes and then all the pruning nodes; both sets of nodes are considered in the order of visitation.

Fix a tree shape $T \in \mathcal{T}$. Let m denote the number of collision nodes in T, and $u_1, \ldots, u_m$ denote the collision nodes in the order of visitation. Consider node u_j. Let K denote a partial configuration including the configurations of the nodes $u_1, \ldots, u_{j-1}$. For the root u_1, we define that the configuration K specifies the request associated to u_1 (but not the course of the two corresponding paths). For $j > 1$, K specifies the request associated to u_j because this requests is included in the configuration of the parent of u_j. Let $E_{\text{coll}}(u_j, K)$ denote the expected number of active configurations for u_j under the assumption that K is active. Furthermore, let $E_{\text{coll}}(u_j)$ denote the maximum, taken over all K, of $E_{\text{coll}}(u_j, K)$.

Lemma 2.5.2 $E_{\text{coll}}(u_j) \leq 1/(1+\varepsilon)$.

Proof. Suppose that the configuration K and the randomization level of the network are fixed arbitrarily. W.l.o.g., we assume that the collision represented by u_j takes place in BF_0.

We have to estimate the expected number of active configurations for u_j, that is, the expected number of possibilities to choose the BF_0-paths of the requests associated to u_j and its children such that these paths collide. The considered random experiment is the mapping of the data blocks to the modules.

Any of the requests associated to u_j and its children has only one BF_0-path for every outcome of the random experiment. Hence, the expected number of active BF_0-paths for these requests is equivalent to the expected number of possibilities to choose these requests.

The request r associated to u_j is specified by K. Suppose that the paths collide on level i, for some $1 \leq i \leq d$. Then we have to calculate the expected number of possibilities to choose a set of requests $\{r_1, \ldots, r_{c_i}\}$ such that the BF_0-paths of these requests collide. Note that it is sufficient to choose a set of requests, rather than a tuple, since the requests associated to the children of u_j are ordered according to their ID.

Let e denote the edge of the BF_0-path of r on level i. The BF_0-paths of $r_1, \ldots, r_{c_i}$ have to traverse e. This edge is reachable from at most 2^{i-1} of the nodes on level 1 of BF_0. Each of these nodes is traversed by two BF_0-paths. Thus, at most 2^i of the requests that are not included in K are candidates for $r_1, \ldots, r_{c_i}$, and, consequently, the number of possibilities to choose candidates for these requests is at most $\binom{2^i}{c_i}$.

Now suppose that the c_i requests and the c_i nodes of level 1 passed by the BF_0-paths of these requests are fixed. Each of the BF_0-paths follows a random course from its node on level 1 to the module on which the respective data block is placed. According to Property 1 all of these paths belong to requests that are not included in K, and, hence, the courses of these paths from level 1 to level i are not revealed by K. Therefore, the probability that all c_i paths traverse edge e of level i is $(2^{-i})^{c_i}$.

As a consequence,

$$E_{\text{coll}}(u_j) \leq \sum_{i=1}^{d} \binom{2^i}{c_i} \cdot \left(\frac{1}{2}\right)^{-i \cdot c_i} \leq \sum_{i=1}^{d} \frac{1}{c_i!} \leq \frac{1}{1+\varepsilon} , \tag{2.1}$$

where the last estimation follows from an assumption in the theorem, which is $\sum_{i=1}^{d} 1/c_i! \geq 1/(1+\varepsilon)$. ■

Now we give an upper bound on the expected number of the active configurations for the pruning nodes. Let $p \leq k$ denote the number of pruning nodes in T, and let $v_1, \ldots, v_p$ denote the pruning nodes in the order of visitation. For a pruning node v_j and a partial configuration K of all collision nodes $u_1, \ldots, u_m$ and the pruning nodes $v_1, \ldots, v_{j-1}$, let $E_{\text{prune}}(v_j, K)$ denote the expected number of active configurations for v_j under the assumption that K is active. Let $E_{\text{prune}}(v_j)$ denote the maximum, taken over all K, of $E_{\text{prune}}(v_j, K)$.

Lemma 2.5.3 $E_{\text{prune}}(v_j) \leq \log n/(n-2M)$.

Proof. W.l.o.g., we assume that the subnetwork in which the paths of the requests represented by v_j and its conflicting node overlap in BF_0. Let w denote the conflicting node of v_j, which is specified by the tree shape T. Let r and r' denote the requests associated to v_j and w, respectively, which are specified by the configuration K.

We have to estimate the expected number of configurations for v_j, that is, the expected number of possibilities to choose the BF_0-path of r and the BF_0-path of r' such that the two paths overlap. The considered random experiments are the random wiring on the randomization level, the random naming of the request IDs, and the random mapping of the data blocks to the modules.

For every outcome of these experiments, there is only one BF_0-path for r'. Hence, the expected number of possibilities to choose this path is 1. It remains to calculate the expected number of possibilities to choose the BF_0-path of r.

Let e denote an arbitrary edge on the path of r'. The number of possibilities to choose e is $\log n$. The number of different paths that connect node level 1 with the module level and traverse e is $n/2$. Hence, the number of admissible paths for the BF_0-path of r is at most $\log n \cdot n/2$.

Let p denote one of the admissible paths, and s its node on level 1 of BF_0. In the following, we calculate the probability that p is the BF_0-path of r.

The configuration K includes the BF_1-path of r. If this path uses a straight edge on the randomization level, then the BF_0-path uses a randomly selected cross edge. The probability that this edge is connected to the node s is at most $1/(n/2-M)$ because at most M of the $n/2$ nodes on level 1 of BF_0 are revealed by configuration K. (Recall that M denotes the maximum number of requests represented by the witness tree.) If the BF_1-path of r uses one of the random cross edges on the randomization level, then the the port of r can be connected only by a straight edge to the node s. Because of the random naming, the port of r is not determined by K. The probability that the port connected by a straight edge to s is the port that issues r is at most $1/(n/2-M)$.

Thus, the probability that the BF_0-path of r traverses the fixed node s is $1/(n/2-M)$. Now assume that this event happens. Then the probability that the BF_0-path follows the course of the fixed path p is $2^{-\log n} = 1/n$. (Note that Property 2 ensures that the BF_0-path is not revealed by K.) As a consequence,

$$E_{\text{prune}}(v_j) \;\leq\; \frac{\log n \cdot n}{2} \cdot \frac{1}{n/2-M} \cdot \frac{1}{n} \;=\; \frac{\log n}{n-2M} \quad .$$

■

The bound for $E_{\text{coll}}(u_j)$ on the expected number of active configurations for a collision node u_j is independent of the configurations of the collision nodes $u_1,\ldots,u_{j-1}$. Furthermore, the bound for $E_{\text{prune}}(v_j)$ on the expected number of active configurations for a pruning node v_j is independent of the configurations of the collision nodes $u_1,\ldots,u_m$ and the pruning nodes $v_1,\ldots v_{j-1}$. Consequently, these bounds are independent estimations of mean values, and, hence, they can be multiplied in order to get an upper bound on the expected number of all configurations. Since the number of choices for the initial configuration K in $E(r_1,K)$, which specifies the request associated with the root, is n, we get the following upper bound on the expected number of active witness tree configurations with tree shape T.

$$\begin{aligned}
E(T) &\leq n \cdot \prod_{j=1}^{m} E_{\text{coll}}(u_j) \cdot \prod_{j=1}^{p} E_{\text{prune}}(v_j) \\
&\leq n \cdot \left(\frac{1}{1+\varepsilon}\right)^m \cdot \left(\frac{\log n}{n-2M}\right)^p \\
&\overset{(*)}{\leq} n \cdot (1+\varepsilon)^{-(k-p)\cdot\log_{1+\varepsilon} n} \cdot \left(\frac{\log n}{n-2M}\right)^p \\
&\leq n \cdot \left(\frac{\log n}{n-2M}\right)^k \quad .
\end{aligned}$$

Estimation $(*)$ follows from Property 3, i.e., $m \geq (k-p)\cdot\log_{1+\varepsilon} n$.

It remains only to sum over all tree shapes $T \in \mathcal{T}$. As the tree shape is determined by the at most k pairs of pruning and conflicting nodes, we have $|\mathcal{T}| \leq M^{2k}$. From this equation we can derive the desired bound on the probability that the collision protocol fails to select a path for every request in t rounds, it is at most

$$\sum_{T\in\mathcal{T}} E(T) \;\leq\; M^{2k} \cdot n \cdot \left(\frac{\log n}{n-2M}\right)^k \;=\; \frac{(\log n)^{O(c)}}{n^{c/2-1}} \quad ,$$

for $k = \lceil c/2 \rceil$, $M \leq C^3 \cdot \log_{1+\varepsilon} n$, and $C \leq \log n$.

Proving a stronger bound on the probability. Finally, we show how to change the pruning rules in order to get the stronger $(\log n)^{O(c)}/n^{c-1}$ bound on the probability. We increase t by one, that is, we set $t = \log_c \log_{1+\varepsilon} n + 3$. Of course, this also increases M, the maximum number of nodes in the witness tree, that is, the upper bound on M becomes $C^4 \cdot \log_{1+\varepsilon} n$.

The modified rules, which we will state in the following, ensure that the number of nodes removed for a pruning node becomes smaller. Analogously to the original pruning, the internal nodes are visited in breadth-first-search order starting at the root. Nodes that are pruned during the traversal or whose children are pruned during the traversal are excluded from the sequel traversal. For a node v visited during the traversal, let $B(v)$ denote the set of nodes visited before v, and let $\Gamma(v)$ denote the set of nodes that are non-pruned children of the nodes in $B(v)$ but not in $B(v)$ themselves. When a node v is visited we apply a pruning if and only if one of v's children has associated a request that is also associated to a node w in $B(v) \cup \Gamma(v)$. We have three different pruning rules.

a) If one of v's children has associated a request that is associated to a node w in $B(v)$, then the tree is pruned below v. v is called the *pruning node*, and w is called the *conflicting node*.

b) Otherwise, if one of v's children has associated a request that is associated to a node w in $\Gamma(v)$ which is not a child of the root, then the tree is pruned below v and below w. v is called the *pruning node*, and w is called the *conflicting node*.

c) Otherwise, at least one of v's children has associated a request that is associated to a node in $\Gamma(v)$ and is a child of the root. Let $x_1, \ldots, x_j$ denote all the children of v fulfilling this condition and $w_1, \ldots, w_j$ the respective children of the root. We remove the subtrees rooted with $x_1, \ldots, x_j$, and we remove the subtrees below $w_1, \ldots, w_j$. In this case, $w_1, \ldots, w_j$ are called *pruning nodes* and $x_1, \ldots, x_j$ are the respective *conflicting nodes*. Note that this definition is contrary to the previous definitions of pruning and conflicting nodes.

We continue the pruning process till either there are no more nodes to visit or there are at least c pruning nodes. In the latter case, we apply a final pruning. If v is the cth pruning node, we remove from the tree all nodes not included in $B(v) \cup \Gamma(v)$. This ensures that the number of pruning nodes is at most c rather than $k = \lceil c/2 \rceil$.

The tree pruned in this fashion fulfills Property 1 and 2 of the original pruning. Note that we have interchanged the definitions of the pruning and the conflicting nodes in Rule c) only because Property 2 does not hold for the original definition. The key feature of the new pruning is that Property 3 becomes stronger now:

> **Property 3:** The number of collision nodes, m, and the number of pruning nodes, p, fulfill the equation $m \geq (c - p) \cdot \log_{1+\varepsilon} n$. This is because, at least $c - p$ of the subtrees rooted at the grandchildren of the root remain untouched by the new pruning, and each of these subtrees have size at least $\log_{1+\varepsilon} n$.

Since Property 1 and 2 hold also for the new pruning, we can estimate $E_{\text{coll}}(u_j)$ and $E_{\text{prune}}(v_i)$ similarly to the Lemmas 2.5.2 and 2.5.3. In fact, the bound on $E_{\text{prune}}(v_i)$ given in Lemma 2.5.3 holds for the new pruning without any change. However, the bound on $E_{\text{coll}}(u_j)$ given in Lemma 2.5.2 changes slightly because Rule c) allows that a collision node representing a collision on butterfly level i has less than c_i children. For $1 \leq i \leq m$, let k_j denote the number of children removed from collision node u_j due to Rule

c). Then Equation 2.1 on page 70 changes to

$$E_{\text{coll}}(u_j) \;\leq\; \sum_{i=1}^{d} \binom{2^i}{c_i - k_j} \cdot (2^{-i})^{c_i - k_j} \;\leq\; \sum_{i=1}^{d} \frac{1}{(c_i - k_j)!} \;\leq\; \frac{C^{k_j}}{1+\varepsilon} \quad .$$

Since each pruned child of a collision node corresponds to a pruning node and the number of pruning nodes is at most c, it holds $\sum_{j=1}^{p} k_j \leq c$, and hence,

$$\prod_{j=1}^{m} E_{\text{coll}}(u_j) \;\leq\; \prod_{j=1}^{m} \frac{C^{k_j}}{1+\varepsilon} \;\leq\; C^c \cdot (1+\varepsilon)^{-m} \quad .$$

As a consequence, the upper bound on the probability that the collision algorithm has not selected a path for every request after $t = \log_c \log_{1+\varepsilon} n + 3$ rounds becomes

$$\begin{aligned}
\sum_{T \in \mathcal{T}} E(T) &= \sum_{T \in \mathcal{T}} n \cdot \prod_{j=1}^{m} E_{\text{coll}}(u_j) \cdot \prod_{j=1}^{p} E_{\text{prune}}(v_j) \\
&\leq \sum_{T \in \mathcal{T}} n \cdot C^c \cdot (1+\varepsilon)^{-m} \cdot \left(\frac{\log n}{n - 2M}\right)^p \qquad (2.2) \\
&\overset{(*)}{\leq} \sum_{T \in \mathcal{T}} n \cdot C^c \cdot (1+\varepsilon)^{-(c-p)\cdot \log_{1+\varepsilon} n} \cdot \left(\frac{\log n}{n - 2M}\right)^p \\
&\leq \sum_{T \in \mathcal{T}} n \cdot C^c \cdot \left(\frac{\log n}{n - 2M}\right)^c \\
&\overset{(**)}{\leq} M^{2c} \cdot n \cdot \left(\frac{C \cdot \log n}{n - 2M}\right)^c \\
&= \frac{(\log n)^{O(c)}}{n^{c-1}} \quad ,
\end{aligned}$$

for $M \leq C^4 \cdot \log_{1+\varepsilon} n$, and $C \leq \log n$. Estimation $(*)$ follows from Property 3, i.e., $m \geq (c-p) \cdot \log_{1+\varepsilon} n$, and Estimation $(**)$ follows from $|\mathcal{T}| \leq M^{2c}$. This completes the proof of Theorem 2.5.1. ■

2.5.3 Numerical analysis of the collision protocol

The asymptotical analysis in the previous section aims at minimizing the bounds on the module and link bandwidths for relatively large servers. For practical purposes, however, results for servers of smaller size, e.g., with 256 ports, are more interesting. In this section, we show how to refine the witness tree analysis such that it also yields interesting bounds also for networks of a smaller size. Table 2.1 shows the results of this analysis. The table lists upper bounds on the probabilities that the collision protocol with $c_d = c$ and $c_1 = \ldots = c_{d-1} = C$ fails to select a routing path for every request within t rounds. Recall that c denotes the bandwidth of the memory modules and C the bandwidth of the links.

The results in the table show that, regardless of the size of the server, a small constant module bandwidth of 3 suffices to ensure that any batch of distinct requests is served with relative high probability, whereas the link bandwidth and the number of rounds has to be increased with the size of the server in order to ensure that any batch can be served with high probability. These characteristics correspond to the results given by the asymptotical analysis. A deeper discussion of the results obtained from our analysis is given in the next section in which we will compare the asymptotic and the numeric bounds to simulation

$\log n$	$c=3$								
	$t=4$			$t=5$					$t=t(n)$
	$C=3$	$C=4$	$C=5$	$C=3$	$C=4$	$C=5$	$t(n)$	$C(n)$	$C=C(n)$
2	9.21e-03	9.21e-03	9.21e-03	9.21e-03	9.21e-03	9.21e-03	3	3	9.21e-03
3	6.68e-02	2.70e-02	2.70e-02	6.68e-02	2.70e-02	2.70e-02	3	3	6.68e-02
4	2.07e-01	6.65e-02	6.05e-02	2.07e-01	6.65e-02	6.05e-02	4	4	6.65e-02
5	4.78e-01	9.25e-02	7.06e-02	4.78e-01	9.25e-02	7.06e-02	4	4	9.25e-02
6	1.00e-00	9.29e-02	5.84e-02	1.00e-00	9.27e-02	5.82e-02	4	4	9.29e-02
7		7.87e-02	4.07e-02		7.53e-02	4.01e-02	4	4	7.87e-02
8		6.38e-02	2.63e-02		5.36e-02	2.54e-02	4	4	6.38e-02
9		5.44e-02	1.64e-02		3.52e-02	1.54e-02	4	4	5.44e-02
10		5.22e-02	1.01e-02		2.20e-02	9.23e-03	5	4	2.20e-02
11		5.96e-02	6.23e-03		1.34e-02	5.45e-03	5	4	1.34e-02
12		8.71e-02	3.84e-03		8.26e-03	3.19e-03	5	4	8.26e-03
13		1.69e-01	2.39e-03		5.56e-03	1.86e-03	5	5	1.86e-03
14		4.35e-01	1.53e-03		4.61e-03	1.07e-03	5	5	1.07e-03
15		1.00e-00	1.05e-03		5.05e-03	6.19e-04	5	5	6.19e-04
16			8.39e-04		7.08e-03	3.53e-04	5	5	3.53e-04
17			8.52e-04		1.29e-02	2.00e-04	5	5	2.00e-04
18			1.17e-03		3.64e-02	1.13e-04	5	5	1.13e-04
19			2.14e-03		1.70e-01	6.32e-05	5	5	6.32e-05
20			4.77e-03		1.00e-00	3.52e-05	5	5	3.52e-05
22			3.15e-02			1.07e-05	5	5	1.07e-05
24			2.37e-01			3.22e-06	5	5	3.22e-06
26			1.00e-00			9.54e-07	5	5	9.54e-07
28						2.79e-07	6	5	2.77e-07
30						8.27e-08	6	5	8.00e-08
32						3.23e-08	6	5	2.28e-08
34						9.50e-08	6	5	6.47e-09
36						1.43e-06	6	5	1.82e-09
38						2.54e-05	6	5	5.08e-10
40						4.45e-04	6	5	1.41e-10
42						7.35e-03	6	5	3.91e-11
44						1.16e-01	6	5	1.11e-11
46						1.00e-00	6	5	6.24e-12

Table 2.1: *Numerical upper bounds on the probability that the collision protocoll on the RB_n-server with module bandwidth c and link bandwidth C fails to select a path for every request within t rounds. The last column shows the probabilities for a slightly increasing C and t, that is, $C = C(n) = \min\{x \in \mathbb{N} \mid x! \geq 2 \cdot \log n\}$ and $t = t(n) = \lceil \log_c \log n + 2 \rceil = \Theta(\log\log n / \log\log\log n)$, which corresponds to the suggestions of Theorem 2.5.1.*

results. We will see that the simulations, in general, follow the predictions of our analyses but allow to choose even smaller values for the bandwidths and the number of rounds.

The results in the table are computed in a recursion that counts the expected number of pruned witness trees. In the following we describe this recursion. First we will explain the general idea behind this approach; then we will describe the exact recursive function that yields the probability bounds given in the table.

2.5.3.1 A recursion for counting witness trees

We estimate the expected number of pruned witness trees by counting the expected number of different pruning traversals. We use the simple pruning rules as described on page 68 in Section 2.5.2. According to these rules, each pruning traversal visits some nodes of the witness tree in breadth-first-search order starting at the root. We change the rules sightly, that is, rather than finishing a traversal after visiting the kth pruning node, a traversal is finished as soon as it has visited m collision and p pruning nodes such that $m \geq (c-2p)\cdot(c^{t-1}-1)/(c-1)$. We can conclude from Property 3 on page 69 that any traversal reaches a node fulfilling this condition. A traversal that has reached its final node is called a *finished traversal*, and a traversal that has not yet reached this node is called a *partial traversal*.

The *configuration* of a traversal is defined to be a collection of the configurations of all visited collision and pruning nodes. Note that the configuration of a finished traversal specifies the complete configuration of the resulting pruned witness tree including the tree shape. A configuration of a partial or finished traversal is called *active* if the corresponding partial or complete configuration of the underlying witness tree is active. Consequently, the expected number of active witness tree configurations is bounded above by the expected number of active finished traversals.

For the configuration K_ϑ of a partial or finished traversal ϑ, let $E(K_\vartheta)$ denote the expected number of possibilities to complete K_ϑ to an active configuration of a finished traversal under the assumption that K_ϑ is active. If ϑ is finished then $E(K_\vartheta)=1$. Let $f(m,p)$ denote the maximum of $E(K_\vartheta)$ over all configurations K_ϑ and traversals ϑ that have visited m collision and p pruning nodes. Then $f(0,0)$ is an upper bound on the expected number of all finished traversals and, hence, an upper bound on the probability that the collision process does not select a path for every request within t rounds. Our numerical analysis computes an upper bound on $f(0,0)$.

Let $E_{\text{coll}}(u_m)$ and $E_{\text{prune}}(v_p)$ denote independent upper bounds on the expected number of active configurations for the mth collision node u_m and the pth pruning node v_p, respectively, in the order of visitation, e.g., the bounds given in Lemma 2.5.2 and Lemma 2.5.3, respectively. Then an upper bound on $f(0,0)$ can be computed recursively as follows.

$$f(m,p) \leq \begin{cases} n\cdot E_{\text{coll}}(u_1)\cdot f(1,0) & \text{if } m=0 \text{ and } p=0\ , \\ E_{\text{coll}}(u_{m+1})\cdot f(m+1,p) & \\ \quad + E_{\text{prune}}(v_{p+1})\cdot f(m,p+1) & \text{if } 1<m<(c-2p)\cdot(c^{t-1}-1)/(c-1)\ , \\ 1 & \text{otherwise}\ . \end{cases}$$

The bound for $m=0$ and $p=0$ holds because there are n possibilities to choose the request associated to the root of the tree; if this request is fixed then the expected number of active configurations for the root is at most $E_{\text{coll}}(u_1)$, and the expected number of possibilities to finish the traversal is $f(1,0)$. If $1<m<(k-p)\cdot(c^{t-1}-1)/(c-1)$ then the next node visited in the traversal is either a collision or a pruning

```
d := 6;                                   # dim. of the butterfly
n := 2^d;                                 # number of input nodes
c := 3;                                   # module contention
C := 4;                                   # network congestion
t := 5;                                   # number of rounds

# expected number of active configurations for a collision node
Ecollc := binomial(n-1,c)/n^c;
EcollC := sum(binomial(2^i-1,C) *2^(-i*C),i=1..d-1)*(1-2^(-C));

# expected number of active configurations for the root
Erootc := binomial(n,c+1)/n^c;
ErootC := sum(binomial(2^i,C+1)*2^(-i*C),i=1..d-1)*(1-2^(-C));

# expected number of active configurations for pruning node
Eprune := (max(1,d-ceil(log[2](C+1))+1)+1) / (2*n);

# recursive function
f:=proc(m,p,q) option remember;
  if  (m=1 and 2*p>=c+1) or (m>=(c-p)*(c^(t-2)-1)/(c-1)+2)  then
    1
  else
    min(1,Ecollc*f(m+1,p,q+c)+EcollC*f(m+1,p,q+C)+q*Eprune*f(m,p+1,q));
  fi;
end;

# probability of failing
result := min(1,Erootc*f(1,0,c)+ErootC*f(1,0,C));
```

Figure 2.2: *Maple program for numerical calculation.*

node. In the first case, $E_{\text{coll}}(u_{m+1})$ is an upper bound on the expected number of active configurations for the collision node, and the expected number of possibilities to finish the traversal is $f(m+1,p)$. In the latter case, $E_{\text{prune}}(v_{p+1})$ is an upper bound on the expected number of active configurations for the pruning node, and the expected number of possibilities to finish the traversal is $f(m,p+1)$. Finally, if $m >= (c-2p)\cdot(c^{t-1}-1)/(c-1)$ then the traversal is finished and hence $f(m,p)=1$.

Solving the recursion for $f(0,0)$ for fixed n, c, C, and t yields a numerical upper bound on the probability that the collision protocol fails.

Improvements on the recursion

The values in Table 2.1 were calculated with the Maple Algebra System. The corresponding Maple program implements a recursive function as described above. However, the recursive function has been improved in several aspects. The program is given in Figure 2.2. The improvements are explained in the following.

The first change is that we give a better bound for the expected number of active configurations for a pruning node v than the bound given by Lemma 2.5.2. We exactly calculate the number of candidates for the conflicting node, i.e., the number of requests represented by the nodes in $B(v) \cup \Gamma(v) \setminus \{v\}$, rather than estimating this number by the maximum number of nodes in the witness tree, M. The number of candidates for the conflicting node is denoted by q, and we use a third parameter in the recursive function that keeps track of this value. In order to compute q we distinguish between *module* and *link collisions*, i.e., a collision of more than c requests at a memory module or a collision of more than C requests at a link in the network, respectively. Initially, q is set to 0. Within the recursion, q is increased by c when the next node of the traversal represents a module collision, and q is increased by C when the next node of the traversal represents a link collision. The expected number of active configurations for a pruning node v is bounded by

$$\frac{q \cdot \max\{1, d - \lceil \log_2(C+1) \rceil + 1\} + 1}{2 \cdot n} .$$

Apart from the replacement of M with q, this bound is about a factor of 2 smaller than the one given in Lemma 2.5.3. This can be shown as follows.

We have to estimate the expected number of configurations for v. Let r denote the request associated to v, and let r' denote the request associated to the conflicting node of v. The number of possibilities to choose r' is q. W.l.o.g., assume that the subnetwork in which the two paths represented by the pruning node overlap is BF_0. We have to give a bound on the expected number of possibilities to choose the BF_0-path p of r such that it overlaps with the BF_0-path p' of r' at some edge e that is involved either in a module or link collision represented by the non-pruned witness tree.

The random experiment that we consider is the random wiring on edge level 0 and the random mapping of the data blocks to the memory modules. The outcome of the mapping of the data block addressed by r to the memory modules in BF_0 is independent from the *revealed paths*, i.e., the paths included in the configurations of nodes visited before v. However, the outcome of the randomized level is influenced by the revealed paths as these paths fix some of the edges such that some of the nodes on level 1 of BF_0 cannot be reached by p anymore. Fortunately, this dependency is in the "right" direction since a path following a random course starting from a switch chosen randomly among the reachable switches of level 1 of BF_0 is less likely to overlap with one of the revealed paths than a path chosen randomly among all switches on this level of BF_0. Hence, we can assume in the following that p starts at a switch that is chosen randomly and uniformly from the set of all switches on level 1 of BF_0 and follows a random course through BF_0 to a randomly selected memory module.

The edge e at which p overlaps with p' has to satisfy the following condition: e can be reached either from at least $C+1$ ports, such that a link collision can take place on this edge, or e is an edge of level d, such that a module collision can take place on it. Consequently, the two paths have to overlap at an edge of one of the $d' = \min\{1, d - \lceil \log_2(C+1) \rceil + 1\}$ highest levels. The following argument takes care that we do not count configurations in which the two paths overlap on more than one of these edges repeatedly. The subpath of p' consisting of the d' possible collision edges is incident to $d'+1$ other edges such that any path overlapping with the subpath has to traverse one of these incident edges first. The probability that p traverses one of these edges is $(d'+1)/n$, and the probability that p follows the course of p' on the next level is $1/2$. Hence, the expected number of active configurations is $q \cdot (d'+1)/(2n)$, which corresponds to the bound on the expected number of active configurations for a pruning node, given above.

Furthermore, we use improved bounds on the expected number of active configurations for the collision nodes. The expected number of active configurations for a node representing a module collision

is bounded by

$$\binom{2^d-1}{c} \cdot 2^{-d} \quad ,$$

whereas the expected number of active configurations for a node representing a link collision is bounded by

$$\sum_{i=1}^{d-1} \binom{2^i-1}{C} \cdot 2^{-i} \cdot \left(1-2^{-C}\right) \quad .$$

Both bounds are slightly strengthened variants of the bound given in Equation 2.1 on page 70 of the asymptotical analysis. They differ only in the following two aspects. First, they take into account that the number of candidates for a colliding request is 2^i-1 rather than 2^i, for $1 \leq i \leq d$. Second, they take into account that collisions that occur on more than one level are overestimated in Equation 2.1 by a small factor because C paths that collide on level i, for $1 \leq i \leq d-1$, collide also on level $i+1$ with probability 2^{-C}. The factor of $\left(1-2^{-C}\right)$ in the bound above ensures that each of these events is only counted once. Similar bounds are used to estimate the expected number of active configurations for the root of the witness tree.

Finally, we have applied some modifications regarding the end of the recursion. In order to get more pruning nodes, we modify the topology of the witness tree such that the root has one more child. The additional child is assigned the same request as the root, it represents the collision of the other path, i.e., the BF_1-path, of this request. Further, we define that a traversal ϑ that has visited m collision and p pruning nodes is finished when

- $m = 1$ and $2 \cdot p \geq c+1$, or
- $m \geq (c-p) \cdot (c^{t-2}-1)/(c-1)+2$, or
- $f(\vartheta) > 1$.

We have to ensure that any traversal fulfills one of these conditions at the latest when it has visited the last non-pruned internal node of the full witness tree. Any traversal which prunes the subtree below all children of the root fulfills the first condition after visiting the $\lceil (c+1)/2 \rceil$-th pruning node. For all other traversals, it holds that at least $c-p$ of the subtrees rooted at the grandchildren of the root are not touched by any pruning. Each of these subtrees include $(c^{t-2}-1)/(c-1)$ collision nodes. Furthermore, at least the root and one of its children have to be a collision node. Consequently, $m \geq (c-p) \cdot (c^{t-2}-1)/(c-1)+2$ at the latest after visiting the last non-pruned internal node. Thus, any traversal can be continued until it fulfills either the first or the second condition. The third condition ensures that $f(m,p) \leq 1$, for any m and p, which allows us to substitute 1 for every recursive call that returns a value larger than 1. This completes the description of the recursive function implemented in the Maple program given in Figure 2.2.

2.5.4 Simulation results

The asymptotical and numerical witness tree analyses show that the collision protocol allows to choose small module and link bandwidths for the data server. However, both analyses slightly overestimate bad events because the probability that the collision algorithm fails is bounded by summing over all witness tree configurations although several of these configurations can be active at the same time.

For small networks, we can simulate the collision protocol, which gives a good impression about the quality of the bounds found in the theoretical analysis. Note that the collision algorithm is very well suited for simulations because the problem to be solved is the same for any batch of distinct requests, i.e., the collision algorithm has to choose one out of two paths for any request, one of them following a random

	$c=2$			$c=3$		$c=4$
$\log n$	$C=2$	$C=3$	$C=4$	$C=3$	$C=4$	$C=4$
2	1.54 %	1.49 %	1.50 %	0.02 %	0.02 %	0.00 %
3	2.90 %	1.44 %	1.34 %	0.01 %	0.00 %	0.00 %
4	4.22 %	1.00 %	0.99 %	0.03 %	0.02 %	0.00 %
5	3.53 %	0.52 %	0.64 %	0.01 %	0.00 %	0.00 %
6	2.90 %	0.32 %	0.33 %	0.01 %	0.00 %	0.00 %
7	2.42 %	0.11 %	0.14 %	0.00 %	0.00 %	0.00 %
8	1.38 %	0.08 %	0.03 %	0.00 %	0.00 %	0.00 %
9	0.87 %	0.03 %	0.07 %	0.00 %	0.00 %	0.00 %
10	0.51 %	0.01 %	0.00 %	0.00 %	0.00 %	0.00 %
11	0.34 %	0.00 %	0.00 %	0.00 %	0.00 %	0.00 %
12	0.18 %	0.00 %	0.01 %	0.00 %	0.00 %	0.00 %
13	34.0 %	0.01 %	0.02 %	0.00 %	0.00 %	0.00 %
14	100 %	0.00 %	0.01 %	0.00 %	0.00 %	0.00 %
15	100 %	0.01 %	0.00 %	0.00 %	0.00 %	0.00 %
16	100 %	0.00 %	0.00 %	0.00 %	0.00 %	0.00 %
17	100 %	0.00 %	0.00 %	0.00 %	0.00 %	0.00 %
18	100 %	0.00 %	0.00 %	0.00 %	0.00 %	0.00 %

Table 2.2: *Results of the simulations. Fraction of batches for which the collision algorithm fails to select a path for every request.*

course through BF_0, the other following a random course through BF_1. Thus, the simulation does not have to take care of different kinds of problem inputs, i.e., different batches. Note that batches with data hot spots do not fulfill this pleasant property.

We have simulated the collision protocol for data servers of different sizes. The smallest server that we have considered has 4 ports and 8 disks, the largest server has $262{,}144 = 2^{18}$ ports and 524,288 disks. For each size, we run 10,000 experiments each of which consisting of one batch of distinct requests. The implemented contention resolution algorithm matches the description of the collision protocol given in Section 2.5.1 except for the termination. We do not stop the algorithm after a fixed number of rounds but we stop it as soon as a path has been selected for every request or when the algorithm selects no path in a round, which means that the algorithm fails.

In each of the experiments, we compute a new random wiring for level 0 and a new random mapping of the data blocks to the memory modules. We use a pseudo-random number generator (`drand48`) from a standard *C* library. Further experiments have been done with simple hash functions based on polynomials of degree 2. However, the different ways of choosing the random numbers had no important effect on the results.

Table 2.2 gives the fraction of experiments in which the collision algorithm fails to select a path for every request for different values of $c = c_d$ and $C = c_1, \ldots, c_{d-1}$, where c corresponds to the bandwidth of the memory module and C to the bandwidth of the links in the network. For example, the table indicates

that a module bandwidth of 2 and a link bandwidth of 3 is sufficient to serve any batch of distinct requests with relative high probability. These results are slightly better than the ones of the numerical analysis which requires the module bandwidth to be 3 and the link bandwidth to be 4 or 5 in order to get a comparable probability for success.

In general, the simulations corroborate the results of the theoretical analysis. For example, they show that the link bandwidth has to be increased from 2 to 3 when the server size grows up to about 2^{13} or 2^{14} ports. A link bandwidth of 3 yields high probability for success for all simulated network sizes. Possibly, a link bandwidth of 3 is sufficient for all networks. Note that neither the theoretical analysis nor the simulation give a definite answer to the question of whether or not a bandwidth 3 is sufficient for all networks. However, we do not believe that a constant bandwidth k is sufficient because, when n is chosen to be sufficiently large, almost every path (i.e., a fraction of $1-o(1)$ of all paths) is involved in a collision with k other paths. Furthermore, the simulations show that the probability that the collision process fails decreases when the network size increases as long as the link bandwidth does not fall below some "magic" border, which is possibly in $\Theta(\log\log n/\log\log\log n)$. This behavior matches exactly the predictions of the witness tree analyses.

The column for $C=2$ and $c=2$ in the table is of particular interest because the theoretical analysis does not cover this case. In Lemma 2.5.2 the probability that a path is involved in a collision is estimated by a term of about $(\log n)/C!$. This term is greater than or equal to 1 for $C=2$ and $n=4$. In this case the witness tree analysis breaks down. However, the simulations show that the collision protocol terminates successfully with a probability of about 98 % even for servers of size $n=2^{12}=4{,}096$. The probability for success suddenly falls to 0 when n is increased to 2^{14}. This is in contrast to the results of the numerical analysis, in which the bounds on the probabilities for success decrease relatively slowly before the protocol fails completely.

Table 2.3 gives the numbers of rounds taken by the collision protocol to select a path for every request in a batch. The values in all columns, except for the first column, increase only slightly with the size of the data server. Recall that the asymptotical analysis gives an upper bound of $\lceil \log_c \log n \rceil + 2$ for the number of rounds. This bound seems to be very tight, possibly the truth is $\lceil \log_c \log n \rceil + 1$. The values in the column for $c=2$ and $C=2$ behave differently from the other values: The number of rounds increase rapidly with the size of the network before the algorithm fails at all. The theoretical analysis does not give an explanation for this phenomenon.

Overall, we can conclude that the results of the theoretical analyses are very close to the results obtained by the simulations as long as the bandwidths are sufficiently large, that is, as long as the assumptions in Theorem 2.5.1 are met. However, the simulations show that the protocol also works for smaller bandwidths. Depending on the success probability that should be guaranteed by the server, there are at least two interesting alternatives: The first alternative is to choose a module bandwidth of $c=2$ and a link bandwidth of $C=3$. This yields a success probability of about 99 % for all investigated server architectures. The second alternative is to set $c=C=3$, which yields a success probability of about 99.99 % or even larger. The most interesting theoretical questions are: how does the protocol behave for small bandwidths? and does a link bandwidth of 3 (or any other constant term) suffice for networks of arbitrary size?

2.6 Serving batches including data hot spots

Up to now we have assumed that all requests in a batch are directed to different data objects. In practice, however, often several users wish to access the same data block at the same time. This event is called a *data hot spot*. We define the *size* of a data hot spot to be the number of users interested in the same block.

	$c=2$			$c=3$		$c=4$
$\log n$	$C=2$	$C=3$	$C=4$	$C=3$	$C=4$	$C=4$
2	1.03 (2)	1.03 (2)	1.03 (2)	1.00 (1)	1.00 (1)	1.00 (1)
3	1.34 (4)	1.21 (3)	1.21 (3)	1.01 (3)	1.01 (2)	1.00 (1)
4	1.94 (5)	1.55 (3)	1.50 (3)	1.10 (2)	1.04 (2)	1.00 (2)
5	2.51 (10)	1.92 (5)	1.83 (4)	1.38 (3)	1.14 (2)	1.01 (2)
6	3.27 (10)	2.11 (4)	2.03 (4)	1.80 (3)	1.35 (3)	1.08 (2)
7	4.17 (11)	2.31 (5)	2.11 (4)	2.00 (3)	1.68 (2)	1.27 (2)
8	5.26 (11)	2.67 (4)	2.22 (4)	2.04 (3)	1.94 (3)	1.61 (2)
9	6.62 (12)	2.97 (5)	2.43 (4)	2.16 (3)	1.99 (2)	1.93 (2)
10	8.46 (15)	3.06 (5)	2.72 (4)	2.52 (3)	2.00 (3)	1.99 (2)
11	10.26 (18)	3.23 (5)	2.94 (4)	2.94 (4)	2.00 (3)	2.00 (2)
12	16.96 (38)	3.62 (5)	3.00 (4)	3.00 (4)	2.00 (3)	2.00 (3)
13	47.22 (144)	3.97 (5)	3.01 (4)	3.01 (4)	2.01 (3)	2.00 (2)
14		4.00 (6)	3.02 (4)	3.04 (4)	2.03 (3)	2.00 (3)
15		4.05 (5)	3.06 (4)	3.26 (4)	2.10 (3)	2.00 (3)
16		4.23 (5)	3.14 (4)	3.85 (4)	2.25 (3)	2.01 (3)
17		4.79 (5)	3.32 (4)	4.00 (4)	2.58 (3)	2.03 (3)
18		5.00 (6)	3.62 (4)	4.00 (5)	2.90 (3)	2.08 (3)

Table 2.3: *Results of the simulation. Average (and maximal) number of rounds over all batches that have been served successfully.*

In this section, we investigate the problems that arise with data hot spots, and give solutions for these problems.

One approach to overcome data hot spots is to substitute several *proxy blocks* for the most requested data blocks: Each data block that is accessed up to at most k times in a batch is replaced by k proxy blocks. Each request that is directed to the original data block is redirected to a randomly chosen one of the proxy blocks. Let n denote the number of user ports. Then this method reduces the size of the data hot spots to $O(\log n/\log\log n)$, w.h.p. We will see that this reduction can help the collision protocol on the RB_n-server. However, when the access patterns vary with time such that they cannot be well predicted, then it is difficult to estimate how many proxy blocks should be created for each data block.

Another way to reduce data hot spots is to break large data blocks into many small pieces. For example, movies that are stored on a video server could be divided into small sequences each of which taking only some seconds. One can expect that this method reduces the size of the data hot spots by a great amount as it is unlikely that all users want to view the same movie at the same time. However, sometimes users behave strangely, e.g., for habitual reasons, they could decide to watch an old "Derrick" crime movie at Friday 8.15 p.m. Nevertheless, the method of breaking the data into many small pieces seems to be a useful method to decrease the size of data hot spots, but it cannot solve the hot spot problem completely.

A third method, which can be used in conjunction with the two other methods, is *combining*. Here requests that are directed to the same block are combined to form a single request when they meet at an

internal switch of the network so that the requested data is sent along a *multicast tree* from a memory module that stores the requested data block to the set of user ports requesting the block. Combining all requests that are directed to the same block guarantees that each link is traversed by at most one routing path for each data block and each memory module has to deliver each data block only once. Hence, combining completely relieves data hot spots at the memory modules. However, combining requires that the contention resolution protocol is able to establish multicast trees rather than only simple paths.

In the following, we investigate how combining can be used to serve batches with data hot spots. At first, we will describe a generalization of the collision protocol that is able to serve batches with data hot spots having a maximum size of $n^{1-\alpha}$, for any constant $\alpha > 0$, on the RB_n-server with module bandwidth $O(1)$ and link bandwidth $O(\log\log n/\log\log\log n)$, w.h.p. Afterwards, we will give a counterexample that shows that the collision protocol fails for some batches with larger data hot spots unless the link bandwidth is set to $\Omega(\log n/\log\log n)$. Interestingly, this counterexample holds for any contention resolution protocol on the RB_n-server, which shows that we have to modify the server architecture in order to be able to serve batches with arbitrary data hot spots efficiently. We will solve this problem by adding further randomization to the network design. The result will be a server with module bandwidth $O(1)$ and link bandwidth $O(\log\log n/\log\log\log n)$ that serves any batch with arbitrary data hot spots. At the end of this section, we will give some simulation results which show that the constants in the asymptotic bounds are very small.

2.6.1 An algorithm for batches with small data hot spots

We describe a generalization of the collision protocol introduced in section 2.5. The generalized protocol uses combining and is able to handle batches with data hot spots of size $n^{1-\alpha}$, for arbitrary constant $\alpha > 0$.

The protocol uses a slightly different rule to select a path in a round. It takes into account that different requests can be directed to the same block. Let c_i denote the bandwidth of the edges on level i, for $1 \leq i \leq \log n$ with $c_{\log n}$ corresponding to the module bandwidth. For an edge e in a fixed round, we define $\ell(e)$ to denote the maximum size of a set of active paths that traverse e and belong to requests that are directed to distinct data blocks. A path p is eligible to be selected in the round if $\ell(e_i) \leq c_i$, for any $1 \leq i \leq \log n$, where e_i denotes the edge of p on level i. If p and its buddy path p' are both eligible to be selected, one of the two paths is selected in an arbitrary fashion. Note that, if all requests are directed to distinct blocks, i.e., the data hot spot size is 1, than the collision protocol with combining is equivalent to the collision protocol described in the previous section.

When each request is satisfied then all selected paths that are directed to the same block are merged to form a multicast tree. It is easy to check that if the protocol finishes then each request is satisfied and at most c_i paths traverse any edge on level i, for $1 \leq i \leq \log n$. In order to execute the collision protocol with combining on the network in a distributed fashion, we need a packet routing algorithm that is able to combine and split packets. The packet routing algorithm of Ranade [66] fulfills this condition. It takes time $O(\log n)$ for each round, w.h.p. The following theorem shows that the collision protocol with combining achieves the same asymptotic bounds on the number of rounds and the module and link bandwidths as the original protocol for batches of distinct requests.

Theorem 2.6.1 *Suppose the maximum data hot spot size is $n^{1-\alpha}$, for some arbitrary constant $\alpha > 0$. Then the collision protocol with combining selects a routing path for any request on a RB_n-server with module bandwidth $O(1)$ and link bandwidth $O(\log\log n/\log\log\log n)$ in $O(\log\log n)$ rounds, w.h.p.*

Proof. We will show the following result that illustrates the influence of the data hot spot size. We use the notations of Theorem 2.5.1. Let $\varepsilon > 0$ denote an arbitrary constant term. Suppose $c_1, \ldots, c_{\log n}$ are chosen sufficiently large such that $\sum_{i=1}^{\log n} 1/c_i! \geq 1/(1+\varepsilon)$ but $c_i \leq \log n$, for $1 \leq i \leq \log n$. Define $b = \min\{c_i \mid 1 \leq i \leq \log n\}$. Then the probability that the collision algorithm takes more than $\log_c \log n + O(1)$ rounds is $(\log n)^{O(c)} \cdot H^c/n^{c-1}$ with H denoting the maximum data hot spot size. Setting $H \leq n^{1-\alpha}$ and $c > 1/\alpha$ yields the theorem.

Not only the above result is a generalization of the result in Theorem 2.5.1 but also the following proof is a generalization of the proof for that theorem. First, we construct a witness tree. Each internal node of the witness tree corresponds to a collision of the request represented by the node itself and its children. During the construction we ensure that the data blocks addressed by the requests involved in each of these collisions are distinct, that is, if more than one request directed to the same block are involved in a collision then we choose one of them arbitrarily. Note that at the combining ensures that at least $c_i + 1$ requests to different data blocks are involved in every collision on level i.

Next, we prune the witness tree and compute the expected number of pruned trees. We use the pruning rules described in Section 2.5.2, but we change them slightly: When a node v is visited we apply a pruning if one of v's children has associated a request that is directed to a data block that is also addressed by a request associated to a node in $B(v) \cup \Gamma(v)$. The rest of the pruning rules remain unchanged. Note that, according to the original pruning rules, a pruning is applied only if one of v's children has associated a request which is also associated to a node in $B(v) \cup \Gamma(v)$. The new pruning is more restrictive as it also prunes because of different requests that are directed to the same block.

We consider the influence of the modifications on the expected number of active configurations. The expected number of active configurations for a collision node is not increased. (In fact, it slightly decreases because the collision probability decreases with the number of requests that are combined.) However, the expected number of active configurations for a pruning node is increasing with the data hot spot size because of the more restrictive pruning rules. In particular, the bound on the expected number of active configuration of a pruning node given in Lemma 2.5.3 does not hold anymore because the conflicting request of a pruning node is not necessarily the request associated to the conflicting node. However, the conflicting request is one of the at most H requests directed to the same block as the request associated to the conflicting node. Therefore, the bound on $E_{\text{prune}}(v_j)$, i.e, the expected number of active configuration for a pruning node v_j, increases only by a factor of H. Consequently,

$$E_{\text{prune}}(v_j) \leq \frac{H \cdot \log n}{n - 2M} \quad .$$

Substituting this bound instead of the original bound into Equation 2.2 yields the following result: The probability that the collision algorithm takes more than $t = \log_c \log_{1+\varepsilon} n + 3$ rounds is at most

$$\begin{aligned} \sum_{T \in \mathcal{T}} E(T) &= \sum_{T \in \mathcal{T}} n \cdot \prod_{j=1}^{m} E_{\text{coll}}(u_j) \cdot \prod_{j=1}^{p} E_{\text{prune}}(v_j) \\ &\leq \sum_{T \in \mathcal{T}} n \cdot C^c \cdot (1+\varepsilon)^{-m} \cdot \left(\frac{H \cdot \log n}{n - 2M}\right)^p \\ &\leq \frac{H^b \cdot (\log n)^{O(c)}}{n^{b-1}} \quad , \end{aligned}$$

which completes the proof of Theorem 2.6.1. ■

2.6.2 A counterexample for batches with large data hot spots

The theorem above shows that the collision protocol works well as long as the data hot spot size is not too large, i.e., not larger than $n^{1-\alpha}$, for arbitrary, constant $\alpha > 0$. Now we investigate what happens if the data hot spot size is larger.

We give an example consisting of a batch of requests having a data hot spot size close to n and requiring a large link bandwidth with relatively high probability. (This probability is with respect to the random wiring on edge level 0 and the random mapping of the data blocks to the memory modules.) The result holds for any contention resolution protocol choosing one out of two randomly placed copies on the RB_n-server. This shows that a "high probability result" for serving batches with arbitrary data hot spots on the RB_n-server is not possible unless the link bandwidth is in $\Omega(\log n/\log\log n)$. Note that this bound is tight in the sense that a link bandwidth of $O(\log n/\log\log n)$ is sufficient to serve any batch of arbitrary requests, w.h.p., (even if one uses a simple random mapping without redundancy instead of the proposed redundant mapping).

Theorem 2.6.2 *Suppose the data blocks are distributed randomly with redundancy 2 among the modules of the RB_n-server as defined in Section 2.2. Let $\alpha > 0$ denote an arbitrary constant term. Suppose the link bandwidth is smaller than or equal to $\alpha \cdot \log n/24\log\log n$. Then a batch of n requests exists such that any contention resolution protocol fails to serve the batch with probability $\Omega(n^{-\alpha})$.*

Proof. Define $h = \lfloor \log((\alpha/2)\cdot(\log n/\log\log n))\rfloor$, and $H = 2^h$. We describe a batch of request with data hot spot size about $n/H = \Theta(n\cdot\log\log n/\log n)$: Each request is directed to a data block that is chosen at random from an arbitrary subset of H data blocks.

We prove two properties holding for this batch of requests. The first property is that the number of edges on level h passed by one of the two alternative paths for the requests is at most $2n/H$ with probability $\Omega(n^{-\alpha})$. The second property is that the contention resolution protocol, which selects one out of two paths and combines some of the paths, does not "eliminate" too many of the routing paths such that the number of combined paths that traverse level h after the contention resolution is $n/3$, w.h.p. (The *number of combined paths* that cross an edge denotes the size of a maximum set of paths that are directed to distinct data blocks and cross the edge.) From this we can conclude that the number of combined paths traversing one of these edges is at least

$$\frac{n/3}{2n/H} = \frac{H}{6} \leq \frac{\alpha\log n}{24\log\log n} \ ,$$

with probability $\Omega(n^{-\alpha})$, which yields the theorem.

The first property can be shown as follows. The node levels 1 to h of RB_n include $2n/H$ disjoint *small subbutterflies* of dimension $h-1$. Each of these small subbutterflies is left by H edges of level h. Consider a path (or multicast tree) passing through one of the small subbutterflies. The edge along which the small subbutterfly is left by the path is determined by the first h bits of the path's final destination, i.e., the memory on which the block addressed by the path is placed.

With probability H^{-H}, the first h bits in the disk id are the same for all copies of the H blocks in BF_0. In this case, all paths that route through the same small subbutterfly of BF_0 leave the small subbutterfly across the same edge on level h. Hence, with probability H^{-H}, all paths in BF_0 use at most n/H edges of level h of BF_0. As the same result holds for the paths in BF_1, the probability that all paths use at most $2n/H$ edges of level h is at least $H^{-2H} = \Omega(n^{-\alpha})$.

Next we show the second property, i.e.,the number of paths traversing level h after the contention resolution is $n/3$, w.h.p. First, we show that not too many of the *candidate paths*, i.e, the two alternative

paths for each request, are combined on the levels 1 to h of RB_n. Consider one of the small subbutterflies, and fix an arbitrary block x from the set of the H accessed data blocks. The probability that none of for one of the requests directed to x starts in the fixed subbutterfly is

$$\left(1-\frac{H}{n}\right)^{n/H} \leq \frac{1}{\mathrm{e}} \ .$$

As a consequence, the expected number of small subbutterflies that are not traversed by a candidate path for a request directed to x is $2n/(\mathrm{e}H)$. Applying a Chernoff bound yields that the number of small subbutterflies that are not traversed by a candidate path for a request directed to x is at most

$$\frac{2n}{\mathrm{e}H} - o\left(\frac{n}{H}\right) \leq \frac{2n}{3H} \ ,$$

w.h.p., for sufficiently large n. As this result holds for any of the H blocks, the number of combined candidate paths that traversing edge level h is at least $2 \cdot n - (2/3) \cdot n = (4/3) \cdot n$, w.h.p.

The contention resolution protocol selects only half of the $2n$ candidate paths that start at the source nodes. In particular, it removes at most n from the candidate paths that traverse edge level h. Consequently, after the path selection, the number of combined paths that traverse level h is at least $(4/3) \cdot n - n = n/3$, w.h.p., which yields the second property and, hence, completes the proof of Theorem 2.6.2. ■

2.6.3 The solution for batches with arbitrary data hot spots

The counterexample from the previous section illustrates the problems that arise with large data hot spots on the RB_n-server: The more requests are directed to the same data block the larger the dependencies between different requests become. Due to these dependencies bad events that occur simultaneously in different parts of the network become more likely. We will solve this problem by adding more randomization to the network such that events in different parts of the network become stochastically independent.

The modified server uses RB'_n as interconnection network instead of RB_n, where RB'_n is defined as follows. Define $h = \lceil (3/4) \cdot \log n \rceil$. RB'_n has the same set of nodes and edges as RB_n, except that, in addition, the straight edges on level 0 and all edges on level h are permuted at random.

The straight edges on level 0 are randomized analogously to the cross edges, that is, each node $\langle 0, v_1 \dots v_{\log n} \rangle$, of RB'_n is connected by a straight edge with node $\langle 1, v'_1 \dots v'_{\log n} \rangle$ if and only if $v_1 = v'_1$ and $\pi'_{v_1}(v_2 \dots v_{\log n}) = v'_2 \dots v'_{\log n}$, where π'_0 and π'_1 are random permutations of the set of $(\log n - 1)$-bit numbers.

The random wiring on level h is defined as follows. The nodes on the levels 1 to h of RB'_n induce $n/2^{h-1}$ disjoint *small subbutterflies* of dimension $h-1$. Let V_i denote the nodes on level h of the ith small subbutterfly, for $0 \leq i < 2^{h-1}$, Further, let ψ_i and ψ'_i denote two random permutations on V_i, for $0 \leq i < 2^{h-1}$. Then each cross edge (v,u) with $v \in V_i$ and u being a node of level $h+1$ is replaced by an edge $(u, \psi_i(v))$ and each straight edge (v,u) with $v \in V_i$ and u being a node of level $h+1$ is replaced by an edge $(u, \psi'_i(v))$, for $0 \leq i < 2^{h-1}$. Figure 2.3 shows an example for the RB'_{16}-server. Note that the picture shows the columns in BF_0 and BF_1 in bit-reversal order, which visualizes the small subbutterflies induced by the nodes on the levels 1 to $h = 3$.

It is easy to see how this modification changes the routing: A path starting at level 1 of the ith small subbutterfly that leaves the subbutterfly along a straight or cross edge (v,u) with $v \in V_i$ and u being a node of level $h+1$ has to be rerouted in such a way that it leaves the subbutterfly across node $\psi_i(v)$ or $\psi'_i(v)$, respectively.

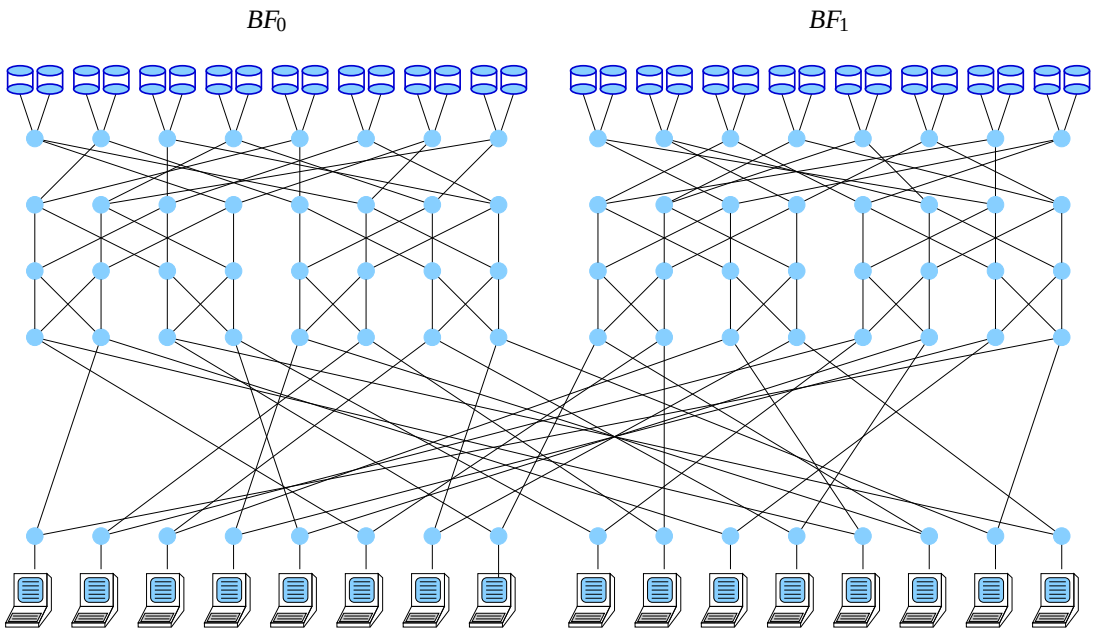


Figure 2.3: *Example for the RB'_{16}-server.*

The collision protocol for RB'_n treats requests that are involved in large data hot spots separately from the other requests. Initially, the *individual hot spot size* for each request is computed, i.e., the number of requests directed to the data block that is addressed by the considered request. Note that this value can be computed in a distributed fashion using Ranade's routing protocol, which takes time $O(\log n)$, w.h.p. Define $H = 2^{h+1} \approx 2 \cdot n^{3/4}$. We divide the bandwidth of each module and each link into two equal parts, One of which is used for the set of requests with individual hot spot size at most H, the other is used for the set of requests that are involved in data hot spots of a size larger than H. This yields two independent service problems. To both of these problems we independently apply the collision protocol with combining as defined in Section 2.6.1. The following theorem shows that the additional randomization of the network yields the desired result.

Theorem 2.6.3 *Any batch of requests with arbitrary data hot spots is served by the RB'_n-server with module bandwidth $O(1)$ and link bandwidth $O(\log\log n/\log\log\log n)$, w.h.p. The collision protocol takes $O(\log\log n)$ rounds.*

Proof. The randomized wiring of the straight edges on level 0 and the edges on level h does not change the analysis for the collision process on the set of requests with small data hot spots. Therefore, we have to give an analysis only for the set of requests with individual hot spot size larger than H.

Let c_i denote the size of the link bandwidth of the edges on level i that is reserved for the requests with large individual hot spot size, for $1 \leq i \leq \log n - 1$, and let $c_{\log n}$ denote the corresponding part of the module bandwidth. Let the *lower levels* denote the edge levels 1 to $h-1$, and the *upper levels* denote the edge levels h to $\log n$. Further, let the *lower and upper paths* denote the parts of the routing paths that cross the lower and upper levels, respectively. First, we consider the upper paths. In particular, we investigate the congestion of the *upper candidate paths*, i.e, the two alternative upper routing paths for each request.

Lemma 2.6.4 *For any δ, the probability that more than δ candidate paths that are directed to different data blocks traverse the same edge on one of the upper levels is at most*

$$\frac{1}{(\delta+1)!\cdot 2^{\delta}\cdot n^{\delta/2-1/4}} \ .$$

Proof. Let x_1 and x_2 denote two of the data blocks. The probability that two paths for requests directed to x_1 and x_2 collide at an edge of an upper level in BF_0 depends on the random mapping of the block to the memory modules in BF_0. In particular, two paths can collide on the upper levels only if the first $h+1$ bits in the module address of their data blocks are the same. The first of these address bits is 0 for all modules in BF_0. However, the other bits are determined at random such that the probability for a collision of two arbitrary requests in the upper level is at most 2^{-h}.

Next we consider collisions of more than two paths. Due to the large individual hot spot size of each request, the total number of accessed data blocks is at most n/H. Thus, the probability that more than δ upper candidate paths share an edge of BF_0 is bounded by the probability that more than δ of the accessed data blocks have the same $h+1$ upper bits in their module address. This probability is at most

$$\begin{aligned}\binom{n/H}{\delta+1}\cdot 2^{-h\cdot\delta} &\leq \left(\frac{\left(\frac{1}{2}\cdot n^{1/4}\right)^{\delta+1}}{(\delta+1)!}\right)\cdot\left(\frac{1}{n^{3/4}}\right)^{\delta}\\ &= \frac{1}{(\delta+1)!\cdot 2^{\delta+1}\cdot n^{\delta/2-1/4}} \ .\end{aligned}$$

Applying the same argument to the upper candidate paths in BF_1 yields the lemma. ∎

Lemma 2.6.4 allows us to choose a constant bandwidth for the upper levels such that no paths are blocked on the upper levels, w.h.p. We set $c_h = \cdots = c_d = \delta = O(1)$ for sufficiently large δ.

Next we describe how to select the c_i's for the lower levels. Let $\varepsilon > 0$ denote an arbitrary constant term. Suppose $c'_1,\ldots,c'_d$ are integers that are chosen sufficiently large such that $\sum_{i=1}^{h-1} 1/c'_i! \geq 1/(1+\varepsilon)$ but $c'_i \leq \log n$, for $1 \leq i < h$. Set $c_i = (c'_i+1)\cdot(2\delta-1)-1$, for $1 \leq i < h$. We will show that the probability that the collision algorithm takes more than $O(\log_{c'}\log n)$ rounds is $(\log n)^{O(c')}/n^{c'/4-1}$, where $c' = \min\{c'_i \mid 1 \leq i < h\}$. This result implies the theorem.

The random wiring on level h removes nearly all dependencies between events occurring in different small subbutterflies below level h. For example, consider two lower paths directed to the same block but traversing different small subbutterflies. These paths follow a random and independent course through the lower levels, which is determined by the random wiring on the levels 0 and h. Note that we only have to consider collisions on the lower levels as the upper routing paths do not collide, w.h.p. In general, the courses of any set of lower paths that do not pass the same node on level h are stochastically independent (except for the fact that the random wiring on the levels 0 and h forms a random permutation rather than a random function).

In the following, we describe the construction of a witness tree. In order to remove dependencies between requests that use the same edge on level h, we define a *dependency set* D_r for every request r. D_r includes all requests whose routing paths share a random edge from level h with one of the two paths of r. The witness tree construction corresponds to the description in the proof of Theorem 2.6.1, except that each internal node representing a collision on level i gets c'_i rather than c_i children. The selection of the requests associated to the children of a node is defined as follows.

Consider a node v of the witness tree representing a collision on an edge e of level $i < h$. At least c_i+1 requests that are directed to distinct blocks are involved in the collision on e. Some of these requests can be included in the dependency set of other requests as one of their routing paths share an edge from level

h with one of the routing paths of the other requests. However, the number of paths that share an edge on level h is at most δ such that the number of blocks that are addressed by the requests in a dependency set is at most $(2\cdot\delta-1)$. As a consequence, there exists a subset of size $(c_i+1)/(2\delta-1) = c_i'+1$ of the requests colliding at e that include only requests that are not included in the dependency set of each other, among them the request corresponding to v. We associate c_i' requests fulfilling this condition to the children of v.

This construction ensures that the paths for the requests involved in a collision represented by a singular witness tree node are pairwise independent (except for the permutation property, which yields that the collisions are negatively correlated). However, collisions represented by different nodes of the tree are not necessarily in dependent. In order to remove also these dependencies we apply a variation of the pruning rules described in Section 2.5.2. The only modification is that, when a node v is visited, we apply a pruning if one of v's children has associated a request which is in the dependency set of a request associated to a node in $B(v)\cup\Gamma(v)$, rather than applying a pruning only if one of the children represents a request that is associated a node in this set.

After the pruning, none of the requests that is represented by the witness tree is included in the dependency set of another request that is also represented by the tree. This ensures that the probabilities for the collisions represented by the tree are stochastically independent.

Unfortunately, the new pruning rules are more restrictive and, hence, the probability that the witness tree is pruned below a node increases. However, each dependency group D_r includes at most H requests because the number of requests which paths share a subbutterfly with one of the two paths of r is at most $2\cdot 2^h = H$. Thus, the pruning probability increases at most by a factor of H. We have considered the same effect already in the proof of Theorem 2.6.1. Analogously to that proof we conclude that the probability that a witness tree of height $O(\log_{c'}\log n)$ exists is at most

$$\frac{H^{c'}\cdot(\log n)^{O(c')}}{n^{c'-1}} = \frac{(\log n)^{O(c')}}{n^{c'/4-1}} .$$

This completes the proof of Theorem 2.6.3. ■

2.6.4 Simulation results

We conclude this section with results of simulations of the collision protocol with combining. The simulation of batches with data hot spots is much more problematic than the simulation of batches with distinct requests because the data hot spots can be constructed in many different ways, e.g., a batch can include data hot spots of different sizes, and requests directed to the same block can be issued at different subsets of ports. Obviously, the combining and, hence, the probability for success of the collision protocol depends on how the batches are composed, which is in contrast to batches of distinct requests for which the probability for success does does not depend on the respective batch.

The number of possibilities to construct batches with different data hot spots is so enormous that we cannot simulate all kinds of different batches. Therefore, we confine ourselves to simulate only some randomly generated access patterns. We produce the hot spots by reducing the number of different data blocks. Let m denote the number of different data blocks. Each data block is represented by two instances that are placed randomly on the disks of BF_0 or BF_1, respectively. Each port selects one out of the m blocks at random. Clearly, the smaller the number of block the larger is the expected data hot spot size. In particular, the expected number of accesses directed to each of the data blocks is n/m, and the maximum data hot spot size of a batch generated in this fashion is in $O(n/m+\log n)$, w.h.p.

Table 2.4 describes the result of the simulations. We simulated the collision protocol with combining on the RB_n-server as described in Section 2.6.1, not the more complicated variant of the protocol on the RB_n'-server. The simulations were done for some interesting server configurations, each of which consists

# different data blocks m	fraction of unsuccessfull exp. $c=2$ $C=2$ $\log n = 6$	$c=2$ $C=3$ $\log n = 8$	$c=2$ $C=3$ $\log n = 10$	avg. (and max.) # rounds $c=2$ $C=2$ $\log n = 6$	$c=2$ $C=3$ $\log n = 8$	$c=2$ $C=3$ $\log n = 10$
2	0.00 %	0.00 %	0.00 %	1.00 (1)	1.00 (1)	1.00 (1)
4	0.18 %	0.00 %	0.00 %	1.04 (8)	1.00 (2)	1.00 (2)
8	0.61 %	0.01 %	0.00 %	1.44 (6)	1.03 (7)	1.06 (3)
16	1.11 %	0.01 %	0.00 %	1.97 (6)	1.21 (3)	1.43 (3)
32	1.80 %	0.00 %	0.00 %	2.29 (8)	1.58 (5)	1.90 (3)
64	1.79 %	0.00 %	0.01 %	2.61 (8)	1.91 (6)	2.00 (4)
128	2.14 %	0.00 %	0.00 %	2.89 (11)	2.02 (4)	2.10 (4)
256	2.27 %	0.03 %	0.00 %	3.07 (9)	2.10 (4)	2.28 (5)
512	2.64 %	0.05 %	0.02 %	3.17 (9)	2.25 (5)	2.47 (5)
1024	2.81 %	0.08 %	0.01 %	3.22 (9)	2.43 (4)	2.71 (5)
∞	2.90 %	0.08 %	0.01 %	3.27 (10)	2.67 (4)	3.06 (5)

Table 2.4: *Results of the simulation. Each request is directed to a data block that is chosen at random out of a set of m data blocks. Each data block is represented by two instances that are placed randomly on the disks of BF_0 or BF_1, respectively.*

of a network of size n with some interesting choices for the module bandwidth c and the link bandwidth C. For each of these server configurations, we did experiments for batches including data hot spots of different sizes. Each row in the table lists the fraction of unsuccessful experiments and the number of rounds taken by the collision protocol on different server configurations for some fixed number m of accessed data blocks. For each of the configurations and each m, we ran 10,000 experiments. The last row, which is marked with ∞, models the situation in which the number of data blocks is so large that all requests are directed to distinct blocks for sure.

The results indicate that the counterexample for large data hot spots, given in Theorem 2.6.2, does not effect data servers of a relatively small size. Recall that the counterexample consists of a batch of requests that requires a link bandwidth of about $\log n/(24 \cdot \log\log n)$. Hence, n has to become very large in order to get a lower bound of $\log\log n/\log\log\log n$. In contrast to the predictions given by the asymptotical analysis, the probability for success even increases with the data hot spot size. The reason for this phenomenon is that the probability for collisions decreases with the number of requests that are combined. For example, if m is very small then collisions on the higher levels become very unlikely. However, this effect cannot be exploited for the asymptotical analysis, which has to deal with arbitrary combinations of small and large data hot spots.

Overall, we can conclude that combining is the right tool to solve the data hot spot problem on our proposed data server architecture. The changes to the data server, i.e., the extra randomization, does not seem to be necessary for small data servers.

2.7 Serving infinite sequences of requests

In this section, we investigate whether the RB_n-server is able to deal with requests that arrive one by one rather than in batches. We consider an infinite sequence of events $\sigma_1, \sigma_2, \ldots$, where σ_i is either an event requesting a data block or an event releasing a data block. We allow arbitrary *sequences of distinct requests*, i.e., sequences in which none of the ports and none of the data blocks is included in two requests that overlap in time. The sequence of events is specified by an *oblivious adversary*, that is, we assume that the sequence is specified before the random wiring and the random mapping is fixed. Hence, the sequence of events is independent of the behavior of the data server.

The sequence is revealed one by one, and the contention resolution protocol has to serve the requests in the order of presentation. *Serving a request* $\sigma_i = (\rho_i, x_i)$ means that a routing path between the port ρ_i and a memory module that holds the data block x_i has to be locked until the block is released in a later event σ_j with $j > i$. Each request σ_i has to be served within the *time step i*, i.e., the time interval between event σ_i and σ_{i+1}, including σ_i but not σ_{i+1}. Thus, the challenging task is to place the decisions for a request σ_i without knowing future events.

In the following, we propose a simple contention resolution protocol, that selects the routing paths for sequentially arriving requests. It is called "minimum protocol". We describe the minimum protocol for the case of unbounded bandwidths, that is, all requests are served, and we aim to minimize the congestion. The *congestion in a time step t* is defined to be the maximum number of routing paths that use the same edge at the end of step t. Note that the congestion is also a measure for the load of the memory modules as it includes the load on the edges that connect the memory modules with the nodes on level $\log n$ of the network. We show that, for any particular time step, the congestion produced by the minimum protocol is in $O(\log\log n)$, w.h.p. The minimum protocol can be easily adapted to the case of bounded bandwidths. In this case any request whose service exceeds the bandwidth is rejected. From the result on the congestion we can conclude immediately that this variant of the minimum protocol requires a module and link bandwidth of only $\Theta(\log\log n)$ to serve each individual request, w.h.p.

2.7.1 The minimum protocol

The minimum protocol for the RB_n-server is a generalization of the load balancing scheme of Azar et al. [8] for throwing n balls into n bins. According to this scheme, two bins are chosen at random for each ball, and the ball is placed in the bin with minimal load. Azar et al. show that this scheme yields load $\log\log n + O(1)$, w.h.p. We adapt the scheme to our path selection problem as follows.

Consider a request $\sigma_i = (\rho_i, x_i)$, where ρ_i is a user port and x_i a data block requested by ρ_i. We assume that the data blocks are distributed randomly, with redundancy 2, among the disks of BF_0 and BF_1 as described in Section 2.2. Hence, the request σ_i corresponds to two paths p_0 and p_1 leading through BF_0 and BF_1, respectively. The minimum protocol has to select one of these paths in order to satisfy the request.

For an edge e, define the congestion $c(e)$ to be the number of paths that traverse e before the request σ_i is served, and for one of the two paths p define the congestion of the path $c(p)$ to be $\max_{e\in p}(c(e))$. The minimum algorithm examines the two paths and computes $c(p_0)$ and $c(p_1)$. The request is satisfied by p_0 if $c(p_0) \leq c(p_1)$, otherwise, the request is satisfied by p_1.

In the following, we examine the congestion produced by the minimum protocol, first, in an asymptotical analysis and, afterwards, by simulations. We will see that the congestion achieved by the minimum protocol is slightly worse than the one achieved by the collision protocol. Note, however, that the minimum protocol has to serve the requests one by one, which is much harder than serving batches of

simultaneously arriving requests because the decisions for each request σ_i have to be made without any knowledge of future requests.

2.7.2 Asymptotical analysis of the minimum protocol

We show that the minimum protocol achieves a congestion of order $\log\log n$ on the RB_n-server at any particular point of time, w.h.p. Note that this congestion guarantee only holds for any particular time step. Clearly, if the minimum process runs over an infinite period of time, then the congestion can exceed this bound at some time steps. For example, if the algorithm has to serve a sequence of n^{3n} requests, all of them directed to different blocks, then, with high probability, there are some time steps at which almost all existing requests are mapped to the the same memory module in BF_0 and in BF_1. In this case, the congestion is $\Theta(n)$ regardless of the decisions of the minimum protocol. However, our bound shows that these events are very rare.

The minimum protocol is deterministic. Nevertheless, the following theorem gives a high probability bound on the congestion. This probability is with respect to the random wiring on level 0 of the network and the random mapping of the data blocks.

Theorem 2.7.1 *At any fixed time t, the probability that the congestion exceeds $O(\log\log n)$ is $n^{-\Omega(\log\log n)}$.*

Proof. Although the minimum protocol and the underlying sequential model is different from the collision protocol and the batch model, we use more or less the same proof technique. As in the analysis for the collision protocol, we first construct a witness tree, then prune this tree, and, finally, bound the probability for the existence of such a tree.

Constructing a witness tree. Fix the links on the randomization level and the mapping of the data blocks to the memory modules. This determines two choices of paths for each request. Assume that there is an edge e with congestion larger than $4c$ at time step t, where c is an integer chosen minimal such that $c \geq \log(\log n + 1) + 1$ and $c! \geq 2 \cdot \log n$. Let p denote the last path mapped to edge e before or in time step t. When p was mapped to e there were already $4c$ other paths present at this edge. Let $p_1, \ldots, p_{4c}$ denote these paths such that p_i was mapped to e at time t_i with $t_i < t_{i+1}$. The root of the tree is the request corresponding to p and the requests corresponding to $p_1, \ldots, p_{4c}$ are its children. Now we consider the buddies $p'_1, \ldots, p'_{4c}$ of these paths. Path p'_i traverses an edge with congestion at least $i-1$ at time t_i, because the congestion of p_i is not larger than the congestion of p'_i at time i, and when p_i was mapped to e there were already $i-1$ other paths present at this edge. As a consequence, we can construct a tree by applying the argument above recursively to $p'_2, \ldots, p'_{4c}$.

The tree constructed above is irregular in that nodes have varying degrees. However, it contains a c-ary tree of height c, which we call the *witness tree*, with the following properties.

- The node on level 0, i.e., the root, has c children that are internal nodes.
- Each internal node on levels $1, \ldots, c-2$ has two children that are internal nodes and $c-2$ children that are leaves, and each internal node on level $c-1$ has c children that are leaves.

We define M to denote the number of nodes in the witness tree, that is, $M = 1 + c \cdot (1 + c \cdot (2^{c-1} - 1)) \leq c^2 \cdot 2^{c-1}$.

Pruning the witness tree. The pruning is done by a breadth-first-search traversal of the tree. We use the same definitions for $B(v)$ and $\Gamma(v)$ as in Section 2.5.2. However, the pruning rules are slightly different. When a node v is visited, the following rules are applied.

a) If one of the children of v has associated a path starting at the same port as another path belonging to a request associated to a node w in $B(v) \cup \Gamma(v)$, then the tree is pruned below v. If w is from $\Gamma(v)$ then the tree is also pruned below w. If there is more than one choice for a child of v fulfilling this condition, we arbitrarily choose one of them. If the request associated to this child is directed to a data block that is also accessed by another request associated to a node w' from $\Gamma(v) \setminus \{w\}$, then we also prune the tree below w'. (Note that the connected pairs of ports and data blocks vary with time. Hence, the request associated to a child of v can be in conflict with two different nodes from $B(v) \cup \Gamma(v)$ with regard to the port and the data block, respectively.)

b) The following rule is applied only if Rule a) is not applied. If one of the children of v has associated a path directed to the same data block as another path belonging to a request associated to a node w in $B(v) \cup \Gamma(v)$, then the tree is pruned below v. If w is from $\Gamma(v)$ then the tree is also pruned below w. If there is more than one choice for the node w, we make the choice arbitrarily.

The node v is called *pruning node*, and w and w' are called the *conflicting nodes* of v. Note that each pruning node has a pruning node w but only some of the pruning nodes have a second pruning node w'.

We continue the pruning process till either there are no more nodes to visit or there are $k = \lceil c/3 \rceil$ pruning nodes. In the latter case, we apply a final pruning. If v is the kth pruning node, we remove from the tree all nodes not included in $B(v) \cup \Gamma(v)$. The remaining tree is called the *pruned witness tree*.

Each internal node u of the pruned witness tree represents a collision of the requests associated to u and its children, and, therefore, the internal nodes are called *collision nodes*. Each pruning node v represents two overlapping paths, one is associated to v and the other is associated to its conflicting node w. (The second pruning node is not considered here.) Analogously to Section 2.5.2, we define the *configuration* of the collision and pruning nodes to consist of the paths involved into the respective collision or pruning event. These configurations have the following three properties, which differ only slightly from the properties in Section 2.5.2.

Property 1: Any request is associated to at most one node of the pruned witness tree, and each BF_0- or BF_1-path is included in the configuration of at most one collision node. In particular, each edge of the randomization level and each randomly placed copy of a data block is involved in at most one of these configurations. This property follows from the pruning of the subtrees below the pruning nodes.

Property 2: Neither the user port nor the data block of the request of a pruning node v is included in the configuration of any collision node or any pruning node visited before v. (Ports and blocks of the requests of the conflicting nodes can be included in more than one pruning configuration.) This property follows from the pruning of the subtrees below the conflicting nodes w and w' from $\Gamma(v)$.

Property 3: The number of collision nodes, m, and the number of pruning nodes, p, fulfill the constraint $m \geq (k-p) \cdot \log n$. This is because, at least $k-p$ of the subtrees rooted at the grandchildren of the root remain untouched by the pruning, and each of these subtrees have size at least $(2^{c-1}-1) \geq \log n$, because $c \geq \log(\log n + 1) + 1)$.

Bounding the probability of the existence of a witness tree. Define the *tree shape* to be a description of the topology of a pruned witness tree, which is determined by the up to k pruning nodes and the at most two conflicting nodes for each of these nodes. Let $\mathcal{T}$ denote the set of all tree shapes, and, for $T \in \mathcal{T}$, let $E(T)$ denote the expected number of active witness tree configurations with tree shape T.

As in Section 2.5, we estimate the expected number of active configurations for the individual collision and pruning nodes. The major difference to the analysis in that section is that the existing requests and their BF_0- and BF_1-paths vary with time. However, each internal node of the witness tree represents a collision that takes place at a fixed point of time. This point of time is determined by the request associated to the node; it is in the time step in which this request was issued. We will use this fact for bounding the expected number of configurations for the collision nodes.

A further problem we have to deal with is that the adversary is allowed to specify which port issues a request in which time step. This possibly adds dependencies to our analysis, e.g., if we assume that a collision occurs in BF_1 but the adversary has inserted only a few requests for some neighbored ports below BF_0 then a pruning event becomes very likely.

One way to solve this problem is, e.g., to randomly permute also the straight edges. However, we prefer a solution which does not change the network. We use random naming for the requests and add dummy requests to the ports that have no existing request in a time step i. For any time step i, define the set R_i to include all requests that exist at the beginning of the time step. For any port ρ that has no existing request in time step i, we add a dummy request to R_i. This dummy request is directed to x_ρ, where x_ρ is a new virtual data block placed randomly on the memory modules of BF_0 and BF_1, analogously to the original data blocks. The arrival time for the dummy request in R_i is assumed to be equivalent to the arrival time of event σ_i. We assign a unique ID to each request in R_i. These IDs are randomly permuted such that the IDs give no evidence about the port that issues a request.

In the following we estimate the expected number of active pruned witness trees. We allow that these witness trees include also dummy requests. The number of these witness trees is an upper bound for the number of active witness trees without dummy requests because the dummy requests only increase the number of admissible configurations for some adversary sequences but do not change the probability for the original configurations to become active.

Fix a tree shape $T \in \mathcal{T}$. Let m denote the number of collision nodes in T, and $u_1, \ldots, u_m$ denote the collision nodes in the order of visitation. Consider node u_j. Let K denote a partial configuration including only configurations of the nodes $u_1, \ldots, u_{j-1}$. Note that, for $j > 1$, K specifies the request associated to u_j. For the root u_1, we assume that K specifies only this request. Let $E_{\text{coll}}(u_j, K)$ denote the expected number of active configurations for u_j under the assumption that K is active. Furthermore, let $E_{\text{coll}}(u_j)$ be the maximum, over all K, of $E_{\text{coll}}(u_j, K)$.

Lemma 2.7.2 $E_{\text{coll}}(u_j) \leq 1/2$.

Proof. Each collision node u_j represents a collision of its associated request with c other requests associated to its children. All these requests are in R_i, where i is the time step in which u_j arrived. W.l.o.g., we assume that the collision represented by u_j takes place in BF_0. Then Property 1 ensures that the BF_0-paths involved in the collision are independent from the paths in K.

We investigated a similar scenario already in the analysis for the collision protocol. In particular, identifying R_i with a static batch of requests and setting $c_\ell = c$, for $1 \leq \ell \leq \log n$, this situation corresponds exactly to the one investigated in the proof of Lemma 2.5.2. Consequently, $E_{\text{coll}}(u_j)$ can be estimated by

Equation 2.1, i.e.,

$$E_{\text{coll}}(u_j) \;\leq\; \sum_{\ell=1}^{d} \binom{2^\ell}{c_\ell} \cdot (2^{-\ell})^{c_\ell} \;\leq\; \sum_{\ell=1}^{d} \frac{1}{c!} \;\leq\; \frac{\log n}{c!} \quad ,$$

which is at most $1/2$ since c is chosen such that $c! \geq 2\log n$. ■

Let $p \leq k$ denote the number of pruning nodes in T, and let $v_1, \ldots, v_p$ denote the pruning nodes in the order of visitation. For a pruning node v_j and a partial configuration K of all collision nodes $u_1, \ldots, u_m$ and the pruning nodes $v_1, \ldots, v_{j-1}$, let $E_{\text{prune}}(v_j, K)$ denote the expected number of active configurations for v_j under the assumption that K is active. Let $E_{\text{prune}}(v_j)$ be the maximum, over all K, of $E_{\text{prune}}(v_j, K)$.

Lemma 2.7.3 $E_{\text{prune}}(v_j) \leq \log n/(n-2M)$.

Proof. Let w denote one of the at most two conflicting nodes corresponding to the pruning node v_j. This node is specified by the tree shape.

The request associated to v_j collides with the request associated to w. W.l.o.g., we assume that this collision takes place in BF_0. Property 2 ensures that the request associated to v_j is neither issued at the same port nor directed to the same data block as one of the requests specified in K. Thus, the BF_0-path of this request does not share a random edge from level 0 with one of the paths described in K, and, hence, it follows a random course, which is independent from the paths in K, to the memory module storing the requested data block.

This scenario is equivalent to the one investigated in the proof of Lemma 2.5.3. In particular, the rest of this proof can be adapted straightforward from there. This yields the desired result. ■

The bounds for $E_{\text{coll}}(u_j)$ and $E_{\text{prune}}(v_j)$ are independent estimations of expected values and can be multiplied in order to get an upper bound on the expected number of all configurations. Since the number of choices for the initial configuration K in $E(u_1, K)$ specifying the request associated with the root is n, we get the following upper bound on the expected number of active witness tree configurations with tree shape T.

$$\begin{aligned} E(T) &\leq n \cdot \prod_{j=1}^{m} E_{\text{coll}}(u_j) \cdot \prod_{j=1}^{p} E_{\text{prune}}(v_j) \\ &\leq n \cdot 2^{-m} \cdot \left(\frac{\log n}{n-2M}\right)^p . \\ &\overset{(*)}{\leq} n \cdot 2^{-(k-p)\cdot\log n} \cdot \left(\frac{\log n}{n-2M}\right)^p \\ &\leq n \cdot \left(\frac{\log n}{n-2M}\right)^k \quad , \end{aligned}$$

where Estimation $(*)$ follows from Property 3, i.e., $m \geq (k-p)\cdot\log n$.

The tree shape is determined by the at most k pruning and $2k$ conflicting nodes. Hence, the number of different tree shapes is at most M^{3k}. As a consequence, the probability that the congestion at time step t is at least $4c = \Theta(\log\log n)$ is at most

$$\sum_{T \in \mathcal{T}} E(T) \;\leq\; M^{3k} \cdot n \cdot \left(\frac{\log n}{n-2M}\right)^k \;=\; n^{-\Theta(\log\log n)} \quad ,$$

for $k = \lceil c/3 \rceil$, and $M \leq c^2 \cdot 2^{c-1}$. This completes the proof of Theorem 2.7.1. ■

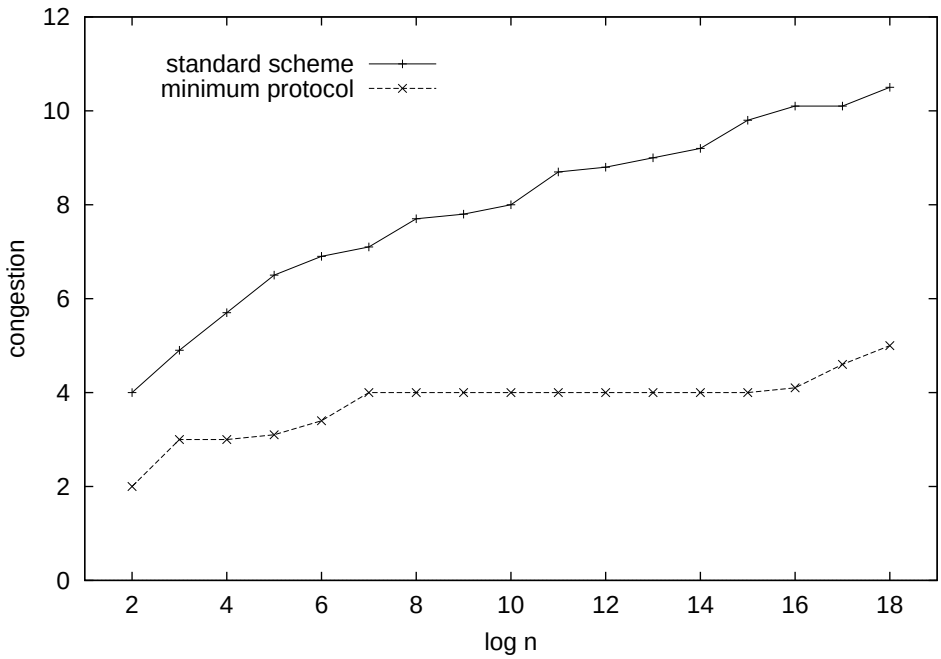


Figure 2.4: *Congestion produced by the minimum protocol and the standard scheme on the RB_n-server.*

2.7.3 Simulation results

We have simulated the minimum protocol on the RB_n-server for different values for n from $2^2 = 4$ to $2^{18} = 262{,}144$. For each choice of n, we have done 10 experiments consisting of a sequence of length $1{,}000 \cdot n$ distinct requests. The sequence is defined as follows. The ports issue their request in a round-robin fashion, that is, the ith port issues a request at time step $n \cdot j + i$, for $0 \leq j \leq 999$. The length of the time interval for each request is n, that is, a port removes a request immediately before it specifies the next one. The random wiring of level 0 and the random mapping of the objects to the memory modules is renewed for each of the 10 sequences. As for the collision protocol, we use a standard pseudo-random number generator to establish the random wiring and random mapping of the blocks to the modules.

We compare the bandwidth required for the minimum protocol on the RB_n-server with redundancy 2 with the bandwidth that is required for a non-redundant *standard scheme*, i.e., the data blocks are distributed at random, without redundancy, among the modules of the RB_n-server. The standard scheme always chooses the unique shortest path between the port that issued the request and the memory module that holds the requested block.

Figure 2.4 shows the results of the simulations for different sizes of the RB_n-server. For each of the 10 sequences, we have measured the maximum congestion over all time steps. The congestion corresponds to the bandwidth required to serve any request in the respective sequence. We only consider the congestion on the links of the network. However, also the module bandwidth is bounded by this value as any memory module is connected with a node on level $\log n$ of the server by a link that is taken into account for computing the congestion.

The simulation results show that the bandwidth required for the minimum algorithm is by about a factor of 2 smaller than the bandwidth required for the standard scheme. For the minimum algorithm, the deviation between the results of different experiments using networks of the same size is very small, that is, the congestion values for the 10 tested sequences for each size deviate only by an additive term of at

most 1. The deviation for the standard scheme is slightly larger, that is, the results for some of sequences on networks of the same size deviate by 3.

Obviously, the simulation results support the results of our theoretical analysis. The theoretical analysis predicts that the congestion produced by the the minimum protocol is $O(\log\log n)$ at any step, where the constant factor hidden in the big "O" is about a factor of 4. Our simulation results, however, indicate that the "real" congestion achieved by the minimum protocol is rather $1 \cdot \log\log n$ than $4 \cdot \log\log n$.

2.8 Conclusions

We have introduced a new proposal for a scalable multimedia data server architecture in which n user ports are connected by a sparse butterfly-like network with $2n$ memory modules. The data server uses a redundant data layout scheme such that each user request can be satisfied by two memory modules alternatively. We have devised and analyzed two contention resolution protocols that exploit the redundancy in order to minimize the bandwidth needed for the memory modules and the links of the network.

One of the presented contention resolution protocols, the collision protocol, is able to serve arbitrary batches of requests arriving in parallel. The protocol requires only small constant module bandwidth, e.g., 2 or 3, and a link bandwidth of $O(\log\log n/\log\log\log n)$. A further presented contention resolution protocol, the minimum protocol, serves requests that arrive one by one rather than in batches. We have shown that the minimum protocol requires a module and link bandwidth of only $O(\log\log n)$ to serve any individual request of an infinite sequence of distinct requests, w.h.p.. The results for both protocols improve exponentially on the known results for non redundant placement.

In addition to the asymptotic bounds on the required bandwidths we have computed numerical and simulation results which show that the "real" constants in the bounds are very small. The simulation results show that the redundant placement in conjunction with a clever contention resolution protocol is clearly superior to a non-redundant placement, even for data servers of a relatively small size. For example, the simulation of the collision protocol shows that a module bandwidth of 2 and a link bandwidth of 3 is sufficient for serving any batch of distinct requests with probability 0.999 on a server with 256 user ports and 512 memory modules. Let us compare this result with an arbitrary alternative server architecture that has the same number of ports and modules but does not use redundancy. Regardless of the distribution of the data blocks among the modules, the non-redundant server requires a module bandwidth of at least 6 (rather than 2) to serve any batch with probability 0.999. Now suppose that only memory modules of bandwidth 2 are available and we want to obtain the same probability bound of 0.999 by increasing the number of modules rather than using redundancy. Then we need at least 52,416 memory modules in order to serve batches of only 256 requests!

The endpoints of the cross edges on the lowest level of the RB_n-server are randomly permuted. The random wiring on this level is defined by two randomly chosen permutations π_0 and π_1. The calculated bounds on the link bandwidths do not hold if the cross edges are chosen as in a standard butterfly, i.e., if π_0 and π_1 are defined to be the identical function. The reason for this effect is that simultaneous collisions in BF_0 and BF_1 become more likely if π_0 and π_1 are chosen symmetrically. For example, $k = o(\log n/\log\log n)$ requests that collide on level $\lceil\log k\rceil$ of BF_0 also collide with a probability about $k^{-k} = \omega(1/\sqrt{n})$ on the same level in BF_1, which yields a congestion of at least $k/2$. Similar examples hold if π_0 and π_1 are defined by arbitrary combinations of bit-permutations, e.g., if π_0 is chosen to be the identical function and π_1 is chosen to be the bit-reversal function. On the other hand, however, we know that at least one deterministic choice of the two permutations π_0 and π_1 exists so that Theorem 2.5.1, which specifies the required bandwidths for the collision protocol on batches of distinct requests, holds

for the deterministic network defined by π_0 and π_1. This follows immediately from the fact that all batches of distinct requests are equivalent as long as the random mapping of the blocks to the memory modules is not fixed. It is an open question what these two permutations look like.

In our analyses, we have considered a scenario with relative small load as the average number of requests directed to a memory module is $1/2$ and the average number of routing paths that traverse a link is $1/2$ as well. The motivation for considering this scenario is the real-time constraints that the server has to fulfill. Obviously, the smaller the number of requests is that a module has to serve at the same time, the smaller the guaranteed bound on the response time can be. Minimizing the maximum load in a scenario with high load is also an interesting problem: Suppose that $m > n \cdot \log n$ requests should be served by n memory modules. It is well known that, if the blocks are distributed randomly without redundancy among the modules, the expected maximum number of requests that have to be served by the same module is $m/n + \Theta(\sqrt{m \cdot \log n/n})$. Obviously, the additive deviation from the expected load depends on the number of balls, m. Possibly, a "two-choice" algorithm is able to achieve an additive deviation that does not depend on m or even a deviation that is independent of m and n. Hence, it is an interesting problem to investigate "two-choice" algorithms for a large number of balls.

We have devised contention resolution protocols for requests that arrive either in parallel batches or sequentially, one by one. The parallel collision protocol achieves slightly smaller congestion than the sequential minimum protocol. The reason for this fact is that the minimum protocol places the requests without considering future requests whereas the collision protocol coordinates the choices for all requests that are included in a batch. Obviously, the minimum protocol can also be used to serve parallel batches of requests. However, the decisions for the requests have to be made sequentially. The collision protocol can be used to serve sequences of requests, too, but this requires that the requests join up to form batches, which eventually causes large delays for some of the requests. We believe that one of the most challenging tasks in the design of a multimedia data server is to devise an access scheme that is able to manage several parallel streams of requests without delaying individual requests. We have done some steps towards a solution of this problem.

Bibliography

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

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

[3] S. Arora, F. T. Leighton, and B. M. Maggs. On–line algorithms for path selection in a nonblocking network. *SIAM Journal on Computing*, 25(3):600–625, 1996.

[4] 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.

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

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

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

[8] Y. Azar, A. Broder, A. Karlin, and E. Upfal. Balanced allocations. In *Proc. of the 26th ACM Symp. on Theory of Computing (STOC)*, pages 593–602, 1994.

[9] J. E. Barnes and P. Hut. A hierarchical $O(N \log N)$ force calculation algorithm. *Nature*, 324(4):446–449, 1986.

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

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

[12] L. A. Bassalygo and M. S. Pinsker. Complexity of an optimum nonblocking switching network without reconnections. *Problems of Information Transmission*, 9:64–66, 1974.

[13] K. E. Batcher. Sorting networks and their applications. In *"AFIPS" Conference Proceedings 32*, pages 307–314, 1968.

[14] B. Beizer. The analysis and synthesis of signal switching networks. In *Proceedings of the Symposium on Mathematical Theory of Automata*, pages 563–576, 1962.

[15] V. E. Beneš. Optimal rearrangeable multistage connecting networks. *Bell System Technical Journal*, 43:1641–1656, 1964.

[16] 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.

[17] G. E. Blelloch, C. E. Leiserson, B. M. Maggs, C. G. Plaxton, S. J. Smith, and M. Zagha. A comparison of sorting algorithms for the Connection Machine CM–2. In *Proc. of the 3rd ACM Symp. on Parallel Algorithms and Architectures (SPAA)*, pages 3–16, 1991.

[18] R. Cole, B. M. Maggs, F. Meyer auf der Heide, M. Mitzenmacher, A. W. Richa, K. Schröder, R. K. Sitaraman, and B. Vöcking. Randomized protocols for low–congestion circuit routing in multistage interconnection networks. In *Proc. of the 30th ACM Symp. on Theory of Computing (STOC)*, pages 378–388, 1998.

[19] R. J. Cole, A. Frieze, B. M. Maggs, M. Mitzenmacher, A. W. Richa, R. K. Sitaraman, and E. Upfal. On balls and bins with deletions. In *Proc. of the RANDOM'98*, 1998.

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

[21] D. E. Culler, A. Dusseau, R. Martin, and K. E. Schauser. Fast parallel sorting under LogP. In *Portability and Performance for Parallel Processing*, pages 71–98, 1994.

[22] 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.

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

[24] R. Diekmann, J. Gehring, R. Lüling, B. Monien, M. Nübel, and R. Wanka. Sorting large data sets on a massively parallel system. In *Proc. of the 6th IEEE Symp. on Parallel and Distributed Processing (SPDP)*, pages 2–9, 1994.

[25] M. Dietzfelbinger and F. Meyer auf der Heide. Simple, efficient shared memory simulations. In *Proc. of the 5th ACM Symp. on Parallel Algorithms and Architectures (SPAA)*, pages 110–119, 1993.

[26] D. Dowdy and D. Foster. Comparative models of the file assignment problem. *Computing Surveys*, 14(2):287–313, 1982.

[27] F. Fabbrocino, J. R. Santos, and R. Muntz. Implicitly scalable, fully interactive multimedia storage server. In *Proc. of the 2nd International Workshop on Distributed Interactive Simulation and Real Time Applications (DIS-RT)*, 1998.

[28] S. Felperin, P. Raghavan, and E. Upfal. Parallel communication with limited buffers. In *Proc. of the 33rd IEEE Symp. on Foundations of Computer Science (FOCS)*, pages 563–572, 1992.

[29] M. J. Garey and D. S. Johnson. *Computers and intractability: A guide to the theory of NP–completeness*. Freeman, San Francisco, 1994.

[30] D. J. Gemmell, H. M. Vin, D. D. Kandlur, P. Venkat Rangan, and L. Rowe. Multimedia storage servers: A tutorial and survey. *IEEE Transactions on Computers*, 28(5):40–49, 1995.

[31] A. J. v. d. Goor. *Computer Architecture and Design*. Addison-Wesley, 1994.

[32] T. Hagerup and C. Rüb. A guided tour of Chernoff bounds. *Information Processing Letters*, 33:305–308, 1989/90.

[33] Ikeno. A limit on crosspoint number. *IRE Transactions on Information Theory*, 5:187–196, 1959.

[34] 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 *Proc. of the 29th ACM Symp. on Theory of Computing (STOC)*, pages 654–655, 1997.

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

[36] A. R. Karlin and E. Upfal. Parallel hashing – an efficient implementation of shared memory. In *Proc. of the 18th ACM Symp. on Theory of Computing (STOC)*, pages 160–168, 1986.

[37] R. Karp, M. Luby, and F. Meyer auf der Heide. Efficient PRAM simulation on a distributed memory machine. *Algorithmica*, 16, 1996.

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

[39] D. E. Knuth. *The Art of Computer Programming, Volume 3*. Addison–Wesley, 1998.

[40] R. R. Koch. Increasing the size of a network by a constant factor can increase performance by more than a constant factor. In *Proc. of the 27th IEEE Symp. on Foundations of Computer Science (FOCS)*, pages 221–230, 1988.

[41] 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, Jan. 1997.

[42] C. Krick, H. Räcke, B. Vöcking, and M. Westermann. *DIVA — The Distributed Variables Library, Version 1.0, Reference Manual, in preparation*. University of Paderborn, 1998.

[43] C. P. Kruskal and M. Snir. The performance of multistage interconnection networks for multiprocessors. *IEEE Transactions on Computers*, c-32(12):1091–198, 1983.

[44] F. T. Leighton. Randomized parallel communication. In *Proceedings of the ACM SIGACT–SIGOPS Symposium on Principles of Distributed Computing*, pages 60–72, 1982.

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

[46] F. T. Leighton. Methods for message routing in parallel machines. In *Proc. of the 24th ACM Symp. on Theory of Computing (STOC)*, pages 77–96, 1992.

[47] F. T. Leighton and B. M. Maggs. Fast algorithms for routing around faults in multibutterflies and randomly–wired splitter networks. *IEEE Transactions on Computers*, c–41(5):578–587, 1992.

[48] 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.

[49] 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.

[50] C. Lund, N. Reingold, J. Westbrook, and D. Yan. On-line distributed data management. In *Proc. of the 2nd European Symposium on Algorithms (ESA)*, 1996.

[51] P. D. Mackenzie, C. G. Plaxton, and R. Rajaraman. On contention resolution protocols and associated probabilistic phenomena. *Journal of the ACM*, 45(2):324–378, 1998.

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

[53] B. M. Maggs and R. K. Sitaraman. Simple algorithms for routing on butterfly networks with bounded queues. In *Proc. of the 24th ACM Symp. on Theory of Computing (STOC)*, pages 150–161, 1992.

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

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

[56] F. Meyer auf der Heide, C. Scheideler, and V. Stemann. Exploiting storage redundancy to speed up randomized shared memory simulations. *Theoretical Computer Science*, 162:245–281, 1996.

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

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

[59] D. Nassimi and S. Sahni. Parallel algorithms to set up the Beneš permutation network. *IEEE Transactions on Computers*, c–31(2):148–154, 1982.

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

[61] B. Ozden, R. Rastogi, and A. Silberschatz. Disk striping in video server environments. In *Proc. of the 7th International Conference on Multimedia Computing Systems (ICMS)*, pages 580–589, 1997.

[62] N. Pippenger. Parallel communication with limited buffers. In *Proc. of the 25th IEEE Symp. on Foundations of Computer Science (FOCS)*, pages 127–136, 1984.

[63] N. Pippenger. Self-routing superconcentrators. *Journal of Computer and System Sciences*, 52(1):53–60, 1996.

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

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

[66] A. G. Ranade. How to emulate shared memory. *Journal of Computer and System Science*, 42:307–326, 1991.

[67] A. G. Ranade, S. Schleimer, and D. S. Wilkerson. Nearly tight bounds for wormhole routing. In *Proc. of the 35th IEEE Symp. on Foundations of Computer Science (FOCS)*, pages 347–355, 1994.

[68] V. Rottmann, P. Berenbrink, and R. Lüling. A simple distributed scheduling policy for parallel interactive continuous media server. *Parallel Computing*, 23:1757–1776, 1998.

[69] J. Santos and R. Muntz. Design of the RIO (Randomized I/O) storage server. Technical Report 970032, UCLA Computer Science Department, May 1997.

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

[71] R. Sedgewick. *Algorithms, Second Edition*. Addison–Wesley, 1988.

[72] P. J. Shenoy, P. Goyal, S. Rao, and H. M. Vin. Design and implementation of Symphony: An integrated multimedia file system. In *Proc. of the ACM/SPIE Multimedia Computing and Networking (MMCN'98)*, pages 124–138, 1998.

[73] P. J. Shenoy and H. M. Vin. Efficient striping techniques for multimedia file servers. In *Proc. of the 7th International Workshop on Network and Operating System Support for Digital Audio and Video (NOSSDAV)*, pages 25–36, 1997.

[74] P. S. Singh, J. L. Hennessy, and A. Gupta. Scaling parallel programs for multiprocessors: Methodology and examples. *IEEE Transactions on Computers*, 26(7):42–50, 1993.

[75] P. S. Singh, W.-D. Weber, and A. Gupta. SPLASH: Stanford parallel applications for shared-memory. *Computer Architecture News*, 20(1):5–44, 1992.

[76] E. Upfal. Efficient schemes for parallel communication. *Journal of the ACM*, 31(3):507–517, 1984.

[77] E. Upfal. An $O(\log N)$ deterministic packet routing scheme. *Journal of the ACM*, 39(1):55–70, 1989.

[78] E. Upfal and A. Wigderson. How to share memory in a distributed system. *Journal of the ACM*, 34:116–127, 1987.

[79] L. G. Valiant. A scheme for fast parallel communication. *SIAM Journal on Computing*, 11(2):350–361, 1982.

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

[81] M. Vernick, C. Venkatramani, and R. Chiueh. Adventures in building the Stony Brook video server. In *Proc. of the ACM Multimedia*, pages 287–295, 1996.

[82] A. Wachsmann and R. Wanka. Sorting on a massively parallel system using a library of basic primitives: Modeling and experimental results. In *Proc. of the 3rd European Conference in Parallel Processing (Euro–Par)*, pages 399–408, 1997.

[83] A. Waksman. A permutation network. *Journal of the ACM*, 15(1):159–163, 1968.

[84] O. Wolfson, S. Jajodia, and Y. Huang. An adaptive data replication algorithm. *ACM Transactions on Data Base Systems (TODS)*, 22(4):255–314, June, 1997.

[85] O. Wolfson and A. Milo. The multicast policy and its relationship to replicated data placement. *ACM Transactions on Data Base Systems (TODS)*, 16(1):181–205, March, 1991.

[86] S. C. Woo, M. Ohara, E. Torrie, J. P. Singh, and A. Gupta. The SPLASH–2 programs: Characterization and methodological considerations. In *Proc. of the 22nd ACM International Symposium on Computer Architecture*, pages 24–36, 1995.