Alcom-FT Technical Report Series
ALCOMFT-TR-01-48

Data Management in Hierarchical Bus Networks*

Friedhelm Meyer auf der Heide
Heinz Nixdorf Institute and
Department of Mathematics
& Computer Science
Paderborn University
Germany

Harald Räcke
Heinz Nixdorf Institute and
Department of Mathematics
& Computer Science
Paderborn University
Germany

Matthias Westermann
Heinz Nixdorf Institute and
Department of Mathematics
& Computer Science
Paderborn University
Germany

Abstract

A hierarchical bus network $T = (V, E)$ uses hierarchically, tree-like connected buses as a communication network. New communication technologies like SCI (Scalable Coherent Interface) (see, e.g., [6, 7]) make such networks very attractive, because they allow their easy construction and guarantee reasonable communication performance. Such networks can be modeled as tree networks: leaves correspond to processors, inner nodes to buses, edges to switches, and bandwidths of inner nodes and edges are related to bandwidths of buses and switches, respectively.

In this paper we address the problem of static data management. Given a set of shared data objects X and the read and write frequencies from the processors to the shared data objects, the goal is to compute a (maybe redundant) placement of the shared data objects to the processors, such that the congestion (the maximum over the load of all edges and inner nodes, induced by the read and write frequencies, divided by the bandwidth of the edge or inner node, respectively) is minimized.

It is known [10] that this problem can be solved optimally in linear time, if inner nodes are allowed to hold copies of shared data objects. In our model, inner nodes correspond to buses and therefore cannot store copies of shared data objects. We show that this restriction increases the complexity of the placement problem drastically: It becomes NP-hard. On the other hand, the main contribution of our paper is an approximation algorithm with runtime $O(|X| \cdot |V| \cdot \text{height}(T) \cdot \log(\text{degree}(T)))$ that increases the congestion by a factor of at most 7.

*Partially supported by the DFG-Sonderforschungsbereich 376 and the IST Programme of the EU under contract number IST-1999-14186 (ALCOM-FT). A preliminary version of this paper appeared in *Proceedings of the 12th ACM Symposium on Parallel Algorithms and Architectures (SPAA)*, pages 109–118, 2000.

1 Introduction

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 data 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 major bottleneck in these systems, because improving communication bandwidth and latency is often more expensive or more difficult than increasing processor speed and memory capacity.

Therefore we investigate data management strategies that aim to minimize the communication overhead caused by remote accesses. However, simply reducing the total communication load, i.e., the sum over all messages multiplied with the length of the routing paths traversed by the messages, 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 or buses, of the amount of data transmitted by the link or bus divided by the respective bandwidth. Known results for store-and-forward routing [9, 11, 14, 15] and wormhole routing [3, 4, 15] 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. In [8] it is shown in an experimental evaluation that the execution time of several applications depends heavily on the congestion produced by the data management strategy.

In this paper, we investigate the data management problem on hierarchical bus networks that use hierarchically, tree-like connected buses as a communication network. New communication technologies like SCI (Scalable Coherent Interface) (see, e.g., [6, 7]) make networks that can be modeled as hierarchical bus networks very attractive, because they allow their easy construction and guarantee reasonable communication performance.

The SCI standard offers many different possibilities for the network topology of an MPP or NOW. Usually large SCI networks are composed of several small, unidirectional SCI ringlets that are linked together via SCI switches. Thereby all processors in a given ringlet share the huge bandwidth of the SCI interconnection. Due to the SCI concept of request-response transactions a message from a node u to a node v is always followed by a response message from v to u. Hence, such a request-response transaction between two nodes on a unidirectional ringlet r can be viewed as a single packet that has to go all the way around r. In the case we only concentrate on the communication load in the network we can therefore neglect the ring structure and model each ringlet as a bus. In the same way tree-like connected ring-networks, as e.g. a ring of rings, can be modeled by a hierarchical bus network. Figure 1 shows an example of a tree-like connected ring network and Figure 2 shows the corresponding hierarchical bus network.

1.1 Hierarchical bus networks and the static data management problem

A hierarchical bus network consists of a set of processors $\mathcal{P}$ and a set of buses $\mathcal{B}$ and can be described by a weighted tree $T = (\mathcal{P} \cup \mathcal{B}, \mathcal{E}, b)$. The processors $\mathcal{P}$ are the leaves, the buses $\mathcal{B}$ are the inner nodes of the tree, and an edge $e \in \mathcal{E}$ describes switches connecting a processor and a bus or two buses. The

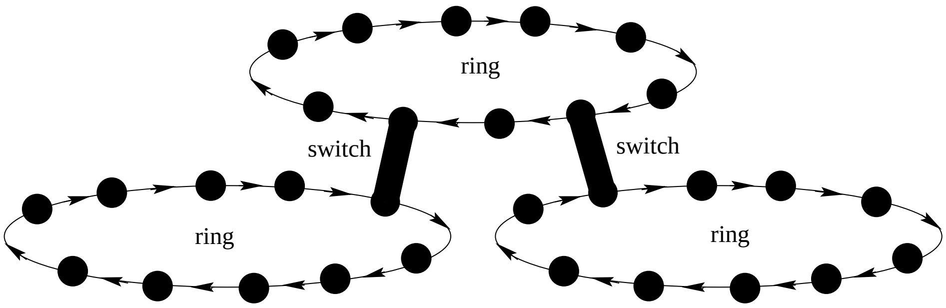

Figure 1: A hierarchical ring network

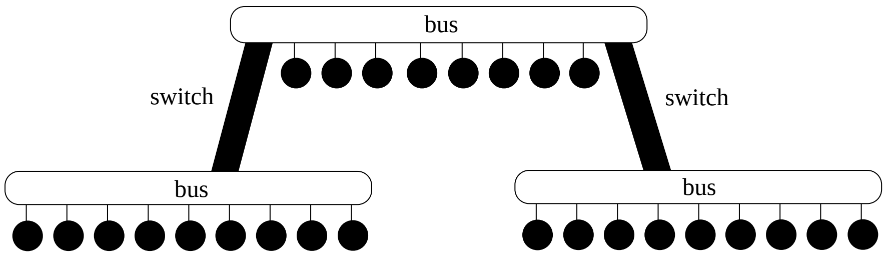

Figure 2: The corresponding bus network

function $b : \mathcal{E} \cup \mathcal{B} \longrightarrow \mathbb{N}$ describes the bandwidths of the switches and buses, respectively. We assume that the switches connecting processors and buses are the "slowest" part of the system, they have bandwidth one, all other bandwidths are at least one.

Our static data management problem is defined as follows (compare [10]). We are given a set X of shared data objects and the read and write frequencies $h_r : \mathcal{P} \times X \longrightarrow \mathbb{N}$ and $h_w : \mathcal{P} \times X \longrightarrow \mathbb{N}$ that describe the number of read and write accesses, respectively, from the processors to the objects. For each object $x \in X$, we have to determine a set of processors $\mathcal{P}_x \subset \mathcal{P}$ holding copies of x, and, for each pair $P \in \mathcal{P}$ and $x \in X$, a processor $c(P, x) \in \mathcal{P}_x$ holding a copy of x that serves P's requests to object x. In case of a read request, P reads the copy on $c(P, x)$, in case of a write request, P sends an update to $c(P, x)$ and $c(P, x)$ initiates a broadcast updating all copies of x. The copy of x on $c(P, x)$ is called the reference copy of x for P. The goal is to compute the above in such a way that the congestion is minimized. The congestion is defined as follows.

- A read request from processor P to object x increases the load by one on each edge on the (unique) path from P to $c(P, x)$.
- A write request from processor P to object x increases the load by one on each edge on the (unique) path from P to $c(P, x)$, and on each edge of the Steiner tree connecting $\mathcal{P}_x$.

The load on a bus $B \in \mathcal{B}$ is the sum over the load of all edges incident to B divided by two (division by two is reasonable, since each message passing B increases the load on two edges incident to B). The relative load of an edge e or a bus B is its load divided by its bandwidth $b(e)$ or $b(B)$, respectively. The congestion is the maximum over the relative loads of all edges and buses.

1.2 Hierarchical bus networks versus trees, and our new results

As described above, hierarchical bus networks are modeled as trees. Data management strategies for trees are described in [10]. There the nibble strategy is presented that can compute an optimal placement, w.r.t. the congestion, of the shared data objects in trees in linear time, given the read and write frequencies h_r and h_w. However, the nibble strategy assumes that inner nodes of the tree may be used to store copies of shared data objects.

Our results in this paper show that in case of hierarchical bus networks, i.e., if inner nodes (buses) are not capable of storing copies of shared data objects, the complexity of the problem increases drastically: Computing a placement that guarantees minimum congestion becomes NP-hard, even on trees with only five nodes. On the other hand, the main result of our paper presents an approximation algorithm that reaches minimal congestion up to a factor of 7. The algorithm needs sequential time $O(|X| \cdot |\mathcal{P} \cup \mathcal{B}| \cdot \text{height}(T) \cdot \log(\text{degree}(T)))$ and can be executed in a distributed fashion on the tree consuming time $O(|X| \cdot |\mathcal{P} \cup \mathcal{B}| \cdot \log(\text{degree}(T)) + \text{height}(T))$.

1.3 Related Work

Static and dynamic data management strategies for trees, for meshes of arbitrary dimension, and for Internet-like clustered networks are presented in [10]. All of these strategies aim to minimize the congestion, but they assume that all nodes of the network may be used to store copies of shared data objects.

A static strategy for trees, the nibble strategy, is presented. It is shown that the placement minimizes the load on any edge of the tree and therefore also the total load and the congestion, regardless of the bandwidths. Further, static strategies computing optimal or close-to-optimal placements for meshes of arbitrary dimension and Internet-like clustered networks are presented. The sequential running time of these algorithms is linear in the input size. Moreover, the algorithms can be efficiently calculated in a distributed fashion by the processors of the underlying network.

In the dynamic model, no knowledge about the access pattern is assumed. The dynamic strategies, are investigated in a competitive model. For example, it is shown that the competitive ratio is 3 for trees and $O(d \cdot \log n)$ for d-dimensional meshes with n nodes. Further, an $\Omega(\log n/d)$ lower bound is presented on the competitive ratio for on-line routing in meshes. This implies that the achieved upper bound on the competitive ratio is optimal for meshes of constant dimension.

Furthermore, in [12] these results are adapted to networks with non-uniform costs for accessing and migrating a shared data object, and in [13] the results are extended also to systems with limited memory capacities.

Earlier theoretical work on dynamic data management concentrates on minimizing the total communication load, i.e., the sum, over all links, of the data transmitted by the link rather than the maximum over the links. We believe that the congestion is more relevant for practice as minimizing the total communication load can lead to some very congested links. However, the theoretical results obtained for the total communication load are more general than the results for the congestion.

For example, Awerbuch et al. investigate data management in arbitrary networks with non-uniform costs for accessing and migrating a data object. In [1] they present a centralized algorithm that achieves optimum competitive ratio $O(\log n)$ and a distributed algorithm that achieves competitive ratio $O\left((\log n)^4\right)$ on an arbitrary network of size n. Both algorithms are deterministic. Furthermore, in [2] the distributed algorithm is adapted also to systems with limited memory capacities.

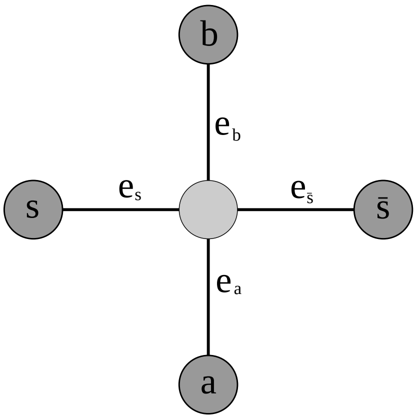

Figure 3: The labeling of the tree

2 The NP-hardness proof

To prove the NP-hardness of static data management, we first have to define the corresponding decision problem. In addition to the usual problem, we are given an integer k. Then the following problem is given: Exists a placement such that the congestion induced by this placement is not larger than k?

Usually the selection of routing paths belongs to the problem definition. But as we consider trees the routing paths are unique. We only consider non-redundant placement. Note, however, that non-redundant is not harder than redundant placement since every optimal placement is non-redundant if all requests are write requests. Thus, NP-completeness of non-redundant placement induces NP-completeness of redundant placement.

Theorem 2.1 *The static placement problem is NP-complete for a* 4*-ary tree of height* 1 *if the inner node is not allowed to store copies of shared data objects.*

Proof. We describe a reduction from PARTITION which is NP-complete [5]. The input of PARTITION are integers $k_1, \ldots, k_n$ with $\sum_{i=1}^{n} k_i = 2k$. 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 4-ary tree. See Figure 3 for the labeling of the tree used in the following. The bandwidth of the edges is one and the bandwidth of the inner node is sufficiently large such that the load on the edges is dominating. The shared objects in the placement problem are $x_1, \ldots, x_n$ and y. The access rates are defined as follows:

$$h_w(a, y) = 4k + 1, \;\; h_w(b, y) = 2k$$
$$\forall v \in \{a, b, s, \bar{s}\}, \; \forall i \in \{1, \ldots, n\} : \;\; h_w(v, x_i) = k_i$$

All other rates are 0. We prove that a placement of congestion at most $4k$ can be realized if and only if there exists a subset $S \subset \{1, \ldots, n\}$ with $\sum_{i \in S} k_i = k$.

Suppose that there is a subset S fulfilling the PARTITION constraint. Each object x_i is placed on node s if $i \in S$ and on node $\bar{s}$ otherwise. The object y is placed on node a. Counting the load on all edge yields a congestion of $4k$.

Now suppose that there is no subset fulfilling the PARTITION constraint. Assume for contradiction that there exists a placement with congestion $4k$ or less. The object y is placed on node a, because otherwise the edge leading to the node where y is placed would have congestion at least $4k+1$. The minimum possible communication load on edge e_a is $4k$. For each object x_i the edge e_a is charged by k_i if the object is not placed on a and by $3k_i$ if it is placed there. Hence, the minimum load induced on edge e_a due to communication for the x_i's is $2k$ and the additional load for object y yields a minimum load of $4k$. The same argument holds for edge e_b, as well. Therefore all objects x_i have to be placed on nodes s and $\bar{s}$. Let S denote the index set of objects placed on s. Without loss of generality we assume $\sum_{i \in S} k_i > k$. The load induced on edge e_s by this placement is

$$3 \sum_{i \in S} k_i + \sum_{i \notin S} k_i = 2k + 2 \sum_{i \in S} k_i > 4k$$

which is a contradiction to the assertion that a placement with congestion at most $4k$ exists. ■

3 The extended-nibble strategy

Assume we are given a hierarchical bus network $T = (\mathcal{P} \cup \mathcal{B}, \mathcal{E}, b)$, a set X of shared objects, and read and write frequencies h_r and $h_w : \mathcal{P} \times X \longrightarrow \mathbb{N}$, respectively. Our extended-nibble strategy consists of the following three steps.

- Step 1: the nibble strategy

  Compute an optimal placement of the shared data objects under the assumption that also inner nodes may hold copies. This is shown in [10] to be possible by a simple, linear time algorithm, the nibble strategy.

- Step 2: the deletion algorithm – removing rarely used copies

  For $x \in X$ let $\kappa_x := \sum_{P \in \mathcal{P}} h_w(P, x)$ denote the write contention induced by x. Modify the above placement so that all copies of x serve at least κ_x read or write requests. We will show that this is possible such that the congestion is at most doubled.

- Step 3: the mapping algorithm – move copies from inner nodes to leaves

  Replace each copy stored on an inner node by one or more copies on leaves. We show how to do this such that the placement we get produces a congestion that is at most a factor of 7 away from optimal. Computing this strategy needs sequential time $O(|X| \cdot |\mathcal{P} \cup \mathcal{B}| \cdot \text{height}(T) \cdot \log(\text{degree}(T)))$, and can be executed in a distributed way on T in time $O(|X| \cdot |\mathcal{P} \cup \mathcal{B}| \cdot \log(\text{degree}(T)) + \text{height}(T))$.

The remainder of this chapter describes the three steps in more detail. In the following chapter the analysis of the algorithm is given.

3.1 Step 1: the nibble strategy

In this section we present the nibble strategy that was introduced in [10]. Note that the nibble strategy places copies also on inner nodes. Therefore, we do not distinguish between processors and buses in the following, i.e., we are given an ordinary tree $T = (V, E)$ with $V = \mathcal{P} \cup \mathcal{B}$ and $E = \mathcal{E}$.

The 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 (processor or bus), 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

```
for l = 0 to height(T(x)) do
    for each node v on level l of T(x) do
        let c denote the copy of x on v
        if s(c) < κ_x then
            M(v) := M(v) \ {c}
            if v is not the root of T(x) then
                let u denote the parent node of v
            else
                let u denote the nearest node holding a copy of x
            s(u) := s(u) + s(v)
        end
    end
end
```

Figure 4: The deletion algorithm for object $x \in X$.

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

The most important result of the following theorem is that the nibble strategy minimizes the load on all edges simultaneously and hence also the congestion, regardless of the bandwidths of the edges.

Theorem 3.1 *The nibble strategy computes a placement that has the following properties.*

- *The nibble placement achieves minimum load on each edge.*
- *All nodes holding a copy of an object x form a connected subgraph $T(x)$.*
- *The load on an arbitrary edge e induced for serving requests to an object x is less or equal than the write contention $\kappa_x := \sum_{v \in V} h_w(v,x)$ of x.*
- *The load on all edges in the connected subgraph $T(x)$ is κ_x.*

3.2 Step 2: the deletion algorithm – removing rarely used copies

The algorithm that removes rarely used copies, called the deletion algorithm, is presented in the following. For a node v let $M(v)$ denote the set of copies placed on v by the nibble strategy. If a processor P issues a read or write request the reference copy $c(P,x)$ is the copy of x that is stored on the node closest to v. For a copy $c \in M(v)$ let $s(c)$ denote the number of read and write requests that are served by c.

The deletion algorithm works independently for each object. Fix an object x. It is assumed that the connected subgraph $T(x)$ induced by the nodes holding a copy of object x is rooted at an arbitrary node. Our algorithm works in height$(T(x))$ rounds. The root is defined to be on level height$(T(x))$ and all nodes that are children of nodes on level $i+1$ are defined to be on level i. In round l all copies of x on level l nodes of $T(x)$ are tested for deletion. A copy c on node v is deleted if the number of requests served by c is less than κ_x. In this case the requests previously served by c are served by the copy of x on the parent node of v. The only exception is the root of $T(x)$, since there exists no parent node. Therefore, if the copy of x on the root is deleted the requests previously served by that copy are served by the nearest undeleted copy of x. The details of the deletion algorithm for object x are described in Figure 4.

The deletion algorithm computes the modified nibble placement in which every copy of object x serves at least κ_x requests. If a copy c of x serves too many request, i.e., c serves more than $2\kappa_x$ requests, additional copies of x are created on v and the requests are split among these copies in such a way that each copy serves at least κ_x and at most $2\kappa_x$ requests. Thus we can make the following observation.

Observation 3.2 *The modified nibble placement has the following properties.*

- *A copy of variable x serves at least κ_x and at most $2\kappa_x$ requests.*
- *On each edge of the connected subgraph $T(x)$ the load due to messages for object x increases at most by κ_x.*
- *The placement achieves optimal load on every edge up to a factor of* 2*.*

3.3 Step 3: the mapping algorithm – moving copies from inner nodes to leaves

In the third and last step, the remaining copies on inner nodes are moved to the leaf nodes. In the following we assume that an arbitrary node is the root of T. The root is defined to be on level height(T) and all nodes that are children of nodes on level $i+1$ are defined to be on level i. In addition we assume that every edge of T is replaced by two directed edges in opposite directions. All edges directed to the root are called *upward edges* and the others are called *downward edges*. Suppose

for each edge $\vec{e}$ **do**
  $L_{\text{map}}(\vec{e}) := 0$
  $L_{\text{acc}}(\vec{e}) := 2 \cdot L_b(\vec{e})$
end
for $l = 0$ **to** height$(T) - 1$ **do**
  for each node v on level l **do**
    while $(L_{\text{map}}(\vec{e}_+) + \tau_{\max} \leq L_{\text{acc}}(\vec{e}_+))$ **do**
      choose $c \in M(v)$ arbitrarily

$$\left.\begin{array}{l} M(u) := M(u) \cup \{c\} \\ M(v) := M(v) \setminus \{c\} \\ L_{\text{map}}(\vec{e}_+) := L_{\text{map}}(\vec{e}_+) + s(c) + \kappa_{x(c)} \end{array}\right\} \begin{array}{c}\text{movement} \\ \text{along} \\ \vec{e}_+ = (v, u)\end{array}$$

    end

$$\left.\begin{array}{l} \delta := L_{\text{acc}}(\vec{e}_+) - L_{\text{map}}(\vec{e}_+) \\ L_{\text{acc}}(\vec{e}_+) := L_{\text{acc}}(\vec{e}_+) - \delta \\ L_{\text{acc}}(\vec{e}_-) := L_{\text{acc}}(\vec{e}_-) - \delta \end{array}\right\} \text{adjustment}$$

  end
end

Figure 5: The upwards phase of the mapping algorithm.

that after step 2 a node v holds a copy c and in step 3 c is mapped to another node u. Then we say that the copy c is moved along the path from v to u. Note that all requests accessing c have to be *forwarded* from v to u and thus increase the load on the respective path.

The mapping algorithm uses the following variables and constants. The *mapping load* $L_{\text{map}}(\vec{e})$ of an edge $\vec{e}$ is the additional load on $\vec{e}$ due to forwarding requests. Let $x(c)$ denote the shared object of which c is a copy. Then the variable $L_{\text{map}}(\vec{e})$ is increased by $s(c) + \kappa_{x(c)}$ whenever a copy c is moved along $\vec{e}$. This increment is smaller than $\tau_{\max} := \max_{c'}\{s(c') + \kappa_{x(c')}\}$. The *acceptable load* $L_{\text{acc}}(\vec{e})$ of an edge $\vec{e}$ is approximately the load on $\vec{e}$ due to forwarding requests the algorithm will accept. In order to define $L_{\text{acc}}(\vec{e})$ initially, we first define the *basic load* $L_b(\vec{e})$ of $\vec{e}$. Suppose a request issued by a leaf node v and served by a copy on a node u in the modified nibble placement. This request is *basic* for the directed edge $\vec{e}$ if $\vec{e}$ lies on the directed path from u to v. Let $L_b(\vec{e})$ denote the number of basic requests for $\vec{e}$. Then initially $L_{\text{acc}}(\vec{e}) := 2L_b(\vec{e})$.

The mapping algorithm consists of two phases: the upwards and the downwards phase. In the upwards phase consisting of height(T) rounds, copies are moved along upward edges in the following way. In round l a node v on level l of T moves copies along the edge $\vec{e}_+$ leading to his parent node u as long as the resulting mapping load on $\vec{e}_+$ does not exceed the acceptable load. Thus, the node v tries to move as much copies as possible out of his subtree. If this would not be done these copies would have to be mapped to the leaf nodes of the subtree of v in the downwards phase. After the movement the acceptable load on the edges $\vec{e}_+$ and $\vec{e}_-$ is possibly reduced ($\vec{e}_-$ denotes the downwards edge incident to v and u). This ensures that in the downwards phase the algorithm does not move too many copies along $\vec{e}_-$ in the subtree of v. Note that $L_{\text{acc}}(\vec{e}_-)$ may become negative by this adjustment. The details of the upwards phase are described in Figure 5.

In the downwards phase consisting of height(T) rounds all copies are moved along downwards edges. In round l a node v on level l of T moves all his copies in the following way. Suppose v holds

for $l = \text{height}(T) - 1$ **to** 1 **do**
  for each node v on level l **do**
    for each $c \in M(v)$ **do**

$$\left.\begin{array}{l}\text{choose a child edge } \vec{e} \text{ of } v \text{ with}\\ L_{\text{map}}(\vec{e}) + s(c) + \kappa_{x(c)} \leq L_{\text{acc}}(\vec{e}) + \tau_{\max}\end{array}\right\} \begin{array}{l}\text{such a free edge } \vec{e}\\ \text{always exists}\end{array}$$

$$\left.\begin{array}{l}M(u) := M(u) \cup \{c\}\\ M(v) := M(v) \setminus \{c\}\\ L_{\text{map}}(\vec{e}) := L_{\text{map}}(\vec{e}) + s(c) + \kappa_{x(c)}\end{array}\right\} \begin{array}{c}\text{movement along}\\ \vec{e} = (v,u)\end{array}$$

    end
  end
end

Figure 6: The downwards phase of the mapping algorithm.

a copy c that serves $s(c)$ requests. Then the algorithm searches a *free* downward child edge $\vec{e}$ of v, i.e., an edge $\vec{e}$ with $L_{\text{map}}(\vec{e}) + s(c) + \kappa_{x(c)} \leq L_{\text{acc}}(\vec{e}) + \tau_{\max}$, where $x(c)$ denotes the shared data object of which c is a copy. The copy is moved along this edge $\vec{e}$ and the mapping load of $\vec{e}$ is increased accordingly. The details of the downwards phase are described in Figure 6. If a free edge can always be found we can make the following observation.

Observation 3.3 *Suppose a node v has moved all its copies downwards and $\vec{e}$ is a downward child edge of v. Then either*

$$L_{map}(\vec{e}) \leq L_{acc}(\vec{e}) + \tau_{\max}$$

or

$$L_{map}(\vec{e}) = 0 \quad \textit{and} \quad L_{acc}(\vec{e}) < -\tau_{\max} \ .$$

4 The Analysis

First, the following lemma ensures that the mapping algorithm can be executed in the described way.

Lemma 4.1 *In the downwards phase of the mapping algorithm a free downward child edge can always be found.*

Proof. To prove the lemma we need the following invariant.

Invariant 4.2 *Let v denote an arbitrary internal node of the tree network and let E_{out} and E_{in} denote the set of outgoing and incoming edges of v, respectively. Then the following inequality holds at any time step during the execution of the mapping algorithm,*

$$\sum_{\vec{e} \in E_{out}} L_{acc}(\vec{e}) - \sum_{\vec{e} \in E_{out}} L_{map}(\vec{e}) \geq \sum_{\vec{e} \in E_{in}} L_{acc}(\vec{e}) - \sum_{\vec{e} \in E_{in}} L_{map}(\vec{e}) + 2 \sum_{c \in M(v)} s(c) \ .$$

Proof. First, we will prove that the invariant holds at the beginning of the mapping algorithm. Since all mapping loads are initially 0 the invariant simplifies to

$$2\sum_{\vec{e}\in E_{\text{out}}} L_{\text{b}}(\vec{e}) \geq 2\sum_{\vec{e}\in E_{\text{in}}} L_{\text{b}}(\vec{e}) + 2\sum_{c\in M(v)} s(c) \enspace .$$

Recall that initially $L_{\text{acc}}(\vec{e}) = 2 \cdot L_b(\vec{e})$. The inequality holds, since a request issued by a leaf u that is a basic request of an incoming edge is also a basic request of the outgoing edge leading to u. Thus, each request contributing to $\sum_{\vec{e}\in E_{\text{in}}} L_{\text{b}}(\vec{e})$ contributes to $\sum_{\vec{e}\in E_{\text{out}}} L_{\text{b}}(\vec{e})$, as well. Furthermore, each request served by a copy placed on v at the beginning of the mapping algorithm is basic for some outgoing edge. This means that each request contributing to $\sum_{c\in M(v)} s(c)$ also contributes to $\sum_{\vec{e}\in E_{\text{out}}} L_{\text{b}}(\vec{e})$.

Now, we show that a movement of a copy or an adjustment of loads does not cause the invariant to become invalid. This shows that it holds throughout the mapping algorithm.

Consider that a copy c is moved along an outgoing edge $\vec{e}$ away from node v. Then $L_{\text{map}}(\vec{e})$ increases by $s(c) + \kappa_{x(c)}$. Thus, the left side of the inequality decreases by this value, whereas the right side decreases by $2s(c)$ because c is removed from the set $M(v)$. From observation 3.2 follows that $s(c) + \kappa_{x(c)} \leq 2s(c)$ holds, and thus the invariant holds after the movement. An analogical argument holds if a copy is moved to v along an incoming edge.

During an adjustment the acceptable load on an incoming and on an outcoming edge are decreased by δ. Hence, both sides of the inequality are decreased by δ and the invariant holds after the adjustment. Altogether this yields the lemma. ■

Consider an arbitrary node v of the tree and a copy c^* that is currently placed on v. We show that there exists a free child edge of v, i.e., a child edge $\vec{e}$ with $L_{\text{map}}(\vec{e}) + s(c^*) + \kappa_{x(c^*)} \leq L_{\text{acc}}(\vec{e}) + \tau_{\max}$.

After the upwards phase of the mapping algorithm the mapping load on upward edges matches the respective acceptable load due to the adjustment. Therefore, these edges can be removed from the sums in the invariant and thus it simplifies to

$$\sum_{\vec{e}\in E_{\text{child}}} L_{\text{acc}}(\vec{e}) - \sum_{\vec{e}\in E_{\text{child}}} L_{\text{map}}(\vec{e}) \geq L_{\text{acc}}(\vec{e}_-) - L_{\text{map}}(\vec{e}_-) + 2\cdot\sum_{c\in M(v)} s(c) \enspace , \tag{1}$$

where E_{child} denotes the set of downward edges leading to children of v, and $\vec{e}_-$ denotes the edge leading from the father of v to v. Now, we distinguish two cases according to observation 3.3.

1. Suppose $L_{\text{map}}(\vec{e}_-) \leq L_{\text{acc}}(\vec{e}_-) + \tau_{\max}$.
   Then $L_{\text{acc}}(\vec{e}_-) - L_{\text{map}}(\vec{e}_-) \geq -\tau_{\max}$ and together with inequality (1) this yields

$$\begin{aligned}\sum_{\vec{e}\in E_{\text{child}}} L_{\text{acc}}(\vec{e}) - \sum_{\vec{e}\in E_{\text{child}}} L_{\text{map}}(\vec{e}) &\geq 2\sum_{c\in M(v)} s(c) - \tau_{\max} \\ &\geq 2s(c^*) - \tau_{\max} \enspace .\end{aligned}$$

   Now assume for contradiction that there exists no child edge $\vec{e}$ of v with $L_{\text{map}}(\vec{e}) + s(c^*) + \kappa_{x(c^*)} \leq L_{\text{acc}}(\vec{e}) + \tau_{\max}$. Then for every child edge $L_{\text{acc}}(\vec{e}) - L_{\text{map}}(\vec{e}) < s(c^*) + \kappa_{x(c^*)} - \tau_{\max}$. The sum over all edges yields

$$\begin{aligned}\sum_{\vec{e}\in E_{\text{child}}} L_{\text{acc}}(\vec{e}) - \sum_{\vec{e}\in E_{\text{child}}} L_{\text{map}}(\vec{e}) &< |E_{\text{child}}| \cdot (s(c^*) + \kappa_{x(c^*)} - \tau_{\max}) \\ &< s(c^*) + \kappa_{x(c^*)} - \tau_{\max} \\ &< 2s(c^*) - \tau_{\max} \enspace ,\end{aligned}$$

   since $\tau_{\max} \geq s(c^*) + \kappa_{x(c^*)}$. This is a contradiction.

2. Suppose $L_{\text{map}}(\vec{e}_-) = 0$ and $L_{\text{acc}}(\vec{e}_-) < -\tau_{\text{max}}$.

   Assume for contradiction that the node v has a copy c^* for which the mapping algorithm tries to find a free child edge. This copy has not been moved to v during the downwards phase because then $L_{\text{map}}(\vec{e}_-)$ would not equal zero. Hence, c^* was already placed on v at the end of the upwards phase. But then v would have moved this copy upwards because the mapping load $L_{\text{map}}(\vec{e}_+)$ on the edge leading to the father of v was small enough to allow this movement.

   This can be seen as follows. $L_{\text{acc}}(\vec{e}_-)$ has been reduced by more than τ_{max} during the adjustment in the upwards phase. Otherwise $L_{\text{acc}}(\vec{e}_-) < -\tau_{\text{max}}$ would not be possible. Therefore $\delta = L_{\text{acc}}(\vec{e}_+) - L_{\text{map}}(\vec{e}_+)$ has been at least τ_{max}. But in this case the condition for moving copies upwards is fulfilled and thus, c^* would have been moved upwards.

   Altogether, it is not possible that the node v has a copy c^* for which the mapping algorithm tries to find a free child edge if $L_{\text{map}}(\vec{e}_-) = 0$ and $L_{\text{acc}}(\vec{e}_-) < -\tau_{\text{max}}$.

Thus, it is shown that the algorithm works correctly. ■

Finally, the following theorem shows that the extended-nibble strategy achieves optimum congestion up to a factor of 7. Further, it gives bounds on the runtime of the strategy.

Theorem 4.3 *The extended-nibble strategy achieves a congestion $C \leq 7 \cdot C_{opt}$ on the tree T, where C_{opt} denotes the optimum congestion. Further, the placement of the strategy can be calculated in sequential time $O(|X| \cdot |\mathcal{P} \cup \mathcal{B}| \cdot height(T) \cdot \log(degree(T)))$. If it is computed in a distributed fashion on the tree T the strategy needs time $O(|X| \cdot |\mathcal{P} \cup \mathcal{B}| \cdot \log(degree(T)) + height(T))$.*

Proof. First, we prove the bounds on the runtime of the extended-nibble strategy which is simply the sum of the runtime of the three steps.

- Nibble strategy

  The nibble placement can be calculated in time $O(|X| \cdot |\mathcal{P} \cup \mathcal{B}|)$. This result can be found in [10].

- Deletion algorithm

  The modified placement can be calculated in time $O(|X| \cdot |\mathcal{P} \cup \mathcal{B}|)$, too, because for each object x each node in the connected component of copies of x is visited only once during the deletion algorithm.

- Mapping algorithm

  An upward movement of a copy costs time $O(1)$. A downward movement at a node v can be performed in time $O(log(\text{degree}(v)))$ if a heap structure is used to find a free edge. A single copy is moved at most $O(\text{height}(T))$ times. Altogether, all movements of a single copy cost time $O(\text{height}(T) \cdot \log(\text{degree}(T)))$.

  The remaining question is how many copies of a single object x have to be mapped. Less than κ_x requests reach a node via the same edge. Therefore at most $\text{degree}(v)$ copies of the same object are created on a node v. Recall that each copy serves at least κ_x requests. Altogether, this yields a number of at most $|\mathcal{P} \cup \mathcal{B})|$ copies of an object. Hence, all copies can be mapped in time $O(|X| \cdot |\mathcal{P} \cup \mathcal{B}| \cdot \text{height}(T) \cdot \log(\text{degree}(T)))$.

If the extended-nibble strategy is computed in a distributed fashion on the tree T, the computation can be pipelined for several copies, which gives the time bound $O(|X| \cdot |\mathcal{P} \cup \mathcal{B}| \cdot \log(\text{degree}(T)) + \text{height}(T))$. Note, that a single node needs only time $O(|X| \cdot |\mathcal{P} \cup \mathcal{B}| \cdot \log(\text{degree}(T)))$ for mapping all copies, because there are only $O(|X| \cdot |\mathcal{P} \cup \mathcal{B}|)$ copies and each copy takes time $O(\log(\text{degree}(T)))$.

In order to prove the bound on the congestion we, first, relate the load of an edge or a bus to the load on the corresponding edge or node, respectively, needed in the nibble placement. Finally we relate the congestion in the nibble placement to the optimal congestion. The following lemma relates the acceptable load to the load in the nibble placement.

Lemma 4.4 *Let $\vec{e}_+$ denote an arbitrary directed edge of T, and let $\vec{e}_-$ denote the edge in the opposite direction of $\vec{e}$. Furthermore, let e denote the corresponding undirected edge. Then*

$$L_{acc}(\vec{e}_+) + L_{acc}(\vec{e}_-) \leq 2L_{nib}(e) \ \ ,$$

where $L_{nib}(e)$ denotes the load on e in the nibble placement.

Proof. First, we show for each object x that the number of basic requests for edges $\vec{e}_+$ and $\vec{e}_-$ that are directed to copies of x is smaller than the load on e in the nibble placement, due to messages for object x. Consider a basic request for the edges $\vec{e}_+$ or $\vec{e}_-$. A corresponding request message has to be sent along e in the modified nibble placement, because the requesting node and the copy that serves the request are only connected via e. Recall that the modified nibble placement is the placement at the end of the deletion algorithm. Thus $L_{\text{b}}(\vec{e}_+) + L_{\text{b}}(\vec{e}_-)$ is less or equal than the load on edge e due to the modified nibble placement. Now, we distinguish two cases in order to show $L_{\text{b}}(\vec{e}_+) + L_{\text{b}}(\vec{e}_-) \leq L_{\text{nib}}(e)$.

1. Suppose e does not belong to the connected component $T(x)$.
   In this case the nibble placement and the modified nibble placement produce the same load on e.
2. Suppose e belongs to the connected component $T(x)$.
   In this case the nibble placement has load κ_x on e. Only requests that were reassigned during the deletion procedure contribute to the basic load on $\vec{e}_+$ and $\vec{e}_-$. These are less than κ_x.

Altogether for the initial value of the acceptable loads on $\vec{e}_+$ and $\vec{e}_-$ holds

$$\begin{aligned} L_{\text{acc}}(\vec{e}_+) + L_{\text{acc}}(\vec{e}_-) &= 2L_{\text{b}}(\vec{e}_+) + 2L_{\text{b}}(\vec{e}_-) \\ &\leq 2L_{\text{nib}}(e) \ \ . \end{aligned}$$

The lemma follows from this inequality, since the acceptable loads are only decreased during the mapping algorithm. ■

Lemma 4.5 *For every edge e in the tree T, $L(e) \leq 4L_{nib}(e) + \tau_{max}$.*

Proof. The load on an edge e consists of the mapping load and the load of the modified nibble placement. Observation 3.2 relates the load of the modified nibble placement to $L_{\text{nib}}(e)$. Observation 3.3 together with the obvious property $L_{\text{map}}(\vec{e}) \leq L_{\text{acc}}(\vec{e})$ for upward edges yields

$$L_{\text{map}}(\vec{e}_+) + L_{\text{map}}(\vec{e}_-) \leq L_{\text{acc}}(\vec{e}_+) + L_{\text{acc}}(\vec{e}_-) + \tau_{\text{max}} \ \ ,$$

where $\vec{e}_+$ and $\vec{e}_-$ denote the directed edges corresponding to e. Hence, the lemma follows from Lemma 4.4 that relates the acceptable loads on a pair of edges to the respective load in the nibble placement.
■

Lemma 4.6 *For every bus v in the tree T holds $L(v) \leq 4L_{nib}(v) + \tau_{max}$, where $L_{nib}(v) := \sum_{e \text{ adj. } v} L_{nib}(e)/2$ denotes the sum of the loads in the nibble placement, over all edges adjacent to v, divided by two.*

Proof. As in the proof of Lemma 4.1, it holds that after the upwards phase of the mapping algorithm the mapping load on upward edges matches the respective acceptable load due to the adjustment. Therefore these edges can be removed from the sums in the Invariant 4.2. This gives

$$\sum_{\vec{e}\in E_{\text{child}}} L_{\text{acc}}(\vec{e}) - \sum_{\vec{e}\in E_{\text{child}}} L_{\text{map}}(\vec{e}) \geq L_{\text{acc}}(\vec{e}_-) - L_{\text{map}}(\vec{e}_-) + 2\sum_{c\in M(v)} s(c) \ . \tag{2}$$

After the downwards phase of the algorithm all copies have been mapped to leaf node and therefore (2) simplifies to

$$\sum_{\vec{e}\in E_{\text{child}}} L_{\text{acc}}(\vec{e}) - \sum_{\vec{e}\in E_{\text{child}}} L_{\text{map}}(\vec{e}) \geq L_{\text{acc}}(\vec{e}_-) - L_{\text{map}}(\vec{e}_-) \ . \tag{3}$$

Now, we distinguish two cases according to Observation 3.3. We only consider the first case of the observation. The other case is obvious, since in this case the node v has moved no copies downwards and thus the mapping load on his adjacent edges equals the corresponding acceptable load (compare case 2 in the proof of Lemma 4.1).

Suppose $L_{\text{map}}(\vec{e}_-) \leq L_{\text{acc}}(\vec{e}_-) + \tau_{\text{max}}$. Inequality (3) together with the property $L_{\text{map}}(\vec{e}) \leq L_{\text{acc}}(\vec{e})$ for upward edges yields

$$\sum_{\vec{e}\in E_{\text{child}}} L_{\text{map}}(\vec{e}) + \sum_{\vec{e}\in E_{\text{upw}}} L_{\text{map}}(\vec{e}) \leq \sum_{\vec{e}\in E_{\text{child}}} L_{\text{acc}}(\vec{e}) + \sum_{\vec{e}\in E_{\text{upw}}} L_{\text{acc}}(\vec{e}) + L_{\text{map}}(\vec{e}_-) - L_{\text{acc}}(\vec{e}_-) \ ,$$

where E_{upw} denotes the set of upward edges incident to v. The only edge incident to v missing in the sums on the left and the right side, is the edge $\vec{e}_-$ leading from the father of v to v. Hence, adding $L_{\text{map}}(\vec{e}_-)$ on both sides and $L_{\text{acc}}(\vec{e}_-) - L_{\text{acc}}(\vec{e}_-)$ on the left side yields

$$\frac{1}{2}\left(\sum_{\vec{e}\in E_{\text{out}}} L_{\text{map}}(\vec{e}) + \sum_{\vec{e}\in E_{\text{in}}} L_{\text{map}}(\vec{e})\right) \leq \frac{1}{2}\left(\sum_{\vec{e}\in E_{\text{out}}} L_{\text{acc}}(\vec{e}) + \sum_{\vec{e}\in E_{\text{in}}} L_{\text{acc}}(\vec{e})\right) + \tau_{\text{max}} \ ,$$

since $L_{\text{map}}(\vec{e}_-) - L_{\text{acc}}(\vec{e}_-) \leq \tau_{\text{max}}$. The left side is exactly the mapping load of bus v. The right side is less than $2L_{\text{nib}}(v) + \tau_{\text{max}}$, according to Lemma 4.4. Altogether this yields $L(v) \leq 4L_{\text{nib}}(v) + \tau_{\text{max}}$. ■

It is obvious that the optimal load for trees that are allowed to hold copies on inner nodes, i.e., the load in the nibble placement, is a lower bound for the optimal load in the hierarchical bus model.

Now we have to relate τ_{max} to the optimal congestion on the tree where the inner nodes correspond to buses. This is done as follows. Let $\hat{x}$ denote an object with maximum write contention κ_{max}. Additionally, assume that in the nibble placement copies of x are on inner nodes. Note that we do not have to consider other objects, because the extended-nibble strategy does not change their placement and thus does not produce any additional load. Let $h_{\hat{x}}$ denote the total number of requests issued to object $\hat{x}$. The inequality $\tau_{\text{max}} \leq 3\kappa_{\text{max}}$ follows from observation 3.2 and $\tau_{\text{max}} \leq h_{\hat{x}}$ is obvious. Now we prove that either $C_{\text{opt}} \geq \kappa_{\text{max}}$ or $C_{\text{opt}} \geq \frac{h_{\hat{x}}}{2}$ holds. We distinguish the following cases.

1. Suppose that the optimal algorithm places more than one copy of $\hat{x}$ on arbitrary leaf nodes.

   The edges leading to these leaf nodes have at least load κ_{max}, because a write has to update all copies. Since these edges have the lowest bandwidth among all edges this yields $C_{\text{opt}} \geq \kappa_{\text{max}}$.

2. Suppose that the optimal algorithm places only one copy of $\hat{x}$ on an arbitrary leaf node l.

   Assume this node issues less than half of the requests directed to the object. Then more than half of the requests traverse the edge e_l leading to l and thus the load is at least $\frac{h_{\hat{x}}}{2}$.

   Now, assume that more than half of the requests to x are issued by the leaf node l. Then the nibble strategy places a copy on l, because the load on e_l would not be minimized, otherwise. Hence, the nibble strategy produces load $\kappa_{\max}$ on e_l, since it places another copy of $\hat{x}$ in the network by definition of $\hat{x}$. Recall that the nibble strategy produces optimum congestion.

Altogether this yields $\tau_{\max} \leq 3C_{\text{opt}}$ and hence $C \leq 7C_{\text{opt}}$. ■

References

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

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

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

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

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

[6] D. B. Gustavson and Q. Li. The scalable coherent interface (SCI). *IEEE Communications Magazine*, 34(5):52–63, 1996.

[7] H. Hellwagner and A. Reinefeld, editors. *SCI – Scalable Coherent Interface*, volume 1734 of *Lecture notes in computer science*. Springer, 1999.

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

[9] 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.

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

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

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

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

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

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