The Dissertation Committee for Mitul Tiwari
certifies that this is the approved version of the following dissertation:

**Algorithms for Distributed Caching and Aggregation**

Committee:

---

Greg Plaxton, Supervisor

---

Mike Dahlin

---

Rajmohan Rajaraman

---

Vijaya Ramachandran

---

Harrick Vin

# Algorithms for Distributed Caching and Aggregation

**by**

**Mitul Tiwari, B.Tech., M.S.**

**DISSERTATION**

Presented to the Faculty of the Graduate School of

The University of Texas at Austin

in Partial Fulfillment

of the Requirements

for the Degree of

**DOCTOR OF PHILOSOPHY**

THE UNIVERSITY OF TEXAS AT AUSTIN

December 2007

*To my parents, Saroj and Dinesh Chandra Tiwari*

# Acknowledgments

First and foremost, I am indebted to my advisor Greg Plaxton for his constant support, encouragement, and guidance. He has shown a lot of patience with me, and has been a very amiable and approachable advisor. He has been an incredible mentor, and always available for discussions. I remember many of our long discussions, including one lasting for 9 hours. His brilliance, perseverance, and pursuit for excellence will continue to inspire me for years.

I would like to thank Harrick Vin for insightful discussions, for serving on my dissertation committee, and for writing a reference letter. His suggestions during our discussions have been quite valuable to improve the applicability of our abstract models and algorithms.

I would like to thank Mike Dahlin, Rajmohan Rajaraman, and Vijaya Ramachandran for serving on my dissertation committee. Mike has been a sounding board for many ideas, and his suggestions have influenced our work on aggregation. I am also thankful to Mike and Vijaya for writing me reference letters during my job search.

I would like to thank Steve Li, Yu Sun, Arun Venkataramani, and Praveen Yalagandula for directly contributing to this dissertation as co-authors of various papers. I am thankful to Yi Lin for helping me to implement our aggregation algorithms.

I would like to thank Lorenzo Alvisi for many of our corridor discussions, and for miscellaneous advice on various topics including public presentation.

I am thankful to Gloria Ramirez and Katherine Utz for making sure that all the paperwork is done properly, and for protecting me from the bureaucracy of a large university.

This research has been funded by National Science Foundation, Texas Advanced Technology Program, and the University of Texas at Austin, and I am thankful to these agencies for providing financial support.

I would like to thank many of my friends in LASR, theory, and data mining groups (and others I am sure I have forgotten) for making my stay in the department enjoyable. This list of friends includes Amit Anand Aiyer, Nalini Belaramani, Paolo Bientinesi, Rezaul Alam Chowdhury, Allen Clement, Ned Dimitrov, Prateek Jain, Navendu Jain, Sugat Jain, Ravi Kokku, Rama Kotla, Chinmayi Krishnappa, Harry Li, Ajay Mahimkar, Jean-Philippe Martin, Raghu Meka, Joseph Modayil, Jayaram Mudigonda, Jeff Napper, Amol Nayate, Anindya Patthak, Amit Prakash, Anup Rao, Taylor Riche, Eric Rozner, Vinay Siddhavanahalli, Upendra Shevade, Suvrit Sra, Edmund Wong, and Jiandan Zheng.

I would like to thank my gym partners Prateek, Suvrit, and Vinay, who have been partners in many crimes both inside and outside the gym, for helping me to stay fit and healthy. Suvrit has been very supportive and encouraging throughout the ups and downs of my graduate studies. I enjoyed many biking trips around Austin, and would like to thank my partners, Aman Garg, Sugat Jain, Vidya Jain, Smitha Shankara, and Praveen Yalagandula.

I am thankful to all of my apartment mates — Subramanyam Mallela, Priyank Misra, Anindya Patthak, Rahul Prashar, Siddhartha Rai, Upendra Shevade, and Vamsi Vutukuru — for not making my life dull outside the department.

Lastly and most importantly, I would like to thank my parents for their constant love, encouragement, and confidence in me and my decisions. My brothers, Manish and Mayank, and sisters-in-law, Deepali and Rashmi, never failed to encourage me and uplift my spirits. I cannot imagine completing this dissertation without the support of my family.

# Algorithms for Distributed Caching and Aggregation

Publication No. ____________________

Mitul Tiwari, Ph.D.
The University of Texas at Austin, 2007

Supervisor: Greg Plaxton

In recent years, there has been an explosion in the amount of distributed data due to the ever decreasing cost of both storage and bandwidth. There is a growing need for automatic distributed data management techniques. The three main areas in dealing with distributed data that we address in this dissertation are (1) cooperative caching, (2) compression caching, and (3) aggregation.

First, we address cooperative caching, in which caches cooperate to locate and cache data objects. The benefits of cooperative caching have been demonstrated by various studies. We address a hierarchical generalization of cooperative caching in which caches are arranged as leaf nodes in a hierarchical tree network, and we call this variant Hierarchical Cooperative Caching. We present a deterministic hierarchical generalization of LRU that is constant-competitive when the capacity blowup is linear in $d$, the depth of the cache hierarchy. Furthermore, we show that any randomized hierarchical cooperative caching algorithm with capacity blowup $b$ has competitive ratio $\Omega(\log \frac{d}{b})$ against an oblivious adversary. Thus we establish that there is no resource competitive algorithm for the hierarchical cooperative caching problem.

Second, we address a class of compression caching problems in which a file can be cached in multiple formats with varying sizes and encode/decode costs. In this work, we address three problems in this class of compression caching. The first problem assumes that the encode cost and decode cost associated with any format of a file are equal. For this problem we present a resource competitive online algorithm. To explore the existence of resource competitive online algorithms for compression caching with arbitrary encode costs and decode costs, we address two other natural problems in the aforementioned class, and for each of these problems, we show that there exists a non-constant lower bound on the competitive ratio of any online algorithm, even if the algorithm is given an arbitrary factor capacity blowup. Thus, we establish that there is no resource competitive algorithm for compression caching in its full generality.

Third, we address the problem of aggregation over trees with the goal of adapting aggregation aggressiveness. Consider a distributed network with nodes arranged in a tree, and each node having a local value. We consider the problem of aggregating values (e.g., summing values) from all nodes to the requesting nodes in the presence of writes. The goal is to minimize the total number of messages exchanged. The key challenges are to define a notion of "acceptable" aggregate values, and to design algorithms with good performance that are guaranteed to produce such values. We formalize the acceptability of aggregate values in terms of certain consistency guarantees. We propose a lease-based aggregation mechanism, and evaluate algorithms based on this mechanism in terms of consistency and performance. With regard to consistency, we adapt the definitions of strict and causal consistency to apply to the aggregation problem. We show that any lease-based aggregation algorithm provides strict consistency in sequential executions, and causal consistency in concurrent executions. With regard to performance, we propose an online lease-based aggregation algorithm, and show that, for sequential executions, the algorithm is constant competitive against any offline algorithm that provides strict consistency. Our online lease-based aggregation algorithm is presented in the form of a fully distributed protocol, and the aforementioned consistency and performance results are formally established with respect to this protocol. We also present experimental results to show that the algorithm performs well under various workloads.

# Table of Contents

# List of Tables

# List of Figures

# Chapter 1

# Introduction

In recent years, there has been an explosion in the amount of distributed data due to ever decreasing cost of both storage and bandwidth. There is a growing need for automatic distributed data management techniques. Two fundamental issues in dealing with the distributed data are data dissemination and data collection, which are the focus of this dissertation.

Data distribution is a key issue in distributing data from a small number of sources. An example is the world wide web, where a single site/page is accessed by many users. Frequently, sites become hot spots due to increased interest. Many techniques for data distribution have been proposed in the literature, such as Harvest [14], Squid [47], content delivery networks [1, 23] and cooperative caching [19]. We see caching as an important component of data dissemination. In the internet environment, caching objects closer to the user reduces client-side latency, network congestion, and server load. We address two problems related to caching: (1) cooperative caching, and (2) compression caching.

Data collection is another key issue in aggregating data from distributed sources to answer various kind of queries. We see aggregation as a basic abstraction for querying distributed data sources, and is useful in many large-scale distributed applications such as system management [25, 41], network monitoring [29], and sensor networks [32]. The main problem that we address in this domain is called aggregation over trees.

Next we describe some of the unifying themes of this dissertation

## 1.1 Unifying Themes

The three main unifying themes of this dissertation are as follows. The first theme is the pursuit of self-tuning resource allocation algorithms. The internet is evolving rapidly, and each new generation of applications reveals a different set of performance bottlenecks. Therefore, we need algorithms that are fundamentally self-tuning in order to operate efficiently under a wide range of conditions. Our goal is to design self-tuning online algorithms and analyze these algorithms in the framework of the competitive analysis, pioneered by Sleator and Tarjan [39]. In this framework, we compare the cost of an online algorithm to that of an optimal offline algorithm. An online algorithm executes each request without any knowledge of future requests. On the other hand, an offline algorithm has knowledge of all the requests in advance. An online algorithm is $c$-competitive if, for any request sequence $\sigma$, the cost incurred by the online algorithm in executing $\sigma$ is at most $c$ times that incurred by an optimal offline algorithm [13]. Often the online algorithm is given extra resources to compensate for its lack of knowledge of future requests. We refer to an online algorithm that is constant competitive with a constant factor advantage in resources as a resource competitive algorithm. A good competitive ratio ensures that the online algorithm provides good performance guarantees even in a worst case scenario.

The second theme is the design of fully distributed protocols for key primitives used in resource allocation schemes. By spreading the resource allocation overhead over all the nodes of the network, a fully distributed protocol has the potential to improve performance, by exploiting parallelism, and to improve fault-tolerance, by ensuring that no single node failure has a drastic impact on the system. It is often quite challenging — or even impossible (see, e.g., [22]) — to design and verify distributed protocols. Still, there are key primitives in resource allocation (e.g., aggregation) for which it is important to address this challenge.

The third theme is our emphasis on hierarchical notions. For example, we address questions concerning caching and aggregation in hierarchical networks. Our motivation for emphasizing such hierarchical notions is that many large-scale distributed systems tend to be hierarchical in nature, both for scalability and to manage complexity. We expect hierarchical notions to be used widely in the design of scalable distributed infrastructure.

Next we describe the three main components of this dissertation. For each of these components, relevant related work is discussed in the later chapters.

## 1.2 Cooperative Caching

In cooperative caching, caches cooperate in order to locate and cache data objects. The benefits of cooperative caching have been demonstrated by various studies [5, 14, 19]. We address a hierarchical generalization of cooperative caching in which caches are arranged as leaf nodes in a hierarchically well-separated tree network [10], and we call this variant hierarchical cooperative caching (HCC).

Initially, we were attracted to the HCC problem in the hope that we would be able to design an efficient online algorithm for solving it. One could hope to design a strategy in which recently accessed files at a cache are placed closer to the cache in the network, and less recent files are placed further away from the cache in the network. One natural question is whether an efficient online algorithm, which is also resource competitive, exists for the HCC problem.

In [31], we present a deterministic hierarchical generalization of LRU that is constant-competitive when the capacity blowup is linear in $d$, the depth of the cache hierarchy. Furthermore, we exhibit an infinite family of depth-$d$ hierarchies such that any randomized hierarchical cooperative caching algorithm with capacity blowup $b$ has competitive ratio $\Omega(\log \frac{d}{b})$ against an oblivious adversary. Thus, we establish that there is no resource competitive algorithm for the HCC problem, and our upper and lower bounds imply a tight bound of $\Theta(d)$ on the capacity blowup required to achieve constant competitiveness.

## 1.3 Compression Caching

Various studies have demonstrated the advantages of compression in caching [2, 4, 28]. A compressed file takes up less space, effectively increasing the size of the fast memory. However, this increase in size comes at the cost of extra processing needed for compression and uncompression. Consequently, it is desirable to keep frequently accessed files uncompressed in the fast memory. More generally, we have the problem of determining — in an online manner — which files to keep in the fast memory, and of these, which to keep in compressed form. The problem becomes even more complicated if there are multiple choices of formats for compressing a file, with varying sizes and costs (for compressing and uncompressing the file).

We define the following class of compression caching problems in which a file can be cached in multiple formats with varying sizes and costs (for compression and uncompression). We are given a cache with a specified capacity, a certain number of compression/uncompression algorithms, and a set of files, each of which can be cached in the uncompressed format or in a compressed format obtained by applying one of the compression algorithms. Each compressed format of a file is specified by three parameters: encode cost, decode cost, and size. The encode cost of a particular format is defined as the cost of creating that format from the uncompressed format of the file. The decode cost of a format is defined as the cost of creating the uncompressed format from the given format. The miss-penalty of a file is defined as the cost of accessing the file if it is not present in the cache. To process a request for a file, the file is required to be loaded into the cache in the uncompressed format. The goal of a compression caching algorithm is to minimize the total cost of processing a given request sequence.

Since compression caching generalizes the disk paging problem considered by Sleator and Tarjan [39] and an efficient resource competitive algorithm (e.g., LRU) exists for the disk paging problem, it is natural to ask whether efficient algorithms exist for compression caching, and whether resource competitive results can be obtained for compression caching.

In this work, we address three problems in the class of compression caching. The first problem assumes that the encode cost and decode cost associated with any format of a file are equal. For this problem we present a resource competitive online algorithm. To explore the existence of resource competitive online algorithms for compression caching with arbitrary encode costs and decode costs, we address two other natural problems in the aforementioned class, and for each of these problems, we show that there exists a non-constant lower bound on the competitive ratio of any online algorithm, even if the algorithm is given an arbitrary factor advantage in cache capacity. Thus, we establish that there is no resource competitive algorithm for compression caching in its full generality. This line of research also leads to problems related to the design of an outsourced storage service (see Chapter 3 for further details).

## 1.4 Aggregation

As mentioned earlier, aggregation is a basic primitive for querying and monitoring distributed data sources, and for building many large-scale distributed applications such as system management [25, 41], file location [12], grid resource monitoring [18], network monitoring [29], and sensor networks [32]. Many generic aggregation frameworks have been proposed in the literature [18, 37, 45]. However, all of these frameworks either use a static aggregation strategy, or require applications to know the read and write access patterns in advance. In this work, we address the problem of aggregation over trees with the goal of adapting aggressiveness in aggregation.

In our work [36], we consider a distributed network with nodes arranged in a tree, and each node having a local value. We formulate the aggregation problem as the problem of aggregating values (e.g., summing values) from all nodes to the requesting nodes in the presence of writes. The goal is to minimize the total number of messages exchanged. The key challenges are to define a notion of "acceptable" aggregate values, and to design algorithms with good performance that are guaranteed to produce such values. We formalize the acceptability of aggregate values in terms of certain consistency guarantees similar to traditional consistency guarantees defined in the distributed shared memory literature. The aggregation problem admits solutions that trade off between consistency and performance. The central question is whether there exists an algorithm that provides strong performance and good consistency guarantees.

We propose a lease-based aggregation mechanism, and evaluate algorithms based on this mechanism in terms of consistency and performance. With regard to consistency, we generalize the definitions of strict and causal consistency for the aggregation problem. We show that any lease-based aggregation algorithm provides strict consistency in sequential executions, and causal consistency in concurrent executions. With regard to performance, we propose an online lease-based aggregation algorithm, and show that, for sequential executions, the algorithm is constant-competitive against any offline algorithm that provides strict consistency. We present our online lease-based aggregation algorithm in the form of a fully distributed protocol, and formally establish the aforementioned consistency and performance results with respect to this protocol. Thus, we provide a positive answer to the central question posed above.

## 1.5 Outline

The rest of this dissertation is organized as follows. In Chapter 2 we describe our results on hierarchical cooperative caching. In Chapter 3 we discuss our results on compression caching. In Chapter 4 we present our results on aggregation over trees. Finally, we conclude in Chapter 5.

# Chapter 2

# Hierarchical Cooperative Caching

## 2.1 Introduction

In the classic disk paging problem, which has been extensively studied, we are given a cache and a sequence of requests for pages. When a page is requested, we incur a miss if it is not already present in the cache. In the event of a miss, we are required to load the requested page into the cache, which may necessitate the eviction of another page. Our goal is to minimize the cost of processing the request sequence, where the cost is defined as the number of misses incurred.

In the seminal paper introducing the notion of competitive analysis, Sleator and Tarjan [39] show that LRU (Least-Recently-Used) and several other online deterministic caching algorithms are $\frac{k}{k-h+1}$-competitive, where $k$ is the cache capacity of the online algorithm and $h$ is the cache capacity of the offline algorithm. They also show that $\frac{k}{k-h+1}$ is the best competitive ratio that can be achieved by any deterministic online caching algorithm. Young [46] proposes the Landlord algorithm that achieves competitive ratio $\frac{k}{k-h+1}$ for a case where the files being cached have nonuniform sizes and retrieval costs. Note that LRU and Landlord are constant-competitive assuming a constant-factor capacity blowup over the corresponding optimal offline algorithm.

In cooperative caching [19], a set of caches cooperate in serving requests for each other and in making caching decisions. The benefits of cooperative caching have been supported by several studies. For example, the Harvest cache [14] introduces the notion of a hierarchical arrangements of caches. Harvest uses the Internet Cache Protocol [43] to support discovery and retrieval of documents from other caches. The Harvest project later became the public domain Squid cache system [42]. Adaptive Web Caching [47] builds a mesh of overlapping multicast trees; the popular files are pulled down towards their users from their origin servers. In local-area network environments, the xFS [5] system utilizes workstations cooperating with each other to cache data and to provide serverless file system services.

A cooperative caching scheme can be roughly divided into three components: placement, which determines where to place copies of files, search, which directs each request to an appropriate copy of the requested file, and consistency, which maintains the desired level of consistency among the various copies of a file. In this work, we study the placement problem, and we assume that a separate mechanism enables a cache to locate a nearest copy of a file, free of cost, and we assume that files are read-only (i.e., copies of a file are always consistent).

We focus on a class of networks where the cost of communication among caches is specified by an ultrametric distance function, the precise definition of which is given in Section 2.2. An ultrametric corresponds to a kind of hierarchical distance function. For this reason, we call the cooperative caching problem in networks with ultrametric distance function the *hierarchical cooperative caching (HCC) problem*. This is an important problem because many actual networks have a hierarchical or approximately hierarchical structure. Furthermore, various caching schemes [14, 16, 42] for a wide area network suggest arranging caches hierarchically. Therefore, we believe that an ultrametric is appropriate for modeling the distance function among caches distributed over a wide area network.

Ultrametrics are equivalent up to a constant-factor to the hierarchically well-separated tree (HST) metrics, as introduced by Bartal [8]. Refining earlier results by Bartal [8, 10], Fakcharoenphol et al. [21] have shown that any metric space can be approximated by the HST metrics with a logarithmic distortion. Hence, many results for the HST metrics imply corresponding results for arbitrary metric spaces, at the expense of an extra logarithmic factor.

For the case where the access distribution of each file at each cache is fixed and known in advance, Korupolu et al. [30] provide a polynomial-time algorithm for the HCC problem that minimizes the average retrieval cost and does not require a capacity blowup. In addition, they provide a faster constant-factor approximation algorithm that does not require a capacity blowup. On the other hand, the assumption in Korupolu et al. [30] of a fixed access distribution is rather strong. Furthermore, even in applications where the access distribution is relatively stable, keeping track of this distribution may be expensive.

Since the HCC problem generalizes the disk paging problem mentioned earlier, we cannot hope to achieve constant competitiveness for the HCC problem without at least a constant-factor capacity blowup. Our main motivation in pursuing the present research has been to determine whether there exists a constant-competitive algorithm with a constant-factor capacity blowup for the HCC problem. Since the LANDLORD algorithm by Young is designed for files with non-uniform retrieval cost, one could think of applying LANDLORD to solve the HCC problem. However, simply running LANDLORD at each cache does not provide a good competitive ratio for the HCC problem, since LANDLORD is not designed to exploit the benefits of cooperation among caches. As stated in Young [46], the focus of LANDLORD "is on simple *local* caching strategies, rather than distributed strategies in which caches cooperate to cache pages across a network".

In this chapter, we show that if an online algorithm is given a sufficiently large capacity blowup, then constant competitiveness can be achieved. In Section 2.4, we present a deterministic hierarchical generalization of LRU that is constant-competitive when the capacity blowup is linear in $d$, the depth of the cache hierarchy. We content ourselves with handling files of unit size only. However, a hierarchical generalization of LANDLORD can be used to deal with files of nonuniform sizes.

Furthermore, we exhibit an infinite family of depth-$d$ hierarchies such that any randomized online HCC algorithm with a capacity blowup $b$ has competitive ratio $\Omega(\log \frac{d}{b})$ against an oblivious adversary. In particular, we construct a hierarchy with a sufficiently large depth and show that an oblivious adversary can generate an arbitrarily long request sequence such that the randomized online HCC algorithm incurs a cost $\Omega(\log \frac{d}{b})$ times that of an optimal offline algorithm. In terms of $n$, the number of caches, our lower bound result shows that the competitive ratio of any randomized HCC algorithm is $\Omega(\log \log n - \log b)$. Our upper and lower bounds imply a tight bound of $\Theta(d)$ on the capacity blowup required to achieve constant competitiveness.

Several paging problems (e.g., distributed paging, file migration, and file allocation) have been considered in the literature, some of which are related to the HCC problem (e.g., see the survey paper by Bartal [9] for the definitions of these problems). In particular, the HCC problem can be formulated as the read-only version of the distributed paging problem on ultrametrics. And the HCC problem without replication is a special case of the constrained file migration problem where the cost accessing a file at distance $d$ is equal to the cost of migrating the file a distance of $d$. Most existing work on these problems focuses on upper bound results, and lower bound results only apply to algorithms without a capacity blowup. For example, for the distributed paging problem, Awerbuch et al. [6] show that, given polylog$(n, \Delta)$ capacity blowup, there exists a deterministic polylog$(n, \Delta)$-competitive algorithm for general networks, where $\Delta$ is the normalized diameter of the network. For the constrained file migration problem, if we let $m$ denote the total capacity of the $n$ caches, Bartal [8] gives a deterministic lower bound of $\Omega(m)$, a randomized lower bound of $\Omega(\log m)$, and a randomized upper bound of $O((\log m) \log^2 n)$.

Applying the recent result of Fakcharoenphol et al. [21], the latter upper bound may be improved to $O((\log m)\log n)$.

The rest of this chapter is organized as follows. Section 2.2 provides some preliminary definitions. Section 2.3 presents our lower bound. Section 2.4 presents our upper bound.

## 2.2 Preliminaries

Assume that we are given a set of caches, each with a specified nonnegative capacity, and a distance function $h$ that specifies the cost of communication between any pair of caches. Such a distance function is a *metric* if it is nonnegative, symmetric, satisfies the triangle inequality, and $h(u, v) = 0$ if and only if $u = v$ for all caches $u$ and $v$. A metric distance function $h$ is an *ultrametric* if $h(u, v) \leq \max(h(u, w), h(v, w))$ for all caches $u$, $v$, and $w$; note that the latter condition subsumes the triangle inequality.

We now describe another method to specify a distance function over a set of caches. In this method, the distance function is encoded as a rooted tree where each node of the tree has an associated nonnegative diameter. There is a one-to-one correspondence between the set of caches and the leaves of the tree, and each leaf has a diameter of zero. The diameter of any node is required to be less than that of its parent. The distance between two caches is then defined as the diameter of the least common ancestor of the corresponding leaves. It is well-known (and easy to prove) that a distance function can be specified by a tree in this manner if and only if it is an ultrametric. We say that such a tree is $\lambda$-*separated*, where $\lambda > 1$, if the diameter of any node is at least $\lambda$ times that of any of its children.

In all of the caching problems addressed in this chapter, we assume that the distance function specifying the cost of communication is an ultrametric, and we adopt the tree view of an ultrametric discussed in the preceding paragraph. The main advantage of this view is that it enables us to leverage standard tree terminology in our technical arguments. Table 2.1 lists a number of useful definitions based on tree terminology.

| Notation | Meaning |
|---:|---|
| $root$ | the root of the tree |
| $\alpha.parent$ | the parent of $\alpha$, where $\alpha \neq root$ |
| $\alpha.ch$ | children of $\alpha$ |
| $\alpha.anc$ | the ancestors of $\alpha$ (including $\alpha$) |
| $\alpha.desc$ | the descendants of $\alpha$ (including $\alpha$) |
| $\alpha.depth$ | the depth of $\alpha$, where the root is considered to be at depth 0 |
| $\alpha.diam$ | the diameter of $\alpha$ |
| $\alpha.caches$ | the set of caches in the subtree rooted at $\alpha$ |
| $\alpha.cap$ | the total capacity of the caches in $\alpha.caches$ |

Table 2.1: Some useful notation. The variable $\alpha$ refers to a tree node.

In the caching problems addressed in this chapter, we refer to the objects to be cached as *files*. The files are assumed to be read-only, so we do not need to deal with the issue of consistency maintenance. Each file is assumed to be indivisible; we do not consider schemes in which a copy of a file may be broken into fragments and spread across multiple caches. Each file $f$ has a specified size, denoted $size(f)$, and penalty, denoted $penalty(f)$. We assume that any file can fit in any cache, that is, the maximum file size is assumed to be at most the minimum cache capacity. The penalty associated with a file represents the cost per unit size to retrieve the file when it is not stored anywhere in the tree of caches, and is assumed to exceed the diameter of the root of the tree.

A copy is a pair $(u, f)$ where $u$ is a cache and $f$ is a file with size at most the capacity of $u$. A set of copies is called a *placement*. If $(u, f)$ belongs to a placement $P$, we say that a copy of $f$ is placed at $u$ in $P$. A placement $P$ is *$b$-feasible* if the total size of the files placed in any cache is at most $b$ times the capacity of the cache.

A caching algorithm maintains a placement. Initially, the placement is empty. Two basic operations, *delete* and *add*, may be used to update a given placement $P$. A delete operation removes a copy from $P$; the algorithm incurs no cost for such a deletion. In an add operation, a copy $(u, f)$ is added to $P$. If, prior to the add, $P$ does not place a copy of $f$ at any cache, then the cost of the add is defined to be $\mathit{penalty}(f)$. Otherwise, the cost is $\mathit{size}(f) \cdot \mathit{dist}(u, v)$, where $v$ is the closest cache at which a copy of $f$ is placed. A caching algorithm $A$ is $b$-feasible if it always maintains a $b$-feasible placement.

A *request* is a pair $(u, f)$ where $u$ is a cache and $f$ is a file. To process such a request, a caching algorithm performs an arbitrary sequence of add and delete operations, subject only to the constraint that $(u, f)$ belongs to at least one of the placements traversed.

The HCC problem is to process a given sequence of requests with the goal of minimizing cost. For any (randomized) HCC algorithm $A$, and any request sequence $\tau$, we define $T_A(\tau)$ as the (expected) cost for $A$ to process $\tau$. An online HCC algorithm $A$ is *$c$-competitive* if for all request sequences $\tau$ and 1-feasible HCC algorithms $B$, $T_A(\tau) \leq c \cdot T_B(\tau)$. (Remark: The asymptotic bounds established in this chapter are unchanged if we allow an additive slack in the definition of $c$-competitiveness, as in [13, Chapter 1].)

An HCC algorithm is $b$-quasifeasible if it maintains a $b$-feasible placement before and after processing a request, and while processing a request, removal of at most one copy of a file from its placement makes its placement $b$-feasible. Observe that any $b$-quasifeasible HCC algorithm is a $(b+1)$-feasible HCC algorithm. In Section 2.4, we present a deterministic constant-competitive $O(d)$-quasifeasible online HCC algorithm; by the preceding observation, our algorithm is $O(d)$-feasible.

An HCC algorithm is *nice* if on a request $(u, f)$, it first adds a copy $(u, f)$ to its placement and then performs an arbitrary sequence of add and delete operations.

Observe that a nice $(b+1)$-feasible HCC algorithm $A$ can simulate any $b$-feasible HCC algorithm $B$ by first retrieving the requested copy, and then exactly following the steps of algorithm $B$. In performing this simulation, algorithm $A$ incurs at most twice the cost of algorithm $B$. Hence, any $b$-feasible $c$-competitive HCC algorithm can be converted into a nice $(b+1)$-feasible $2c$-competitive HCC algorithm. (Remark: With additional care it may be possible to argue that the factor of 2 appearing in the preceding observation can be eliminated.) In Section 2.3 we prove that for any nice $b$-feasible randomized online HCC algorithm $A$, there exists a request sequence $\tau$ and a 1-quasifeasible offline HCC algorithm $B$ such that $T_A(\tau) = \Omega(\log \frac{d}{b}) \cdot T_B(\tau)$. By the foregoing observation, together with that of the preceding paragraph, we can conclude that the competitive ratio of any $b$-feasible randomized online HCC algorithm is $\Omega(\log \frac{d}{b})$.

## 2.3 The Lower Bound

In this section, we present a lower bound that holds for an arbitrary nice randomized $b$-feasible online HCC algorithm ON, where $b$ is a positive integer. The lower bound holds for ON with respect to the trees drawn from an infinite family of $k$-ary, depth $d$ trees, parameterized by integers $d$ and $k$ such that $d = 8bk - 1$. We find it convenient to refer to an arbitrary tree in this family as $T(d,k)$. Furthermore, the diameter of an internal node, $\alpha$, of tree $T(d,k)$ is set to be $\lambda^{d-i-1}$, where $\lambda = \max(\frac{15}{7}, \Omega(\log k))$, $i = \alpha.\mathit{depth}$, and $0 \leq \alpha.\mathit{depth} < d$. Recall from the previous section that the diameter of a leaf is 0. For any file $f$ placed in $T(d,k)$, $\mathit{penalty}(f)$ is set to be $\lambda \cdot \mathit{root.diam}$. Note that $T(d,k)$ is a $\lambda$-separated tree.

The lower bound discussed above is established in Sections 2.3.2 through 2.3.8. Section 2.3.1 illustrates some of the central ideas to be used in the proof of the lower bound by addressing a simpler problem. It is not necessary to read Section 2.3.1 before continuing to Section 2.3.2, but it might provide some useful intuition.

### 2.3.1 A Simple Lower Bound Result

In the present section, we restrict attention to instances of the HCC problem satisfying the following characteristics.

- There are $n$ caches, each with capacity $\ell$, where $\ell$ is a positive integer.
- Each pair of distinct caches are unit distance apart. Note that such a uniform metric space is a special case of an ultrametric; it corresponds to a one-level hierarchy in which the root (which does not have an associated cache) is at distance $1/2$ to each of its $n$ children (the caches), and the shortest path between any pair of caches goes through the root.
- There are two sets of $\ell$ unit-sized files $X$ and $Y$, and every access is to a file in $X \cup Y$.
- Each file in $X \cup Y$ has the same associated penalty $\rho = \Omega(\log n)$.

In addition, we make the following simplifying assumptions.

- The online algorithm is deterministic. We make this assumption primarily for ease of presentation. The $\Omega(\log n)$ lower bound presented in this section is easily generalized to hold for randomized online algorithms. The proof of our main lower bound is generalized to handle the randomized case.

- The capacity blowup $b$ afforded to the online algorithm is 1. Our assumption that $b = 1$ allows for an easy $\Omega(\ell)$ lower bound argument, since the adversary can restrict all accesses to a single cache, and the results of Sleator and Tarjan [39] then imply an $\Omega(\ell)$ lower bound. But the results of Sleator and Tarjan also imply that if we allow $b$ to be a constant greater than one, this approach cannot yield a non-constant lower bound. In fact, as we show in Section 2.4, it is possible to give a constant-competitive algorithm for the general HCC problem if the capacity blowup $b$ is at least the depth of the hierarchy. Since we are currently focusing on the special case of a constant-depth hierarchy, we cannot hope to establish a non-constant lower bound on the competitive ratio while allowing an arbitrary constant capacity blowup $b$.

Given the foregoing assumptions, we now sketch a proof of an $\Omega(\log n)$ lower bound on the competitive ratio. At the end of this section, we briefly indicate how the ideas used to prove this $\Omega(\log n)$ lower bound are generalized in Sections 2.3.2 through 2.3.8 to establish our main lower bound for the HCC problem.

Fix an online algorithm $A$. Our objective is to produce a request sequence $\sigma$ and an offline algorithm $B$ such that $A$'s cost to serve $\sigma$ is $\Omega(\log n)$ times that of $B$. Initially, the offline algorithm $B$ loads one of its caches with the files in the set $X$, and loads each of the remaining caches with the files in the set $Y$. Note that the offline algorithm $B$ incurs $\Theta((\rho + n)\ell)$ cost to establish this initial configuration. We generate a sufficiently long request sequence

$\sigma$ so that the cost incurred by $A$ exceeds the initial configuration cost of $B$ by an $\Omega(\log n)$ factor. This allows us to ignore the initial configuration cost of $B$ in the remainder of this proof sketch.

The offline algorithm $B$ maintains the invariant that one cache holds the set of files $X$, and every other cache holds the set of files $Y$. Typically, the configuration of $B$ does not change when a read request is processed. Occasionally, $B$ selects a new cache to hold $X$, and updates its configuration accordingly, paying $\Theta(\ell)$ cost. These updates to $B$'s configuration partition time into epochs. Within each epoch, we ensure that the online algorithm $A$'s cost to service each request is $\Omega(\log n)$ times that of $B$. Furthermore, we ensure that the total cost paid by $A$ within each completed epoch is $\Omega(\ell \log n)$. Since the cost required by $B$ to update its configuration at the end of each epoch is $\Theta(\ell)$, the desired lower bound follows.

To ensure that the online algorithm $A$'s cost to service each request within an epoch is $\Omega(\log n)$ times that of the offline algorithm $B$, $B$ maintains a partition of the $n$ caches into two sets $U$ and $V$. At the beginning of the epoch, all of the caches are in the set $U$. During the epoch, $B$ periodically shifts caches from $U$ to $V$. The epoch ends when there is exactly one cache remaining in $U$. The offline algorithm $B$ ensures that the cache belonging to $U$ throughout the epoch is the one that stores $X$ in this epoch. Consequently, during the epoch, a request for a file in $Y$ at a cache in $V$ costs $B$ nothing. At a general point within the epoch, the next request to be appended to $\sigma$ is determined in the following manner.

- First, if some file $x$ in $X$ is not stored in any of $A$'s caches, then a request is generated for $x$ at a particular fixed cache, say the cache with the lowest numerical identifier. In this case $A$ pays at least $\rho = \Omega(\log n)$ to service the request, while $B$ pays at most one unit.

- Second, if there exists a file $y$ in $Y$ that is not stored by $A$ at some cache $v$ in $V$, then we generate a request for $y$ at $v$. In this case $A$ pays at least one unit to service the request, while $B$ pays zero.

- Otherwise, there exists (by a simple averaging argument) a cache $u$ in $U$ at which $A$ stores at least $|X|/|U|$ files belonging to the set $X$. The offline algorithm shifts $u$ from $U$ to $V$, and a request is generated at $u$ for a file $y$ in $Y$ that $A$ does not currently store at $u$. As argued in the next paragraph, as long as $|U| > 1$, we are able to ensure that the cache $u$ shifted to $V$ is not the cache that the offline algorithm $B$ is using to store $X$ in the current epoch. Therefore, $A$ pays at least one unit to service the request, while $B$ pays zero.

Therefore, in every case, $A$'s cost to service a request within an epoch is $\Omega(\log n)$ times that of $B$.

To maintain the invariant that $U$ contains the cache $w$ currently used by the offline algorithm $B$ to store $X$, $w$ is chosen in an adversarial manner. Informally, the online algorithm is forced to play a shell game in each epoch, where the shells are the $n$ caches and the online algorithm makes successive guesses as to the identity of $w$. Since the online algorithm is deterministic, $w$ can be chosen to be the $n$th guess of the online algorithm.

It remains to prove that the total cost paid by the online algorithm $A$ within each completed epoch is $\Omega(\ell \log n)$. As we have argued above, $A$ stores at least $|X|/|U|$ files belonging to the set $X$ in cache $u$ when $u$ is shifted to $V$. Since no other cache is shifted to $V$ until $A$ stores all of $Y$ in $u$, $A$ incurs a cost at least $|X|/|U|$ before $V$ grows again. Thus the total cost incurred by $A$ during a completed epoch is at least $|X| \cdot \sum_{2 \le i \le n} 1/i = \Omega(\ell \log n)$, establishing the desired lower bound.

In the sections that follow, we establish our main lower bound by generalizing the shell game described above to ultrametrics corresponding to complete, regular trees of non-constant depth. The basic intuition is that the online algorithm is forced to play many shell games in parallel, one corresponding to each node of the tree. The adversarial nature of the shell games ensures that the online algorithm generally makes incorrect guesses related to the hierarchical configuration maintained by the offline algorithm. These incorrect guesses erode the capacity advantage $b$ enjoyed by the online algorithm: By employing a sufficiently deep tree, the online algorithm can be forced to devote all of the space in certain caches to files associated with such incorrect guesses. The next request is introduced at such a cache.

### 2.3.2 **Algorithm** ADV

In Figure 2.1, we present an algorithm for an *oblivious adversary* [13, Chapter 4], ADV, that constructs a request sequence $\sigma$ of any given length $N$. The following definitions are useful for developing the ADV algorithm.

- For any nonnegative integer $i$ and positive integer $j$, let

$$g(i, j) = k^{d-i} \cdot \left( \frac{i-1}{8k} + \frac{1}{4j} \right).$$

- For any node $\alpha$, we define associated "reactivation" and "deactivation" values $\alpha.react = g(\alpha.depth, k)$ and $\alpha.deact = g(\alpha.depth, 2k)$.

- We define the "activation" value of $\alpha$, denoted $\alpha.act$, as $g(\alpha.depth, r)$ where $r = |\{\beta : \beta \in \alpha.parent.ch : \beta.x = 0\}|$. Note that $\alpha.act$ is a function of the program state, since it depends on the values of certain program variables (i.e., $\beta.x$ for all $\beta$ in $\alpha.parent.ch$).

- We fix $d + 1$ disjoint sets of unit-sized files $F(i)$, $0 \leq i \leq d$, such that $|F(i)| = \lceil k^{d-i-1} \rceil$ for $0 \leq i \leq d$. Each request in the request sequence $\sigma$ generated by ADV involves a file drawn from these sets.

- We define $\alpha.placed$ as the set of distinct files placed by ON in $\alpha.caches$ after processing the request sequence given by the program variable $\sigma$. Since ON is a randomized algorithm, $\alpha.placed$ is a random set.

- We define $\alpha.load$ as the expected value of $|(\cup_{0 \leq i < \alpha.depth} F(i)) \cap \alpha.placed|$.

- We define $\alpha.missing$ as the set of all files $f$ in $F(\alpha.depth)$ and $\Pr(f \in \alpha.placed) \leq \frac{1}{2}$.

Algorithm ADV is oblivious since it constructs the request sequence without examining the random bits used by ON during its execution. Whenever line 18 of ADV is executed, a request is appended to $\sigma$ and ON processes this request. The main technical result to be established in this section is that, for $N$ sufficiently large, $T_{\text{ON}}(\sigma)$ is $\Omega(\log \frac{d}{b})$ times $T_A(\sigma)$ for an optimal 1-quasifeasible offline algorithm $A$.

### 2.3.3 Correctness of ADV

We show in this section that ADV is well-defined (i.e., $\pi \neq root$ just before line 5, $\pi$ is not a leaf just before line 8, and line 14 finds a child) and that each round terminates with the generation of a request. For the sake of brevity, in our reasoning below, we call a predicate a *global invariant* if it holds everywhere in ADV (i.e., it holds initially and it holds between any two adjacent lines of the pseudocode in Figure 2.1).

**Lemma 2.3.1.** *Let $I_1$ denote that every internal node has a child with $x$ field equal to 0, $I_2$ denote that $\pi$ is a node, and $I_3$ denote that $\pi.load \geq \pi.deact$. Then $I_1 \wedge I_2$ is a global invariant and $I_3$ holds everywhere in the down loop.*

*Proof.* The predicate $I_1 \wedge I_2$ holds initially because $\pi = root$ and $\alpha.x = 0$ for all $\alpha$, and $I_3$ holds just before the down loop due to the guard of the up loop. We next show that every line of code outside the down loop preserves $I_1 \wedge I_2$

```
   {initially, $N \geq 0$, $count = 0$, $root.x = root.y = root.act = g(0, k)$,
     $\pi = root$, $\alpha.x = \alpha.y = 0$ for all $\alpha \neq root$, and $\sigma$ is empty}
1  while $count < N$ do {main loop}
2    while $\pi.load < \pi.deact$ do {up loop}
3      $\pi.y := \pi.react$;
4      for every child $\delta$ of $\pi$, set both $\delta.x$ and $\delta.y$ to 0;
5      $\pi := \pi.parent$
6    od; {end of up loop}
7    while $\pi.missing = \emptyset$ do {down loop}
8      if a child $\delta$ of $\pi$ satisfies $\delta.x > 0 \wedge \delta.load \geq \delta.react$ then
9        $\pi := \delta$
10     else
11       if $\pi$ has exactly one child with $x$ equal to 0 then
12         for every child $\delta$ of $\pi$, set both $\delta.x$ and $\delta.y$ to 0
13       fi;
14       $\pi :=$ a child $\delta$ of $\pi$ such that $\delta.x = 0 \wedge \delta.load \geq \delta.act$;
15       set both $\pi.x$ and $\pi.y$ to $\pi.act$
16     fi
17   od; {end of down loop}
18   append to $\sigma$ a request for an element in $\pi.missing$
         at an arbitrary cache in $\pi.caches$;
19   $count := count + 1$
20 od {end of main loop}
```

Figure 2.1: The ADV algorithm. We remark that the $y$ field maintained at each node has no impact on the computation of the request sequence $\sigma$. (To see this, note that the $y$ field is written, but never read.) The $y$ field has been introduced to facilitate our analysis.

(i.e., if $I_1 \wedge I_2$ holds before the line, then it holds after the line) and every line of code in the down loop preserves $I_1 \wedge I_2 \wedge I_3$.

Each line of code outside the down loop preserves $I_1$ because such lines do not assign a nonzero value to an $x$ field. The only line that affects $I_2$ is line 5. We observe that $\pi \neq \mathit{root}$ just before line 5, due to the guard of the up loop and the observation that $\mathit{root.load} \geq \mathit{root.deact} = 0$. Hence, line 5 preserves $I_2$. It follows that every line of code outside the down loop preserves $I_1 \wedge I_2$.

In the down loop, the only line that affects $I_1$ is 15, but $I_3$ and the inner **if** statement establish that $\pi$ has at least two children with $x$ field equal to 0 just before line 14. Hence, if $I_1 \wedge I_3$ holds before line 15, then $I_1$ holds after line 15.

We now argue that for each line of code in the down loop, if $I_1 \wedge I_2 \wedge I_3$ holds before execution of the line, then $I_2$ holds after execution of the line. It is sufficient to prove that if the assignment statement of line 14 is executed, the RHS is well-defined (i.e., some child $\delta$ of $\pi$ satisfies $\delta.x = 0$ and $\delta.\mathit{load} \geq \delta.\mathit{act}$). To establish the claim, first we need to show that $\pi.\mathit{depth} < 8bk - 1$ (i.e., $\pi$ is not a leaf) just before line 8. Let us assume to the contrary that $\pi.\mathit{depth} = 8bk - 1$ just before line 8. (Note that $I_2$ implies that $\pi.\mathit{depth}$ cannot take on a higher value.) By the guard of the down loop, the probability that $\pi.\mathit{placed}$ contains the lone file in $F(8bk - 1)$ is at least $\frac{1}{2}$. Furthermore, $I_3$ implies that $\pi.\mathit{load} \geq \pi.\mathit{deact} = g(8bk - 1, 2k) = b - \frac{1}{8k}$. It follows that the expected number of files stored by ON in the cache associated with the leaf $\pi$ is at least $b - \frac{1}{8k} + \frac{1}{2} > b$, which is a contradiction since $\pi.\mathit{cap} = 1$ and ON is

$b$-feasible. Hence, $\pi.depth < 8bk - 1$ just before line 8.

We now argue that if the assignment statement of line 14 is executed, the RHS is well-defined. Let $A = \{\alpha : \alpha \in \pi.ch \wedge \alpha.x = 0\}$ and $B = \{\beta : \beta \in \pi.ch \wedge \beta.x > 0\}$. Let $r$ denote $|A|$ and $i$ denote $\pi.depth$. We observe that

$$\begin{aligned}
& \sum_{\alpha \in A} \alpha.load \\
= \; & \sum_{\alpha \in \pi.ch} \alpha.load - \sum_{\beta \in B} \beta.load \\
\geq \; & \pi.load + \frac{|F(i)|}{2} - \sum_{\beta \in B} \beta.load \\
\geq \; & \pi.deact + \frac{|F(i)|}{2} - \sum_{\beta \in B} \beta.react \\
= \; & g(i, 2k) + \frac{k^{d-i-1}}{2} - \sum_{\beta \in B} g(i+1, k) \\
= \; & k^{d-i} \cdot \frac{i}{8k} + \frac{k^{d-i-1}}{2} - (k - r) \cdot k^{d-i-1} \cdot \frac{(i+2)}{8k} \\
= \; & r \cdot k^{d-i-1} \cdot \left( \frac{i}{8k} + \frac{1}{4r} + \frac{1}{4k} \right).
\end{aligned}$$

(In the derivation above, the first inequality is due to the definition of *load* and the guard of the down loop, i.e, for each file $f$ in $|F(i)|$, $\Pr(f \in \alpha.placed) > \frac{1}{2}$, and the second inequality is due to $I_3$ and the guard of the outer **if** statement. The formula for $|F(i)|$ is valid in the second equality since $i < 8bk - 1$.) Hence, by an averaging argument (note that $r > 0$ by $I_1$), there exists a child $\delta$ of $\pi$

such that $\delta.x = 0$ and

$$\begin{aligned} & \delta.load \\ \geq\ & k^{d-i-1} \cdot \left(\frac{i}{8k} + \frac{1}{4r}\right) \\ =\ & \delta.act. \end{aligned}$$

Hence, the RHS of line 14 evaluates to a node.

Recall that we wish to show that every line of code in the down loop preserves $I_1 \wedge I_2 \wedge I_3$. Thus far we have established that for each line of code in the down loop, if $I_1 \wedge I_2 \wedge I_3$ holds before execution of the line, then $I_1 \wedge I_2$ holds after execution of the line. It remains to show that for each line of code in the down loop, if $I_1 \wedge I_2 \wedge I_3$ holds before execution of line, then $I_3$ holds after execution of the line. The only lines in the down loop that affect $I_3$ are 9 and 14. By $I_2$, $\pi$ is a node and by definition, $\alpha.react \geq \alpha.deact$ and $\alpha.act \geq \alpha.deact$ for all $\alpha$. Hence, if $I_2 \wedge I_3$ holds before lines 9 and 14, then $I_3$ holds after both of these lines.

This completes our proof of the lemma. □

**Lemma 2.3.2.** *The up loop terminates.*

*Proof.* Every iteration of the up loop moves $\pi$ to its parent, and $root.load \geq root.deact$ by definition. Hence, the up loop terminates. □

**Lemma 2.3.3.** *The down loop terminates.*

*Proof.* Every iteration of the down loop moves $\pi$ to one of its children. By $I_2$ of Lemma 2.3.1, $\pi$ is always a well defined node. Hence, the down loop terminates. $\square$

**Lemma 2.3.4.** *After generating a sequence $\sigma$ of $N$ requests,* ADV *terminates.*

*Proof.* Follows from Lemmas 2.3.2 and 2.3.3. $\square$

### 2.3.4 Some Properties of ADV

We first prove some properties of ADV that follow directly from its structure. For the sake of brevity, for a property that is a global invariant, we sometimes only state the property but omit stating that the property holds everywhere.

**Lemma 2.3.5.** *For all $\alpha$, $\alpha.x = 0$ or $\alpha.x \geq \alpha.react$.*

*Proof.* The claim holds initially because $\alpha.x = 0$ for all $\alpha$. The only line that assigns a nonzero value to $x$ is 15, which preserves the claim because by definition, $\alpha.act \geq \alpha.react$ for all $\alpha$. $\square$

**Lemma 2.3.6.** *For all $\alpha$, $\alpha.y$ equals 0 or $\alpha.react$ or $\alpha.x$.*

*Proof.* The claim holds initially because $\alpha.y = 0$ for all $\alpha$. The only lines that modify $x$ are 4, 12, and 15. The only lines that modify $y$ are 3, 4, 12, and 15. By inspection of the code, all of these lines trivially preserve the claim. $\square$

**Lemma 2.3.7.** *Let $P$ denote the predicate that every node in $\pi.anc$ has a positive $x$ value, and every node that is neither in $\pi.anc$ nor a child of a node in $\pi.anc$ has a zero $x$ value. Then $P$ is a loop invariant of the up loop, the down loop, and the main loop.*

*Proof.* Let $X$ denote $\pi.anc$ and let $Y$ denote the set of nodes that are neither in $X$ nor children of the nodes in $X$.

Every iteration of the up loop moves $\pi$ to its parent. To avoid confusion, we use $\pi$ to denote the old node (i.e., child) and $\pi'$ to denote the new node (i.e., parent). An iteration of the up loop removes $\pi$ from $X$, adds $\pi.ch$ to $Y$, and sets the $x$ value of $\pi.ch$ to 0. Therefore, it preserves $P$.

Every iteration of the down loop moves $\pi$ to one of its children. To avoid confusion, we use $\pi$ to denote the old node (i.e., parent) and $\pi'$ to denote the new node (i.e., child). Suppose the down loop takes the first branch of the outer **if** statement. Then it adds $\pi'$, which has a positive $x$ value, to $X$ and removes $\pi'.ch$ from $Y$. Hence it preserves $P$. Suppose the down loop takes the second branch of the outer **if** statement. If line 12 is executed, $P$ is preserved because line 12 leaves $X$ and $Y$ unchanged and only changes the $x$ value of the nodes in neither $X$ nor $Y$. Then lines 14 and 15 preserves $P$ because they add $\pi'$, which has a positive $x$ value after line 15, to $X$ and removes $\pi'.ch$ from $Y$. Hence, it preserves $P$.

The main loop preserves $P$ because both the up loop and the down loop preserve $P$. □

**Lemma 2.3.8.** *For all $\alpha$, $\alpha.y \leq \alpha.x$.*

*Proof.* The claim holds initially because $\alpha.x = \alpha.y = 0$ for all $\alpha$. The only lines that modify the $x$ or $y$ field are 3, 4, 12, and 15. At lines 4, 12, and 15, the $x$ and $y$ fields become the same value. It follows from Lemma 2.3.7 and the guard of the up loop that just before line 3, $\pi \neq \mathit{root}$ and $\pi.x > 0$. It then follows from Lemmas 2.3.5 and 2.3.6 that line 3 preserves $\pi.y \leq \pi.x$. $\square$

We now introduce the notion of an active sequence to facilitate our subsequent proofs. A sequence $\langle a_0, a_1, \ldots, a_r \rangle$, where $0 \leq r < k$, is called *$i$-active* if $a_j = g(i+1, k-j)$ for all $0 \leq j \leq r$.

**Lemma 2.3.9.** *For every internal node $\alpha$, the nonzero $x$ fields of the children of $\alpha$ form an $i$-active sequence, where $i = \alpha.\mathit{depth}$.*

*Proof.* The claim holds initially because $\alpha.x = 0$ for all $\alpha$. The only lines that modify the $x$ field are 4, 12, and 15. Lines 4 and 12 preserve the claim because the $x$ fields of the children of $\pi$ all become 0. Line 15 preserves the claim (for $\pi.\mathit{parent}$) because $\pi.x$ becomes $\pi.\mathit{act}$, which by definition equals $g(i+1, k-j)$, where $i = \pi.\mathit{parent}.\mathit{depth}$ and $j$ equals the number of children of $\pi.\mathit{parent}$ that have a positive $x$ field. $\square$

**Lemma 2.3.10.** *Let $P(\alpha)$ denote the predicate that for all $\beta$ that are not ancestors of $\alpha$, $\beta.y \leq \beta.\mathit{react}$. Then $P(\pi)$ holds initially and $P(\pi)$ is a loop invariant of the up loop, the down loop, and the main loop.*

*Proof.* The predicate $P(\pi)$ holds initially because $\pi = \textit{root}$ and $\alpha.y = 0$ for all $\alpha$. The up loop preserves $P(\pi)$ because every iteration first establishes $\pi.y = \pi.\textit{react}$ and then moves $\pi$ to its parent. The down loop preserves $P(\pi)$ because it does not set the $y$ field to a nonzero value. The main loop preserves $P(\pi)$ because both the up loop and the down loop preserve $P(\pi)$. □

### 2.3.5 Colorings

In order to facilitate the presentation of an offline algorithm in Section 2.3.6, we introduce the notion of colorings in this section and the notion of consistent placements in the next.

A *coloring* of $T(d, k)$ (recall that $T(d, k)$ is the tree of caches) is an assignment of one of the colors {white, black} to every node in $T(d, k)$ so that the following rules are observed: (1) *root* is white, (2) every internal white node has exactly one black child and $k - 1$ white children, and (3) the children of a black node are black. A coloring is called *consistent* (with ADV) if for every $\alpha$, if $\alpha.x > 0$, then $\alpha$ is white.

For any coloring $C$ and any pair of sibling nodes $\alpha$ and $\beta$, we define $\textit{swapc}(C, \alpha, \beta)$ (swap coloring) as the coloring obtained from $C$ by exchanging the color of each node in the subtree rooted at $\alpha$ with that of the corresponding node in the subtree rooted at $\beta$. (Note that the subtrees rooted at $\alpha$ and $\beta$ have identical structure.)

### 2.3.6 Consistent Placements

A placement is *colorable* if there exists a coloring $C$ such that: (1) for each white internal node $\alpha$ of $T(d,k)$, the files in $F(\alpha.depth)$ are stored in (and fill) the caches associated with the unique black child of $\alpha$; (2) for each white leaf $\alpha$ of $T(d,k)$, the (singleton) file in $F(\alpha.depth)$ is stored in (and fill) the cache $\alpha$. Note that in the preceding definition of a colorable placement, the coloring $C$, if it exists, is unique. A placement is called *consistent* if it is colorable and the associated coloring is consistent.

For any placement $P$ and any pair of siblings $\alpha$ and $\beta$, we define $swapp(P,\alpha,\beta)$ (swap placement) as the placement obtained from $P$ by exchanging the contents of each cache in $\alpha$ with that of the corresponding cache in $\beta$. (It is convenient to assume that the children of each node in $T(d,k)$ are ordered from left to right. This induces an overall left to right ordering of $\alpha.caches$ and $\beta.caches$. For all $i$, the $i$th cache in $\alpha.caches$ corresponds to the $i$th cache in $\beta.caches$.) Note that for any colorable placement $P$ with associated coloring $C$ and any pair of sibling nodes $\alpha$ and $\beta$, the placement $swapp(P,\alpha,\beta)$ is colorable, and its associated coloring is $swapc(C,\alpha,\beta)$.

### 2.3.7 The Offline Algorithm OFF

For every internal node $\alpha$, we maintain an additional variable $\alpha.last$ defined as follows. First, we partition the execution of the adversary algorithm into epochs with respect to $\alpha$. The first epoch begins at the start of the execution. Each subsequent epoch begins when either line 4 or line 12 is executed with $\pi = \alpha$. The variable $\alpha.last$ is updated at the start of each epoch, when it is set to the child $\beta$ of $\alpha$ for which line 15 is executed with $\pi = \beta$ furthest in the future in that epoch. (If one or more children $\beta$ of $\alpha$ are such that line 15 is never executed with $\pi = \beta$ in the future, then $\alpha.last$ is set to an arbitrary such child $\beta$.) Note that the variables $\alpha.last$ are introduced solely for the purpose of analysis and have no impact on the execution of ADV.

At any point in the execution of ADV, the values of the *last* fields determine a unique coloring, denoted by $C_{\text{OFF}}$, as follows: *root* is white and the black child of each internal white node $\alpha$ is $\alpha.last$.

We define a 1-quasifeasible offline algorithm OFF that maintains a placement $P_{\text{OFF}}$ as follows. We initialize $P_{\text{OFF}}$ to an arbitrary consistent placement with associated coloring $C_{\text{OFF}}$. We update $P_{\text{OFF}}$ to $swapp(P_{\text{OFF}}, \alpha, \beta)$ whenever line 4 or line 12 is executed, where $\alpha$ and $\beta$ denote the values of $\pi.last$ before and after the execution of the line. Whenever line 18 is executed, a request is generated and the algorithm OFF uses the placement $P_{\text{OFF}}$ to process this request. On a request $(u, f)$, if there is not already a copy of file $f$ at $u$, OFF creates a copy $(u, f)$ in order to process the request and then immediately discards the copy. Note that the capacity constraint can be violated at $u$ by one unit when the copy $(u, f)$ is created, but the capacity constraint is satisfied before processing the next request. Hence, the placement $P_{\text{OFF}}$ remains the same before and after line 18, and $P_{\text{OFF}}$ is updated only at lines 4 and 12.

**Lemma 2.3.11.** *Throughout the execution of ADV, $P_{\text{OFF}}$ is colorable and has associated coloring $C_{\text{OFF}}$.*

*Proof.* Immediate from the way $P_{\text{OFF}}$ is updated whenever a *last* field is updated. ☐

**Lemma 2.3.12.** *Execution of line 4 or line 12 preserves the consistency of $C_{\text{OFF}}$.*

*Proof.* Assume that $C_{\text{OFF}}$ is consistent before line 4. So $\pi$ is white in $C_{\text{OFF}}$ before line 4, because by Lemma 2.3.7, $\pi.x$ is positive before line 4. By the definition of $C_{\text{OFF}}$, before line 4, $\pi.last$ is black. Let $\alpha$ be $\pi.last$ before line 4, and let $\beta$ be $\pi.last$ after line 4. Before and after line 4, the $x$ values of the descendants of $\alpha$ are equal to 0. By Lemma 2.3.7, the $x$ values of all proper descendants of $\beta$ are equal to 0 before and after line 4. Since $\beta.x = 0$ after line 4, the $x$ values of all descendants of $\alpha$ and $\beta$ are equal to 0 after line 4. Hence, the *swapp* operation preserves the consistency of $C_{\text{OFF}}$. The same argument applies to line 12. ☐

**Lemma 2.3.13.** *Execution of line 15 preserves the consistency of $C_{\text{OFF}}$.*

*Proof.* Assume that $C_{\text{OFF}}$ is consistent before line 15. Line 14 implies that $\pi \neq root$ just before line 15. Let $\pi'$ denote $\pi.parent$. By Lemma 2.3.7, $\pi'.x > 0$ and hence $\pi'$ is white before line 15. Therefore, by construction of ADV, $\pi'.last$ is the black child of $\pi'$.

Let $t$ denote the start of the current epoch for $\pi'$, i.e., $t$ is the most recent time at which $\pi'.last$ was assigned. Just after time $t$, the $x$ values of all children of $\pi'$ were equal to 0. By the definition of $t$, no child of $\pi'$ has been set to 0 since time $t$. By Lemma 2.3.1, every internal node has at least one child with $x$ equal to 0. Therefore, from time $t$ until after the execution of line 15, at most $k-1$ children of $\pi'$ have had their $x$ value set to a nonzero value. (Note that line 15 is the only line that sets $x$ to a nonzero value.) Thus, by the definition of *last*, $\pi'.last.x$ remains 0 after the execution of line 15. Thus, $\pi'.last \neq \pi$. Since $\pi'$ is white and $\pi'.last$ is black in $C_{\mathrm{OFF}}$, we conclude that $\pi$ is white in $C_{\mathrm{OFF}}$. So $C_{\mathrm{OFF}}$ remains consistent even with the additional constraint that $\pi$ is required to be white. (Note that $\pi.x$ is set to a positive value by line 15.) □

**Lemma 2.3.14.** *The placement $P_{\mathrm{OFF}}$ is always consistent.*

*Proof.* Lines 4, 12, and 15 are the only lines that can affect the consistency of $C_{\mathrm{OFF}}$ since they are the only lines that modify the *last* field or the $x$ field of any node. From Lemmas 2.3.12 and 2.3.13, these lines preserve the consistency of $C_{\mathrm{OFF}}$. From Lemma 2.3.11 it follows that $P_{\mathrm{OFF}}$ is always consistent. □

### 2.3.8 A Potential Function Argument

In this section, we use a potential function argument to show that ON is $\Omega\left(\frac{\nu}{\nu'}\right)$-competitive, where

$$\nu = \min\left(\frac{\lambda}{16}, \frac{\ln k}{8} - \frac{1}{8}\right)$$

and $\nu' = \frac{\lambda}{\lambda - 1}$. Let $T'_{\mathrm{OFF}}(\sigma)$ denote the total cost incurred by OFF to process request sequence $\sigma$, except that we exclude from $T'_{\mathrm{OFF}}(\sigma)$ the cost of initializing $P_{\mathrm{OFF}}$. (This initialization cost is taken into account in the proof of Theorem 1 below.) We define $\Phi$, a potential function, as:

$$\begin{aligned} \Phi \;=\; & \nu \cdot T'_{\mathrm{OFF}}(\sigma) - \nu' \cdot T_{\mathrm{ON}}(\sigma) + \\ & \sum_{\alpha \in \pi.anc \wedge \alpha \neq root} \alpha.parent.diam \cdot \alpha.x + \\ & \sum_{\alpha \notin \pi.anc} \alpha.parent.diam \cdot (\alpha.x - \alpha.y + \alpha.load) \end{aligned} \tag{2.1}$$

**Lemma 2.3.15.** *The cost incurred by swapp$(P, \alpha, \beta)$ is at most $2 \cdot k^{d-i} \cdot$ $\alpha.parent.diam$, where $i = \alpha.depth$.*

*Proof.* The cost incurred is the cost of exchanging the files placed in $\alpha$ and $\beta$ with each other, which is at most $2 \cdot \alpha.cap \cdot \alpha.parent.diam = 2 \cdot k^{d-i} \cdot$ $\alpha.parent.diam$. Note that $\alpha$ and $\beta$ have the same capacity. □

**Lemma 2.3.16.** *The predicate $\Phi \leq 0$ is a loop invariant of the up loop.*

*Proof.* Every iteration of the up loop moves $\pi$ to its parent. To avoid confusion, we use $\pi$ to refer to the old node (i.e., child) and we use $\pi'$ to refer to the new node (i.e., parent). Consider the change in $\Phi$ in a single iteration of the up loop. ON incurs no cost in the up loop. By the definition of $\Phi$, line 3 preserves $\Phi$. By Lemma 2.3.8, line 4 does not increase $\Phi$. Let $i = \pi.depth$. By Lemma 2.3.15, after the execution of line 4, OFF incurs a cost of at most

$c = 2 \cdot k^{d-i-1} \cdot \pi.diam$ to move from the current consistent placement to the next. Thus, the total change in $\Phi$ in an iteration is at most

$$\begin{aligned}
& \nu \cdot c - \pi'.diam \cdot (\pi.y - \pi.load) \\
\leq\ & \nu \cdot c - \pi'.diam \cdot (\pi.react - \pi.deact) \\
=\ & \nu \cdot c - \pi'.diam \cdot (g(i,k) - g(i,2k)) \\
=\ & \nu \cdot c - \pi'.diam \cdot k^{d-i-1} \cdot \frac{1}{8} \\
\leq\ & \nu \cdot c - \frac{\lambda}{16} \cdot c \\
\leq\ & 0.
\end{aligned}$$

(In the derivation above, the first inequality is due to the guard of the up loop and line 3, and the second inequality is due to the assumption that $T(d,k)$ is $\lambda$-separated.) □

**Lemma 2.3.17.** *The predicate $\Phi \leq 0$ is a loop invariant of the down loop.*

*Proof.* Every iteration of the down loop moves $\pi$ to one of its children. To avoid confusion, we use $\pi$ to refer to the old node (i.e., parent) and $\pi'$ to refer to the new node (i.e., child). ON incurs no cost in the down loop. We consider the following three cases.

Suppose that the outer **if** statement takes the first branch. In this case, OFF does not incur any cost. Thus, the change in $\Phi$ is

$$\begin{aligned} & \pi.diam \cdot (\pi'.y - \pi'.load) \\ \leq \; & \pi.diam \cdot (\pi.react - \pi.react) \\ = \; & 0, \end{aligned}$$

where the inequality is due to Lemma 2.3.10 and the guard of the outer **if** statement.

Suppose that the outer **if** statement takes the second branch and that line 12 is not executed. In this case, OFF does not incur any cost. Thus, the change in $\Phi$ is

$$\begin{aligned} & \pi.diam \cdot (\pi'.y - \pi'.load) \\ = \; & \pi.diam \cdot (\pi'.x - \pi'.load) \\ \leq \; & 0, \end{aligned}$$

where the equality is due to line 15 and the inequality is due to lines 14 and 15.

Suppose that the outer **if** statement takes the second branch and that line 12 is executed. By Lemma 2.3.15, in this case, OFF incurs a cost of $c = 2 \cdot k^{d-i-1} \cdot \pi.diam$, where $i = \pi.depth$. Thus, the change in $\Phi$ due to line

12 is at most

$$\begin{aligned}
& \nu \cdot c - \pi.diam \cdot \sum_{\delta \in \pi.ch} (\delta.x - \delta.y) \\
\leq\ & \nu \cdot c - \pi.diam \cdot \sum_{\delta \in \pi.ch} (\delta.x - \delta.react) \\
=\ & \nu \cdot c - \pi.diam \cdot \sum_{j=0}^{k-2} (g(i+1, k-j) - g(i+1, k)) \\
=\ & \nu \cdot c - \pi.diam \cdot k^{d-i-1} \sum_{j=0}^{k-2} \left( \frac{1}{4(k-j)} - \frac{1}{4k} \right) \\
\leq\ & \nu \cdot c - \left( \frac{\ln k}{8} - \frac{1}{8} \right) \cdot c \\
\leq\ & 0.
\end{aligned}$$

(In the above derivation, $\delta.x$ and $\delta.y$ denotes the values just before the execution of line 12, the first inequality follows from Lemma 2.3.10, the first equality follows from Lemma 2.3.9, and the second inequality follows from the fact that $H_{k-1} > \ln k$, where $H_{k-1}$ denotes the $(k-1)$th harmonic number, that is, $H_{k-1} = \sum_{i=1}^{k-1} \frac{1}{i}$.) By the analysis of the previous case (i.e., the outer **if** statement takes the second branch but line 12 is not executed), lines 14 and 15 do not increase $\Phi$. Thus, every iteration of the down loop preserves $\Phi \leq 0$. □

**Lemma 2.3.18.** *Lines 18 to 19 preserve* $\Phi \leq 0$.

*Proof.* Let the request appended to $\sigma$ in line 18 be $(u, f)$. The guard of the down loop ensures that $f$ is in $\pi.missing$. Algorithm OFF incurs cost at most

$\pi.diam$ to process such a request because it stores all the files in $F(\pi.depth)$ in a child of $\pi$, and $\pi.missing \subseteq F(\pi.depth)$.

Since ON is nice, it processes a request $(u, f)$ in two phases as follows: in the first phase, ON adds a copy $(u, f)$ to its placement; in the second phase, ON performs an arbitrary sequence of add and delete operations. If $\pi$ is equal to *root*, then ON incurs expected cost at least $\frac{\lambda}{2} \cdot \pi.diam$ in the first phase since the miss penalty associated with any file is $\lambda \cdot root.diam$. If $\pi$ is not equal to *root*, then ON incurs expected cost at least $\frac{1}{2} \cdot \pi.parent.diam = \frac{\lambda}{2} \cdot \pi.diam$ in the first phase. Thus, in either case, ON incurs expected cost at least $\frac{\lambda}{2} \cdot \pi.diam$ in the first phase. Let $X$ be the set of nodes on the path from $\pi$ to $u$, excluding $\pi$. Note that $\alpha.load$, for $\alpha \in X$, may increase by 1 during the first phase.

The change in $\Phi$ due to the first phase of ON and due to OFF in processing a request is at most

$$\begin{aligned}
& \nu \cdot \pi.diam - \frac{\nu' \cdot \lambda \cdot \pi.diam}{2} + \sum_{\alpha \in X} \alpha.parent.diam \\
\leq\ & \pi.diam \cdot \left(\nu - \frac{\nu' \cdot \lambda}{2}\right) + \pi.diam \cdot \sum_{j \geq 0} \lambda^{-j} \\
\leq\ & \pi.diam \cdot \left(\nu - \frac{\nu' \cdot \lambda}{2} + \frac{\lambda}{\lambda - 1}\right) \\
=\ & \pi.diam \cdot \left(\nu - \frac{\lambda^2 - 2\lambda}{2(\lambda - 1)}\right) \\
\leq\ & \pi.diam \cdot \frac{\lambda}{16} \cdot \left(\frac{15 - 7\lambda}{\lambda - 1}\right) \\
\leq\ & 0,
\end{aligned}$$

(In the above derivation, the second last inequality follows from $\nu \leq \frac{\lambda}{16}$ and the last inequality follows from $\lambda \geq \frac{15}{7}$.)

For analyzing the second phase of ON in processing a request, it is convenient to view the randomized online algorithm ON as a probability distribution over a collection of deterministic online algorithms. For each such deterministic algorithm $A$, we define an associated potential function $\Phi_A$ as in Equation 2.1, but with $T_{\text{ON}}(\sigma)$ replaced by the cost incurred by $A$ on $\sigma$, denoted $T_A(\sigma)$, and each term $\alpha.load$ appearing in the second summation replaced by $|(\cup_{0 \leq i < \alpha.depth} F(i)) \cap \alpha.placed_A|$, where $\alpha.placed_A$ denotes the set of distinct files placed by $A$ in $\alpha.caches$ after processing the request sequence $\sigma$. We denote $|(\cup_{0 \leq i < \alpha.depth} F(i)) \cap \alpha.placed_A|$ by $\alpha.load_A$. Note that $T_{\text{ON}}(\sigma)$ is the expected value of $T_A(\sigma)$ when $A$ is chosen at random from the probability distribution associated with ON. Similarly, for any node $\alpha$, $\alpha.load$ is the expected value of $\alpha.load_A$ and $\Phi$ is the expected value of $\Phi_A$. Thus it is sufficient to prove that for any $A$, each individual operation (i.e., each addition or deletion of a file) performed by $A$ during the second phase does not increase $\Phi_A$. For deletions, this claim is immediate since all terms in $\Phi_A$ are unchanged except that terms of the form $\alpha.load_A$ may decrease by one. When a file is added, the set of nodes with an increased $load_A$ value form a path $P$ from some node, say $\alpha$, to a leaf, and $A$ incurs a cost of $\alpha.parent.diam$. Let the set of nodes on path $P$ be $Y$. (Note that *root* does not belong to $Y$ since *root.load* is always zero.) Since the diameters of the nodes of $P$ are $\lambda$-separated, the change in

$\Phi_A$ is at most

$$\begin{aligned} & -\nu' \cdot \alpha.parent.diam + \sum_{\beta \in Y} \beta.parent.diam \\ \leq \quad & -\nu' \cdot \alpha.parent.diam + \alpha.parent.diam \cdot \sum_{j \geq 0} \lambda^{-j} \\ = \quad & -\nu' \cdot \alpha.parent.diam + \frac{\lambda}{\lambda - 1} \cdot \alpha.parent.diam \\ = \quad & 0. \end{aligned}$$

The claim of the lemma then follows. □

**Lemma 2.3.19.** ON *is* $\Omega\left(\frac{\nu}{\nu'}\right)$*-competitive.*

*Proof.* Initially, $\Phi = 0$. By Lemmas 2.3.16, 2.3.17, and 2.3.18, $\Phi \leq 0$ is a loop invariant of the main loop. Therefore, by Lemmas 2.3.5 and 2.3.8, $T_{\text{ON}}(\sigma) \geq \frac{\nu}{\nu'} \cdot T'_{\text{OFF}}(\sigma)$ holds initially and is a loop invariant of the main loop. Let $C$ be the cost incurred by OFF in moving from the empty placement to the first placement. Note that ON serves every request with a cost at least 1 (because the diameter of an internal node is at least 1). Hence, $T_{\text{ON}}(\sigma)$ tends to $\infty$ as $N$ (the length of the request sequence $\sigma$) tends to $\infty$. Therefore, we can ensure that $\frac{T_{\text{ON}}(\sigma)}{T'_{\text{OFF}}(\sigma)+C} = \Omega\left(\frac{\nu}{\nu'}\right)$ by choosing $N$ sufficiently large. □

**Theorem 1.** ON *is* $\Omega\left(\log \frac{d}{b}\right)$*-competitive.*

*Proof.* Recall that $\lambda$ is $\Omega(\log k)$. Hence, $\nu = \Theta(\log k)$ and $\nu' = \Theta(1)$. From Lemma 2.3.19, it implies that ON is $\Omega(\log k)$-competitive. The theorem follows since $d = 8bk - 1$, that is, $k = \Theta(d/b)$. $\square$

It is also possible to express the preceding lower bound in terms of the number of caches $n$ and the capacity blowup $b$. Since, $n = k^d$ and $d = 8bk - 1$, we have $n = k^{8bk-1}$. Solving these equations for $k$ (e.g., using bootstrapping), we find that $k = \Theta(\frac{\log n}{b(\log\log n - \log b)})$ and hence,

$$\begin{aligned}\log k &= \Theta(\log\log n - \log b - \log(\log\log n - \log b)) \\ &= \Theta(\log\log n - \log b).\end{aligned}$$

It follows that ON is $\Omega(\log\log n - \log b)$-competitive.

## 2.4 An Upper Bound

We show in this section that, given $O(d)$ capacity blowup, where $d$ is the depth of the cache hierarchy, a simple LRU-like algorithm, which we refer to as *Hierarchical LRU* (HLRU), is constant-competitive with respect to an optimal offline algorithm OPT. For the sake of simplicity, we assume that every file has unit size and uniform miss penalty. Our result can easily be extended to handle variable file sizes and nonuniform miss penalties using an approach similar to Landlord [46].

### 2.4.1 The HLRU Algorithm

In this section we present a $2(d+1)$-quasifeasible HLRU algorithm that is constant-competitive with respect to OPT. HLRU divides every cache into $d+1$ equal-sized segments numbered from 0 to $d$. (For generalizing our results to variable sized files, the segments should be contiguous. For the case of unit sized files considered here, the segments need not be contiguous.) For a node $\alpha$, we define $\alpha.small$ to be the union of segment $\alpha.depth$ of all the caches in $\alpha.caches$, and we define $\alpha.big$ to be the union of $\beta.small$ for all $\beta \in \alpha.desc$.

For the rest of this section, we extend the definitions of a *copy* and a *placement* (defined in Section 2.2) to internal nodes as well. A copy is a pair $(\alpha, f)$ where $\alpha$ is a node and $f$ is a file that is stored in $\alpha.small$. A placement refers to a set of copies. The HLRU algorithm, shown in Figure 2.2, maintains a placement $P$. Note that when a copy $(\alpha, f)$ is added to $P$ in line 4, file $f$ is added to $\alpha.small$. In HLRU, a node $\alpha$ uses a variable $\alpha.ts[f]$ to keep track of the timestamp of a file $f$. For the convenience of presentation, we define $root.parent$ to be a fake node that has every file in $root.parent.small$ (and hence also in $root.parent.big$), and we define $root.parent.diam$ to be the uniform miss penalty.

### 2.4.2 Analysis of the HLRU Algorithm

For any node $\alpha$ and file $f$, we partition time into *epochs* with respect to $\alpha$ and $f$ as follows. The first epoch begins at the start of the execution, which is defined to be time 1. Subsequent epochs begin just after the execution of line 11.

We define $\alpha.ts^*[f]$ to be the time of the most recent access to file $f$ in a cache in $\alpha.caches$ in the current epoch with respect to node $\alpha$ and file $f$. If no such access exists, we define $\alpha.ts^*[f]$ to be 0.

```
    {On a request $(\alpha,\ f)$}
 1  $t$ := **now**;
 2  **do**
 3    *flag* := **false**;
 4    $P := P \cup \{(\alpha, f)\}$;
 5    $\alpha.ts[f] := \max(\alpha.ts[f], t)$;
 6    **if** capacity is violated at $\alpha.small$ **then**
 7      $f$ := file with smallest nonzero $\alpha.ts[f]$;
 8      $P := P \setminus \{(\alpha, f)\}$;
 9      **if** $f \notin \alpha.big$ **then**
10        $t := \alpha.ts[f]$;
11        $\alpha.ts[f] := 0$;
12        $\alpha := \alpha.parent$;
13        *flag* := **true**
14      **fi**
15    **fi**
16  **while** *flag*
```

Figure 2.2: The HLRU algorithm.

For the purpose of our analysis, we categorize the file movements in HLRU into two types: *retrievals* and *evictions*. On a request $(u,\ f)$, the HLRU algorithm first performs a retrieval (this corresponds to the block of code from the beginning of the code to line 5 of the first iteration of the loop) of $f$ from the nearest cache $v$ that has a copy. Let $\alpha$ be the least common ancestor of $u$ and $v$. Then the cost of such a retrieval is $\alpha.diam$. Let $X$ denote the set of nodes on the path from $\alpha$ to $u$, excluding $\alpha$ but including $u$. For every node $\beta$ in $X$, we charge a *pseudocost* of $\beta.parent.diam$ to node $\alpha$ for such a retrieval.

Each subsequent iteration of the loop performs an eviction (this corresponds to the block of code from line 6 of an iteration to line 5 of the next iteration) of a file from $\alpha.small$ to $\alpha.parent.small$ for some node $\alpha$. We charge a pseudocost of $\alpha.parent.diam$ to $\alpha$ for such an eviction.

The only cost incurred by OPT is due to retrievals. Let OPT add (or retrieve) a copy $(u, f)$ by fetching $f$ from $v$, let $\alpha$ be the least common ancestor of $u$ and $v$, and let $X$ be the set of nodes on the path from from $\alpha$ to $u$, excluding $\alpha$ but including $u$. Then the cost of such a retrieval is $\alpha.diam$. For every node $\beta$ in $X$, we charge a pseudocost of $\beta.parent.diam$ to node $\alpha$ for such a retrieval by OPT.

For any node $\alpha$ and file $f$, we define auxiliary variables $\alpha.in[f]$ and $\alpha.out[f]$ for the purpose of our analysis. These variables are initialized to 0. We increment $\alpha.in[f]$ whenever a retrieval of file $f$ charges a pseudocost to node $\alpha$. We increment $\alpha.out[f]$ whenever eviction of file $f$ charges a pseudocost to node $\alpha$.

**Lemma 2.4.1.** *Before and after every retrieval or eviction, for any node $\alpha$ and file $f$, $f \in \alpha.big$ iff $\beta.ts[f] > 0$ for some $\beta \in \alpha.desc$.*

*Proof.* Initially, both sides of the equivalence are false. If both sides of the equivalence are false, then according to the code in Figure 2.2, the only event that sets either side to true is a retrieval of $f$ at a cache $u$ in $\alpha.caches$, which in fact sets both sides to true. It remains to prove that if both sides of the equivalence are true, and if one side becomes false, then the other side becomes false.

The only event that falsifies the left side is an eviction of the last copy of $f$ in $\alpha.big$ from $\alpha.small$. Prior to this eviction, $\beta.ts[f] = 0$ for all proper descendants $\beta$ of $\alpha$ (note that the equivalence holds for $\beta$) and $\alpha.ts[f] > 0$. The eviction then sets $\alpha.ts[f]$ to 0, falsifying the right side.

The only event that can falsify the right side (i.e., line 11) is an eviction of $f$ from $\alpha.small$ such that, after the eviction, $f \notin \alpha.big$. Note that eviction of $f$ from $\beta.small$, for a proper descendant $\beta$ of $\alpha$, cannot falsify the right side because such an eviction ensures $\beta.parent.ts[f] > 0$ (line 5). Thus, falsification of the right side implies falsification of the left side. □

**Lemma 2.4.2.** *Before and after every retrieval or eviction, for any node $\alpha$ and file $f$,*

$$\alpha.ts^*[f] = \max_{\beta \in \alpha.desc} \beta.ts[f].$$

*Proof.* Initially, both sides of the equality are zero. By the definition of $\alpha.ts^*[f]$, the value of $\alpha.ts^*[f]$ changes from nonzero to 0 (i.e., a new epoch with respect to $\alpha$ and $f$ begins) after line 11. By the guard of the inner **if** statement, $f \notin \alpha.big$ just before line 11. Hence, by Lemma 2.4.1, $\beta.ts[f]$ is 0 for all $\beta \in \alpha.desc$.

The value $\alpha.ts^*[f]$ increases due to some access of $f$ at a cache $u$ in $\alpha.caches$. The equality holds because the max value on the right side is at $u$.

Between the changes of $\alpha.ts^*[f]$, only the eviction of $f$ from $\alpha.big$ can change the max (reset it to 0) on the right side of the equality. This eviction also resets $\alpha.ts^*[f]$ to 0 because a new epoch begins. $\square$

**Lemma 2.4.3.** *Before and after every retrieval or eviction, for any node $\alpha$ and file $f$, $\alpha.ts[f] \leq \alpha.ts^*[f]$. Furthermore, just after line 8, if $f \notin \alpha.big$, then $\alpha.ts[f] = \alpha.ts^*[f]$.*

*Proof.* The first claim of the lemma follows immediately from Lemma 2.4.2. For the second claim, note that we are evicting the last copy of $f$ in $\alpha.big$ from $\alpha.small$. By Lemma 2.4.1, all proper descendants $\beta$ of $\alpha$ have $\beta.ts[f] = 0$. So $\alpha.ts[f] = \alpha.ts^*[f]$ by Lemma 2.4.2. $\square$

**Lemma 2.4.4.** *If a file movement (between two caches) has actual cost $C$ and charges a total pseudocost of $C'$, then*

$$C \leq C' \leq \frac{\lambda}{\lambda - 1} C.$$

*Proof.* Suppose the file movement is from cache $u$ to cache $v$. Let $\alpha$ be the least common ancestor of $u$ and $v$ and let $B$ be the nodes on the path from $\alpha$

to $v$, excluding $\alpha$ but including $u$. Then

$$\begin{aligned} & C \\ = \;& \alpha.diam \\ \leq \;& \sum_{\beta \in B} \beta.parent.diam \\ = \;& C' \\ \leq \;& \alpha.diam \cdot \sum_{j \geq 0} \lambda^{-j} \\ = \;& \frac{\lambda}{\lambda - 1} \cdot C. \end{aligned}$$

□

**Lemma 2.4.5.** *For any node $\alpha$, the total pseudocost charged to node $\alpha$ due to retrievals is*

$$\sum_{f} \alpha.in[f] \cdot \alpha.parent.diam.$$

*Proof.* Follows from the observation that whenever a pseudocost is charged to node $\alpha$ due to a retrieval, the pseudocost is $\alpha.parent.diam$. □

**Lemma 2.4.6.** *For any node $\alpha$, the total pseudocost charged to node $\alpha$ due to evictions is at most*

$$\sum_{f} \alpha.out[f] \cdot \alpha.parent.diam.$$

*Proof.* Follows from the observation that whenever a pseudocost is charged to node $\alpha$ due to an eviction, the pseudocost is at most $\alpha.parent.diam$. □

**Lemma 2.4.7.** *For any node $\alpha$ and file $f$,*

$$\alpha.out[f] \leq \alpha.in[f].$$

*Proof.* We observe that if a pseudocost is charged to a node $\alpha$ as a result of a retrieval, then $f \notin \alpha.big$ before the retrieval and $f \in \alpha.big$ after the retrieval. Similarly, if a pseudocost is charged to node $\alpha$ as a result of an eviction, then the eviction falsifies $f \in \alpha.big$. It then follows that

$$\alpha.out[f] \leq \alpha.in[f] \leq \alpha.out[f] + 1$$

because $f \notin \alpha.big$ initially. □

**Lemma 2.4.8.** *For any node $\alpha$, $\alpha.big$ always contains the most recently accessed $2 \cdot \alpha.cap$ files by $\alpha.caches$.*

*Proof.* Let $X$ denote the set of the most recently accessed $2 \cdot \alpha.cap$ files. We consider the places where a file is added to $X$ or removed from $\alpha.big$.

A file $f$ can be added to $X$ only when $f$ is requested at a cache $u$ in $\alpha.caches$. In this case, $f$ is added to $u.small$ and is not evicted from $u.small$ because it is the most recently accessed item. Hence, $f \in \alpha.big$.

A file $f$ can be removed from $\alpha.big$ only when it is moved from $\alpha.small$ to $\alpha.parent.small$ as the result of an eviction and there is no other copy of $f$ in $\alpha.big$. This means that $f$ is chosen as the LRU item at line 7. Since $f$ is the LRU item, there are $2 \cdot \alpha.cap$ items $f'$ in $\alpha.small$ such that $\alpha.ts[f] <$

$\alpha.ts[f'] \leq \alpha.ts^*[f']$. By Lemma 2.4.3, $\alpha.ts[f] = \alpha.ts^*[f]$ just after line 8. It follows then from the definition of $ts^*$ that $f \notin X$. □

**Lemma 2.4.9.** *For any node $\alpha$, the total pseudocost due to retrievals charged to $\alpha$ by HLRU is at most twice the pseudocost charged to $\alpha$ by OPT.*

*Proof.* Fix a node $\alpha$. For OPT, we say that a request for a file $f$ at a cache in $\alpha.caches$ results in a miss if no copy of $f$ exists at any cache in $\alpha.caches$ at the time of the request. For HLRU, a miss occurs if no copy of $f$ is in $\alpha.big$. By Lemma 2.4.8, HLRU incurs at most as many misses as an LRU algorithm with capacity $2 \cdot \alpha.cap$ running on the subsequence of requests originating from the caches in $\alpha.caches$. (Note that LRU misses whenever HLRU misses.) By the well-known result of Sleator and Tarjan [39], such an LRU algorithm incurs at most twice as many misses as OPT.

Note that a miss results in a pseudocost of $\alpha.parent.diam$ being charged to $\alpha$. Therefore, the total pseudocost charged to node $\alpha$ in OPT is at least the number of misses in OPT times $\alpha.parent.diam$. Furthermore, within HLRU, a pseudocost is charged to node $\alpha$ only on a miss. Therefore, the total pseudocost charged to node $\alpha$ in HLRU is at most the number of misses incurred by HLRU times $\alpha.parent.diam$. The claim of the lemma then follows. □

**Lemma 2.4.10.** *For any node $\alpha$, the total pseudocost charged to $\alpha$ by HLRU is at most four times the total pseudocost charged to $\alpha$ by OPT.*

*Proof.* Follows immediately from Lemmas 2.4.5, 2.4.6, 2.4.7, and 2.4.9. □

**Theorem 2.** *HLRU is constant-competitive.*

*Proof.* Follows immediately from Lemmas 2.4.4 and 2.4.10. □

# Chapter 3

# Compression Caching

## 3.1 Introduction

Recently we have seen an explosion in the amount of data distributed over handheld devices, personal computers, and local and wide area networks. There is a growing need for self-tuning data management techniques that can operate under a wide range of conditions, and optimize various resources such as storage space, processing, and network bandwidth. There is a large body of work addressing different aspects of this domain of self-tuning data management.

An important aspect of this domain that merits further attention is that data can be stored in different formats. For example, one can compress a text file using different traditional compression techniques such as gzip and bzip. Various studies [2, 4, 28] have experimentally demonstrated the advantages of compression in caching. A compressed file takes up less space, effectively increasing the size of the memory. However, this increase in size comes at the cost of extra processing needed for compression and uncompression. Consequently, it may be desirable to keep frequently accessed files uncompressed in the memory.

As another example, consider the option of storing only a TEX file or the corresponding pdf file along with the TEX file. One can save space by storing only the TEX file, but then one has to run a utility (such as pdflatex) to generate the pdf file when needed. On the other hand, storing the pdf file may require an order of magnitude more storage space than the TEX file, but then the pdf file is readily accessible when needed. In general, many files are automatically generated using some utility such as a compiler or other translator. If the utility generates a large output compared to the input, then by storing only the input one achieves a form of "compression", not in the traditional sense, but with analogous consequences. In this chapter, when we refer to compression, we have in mind this broader notion of compression where one can have a wide separation between storage space and processing costs associated with different formats of a file.

By extending this notion further, one can create further room for optimization. Consider a distributed storage system to hold a collection of files, where each file has an associated "makefile" that specifies how to generate that file. Given requests for files at different locations, it is desirable to have self-tuning techniques that decide which files to keep explicitly at which locations, and which files to generate dynamically. The problem of designing such a storage system is quite challenging due to various aspects such as the different sizes and processing costs associated with the different formats of a file, dependencies among different files, and files distributed over the network.

In this work, we initiate a study towards modeling and designing such a storage system. We address the notion of compression and uncompression of files, while contemplating the possibility of a richer variety of separation between the sizes and processing costs associated with the different formats of a file. We focus primarily on the single machine setting, however one of our upper bound results (see Section 3.3.3) is applicable to a simple, but well-motivated, special case of the distributed storage problem.

**Problem Formulation**. We define a class of compression caching problems in which a file can be cached in multiple compressed formats with varying sizes, and costs for compression and uncompression (see Section 3.2 for a formal description). We are given a cache with a specified capacity. Also assume that for each file, there are multiple associated formats. Each format is specified by three parameters: encode cost, decode cost, and size. The encode cost of a particular format is defined as the cost of creating that format from the uncompressed format of the file. The decode cost of a format is defined as the cost of creating the uncompressed format. The miss penalty of a file is defined as the cost of accessing the file if no format of the file is present in the cache. To execute a request for a file, the file is required to be loaded into the cache in the uncompressed format. The goal of a compression caching algorithm is to minimize the total cost of executing a given request sequence.

The main challenge is to design algorithms that determine — in an online manner — which files to keep in the fast memory, and of these, which to keep in compressed form. The problem is further complicated by the multiple compression formats for a file, with varying sizes and encode/decode costs.

Since compression caching has the potential to be useful in many different scenarios, a desirable property of an online algorithm is to provide a good competitive ratio, which is defined as the maximum ratio of the cost of the online algorithm and that of the offline algorithm over any request sequence (see [13] for more details).

In a seminal work, Sleator and Tarjan [39] show that the competitive ratio of any deterministic online paging algorithm without any capacity blowup is the size of the cache, and they also show that LRU is resource competitive for the disk paging problem. Since compression caching generalizes the disk paging problem, it is natural to ask whether similar resource competitive results can be obtained for compression caching.

**Contributions**. In this chapter, we address three problems in the class of the compression caching. Our contributions for each of these three problems are as follows.

- The first problem assumes that the encode cost and decode cost associated with any format of a file are equal. For this problem we generalize the Landlord algorithm [46] to obtain an online algorithm that is resource competitive. We find that this problem also corresponds to a special case of the distributed storage problem, and hence, our algorithm is applicable to this special case. (See Section 3.3.3 for further details.)
- The second problem assumes that the decode costs associated with different formats of a file are the same. For this problem, we show that any

deterministic online algorithm (even with an arbitrary factor capacity blowup) is $\Omega(m)$-competitive, where $m$ is the number of possible formats of a file. The proof of this lower bound result is the most technically challenging part of the chapter. Further, we give an online algorithm for this problem that is $O(m)$-competitive with $O(m)$ factor capacity blowup. Thus, we tightly characterize the competitive ratio achievable for this problem.

- The third problem assumes that the encode costs associated with different formats of a file are the same. For this problem we show that any deterministic online algorithm (even with an arbitrary factor capacity blowup) has competitive ratio $\Omega(\log m)$. We also present an online algorithm for this problem that is $O(m)$-competitive with $O(m)$ factor capacity blowup.

**Related Work**. The competitive analysis framework was pioneered by Sleator and Tarjan [39]. For the disk paging problem, it has been shown that LRU is $\frac{k}{k-h+1}$-competitive, where $k$ is the cache capacity of LRU and $h$ is the cache capacity of any offline algorithm [39]. In the same paper, it has been shown that $\frac{k}{k-h+1}$ is the best possible competitive ratio for any deterministic online paging algorithm. For the variable size file caching problem, which is useful in the context of web-caching, Young [46] proposes the Landlord algorithm, and shows that Landlord is $\frac{k}{k-h+1}$-competitive. For the variable size file caching problem, Cao and Irani [15] independently propose the greedy-dual-size algorithm and show that it is $k$-competitive against any offline algorithm, where $k$ is cache capacity of both greedy-dual-size and the offline algorithm. For the distributed paging problem, Awerbuch et al. [6] give an algorithm that is polylog$(n, \Delta)$-competitive with polylog$(n, \Delta)$ factor capacity blowup, where $n$ is the number of nodes and $\Delta$ is the diameter of the network.

Various studies [2, 4, 28] have shown experimentally that compression effectively increases on-chip and off-chip chip cache capacity, as well as off-chip bandwidth, since the compressed data is smaller in size. Further, these studies show that compression in caching increases the overall performance of the system.

**Outline**. The rest of this chapter is organized as follows. In Section 3.2 we provide some definitions. In Section 3.3 we present our results for the compression caching problem with equal encode and decode costs. In Section 3.4 we describe our results for the compression caching problem with varying encode costs and uniform decode costs. In Section 3.5 we discuss our results for the compression caching problem with uniform encode costs and varying decode costs.

## 3.2 Preliminaries

Assume that we are given a cache with a specified capacity and $m$ different functions for encoding and decoding any file, denoted $h_i$ and $h_i^{-1}$, where $0 \leq i < m$. Without loss of generality, we assume that $h_0$ and $h_0^{-1}$ are the identity functions. We define index $i$ as an integer $i$ such that $0 \leq i < m$. For any index $i$, we obtain the $i$-encoding of any file $f$ by evaluating $h_i(f)$, and we obtain the file $f$ from the $i$-encoding $\mu$ of $f$ by evaluating $h_i^{-1}(\mu)$. For any file $f$, we refer to the 0-encoding of $f$ as the *trivial* encoding $f$, and for $i > 0$, we refer to the $i$-encoding of $f$ as a *nontrivial* encoding of $f$. For any file $f$ and index $i$, the $i$-encoding of $f$ is also referred to as an encoding of $f$, and we say $f$ is present in the cache if any encoding of $f$ is present in the cache. For any file $f$ and index $i$, the $i$-encoding of $f$ is characterized by three parameters: encode cost, denoted $\mathit{encode}(i, f)$; decode cost, denoted $\mathit{decode}(i, f)$; and size, denoted $\mathit{size}(i, f)$. The encode cost $\mathit{encode}(i, f)$ is defined as the cost of evaluating $h_i(f)$, and the decode cost $\mathit{decode}(i, f)$ is defined as the cost of evaluating $h_i^{-1}(\mu)$, where $\mu$ is the $i$-encoding of $f$. Note that for any file $f$, $\mathit{encode}(0, f)$ and $\mathit{decode}(0, f)$ are 0.

For any file $f$, the *access cost* of $f$ is defined as follows: if for some index $i$, the $i$-encoding of $f$ is present in the cache (break ties by picking minimum such $i$), then the access cost is $\mathit{decode}(i, f)$; if none of the encodings of $f$ is present in the cache, then the access cost is defined as the miss penalty $p(f)$. Without loss of generality, we assume that the miss penalty for any file $f$ is at least the decode cost of any of the encodings of $f$. The cost of deleting any encoding of any file from the cache is 0. For any file $f$ and index $i$, the $i$-encoding of $f$ can be added to the cache if there is enough free space to store the $i$-encoding of $f$. For any file $f$ and index $i$, the cost of adding the $i$-encoding of $f$ to the cache is the sum of the access cost of $f$ and $\mathit{encode}(i, f)$.

To execute a request for a file $f$, an algorithm $A$ is allowed to modify its cache content by adding/deleting encodings of files, and then incurs the access cost for $f$. The goal of the compression caching problem is to minimize the total cost of executing a given request sequence. An online compression caching algorithm $A$ is $c$-competitive if for all request sequences $\tau$ and compression caching algorithms $B$, the cost of executing $\tau$ by $A$ is at most $c$ times that of executing $\tau$ by $B$.

Any instance $I$ of the compression caching problem is represented by a triple $(\sigma, m, k)$, where $\sigma$ is the sequence of request for the instance $I$, $m$ is the number of possible encodings for files in $\sigma$, and $k$ is the cache capacity. For any instance $I = (\sigma, m, k)$, we define $\mathit{reqseq}(I) = \sigma$, $\mathit{numindex}(I) = m$, and $\mathit{space}(I) = k$.

We define a *configuration* as a set of encodings of files. For any configuration $S$, we define the size of $S$ as the sum, over all encodings $\mu$ in $S$, of size of $\mu$. We define a *trace* as a sequence of pairs, where the first element of the pair is a request for a file and the second element of the pair is a configuration. For any configuration $S$ and any integer $k$, $S$ is *$k$-feasible* if the size of $S$ is at most $k$. For any trace $T$ and integer $k$, $T$ is $k$-feasible if and only if any configuration in $T$ is $k$-feasible. For any two sequences $\tau$ and $\tau'$, we define $\tau \circ \tau'$ as the sequence obtained by appending $\tau'$ to $\tau$. For any trace $T$, we define $\mathit{requests}(T)$ as the sequence of requests present in $T$, in the same order as in $T$.

For any file $f$, any trace $T$, and any configuration $S$, we define $\mathit{cost}_f(T, S)$ inductively as follows. If $T$ is empty, then $\mathit{cost}_f(T, S)$ is zero. If $T$ is equal to $\langle (f', S') \rangle \circ T'$, then $\mathit{cost}_f(T, S)$ is $\mathit{cost}_f(T', S')$ plus the sum, over all $i$-encodings

$\mu$ of $f$ such that $\mu$ is present in $S'$ and $\mu$ is not present in $S$, of $\mathit{encode}(i, f)$, plus the access cost of $f$ in $S$ if $f = f'$. For any file $f$ and any trace $T$, we define $\mathit{cost}_f(T)$ as $\mathit{cost}_f(T, \emptyset)$. For any trace $T$ and any configuration $S$, we define $\mathit{cost}(T, S)$ as the sum, over all files $f$, of $\mathit{cost}_f(T, S)$. For any trace $T$, we define $\mathit{cost}(T)$ as $\mathit{cost}(T, \emptyset)$.

## 3.3 Equal Encode and Decode Costs

In this section, we present a symmetric instance of the compression caching problem which assumes that the encode cost and decode cost associated with any encoding of a file are equal. We present an online algorithm for this problem, and show that the algorithm is resource competitive. Interestingly, this problem also corresponds to a multilevel storage scenario, and so, our algorithm applies to this scenario (see Section 3.3.3).

In this problem, we consider that for any file $f$ and index $i$, $\mathit{encode}(i, f) = \mathit{decode}(i, f)$. At the expense of a small constant factor in the competitive ratio, we can assume that, for any file $f$, the miss penalty $p(f)$ is at least $q \cdot \mathit{encode}(m - 1, f)$, where $q > 1$; and by preprocessing, we can arrange encodings of files in geometrically decreasing sizes and geometrically increasing encode-decode costs. The basic idea behind the preprocessing phase is as follows. First, consider any two encodings with similar sizes (resp., similar encode-decode costs) within a constant factor. Second, from these two encodings, select the one with smaller encode-decode cost (resp., smaller size), and eliminate the other. While an encoding can be eliminated by one of the above preprocessing steps, we do so. After the above preprocessing phase, we can arrange the encodings of files in geometrically decreasing sizes and geometrically increasing encode-decode costs.

For ease of presentation, we assume that $m$ encodings are selected for each file in the preprocessing phase. More precisely, after the preprocessing phase, for any file $f$ and index $i < m-1$, we have $\mathit{size}(i+1, f) \leq \frac{1}{r} \cdot \mathit{size}(i, f)$ and $\mathit{encode}(i+1, f) \geq q \cdot \mathit{encode}(i, f)$, where $r > 1$. Also, we assume that the capacity of the cache given to an online algorithm is $b$ times that given to an offline algorithm.

### 3.3.1 Algorithm

In Figure 3.1, we present our online algorithm ON. At a high level, ON is a credit-rental algorithm. Algorithm ON maintains a "containment" property on the encodings in the cache, defined as follows: If ON has the $i$-encoding of some file $f$ in the cache, then ON also has all the $j$-encodings of $f$ for any index $j \geq i$ in the cache. A credit is associated with each encoding present in the cache. For any file $f$ and index $i$, the $i$-encoding of $f$ is created with an initial credit $\mathit{decode}(i+1, f)$, for $i < m-1$, and credit $p(f)$, for $i = m-1$. On a request for a file $f$, if the 0-encoding of $f$ is not present in the cache, then ON creates space for the 0-encoding of $f$, and for other $i$-encoding of $f$ that are necessary to maintain the containment property. Then, ON creates the 0-encoding of $f$, and any other $i$-encodings of $f$ that are necessary to maintain the containment property, with an initial credit as described above.

```
{Initially, for any encoding $\mu$ of any file, $credit(\mu) = 0$}
On a request for a file $f$
if $f$ is not present in the cache then
    $createspace(f, m-1)$
    for all indices $i$, add the $i$-encoding $\mu$ of $f$, with $credit(\mu) := decode(i+1, f)$, if $i < m-1$,
        and with $credit(\mu) := p(f)$, if $i = m-1$
else if the $i$-encoding $\mu$ of $f$ is present in the cache (break ties by picking minimum such $i$) then
    evaluate $h_i^{-1}(\mu)$
    $credit(\mu) := decode(i+1, f)$
    if $i > 0$ then
        $createspace(f, i-1)$
        for all indices $j < i$, add the $j$-encoding $\nu$ of $f$, with $credit(\nu) := decode(j+1, f)$
    fi
fi

$createspace(f, i)$
    $sz := \sum_{j=0}^{i} size(f, j)$
    while free space in the cache $< sz$ do
        $\delta := \min_{\mu \in X} \frac{credit(\mu)}{size(j, f')}$, where $\mu$ is the $j$-encoding of $f'$
        for each file $f'$ such that there is an encoding of $f'$ in the cache do
            let $\mu$ be the largest (in size) encoding of $f'$ in the cache
            let $j$ be the index of $\mu$
            $credit(\mu) := credit(\mu) - \delta \cdot size(j, f')$
            if $credit(\mu) = 0$ then
                delete $\mu$
            fi
        od
    od
```

Figure 3.1: The online algorithm ON for any symmetric instance of the compression caching problem. Here, $X$ is the cache content of ON.

In order to create space, for each file present in the cache, ON charges rent from the credit of the largest encoding of the file, where rent charged is proportional to the size of the encoding, and deletes any encoding with 0 credit. The credit-rental algorithm described here can be viewed as a generalization of Young's Landlord algorithm [46].

### 3.3.2 Analysis

We use a potential function argument similar to that of Young to show that ON is resource competitive.

For any $i$-encoding $\mu$ and any $j$-encoding $\nu$ of the same file $f$, we say that $\mu \leq \nu$ if $i \leq j$. Let $X$ be the cache content of ON and $Y$ be that of the offline algorithm. We define $Y^*$ as $\cup_{\mu \in Y}\{\nu \mid \mu \leq \nu\}$. For any $i$-encoding $\mu$ of any file $f$, we define $\mathit{cost}(\mu)$ as $\mathit{decode}(i+1, f)$ if $i < m-1$, and as $p(f)$, if $i = m-1$. We define a potential function $\Phi$ as follows, where $\beta$ is defined as $\frac{r-1}{r} \cdot b$.

$$\Phi = \sum_{\mu \in X} \mathit{credit}(\mu) + \beta \sum_{\nu \in Y^*} (\mathit{cost}(\nu) - \mathit{credit}(\nu))$$

Consider the following model of executing a given request. To execute a request for a file $f$, an algorithm performs an arbitrary sequence of addition and deletion of encodings, subject only to the following two constraints: (1) the 0-encoding of $f$ is present in the cache immediately after executing the request, and (2) for any file $f'$, the $i$-encoding of $f'$ can be added to the cache only if the 0-encoding of $f'$ is also present in the cache.

By using a scratch buffer space (of capacity at most the size of the 0-encoding of any file), it is not hard to see that any offline algorithm can be simulated by another offline algorithm OFF that incurs the same costs as the original offline algorithm, and adheres to the above model. Hence, in the following, without loss of generality we restrict attention to algorithms that adhere to the above model. Note that ON is such an algorithm.

To analyze the performance of ON, we execute ON alongside OFF. As in the case of Young's analysis of Landlord in [46], we execute each successive request with OFF, and then with ON. Then, we observe the effect of each action on the potential function $\Phi$. The actions taken by OFF to execute a request for $f$ can be broken down into a sequence of steps, with each step being one of the following: OFF removes an encoding; OFF adds the 0-encoding of $f$; OFF adds a nontrivial encoding of $f$. Actions taken by ON to execute a request for $f$ can be broken down into a sequence of steps, with each step being one of the following: ON charges rent; ON removes an encoding; ON adds a set of encodings of $f$ (note that as long as this set is not empty, the 0-encoding of $f$ is in the set).

**Lemma 3.3.1.** *The potential $\Phi$ is nonnegative.*

*Proof.* Both of the terms in the two summations in $\Phi$ are nonnegative. □

**Lemma 3.3.2.** *If* OFF *removes an encoding, $\Phi$ does not increase.*

*Proof.* In this case, a nonnegative term from the second summation in $\Phi$ disappears. Hence, $\Phi$ does not increase. □

**Lemma 3.3.3.** *If* OFF *adds the 0-encoding of a file $f$ by decoding from the $i$-encoding $\mu$ of $f$, then $\Phi$ increases by at most $\beta \cdot \sum_{\nu:\nu<\mu} cost(\nu)$.*

*Proof.* The first summation does not change. The second summation increases by at most $\beta \cdot \sum_{\nu:\nu<\mu} cost(\nu)$. □

**Lemma 3.3.4.** *If* OFF *adds the 0-encoding $\mu$ of a file $f$ by retrieving $f$ remotely, then $\Phi$ increases by at most $\beta \cdot \sum_{\nu:\mu\leq\nu} cost(\nu)$.*

*Proof.* The first summation does not change. The second summation increases by at most $\beta \cdot \sum_{\nu:\mu\leq\nu} cost(\nu)$. □

**Lemma 3.3.5.** *If* OFF *adds a nontrivial encoding of a file $f$, then $\Phi$ does not change.*

*Proof.* The first summation does not change. When OFF adds a nontrivial encoding of $f$, the 0-encoding of $f$ is in the cache. Hence, the second summation does not change. □

**Lemma 3.3.6.** *If* ON *charges rent $\delta$, then $\Phi$ does not increase.*

*Proof.* Let $h$ be the cache capacity of OFF. Then the cache capacity of ON is $bh$. The sum over all files $f$ present in $X$, the size of the largest encoding of $f$ in $X$, is at least $\frac{r-1}{r} \cdot bh$. Since ON charges rent from the largest encodings of all files present in $X$, if ON charges rent $\delta$, then the first summation decreases by at least $\frac{r-1}{r} \cdot bh \cdot \delta$.

Since $h$ is the cache capacity of OFF, the second summation increases by at most $\beta \cdot \delta \cdot h$. Since $\beta = \frac{r-1}{r} \cdot b$, $\Phi$ does not increase. □

**Lemma 3.3.7.** *If* ON *removes an encoding, then $\Phi$ does not change.*

*Proof.* Algorithm ON removes an encoding $\mu$ only when $credit(\mu)$ is 0. Hence, $\Phi$ does not change. □

**Lemma 3.3.8.** *If* ON *adds $i$ encodings of a file $f$, for $0 < i < m$, then $\Phi$ decreases by at least $(\beta - 1) \cdot \sum_{\nu:\nu<\mu} cost(\nu)$, where $\mu$ is the $i$-encoding of $f$.*

*Proof.* Algorithm ON adds to the cache all encodings $\nu$ of $f$ such that $\nu < \mu$, setting *credit*$(\nu)$ equal to *cost*$(\nu)$ in each case. Hence, the first summation in $\Phi$ increases by at most $\sum_{\nu:\nu<\mu} cost(\nu)$. Since OFF executes the request first, and the 0-encoding of $f$ is present in OFF's cache at the end of the execution of the request, the second summation decreases by at least $\beta \cdot \sum_{\nu:\nu<\mu} cost(\nu)$. Hence, $\Phi$ decreases by at least $(\beta - 1) \cdot \sum_{\nu:\nu<\mu} cost(\nu)$. □

**Lemma 3.3.9.** *If* ON *adds $m$ encodings of a file $f$, then $\Phi$ decreases by at least $(\beta - 1) \cdot \sum_{\nu:\mu\leq\nu} cost(\nu)$, where $\mu$ is the 0-encoding of $f$.*

*Proof.* Algorithm ON adds to the cache all encodings $\nu$ of $f$ such that $\mu \leq \nu$, setting *credit*$(\nu)$ equal to *cost*$(\nu)$ in each case. Hence, the first summation increases by at most $\sum_{\nu:\mu\leq\nu} cost(\nu)$. Since OFF executes the request for $f$ first, and adds the 0-encoding of $f$ to its cache, the second summation decreases by at least $\beta \cdot \sum_{\nu:\mu\leq\nu} cost(\nu)$. Hence, $\Phi$ decreases by at least $(\beta - 1) \cdot \sum_{\nu:\mu\leq\nu} cost(\nu)$. □

**Theorem 3.** *Algorithm* ON *is resource competitive for any symmetric instance of the compression caching problem.*

*Proof.* Consider a request for a file $f$. By Lemmas 3.3.2 through 3.3.5, if the total cost incurred by OFF to execute the request is $c$, then the potential $\Phi$ increases by at most $\frac{q}{q-1} \cdot \beta \cdot c$; other actions of OFF do not increase $\Phi$. By Lemmas 3.3.6 through 3.3.9, if the total cost incurred by ON to execute the request is $c$, the potential $\Phi$ decreases by at least $(\beta - 1) \cdot c$; other actions of ON do not increase $\Phi$. By Lemma 3.3.1, the potential $\Phi$ is always nonnegative. Hence, the ratio of the cost of ON to that of OFF in executing any request sequence $\sigma$ is at most $\frac{q}{q-1} \cdot \frac{\beta}{\beta-1}$.

Since $\beta = \frac{r-1}{r} \cdot b$, the competitive ratio is at most $\frac{q}{q-1} \cdot \frac{r-1}{r} \cdot b / (\frac{r-1}{r} \cdot b - 1)$. By choosing the values of $q$, $r$, and $b$ appropriately, we find that ON is resource competitive. For example, ON is 4-competitive with $q = 2$, $r = 2$, and $b = 4$. Hence, Theorem 3 holds. □

### 3.3.3 Multilevel Storage

Consider an outsourced storage service scenario (for simplicity, here we describe the problem for a single user) where we have multiple levels of storage. Each storage space is specified by two parameters: storage cost and access latency to the user. The user specifies a fixed overall budget to buy storage space at the various levels, and generates requests for files. The goal is to manage the user budget and minimize the total latency incurred in processing a given request sequence.

Our credit-rental algorithm for the compression caching problem with equal encode and decode costs can be easily generalized to this scenario, and we can show (using a similar analysis as above) that the generalized algorithm is constant competitive with a constant factor advantage in the budget for the aforementioned multilevel storage problem.

## 3.4 Varying Encode Costs and Uniform Decode Costs

We say that an instance $I = (\sigma, m, k)$ of the compression caching problem is a uniform-decode instance if any file in $\sigma$ satisfies the following properties. First, we consider that the decode cost associated with different encodings of any file in $\sigma$ are the same; for any file $f$ and any index $i > 0$, we abbreviate $\mathit{decode}(i, f)$ to $\mathit{decode}$. Second, we consider that for any index$i$, any file $f$ and $f'$ in $\sigma$, $\mathit{size}(i, f) = \mathit{size}(i, f')$, $p(f) = p(f')$, and $\mathit{encode}(i, f) = \mathit{encode}(i, f')$. For the sake of brevity, we write $\mathit{encode}(i, f)$ as $\mathit{encode}(i)$.

We formulate this problem to explore the existence of resource competitive algorithms for the problems in the class of compression caching. This problem is also motivated by the existence of multiple formats of a multimedia file with varying sizes and encode costs, and with roughly similar decode costs.

One might hope to generalize existing algorithms like Landlord for this problem, and to achieve resource competitiveness. However, in this section we show that any deterministic online algorithm (even with an arbitrary factor capacity blowup) for this problem is $\Omega(m)$-competitive, where $m$ is the number of possible encodings of each file. We also give an online algorithm for this problem that is $O(m)$-competitive with $O(m)$ factor capacity blowup.

### 3.4.1 The Lower Bound

In this section we show that any deterministic online algorithm (even with an arbitrary factor capacity blowup) for any uniform-decode instance of the compression caching problem is $\Omega(m)$-competitive. We use an adversarial request generating algorithm and an associated offline strategy to demonstrate the desired result.

For any algorithm $A$, any request sequence $\sigma$, and any real number $k$, we define $\mathit{config}(A, \sigma, k)$ as the configuration of $A$ after executing $\sigma$ with a cache of size $k$, starting with an empty configuration.

For any algorithm $A$, any request sequence $\sigma$, any real number $k$, any index $i$, and any $k$-feasible configuration $S$, we define $\mathit{pseudocost}(A, \sigma, k, i, S)$ as follows. If $\sigma$ is empty, then $\mathit{pseudocost}(A, \sigma, k, i, S)$ is zero. If $\sigma$ is equal to $\langle f \rangle$, then $\mathit{pseudocost}(A, \langle f \rangle, k, i, S)$ is defined as follows. Let $S'$ be the configuration of $A$ after executing the request for $f$ starting with configuration $S$ and with a cache of size $k$. For $i = 0$, we define $\mathit{pseudocost}(A, \langle f \rangle, k, 0, S)$ as $p(f)$, if $f$ is not present in $S$, plus the sum, over all files $f' \neq f$ such that $f'$ is present in $S'$ while $f'$ is not present in $S$, of $p(f')$. For $i = 1$ to $m - 1$, we define $\mathit{pseudocost}(A, \langle f \rangle, k, i, S)$ as $\mathit{pseudocost}(A, \langle f \rangle, k, i - 1, S)$ plus the sum, over all files $f'$ such that for any index $j \geq i$, the $j$-encoding $\mu$ of $f'$ is present in $S'$ while $\mu$ is not present in $S$, of $\mathit{encode}(i)$. If $\sigma$ is equal to $\sigma' \circ \langle f \rangle$, then $\mathit{pseudocost}(A, \sigma, k, i, S)$ is the sum of $\mathit{pseudocost}(A, \sigma', k, i, S)$ and $\mathit{pseudocost}(A, \langle f \rangle, k, i, \mathit{config}(A, \sigma', k))$. We define $\mathit{pseudocost}(A, \sigma, k, i)$ as $\mathit{pseudocost}(A, \sigma, k, i, \emptyset)$.

#### 3.4.1.1 Informal overview

At a high level, the adversarial request generating algorithm *Adversary* works recursively as follows. For a given online algorithm ON, a given number of encodings for a file $m$, a given cache capacity of the offline algorithm $k$, and a blowup $b$, the algorithm $\mathit{Adversary}(\mathrm{ON}, m, k, b)$ picks a set of files $X$ such that any file in $X$ is not in ON's cache, and invokes a recursive request generating procedure $\mathit{AdversaryHelper}(X, i, \sigma, \mathrm{ON}, k, b)$, where initially $|X|$ is the number of $(m-1)$-encodings that can fit in a cache of size $k$, $i = m - 1$, and $\sigma$ is empty. This procedure returns a trace of the offline algorithm OFF. (See Section 3.4.1.2 for formal definitions and a description of the algorithm.)

Consider an invocation of procedure $\mathit{AdversaryHelper}(X, i, \sigma, \mathrm{ON}, k, b)$. The adversary picks a subset of the files $Y$ from $X$ such that any file $f$ in $Y$ satisfies certain conditions. For $i > 1$, if $Y$ contains sufficiently many files, then the adversary invokes $\mathit{AdversaryHelper}(Y, i-1, \sigma', \mathrm{ON}, k, b)$, where $\sigma'$ is the request sequence generated; otherwise, $\mathit{AdversaryHelper}(X, i, \sigma, \mathrm{ON}, k, b)$ is terminated. For $i = 1$, the adversary picks a file $f$ in $Y$, and repeatedly generates requests for $f$ until either ON adds an encoding of $f$ to its cache, or a certain number of requests for $f$ are generated. Finally, $\mathit{AdversaryHelper}(X, 1, \sigma)$ terminates when $Y$ is empty.

At a high level, the offline algorithm OFF works as follows. When procedure $\mathit{AdversaryHelper}(X, i, \sigma, \mathrm{ON}, k, b)$ terminates, OFF decides the encodings for the files in $X$. For any index $j \geq i$, if ON adds the $j$-encodings of less than a certain fraction of files in $X$ any time during the execution of the request sequence generated by $\mathit{AdversaryHelper}(X, i, \sigma, \mathrm{ON}, k, b)$, then OFF adds the $i$-encodings of all the files in $X$, and incurs no miss penalties in executing the request sequence generated by $\mathit{AdversaryHelper}(X, i, \sigma, \mathrm{ON}, k, b)$. Otherwise, OFF returns the concatenation of the traces generated during the execution of procedure $\mathit{AdversaryHelper}(X, i, \sigma, \mathrm{ON}, k, b)$. By adding the $j$-encodings of a certain fraction of files in $X$, ON incurs much higher cost than OFF in executing the request sequence generated by $\mathit{AdversaryHelper}(X, i, \sigma, \mathrm{ON}, k, b)$.

Using an inductive argument, we show that ON is $\Omega(m)$-competitive for the compression caching problem with varying encode and uniform decode costs.

#### 3.4.1.2 Adversarial request generating algorithm

Some key notations used in the adversarial request generating algorithm are as follows.

For any file $f$ and any real number $b$, $\mathit{eligible}(f, m, b)$ holds if the following conditions hold: (1) for any index $i$, $\mathit{size}(i, f) = r^{m-i-1}$, where $r = 8b$; (2) $p(f) = p$; (3) for any index $i$, $\mathit{encode}(i, f) = p \cdot q^i$, where $q = \frac{m}{20}$; and (4) $\mathit{decode} = 0$. The number of $i$-encodings of files that can fit in a cache of size $k$ is denoted $\mathit{num}(k, i)$. Note that, for eligible files, $\mathit{num}(k, i)$ is equal to $r \cdot \mathit{num}(k, i-1)$.

For any algorithm $A$, any request sequence $\sigma$, any real number $k$, any file $f$, and any index $i$, we define a predicate $\mathit{aggressive}(A, \sigma, k, f, i)$ as follows. If $\sigma$ is empty, then $\mathit{aggressive}(A, \sigma, k, f, i)$ does not hold. If $\sigma$ is equal to $\sigma' \circ \langle f' \rangle$, then $\mathit{aggressive}(A, \sigma, k, f, i)$ holds if either $\mathit{aggressive}(A, \sigma', k, f, i)$ holds or, for some index $j \geq i$, the $j$-encoding of $f$ is present in $\mathit{config}(\mathrm{ON}, \sigma, k)$.

For any request sequence $\sigma$, and any index $i$, any set of files $X$, we define $\mathit{trace}(\sigma, i, X)$ as follows. If $\sigma$ is empty, then $\mathit{trace}(\sigma, i, X)$ is empty. If $\sigma$ is equal to $\sigma' \circ \langle f \rangle$, then $\mathit{trace}(\sigma, i, X)$ is $\mathit{trace}(\sigma', i, X) \circ \langle (f, Y) \rangle$, where $Y$ is the set of the $i$-encodings of files in $X$.

In Figure 3.2 we describe the adversarial request generating algorithm *Adversary*.

#### 3.4.1.3 Analysis

The caching decisions of the offline algorithm OFF are given by the trace $T$ generated during the execution of *Adversary* (Figure 3.2).

**Lemma 3.4.1.** *Consider any deterministic online algorithm* ON *with a cache of size $k$. Let $S'$ be the new configuration of* ON *after executing a request for any eligible file $f$ in some configuration $S$. The total cost of* ON *in the configuration change from $S$ to $S'$ and in executing request for $f$ in configuration $S$ is at least $\frac{q-1}{q} \cdot \mathit{pseudocost}(\mathrm{ON}, \langle f \rangle, k, i, S)$, where $0 \le i < m$.*

*Proof.* The lemma follows from the definition of *pseudocost*, that is, for any file $f'$ and index $j \ge i$, if the $j$-encoding $\mu$ of $f'$ is present in $S'$ while $\mu$ is not present in $S$, then we add $\mathit{encode}(i)$ to $\mathit{pseudocost}(\mathrm{ON}, \langle f \rangle, k, i, S)$. □

**Lemma 3.4.2.** *Consider any execution of $\mathit{AdversaryHelper}(X, 1, \sigma, \mathrm{ON}, k, b)$. Let $\sigma'$ be the sequence of requests for some file $f$ generated in the loop of Lines 15 to 19, and let $T''$ be the trace $T$ just before Line 20. Then, in Line 20, $\mathit{pseudocost}(\mathrm{ON}, \sigma', bk, 0, \mathit{config}(\mathrm{ON}, \sigma \circ \mathit{requests}(T''), bk))$ is at least $\mathit{cost}(\mathit{trace}(\sigma', 0, \{f\}))$.*

```
$\mathit{Adversary}(\mathrm{ON}, m, k, b)$
$T := \emptyset$
**while** $|T| < N$ **do**
  $X :=$ set of $\mathit{num}(k, m-1)$ files $f$ such that (1) $\mathit{eligible}(f, m, b)$ holds; and
      (2) $f$ is not present in $\mathit{config}(\mathrm{ON}, \mathit{requests}(T), bk)$
  $T := T \circ \mathit{AdversaryHelper}(X, m-1, \mathit{requests}(T), \mathrm{ON}, k, b)$
**od**
**return** $T$

$\mathit{AdversaryHelper}(X, i, \sigma, \mathrm{ON}, k, b)$
$T, \sigma' := \emptyset, \emptyset$
$Y := X$
**repeat**
  **if** $i = 1$ **then**
    Let $f$ be an arbitrary file in $Y$
    $\mathit{count} := 0$
    **repeat**
      $\sigma' := \sigma' \circ \langle f \rangle$
      $S := \mathit{config}(\mathrm{ON}, \sigma \circ \mathit{requests}(T) \circ \sigma', bk)$
      $\mathit{count} := \mathit{count} + 1$
    **until** $f$ is not present in $S$ or $\mathit{count} \geq 8$
    $T := T \circ \mathit{trace}(\sigma', 0, \{f\})$
    $\sigma' := \emptyset$
  **else**
    $X' :=$ arbitrary subset of $\mathit{num}(k, i-1)$ files in $Y$
    $T' := \mathit{AdversaryHelper}(X', i-1, \sigma \circ \mathit{requests}(T), \mathrm{ON}, k, b)$
    $T := T \circ T'$
  **fi**
  reassign $Y$ as follows: for any file $f$, $f$ is in $Y$ if and only if (1) $f$ is in $X$
      (2) $f$ is not present in $\mathit{config}(\mathrm{ON}, \sigma \circ \mathit{requests}(T), bk)$;
      (3) $\mathit{cost}_f(T) < (8 \cdot e_i - 8 \cdot e_{i-1})$; and
      (4) $\mathit{aggressive}(\mathrm{ON}, \mathit{requests}(T), bk, f, i)$ does not hold
**until** ($i = 1$ and $|Y| = \emptyset$) or ($|Y| < \mathit{num}(k, i-1)$)
**if** $|\{f \in X | \mathit{aggressive}(\mathrm{ON}, \mathit{requests}(T), bk, f, i)\}| < 2b \cdot \mathit{num}(k, i-1)$ **then**
  $T := \mathit{trace}(\mathit{requests}(T), i, X)$
**fi**
**return** $T$
```

Figure 3.2: The adversarial request generating algorithm for the compression caching problem with varying encode and uniform decode costs. Here, $N$ is the sufficient large number of requests to be generated.

*Proof.* In Line 20, $\mathit{pseudocost}(\mathrm{ON}, \sigma', bk, 0, \mathit{config}(\mathrm{ON}, \sigma \circ \mathit{requests}(T''), bk))$ is at least $\mathit{count} \cdot p(f)$, where $\mathit{count}$ is at least 1. On the other hand, the value of $\mathit{cost}(\mathit{trace}(\sigma', 0, \{f\}))$ is at most $p(f)$. □

**Lemma 3.4.3.** *Consider any execution of* $\mathit{AdversaryHelper}(X, 1, \sigma, \mathrm{ON}, k, b)$*. Let trace* $T$ *be the return value of* $\mathit{AdversaryHelper}(X, 1, \sigma, \mathrm{ON}, k, b)$*,* $\sigma'$ *be* $\mathit{requests}(T)$*, and* $S$ *be* $\mathit{config}(\mathrm{ON}, \sigma, bk)$*. Then,* $\mathit{pseudocost}(\mathrm{ON}, \sigma', bk, 1, S)$ *is at least* $\frac{1}{40} \cdot \mathit{cost}(T)$*.*

*Proof.* Let $Z$ be $\{f \in X \mid \mathit{aggressive}(\mathrm{ON}, \sigma', bk, f, i)\}$.

Case (1): In the execution of $\mathit{AdversaryHelper}(X, 1, \sigma, \mathrm{ON}, k, b)$, just before Line 29, $|Z|$ is less than $2b \cdot \mathit{num}(k, 0)$. Let $T'$ be the trace $T$ in Line 28. Procedure $\mathit{AdversaryHelper}(X, 1, \sigma, \mathrm{ON}, k, b)$ terminates when $Y$ is empty in Line 28. That is, $r \cdot \mathit{num}(k, 0)$ files in $X$ do not satisfy one of the last three conditions to be in $Y$. Since less than $2b \cdot \mathit{num}(k, 0)$ files are in $|Z|$, and ON can keep at most $(b \cdot \mathit{num}(k, 0))$ 0-encodings of files, just before Line 29, there are at least $(r - 2b - b) \cdot \mathit{num}(k, 0)$ files $f$ for which $\mathit{cost}_f(T')$ is at least $(8 \cdot \mathit{encode}(1) - 8 \cdot \mathit{encode}(0))$. Hence, by Lemma 3.4.2, $\mathit{pseudocost}(\mathrm{ON}, \sigma', bk, 0, S) \geq (r - 3b) \cdot \mathit{num}(k, 0) \cdot 8 \cdot (\mathit{encode}(1) - \mathit{encode}(0))$.

In this case, OFF overrides the trace (Line 30), adds 1-encodings of all the files in $X$, and executes all the requests in $\mathit{requests}(T)$. Hence, after Line 30, $\mathit{cost}(T) = r \cdot \mathit{num}(k, 0) \cdot \mathit{encode}(1)$. Therefore, $\mathit{pseudocost}(\mathrm{ON}, \sigma', bk, 0, S) \geq \frac{(r-3b)}{r} \cdot 8 \cdot \frac{(\mathit{encode}(1) - \mathit{encode}(0))}{\mathit{encode}(1)} \cdot \mathit{cost}(T)$. Since $\mathit{pseudocost}(\mathrm{ON}, \sigma', bk, 1, S) \geq \mathit{pseudocost}(\mathrm{ON}, \sigma', bk, 0, S)$, and $r = 8b$, the lemma follows.

Case (2): In the execution of $\mathit{AdversaryHelper}(X, 1, \sigma, \mathrm{ON}, k, b)$, just before Line 29, if $|Z|$ is at least $2b \cdot \mathit{num}(k, 0)$, then OFF does not commit any additional costs to execute requests in $\sigma'$ (Line 32). Since

$$\mathit{pseudocost}(\mathrm{ON}, \sigma', bk, 1, S) \geq \mathit{pseudocost}(\mathrm{ON}, \sigma', bk, 0, S)$$

by the definition of *pseudocost*, and

$$\mathit{pseudocost}(\mathrm{ON}, \sigma', bk, 0, S) \geq \mathit{cost}(T)$$

by Lemma 3.4.2. Hence, the lemma follows. □

**Lemma 3.4.4.** *Consider any execution of* $\mathit{AdversaryHelper}(X, i, \sigma, \mathrm{ON}, k, b)$*. Let trace* $T$ *be the return value of* $\mathit{AdversaryHelper}(X, i, \sigma, \mathrm{ON}, k, b)$*,* $\sigma'$ *be* $\mathit{requests}(T)$*, and* $S$ *be* $\mathit{config}(\mathrm{ON}, \sigma, bk)$*. Then,* $\mathit{pseudocost}(\mathrm{ON}, \sigma', bk, i, S)$ *is at least* $\frac{i}{40} \cdot \mathit{cost}(T)$ *for* $i \geq 1$*.*

*Proof.* We prove this lemma by induction on $i$. The base case ($i = 1$) follows from Lemma 3.4.3.

Induction hypothesis: Let $T''$ be the trace $T$ just before the execution of Line 24, and $T'$ be the return value of any execution of $\mathit{AdversaryHelper}(X', i - 1, \sigma \circ \mathit{requests}(T''), \mathrm{ON}, k, b)$ in Line 24 during the execution of procedure $\mathit{AdversaryHelper}(X, i, \sigma, \mathrm{ON}, k, b)$. Let $S'$ be $\mathit{config}(\mathrm{ON}, \sigma \circ \mathit{requests}(T''), bk)$ Then, $\mathit{pseudocost}(\mathrm{ON}, \mathit{requests}(T'), bk, i - 1, S')$ is at least $\frac{i-1}{40} \cdot \mathit{cost}(T')$.

Induction step: Let $Z$ be $\{f \in X \mid \mathit{aggressive}(\mathrm{ON}, \sigma', bk, f, i)\}$.

Case (1): In the execution of $\mathit{AdversaryHelper}(X, i, \sigma, \mathrm{ON}, k, b)$, just before Line 29, $|Z|$ is less than $2b \cdot \mathit{num}(k, i-1)$. Let $T'$ be the trace $T$ in Line 28. For $i > 1$, $\mathit{AdversaryHelper}(X, i, \sigma, \mathrm{ON}, k, b)$ terminates when there are less than $\mathit{num}(k, i-1)$ files in set $Y$ (in Line 28). That is, at least $(r-1) \cdot \mathit{num}(k, i-1)$ files do not satisfy one of the last three conditions to be in $Y$. Since less than $2b \cdot \mathit{num}(k, i-1)$ files are in $Z$, and ON can keep at most $(b \cdot \mathit{num}(k, i-1))$ $j$-encodings of files such that $j \leq i$, there are at least $(r - 1 - 2b - b) \cdot \mathit{num}(k, i-1)$ files $f$, for which $\mathit{cost}_f(T')$ is at least $8 \cdot \mathit{encode}(i) - 8 \cdot \mathit{encode}(i-1)$.

By the induction hypothesis,

$$\begin{aligned}
&\mathit{pseudocost}(\mathrm{ON}, \mathit{requests}(T'), bk, i-1, \mathit{config}(\mathrm{ON}, \sigma \circ \mathit{requests}(T'), bk)) \\
&\geq \frac{i-1}{40} \cdot (r - 3b - 1) \cdot \mathit{num}(k, i-1) \cdot 8 \cdot (\mathit{encode}(i) - \mathit{encode}(i-1))
\end{aligned}$$

In this case OFF overrides the trace $T'$ (Line 30), and adds the $i$-encodings of all the files in $X$, and executes all the requests $\mathit{requests}(T)$. Hence, after Line 30, $\mathit{cost}(T)$ is $r \cdot \mathit{num}(k, i-1) \cdot \mathit{encode}(i)$. Finally,

$$\begin{aligned}
& \mathit{pseudocost}(\mathrm{ON}, \sigma', i, S) \\
\geq\ & \mathit{pseudocost}(\mathrm{ON}, \sigma', i-1, S) \\
\geq\ & \frac{i-1}{40} \cdot (r - 3b - 1) \cdot \mathit{num}(k, i-1) \cdot 8 \cdot (\mathit{encode}(i) - \mathit{encode}(i-1)) \\
\geq\ & \frac{i-1}{40} \cdot (r - 3b - 1) \cdot \frac{8}{r} \cdot (1 - \frac{\mathit{encode}(i-1)}{\mathit{encode}(i)}) \cdot \mathit{cost}(T) \\
\geq\ & \frac{i}{40} \cdot \mathit{cost}(T)
\end{aligned}$$

(The first inequality follows from the definition of *pseudocost*. The second inequality follows from the induction hypothesis. The third inequality follows from the fact that $\mathit{cost}(T)$ is $r \cdot \mathit{num}(k, i-1) \cdot \mathit{encode}(i)$. The fourth inequality follows from the fact that $\frac{i-1}{40} \cdot (r - 3b - 1) \cdot \frac{8}{r} \cdot (1 - \frac{\mathit{encode}(i-1)}{\mathit{encode}(i)}) \geq \frac{i}{40}$, for $q \geq 2$ and $i \geq 2$.)

Case (2): In the execution of $\mathit{AdversaryHelper}(X, i, \sigma, \mathrm{ON}, k, b)$, just before Line 29, $|Z|$ is at least $2b \cdot \mathit{num}(k, i-1)$.

Let $\Delta = \mathit{cost}(T)$. In this case, OFF does not incur any additional costs to execute requests in $\sigma'$ (Line 32). By the design of *AdversaryHelper*, $\Delta \leq r \cdot \mathit{num}(k, i-1) \cdot (p + 8 \cdot \mathit{encode}(i))$. (For any file $f$ in $X$ selected in the set $X'$ in Line 24, i.e., $T' := \mathit{AdversaryHelper}(X', i-1, \sigma \circ \mathit{requests}(T), \mathrm{ON}, k, b)$, $\mathit{cost}_f(T)$ is at most $8 \cdot \mathit{encode}(i) - 8 \cdot \mathit{encode}(i-1)$. Since $\mathit{cost}_f(T')$ is at most $(p + 8 \cdot \mathit{encode}(i-1))$, $\mathit{cost}_f(T)$ is at most $(p + 8 \cdot \mathit{encode}(i))$.)

By the induction hypothesis,

$$\begin{aligned}
& \mathit{pseudocost}(\mathrm{ON}, \sigma', bk, i, S) \\
\geq\ & \mathit{pseudocost}(\mathrm{ON}, \sigma', bk, i-1, S) + 2b \cdot \mathit{num}(k, i-1) \cdot \mathit{encode}(i) \\
\geq\ & \frac{i-1}{40} \cdot \Delta + 2b \cdot \mathit{num}(k, i-1) \cdot \mathit{encode}(i) \\
\geq\ & \frac{i}{40} \cdot \Delta
\end{aligned}$$

(The first inequality follows from the fact that the second term of the RHS is not included in the LHS. The second inequality follows from induction hypothesis. The third inequality holds since $2b \cdot \mathit{encode}(i) \geq \frac{r}{40} \cdot (p + 8 \cdot$

*encode*$(i)$).) □

The following theorem is immediate from Lemmas 3.4.1 and 3.4.4.

**Theorem 4.** *Any deterministic online algorithm with an arbitrary factor capacity blowup is $\Omega(m)$-competitive for any uniform-decode instance $I$ of the compression caching problem, where $m = numindex(I)$.*

### 3.4.2 An Upper Bound

In this section we present an online algorithm that is $O(m)$-competitive with $O(m)$ factor capacity blowup for any uniform-decode instance $I$ of the compression caching problem, where $m = numindex(I)$. As in Section 3.3, by preprocessing, we can arrange the encodings of files in decreasing sizes and increasing encode costs; that is, after preprocessing, for any file $f$ and any index $i < m - 1$, $size(i + 1, f) = \frac{1}{r} size(i, f)$, and $encode(i + 1, f) = q \cdot encode(i, f)$, where $r = 1 + \epsilon$, $q = 1 + \epsilon'$, $\epsilon > 0$, and $\epsilon' > 0$.

#### 3.4.2.1 Algorithm

In this section we present an online algorithm ON for any uniform-decode instance of the compression caching problem.

Algorithm ON divides its cache into $m$ blocks. For any index $i$, block $i$ keeps only the $i$-encodings of files. For any integer $k$ and index $i$, let $num(k, i)$ be the maximum number of $i$-encodings of files that can fit in any block of size $k$.

```
    ON($\sigma, m, k$)
    { Initially, for each index $j$ and file $g$, $\mathit{misscount}(g, j) := 0$, $\mathit{filecount}(j) := 0$, and
        trace $T := \emptyset$}
1   **for** $i = 0$ to $|\sigma| - 1$ **do**
2   Let $\sigma_i$ be a request for a file $f$
3   **if** the $j$-encoding $\mu$ of $f$ is present the cache
            (break ties by picking minimum such $j$) **then**
4       evaluate $h_j^{-1}(\mu)$
5   **else**
6       **foreach** index $j$, increment $\mathit{misscount}(f, j)$
7       **if** for some $j$, $\mathit{misscount}(f, j) \cdot p(f) \geq \mathit{encode}(j)$
                (break ties by picking maximum such $j$) **then**
8           **if** $\mathit{filecount}(j) = \mathit{num}(k/m, j)$ **then**
9               **foreach** $j' \leq j$ **do**
10                  $\mathit{filecount}(j') := 0$;
11                  for all the $j'$-encodings $\mu$ of files $f'$, $\mathit{misscount}(f', j') := 0$; and unmark $\mu$
12              **od**
13          **fi**
14          remove an unmarked encoding from block-$j$
15          add the $j$-encoding $\mu$ of $f$ to block $j$, and mark $\mu$
16          increment $\mathit{filecount}(j)$
17      **fi**
18  **fi**
19  $S :=$ the set of encodings present in the cache
20  $T = T \circ (\sigma_i, S)$
21  **od**
```

Figure 3.3: The online algorithm ON for any uniform-decode instance of the compression caching problem.

Roughly speaking, ON works as follows. For any index $i$, ON adds the $i$-encoding of a file $f$ after the miss penalties incurred by ON on $f$ sum to at least $\mathit{encode}(i, f)$. We use a standard marking algorithm as an eviction procedure for each block. The complete description of ON is presented in Figure 3.3.

#### 3.4.2.2 Analysis

For any trace $T$ and any configuration $S$, we define *encodecost*$(T, S)$ inductively as follows. If $T$ is empty, then *encodecost*$(T, S)$ is zero. If $T$ is equal to $\langle (f, S') \rangle \circ T'$, then *encodecost*$(T, S)$ is *encodecost*$(T', S')$ plus the sum, over all $i$-encoding $\mu$ of $f'$ such that $\mu$ is present in $S'$ while $\mu$ is not present in $S'$, of *encode*$(i)$. For any trace $T$, we define *encodecost*$(T)$ as *encodecost*$(T, \emptyset)$.

For any trace $T$ and any configuration $S$, we define *penalty*$(T, S)$ inductively as follows. If $T$ is empty, then *penalty*$(T, S)$ is zero. If $T$ is equal to $\langle (f, S') \rangle \circ T'$, then *penalty*$(T, S)$ is *penalty*$(T', S')$ plus $p(f)$, if $f$ is not present in $S$; otherwise, it is zero. For any trace $T$, we define *penalty*$(T)$ as *penalty*$(T, \emptyset)$.

**Lemma 3.4.5.** *Consider any uniform-decode instance $I = (\sigma, m, k)$ of the compression caching problem. Let $T = \mathrm{ON}(\sigma, m, k)$. Then, requests$(T) = \sigma$, and space$(T) \leq k$.*

*Proof.* Follows from inspection of the algorithm in Figure 3.3. $\square$

**Lemma 3.4.6.** *Consider any uniform-decode instance $I = (\sigma, m, k)$ of the compression caching problem. Let $T = \mathrm{ON}(\sigma, m, k)$. Then, encodecost$(T) \leq m \cdot$ penalty$(T)$.*

*Proof.* For any file $f$ and index $i$, ON adds the $i$-encoding of $f$ only when *misscount*$(f, i) \cdot p(f)$ is at least *encode*$(i)$. Since on a miss by ON, we increment *misscount*$(f, i)$ for all indices $i$, the lemma follows. $\square$

For any trace $T$, *miss*$(T)$ returns a sequence of requests constructed as follows. If $T$ is empty, then *miss*$(T)$ is empty. If $T$ is $\langle(f, S)\rangle$, then *miss*$(T)$ is $\langle f\rangle$. If $T$ is equal to $T' \circ \langle(f, S)\rangle \circ \langle(f', S')\rangle$, then *miss*$(T)$ is *miss*$(T' \circ \langle(f, S)\rangle) \circ \langle f'\rangle$, if $f'$ is not present in $S$, and is *miss*$(T' \circ (f, S))$ otherwise.

An interval is a pair of nonnegative integers $(i, j)$ such that $i < j$. For any file $f$ and any configuration $S$, we define *configuration*$(f, S)$ as $S$.

For any trace $T$, any encoding $\mu$, and any interval $(i, j)$ such that $0 \le i$ and $j \le |T|$, *added*$(T, (i, j), \mu)$ holds if, for some $j'$ such that $i \le j' < j$, one of the following conditions holds: (1) $j' = 0$ and $\mu$ is present in *configuration*$(T_0)$; or (2) $j' > 0$ and $\mu$ is present in *configuration*$(T_{j'})$ while $\mu$ is not present in *configuration*$(T_{(j'-1)})$. For any trace $T$, any encoding $\mu$, and any interval $(i, j)$ such that $0 \le i$ and $j \le |T|$, *deleted*$(T, (i, j), \mu)$ holds if, for some $j'$ such that $i \le j' < j$ and $j' > 0$, $\mu$ is not present in *configuration*$(T_{j'})$ while $\mu$ is present in *configuration*$(T_{(j'-1)})$. For any trace $T$, any encoding $\mu$, and any interval $(i, j)$ such that $0 \le i$ and $j \le |T|$, *throughout*$(T, (i, j), \mu)$ holds if $i > 0$ and $\mu$ is present in *configuration*$(T_{j'})$ for all $j'$ in $[i - 1, j]$. For any trace $T$ and any encoding $\mu$, *present*$(T, \mu)$ holds if $\mu$ is present in any of the configurations in $T$.

**Lemma 3.4.7.** *Consider any uniform-decode instance* $I = (\sigma, m, k)$ *of the compression caching problem. Let* $T$ *be any trace such that requests*$(T) = \sigma$, *and space*$(T) \le k$. *Let* $T' = \mathrm{ON}(\sigma, m, bmk)$. *Then penalty*$(T') = O(\mathit{cost}(T))$.

*Proof.* Follows from Claims 1 through 5 (see below).

First we introduce some useful definitions. Consider the execution $T' = \text{ON}(\sigma, m, bmk)$. For any index $\ell$, we define an $\ell$-epoch as follows. The first $\ell$-epoch starts at the beginning of the request sequence $\sigma$. For any integer $i > 0$, the $i$th $\ell$-epoch ends if the request sequence $\sigma$ ends or just before a request such that, during the execution of the request, Line 11 (Figure 3.3) is executed with $j' = \ell$. The $(i+1)$st $l$-epoch starts when the $i$th $\ell$-epoch ends. Any $\ell$-epoch is specified by an interval $(i, j)$ such that the epoch starts just before $\sigma_i$, and ends just before $\sigma_j$. For any index $\ell$ and any integer $i$ such that $0 \leq i < |\sigma|$, $\mathit{enclosing}(\sigma, i, \ell)$ returns an interval $(j, j')$ for an $\ell$-epoch such that $j \leq i < j'$.

For the request sequence $\sigma$, for any index $\ell$, and any $\ell$-epoch $(i, j)$, we define the predicate $\mathit{incomplete}(\ell, (i, j))$ to hold if $j = |\sigma|$. For the request sequence $\sigma$, for any index $\ell$, and any $\ell$-epoch $(i, j)$, the predicate $\mathit{complete}(\ell, (i, j))$ holds if $\mathit{incomplete}(\ell, (i, j))$ does not hold and after the execution of $\sigma_j$, the $\mathit{filecount}(\ell)$ variable of ON is $\mathit{num}(k, \ell)$. For the request sequence $\sigma$, for any index $\ell$, and any $\ell$-epoch $(i, j)$, the predicate $\mathit{terminated}(\ell, (i, j))$ holds if $\mathit{incomplete}(\ell, (i, j))$ does not hold and after the execution of $\sigma_j$, the $\mathit{filecount}(\ell)$ variable of ON is less than $\mathit{num}(k, \ell)$.

Let $\sigma_i$ be a request for file $f$. Then, $\mathit{penalty}(T', i) = p(f)$, if $\sigma_i$ is in $\mathit{miss}(T')$, and is zero otherwise.

We color all of the requests in $\sigma$ using a coloring algorithm presented

```
$color(\sigma, i)$
begin
  color := nil    Let the $i$th request in $\sigma$ be for a file $f$
  if $\sigma_i$ is not in $miss(T')$ then
    // do not color $\sigma_i$
  else if for some index $\ell > 0$, $added(T, enclosing(\sigma, i, \ell), \mu)$ holds,
      where $\mu$ is the $\ell$-encoding of $f$, then
    color := $white$
  else if for some index $\ell > 0$, $deleted(T, enclosing(\sigma, i, \ell), \mu)$ holds,
      where $\mu$ is the $\ell$-encoding of $f$, then
    color := $red$
  else if for some index $\ell > 0$, $incomplete(\ell, enclosing(\sigma, i, \ell))$ holds and
      $present(T, \mu)$ holds, where $\mu$ is the $\ell$-encoding of $f$, then
    color := $blue$
  else if for some index $\ell$, $complete(\ell, enclosing(\sigma, i, \ell))$ holds and
      $throughout(T, enclosing(\sigma, i, \ell), \mu)$ holds, where $\mu$ is the $\ell$-encoding of $f$, then
    color := $black$
  else
    color := $gray$
  fi
  return color
end
```

Figure 3.4: The coloring algorithm.

in Figure 3.4.

Note that $encodecost(T)$ is equal to the following sum:

$$\sum_{(\forall index\ \ell)} \sum_{(\forall \ell-epoch(i,j))} \sum_{(\forall \ell-\text{encoding }\mu \text{ s.t. } added(T,(i,j),\mu))} encode(\ell)$$

Claim 1. $\sum_{\forall i: color(\sigma,i)=white} penalty(T', i) \le encodecost(T)$

Our proof of Claim 1 is as follows. For any index $\ell$, consider any $\ell$-epoch $(i, j)$. For any file $f$ such that $added(T, (i, j), \mu)$ holds, where $\mu$ is the $\ell$-encoding of $f$, we color *white* all of the misses for $f$ by ON during the $\ell$-epoch.

Since ON adds the $\ell$-encoding of $f$, $\mu$, when $misscount(f, \ell) \cdot p(f) \ge$

$encode(\ell)/p(f)$, and does not remove $\mu$ until the end of the epoch, there are less than $encode(\ell)/p(f) + 1$ misses for $f$ by ON in the $\ell$-epoch.

Claim 2. $\sum_{\forall i: color(\sigma,i)=red} penalty(T', i) \leq encodecost(T)$

Our proof for Claim 2 is as follows. Since there could be at most one deletion per addition of any encoding, we can charge each addition twice, and account for the misses in an epoch in which the encoding is deleted.

Claim 3. $\sum_{\forall i: color(\sigma,i)=blue} penalty(T', i) \leq encodecost(T)$

Our proof for the Claim 3 is as follows. Consider any incomplete $\ell$-epoch $(i, j)$. For any file $f$ such that $present(T, \mu)$ holds, where $\mu$ is the $\ell$-encoding of $f$, we color blue all of the misses for $f$ during the $\ell$-epoch. As argued above, there are less than $encode(\ell)/p(f) + 1$ misses for $f$ in $T'$ during any $\ell$-epoch.

Since $present(T, \mu)$ holds and there could be at most one incomplete $\ell$-epoch, we can charge the miss penalties incurred for $f$ in $T'$ during the epoch to the encode cost incurred for $\mu$ in $T$.

Claim 4. At most a constant fraction of all the requests in $miss(T')$ are colored *black* by the coloring algorithm in Figure 3.4.

Our proof for Claim 4 is as follows. Consider any $\ell$-epoch $(i, j)$ such that $complete(i, j)$ holds. Note that if there are no complete epochs, then there are no requests that are colored *black*.

By definition, a set of $num(k, \ell)$ files are marked in the $\ell$-epoch. The number of misses incurred by any file $f$ in the set is at most $encode(\ell)/p(f)+1$

misses in the epoch. Since $\mathit{space}(T) \leq k$ and capacity of any block of ON is $bk$, for at most $\mathit{num}(k, \ell)/b$ files $f$ $\mathit{throughout}(T, (i, j), \mu)$ can hold, where $\mu$ is the $\ell$-encoding of $f$. Hence, at most $\mathit{num}(k, \ell)/b \cdot \mathit{encode}(\ell)/p(f)$ misses in the epoch can be colored *black*.

Since $\mathit{num}(k, \ell - 1) \leq 1/r \cdot \mathit{num}(k, \ell)$, at most $\mathit{num}(k, \ell)/(r-1)$ files can be flushed from lower-indexed blocks. At most $\mathit{num}(k, \ell)/(b \cdot (r-1)) \cdot \mathit{encode}(\ell - 1)/p(f)$ misses can be colored *black*. Hence, at most $2\frac{\mathit{num}(k,l)}{b} \cdot \frac{\mathit{encode}(\ell)}{p(f)}$ misses can be colored *black*, Since there are at least $2\mathit{num}(k, l) \cdot \frac{\mathit{encode}(\ell)}{p(f)}$ misses in the $\ell$-epoch, at most a constant fraction of misses attributed to a complete epoch can be colored *black*. Hence, Claim 4 follows.

Claim 5. For any request $q$ in $\mathit{miss}(T')$ that is colored *gray* in the coloring algorithm in Figure 3.4, $q$ belongs to $\mathit{miss}(T)$.

Our proof of Claim 5 is as follows. Consider any request $\sigma_i$ such that $\mathit{color}(\sigma, i)$ returns *gray*. Since $\mathit{color}(\sigma, i)$ returns *gray*, all of the following conditions hold: $(a)$ there is no index $\ell > 0$, such that $\mathit{added}(T, \mathit{enclosing}(\sigma, i, \ell), \mu)$ or $\mathit{deleted}(T, \mathit{enclosing}(\sigma, i, \ell), \mu)$ holds, where $\mu$ is the $\ell$-encoding of $f$; $(b)$ there is no index $\ell > 0$ such that $\mathit{incomplete}(\ell, \mathit{enclosing}(\sigma, i, \ell))$ holds and $\mathit{present}(T, \mu)$ holds, where $\mu$ is the $\ell$-encoding of $f$; and $(c)$ there is no index $\ell$ such that $\mathit{complete}(\ell, \mathit{enclosing}(\sigma, i, \ell))$ and $\mathit{throughout}(T, \mathit{enclosing}(\sigma, i, \ell), \mu)$ hold, where $\mu$ is the $\ell$-encoding of $f$. These conditions imply that $\sigma_i$ belongs to $\mathit{miss}(T)$.

From Claims 1 through 5, it follows that $\mathit{penalty}(T') = O(\mathit{cost}(T))$. $\square$

**Lemma 3.4.8.** *Consider any uniform-decode instance $I = (\sigma, m, k)$ of the compression caching problem. Let $T$ be any trace with requests$(T) = \sigma$ and space$(T) \leq k$. Then there exists an online algorithm $A$, and a positive constant $b$, such that $T' = A(\sigma, m, bmk)$, space$(T') \leq bmk$, and cost$(T') = O(m) \cdot$ cost$(T)$.*

*Proof.* Follows from Lemmas 3.4.5, 3.4.6 and 3.4.7. □

**Theorem 5.** *For any uniform-decode instance $I$ of the compression caching problem, there exists an online algorithm that is is $O(m)$-competitive with $O(m)$ factor capacity blowup, where $m =$ numindex$(I)$.*

*Proof.* Follows from Lemma 3.4.8. □

## 3.5 Uniform Encode Costs and Varying Decode Costs

We say that an instance $I(\sigma, m, k)$ of the compression caching problem is a uniform-encode instance if any file in $\sigma$ satisfies the following properties. First, we consider that the encode costs of all the nontrivial encodings of any file $f$ in $\sigma$ are the same; for any index $i > 0$, we abbreviate *encode*$(i, f)$ to *encode*. Second, we consider that for any index $i$, any file $f$ and $f'$ in $\sigma$, *size*$(i, f) =$ *size*$(i, f')$, $p(f) = p(f')$, and *decode*$(i, f) =$ *decode*$(i, f')$. For the sake of brevity, for any file $f$ in $\sigma$, we write *decode*$(i, f)$ as *decode*$(i)$.

This problem is a mathematically interesting compression caching problem, and analogous to the problem considered in Section 3.4.

In this section we show that any deterministic online algorithm (even with an arbitrary factor capacity blowup) for any uniform-encode instance of the compression caching problem is $\Omega(\log m)$-competitive, where $m$ is the number of possible encodings for each file. Further, we present an online algorithm for this problem that is $O(m)$-competitive with $O(m)$ factor capacity blowup.

### 3.5.1 The Lower Bound

In this section, we show that any deterministic online algorithm (even with an arbitrary capacity blowup) for any uniform-encode instance of the compression caching problem is $\Omega(\log m)$-competitive.

For any given online algorithm ON with a capacity blowup $b$, we construct a uniform-encode instance of the compression caching problem. For any file $f$ and index $i < m-1$, we consider that $\mathit{size}(i+1, f) \leq \frac{1}{r} \cdot \mathit{size}(i, f)$, where $r > b$. For any file $f$ and index $i$ such that $0 < i < m-1$, we consider that $\mathit{decode}(i+1, f) \geq \mathit{decode}(i, f) \cdot \log m$. We also set the miss-penalty $p(f)$ to be $\mathit{encode}$, and $\mathit{encode} \geq \mathit{decode}(m-1, f) \cdot \log m$.

#### 3.5.1.1 Adversarial request generating algorithm

Our adversarial request generating algorithm ADV takes ON as input, and generates a request sequence $\sigma$ and an offline algorithm OFF such that ON incurs at least $\log m$ times the cost incurred by OFF in executing $\sigma$. For any file $f$, ADV maintains two indices denoted $w_u(f)$ and $w_\ell(f)$; initially,

$w_u(f) = m$ and $w_\ell(f) = 0$. The complete description of ADV is presented in Figure 3.5.

Roughly, ADV operates as follows. Algorithm ADV forces ON to do a search over the encodings of a file to find the encoding that OFF has chosen for that file. Before any request is generated, ADV ensures that for any $f$, there is no $i$-encoding of $f$ in ON's cache such that $w_\ell(f) \leq i < w_u(f)$. On a request for any file $f$, if ON adds the $i$-encoding of $f$ such that $w_\ell(f) \leq i < w_u(f)$, then ADV readjusts $w_\ell(f)$ and $w_u(f)$ to ensure that the above condition is satisfied. If ON does not keep the $i$-encoding of $f$ such that $i < w_u(f)$, then ADV continues to generate requests for $f$. Finally, when $w_u(f) = w_\ell(f)$, OFF claims that OFF has kept the $i$-encoding of $f$, where $i = w_\ell(f)$, throughout this process, and has executed the requests for $f$. Then, ADV resets the variables $w_u(f)$ and $w_\ell(f)$ to $m$ and 0, respectively, and OFF deletes the encoding of $f$ from its cache. In this process, OFF incurs encoding cost of adding only one encoding of file $f$. On the other hand ON incurs much higher cost than OFF because of adding multiple encodings of $f$.

#### 3.5.1.2 Analysis

We define an *epoch* for a file $f$ as follows. The first epoch starts when OFF executes the first request for $f$. An epoch for $f$ ends when $w_u(f)$ is equal to $w_\ell(f)$. A new epoch for $f$ starts when OFF executes the first request for $f$ after the previous epoch ended.

**Lemma 3.5.1.** *For any request, if* OFF *incurs decode cost $c$, then* ON *incurs*

```
{Initially, for any file $f$, $w_u(f) = m$ and $w_\ell(f) = 0$. }
1   $\sigma := \emptyset$
2   while $|\sigma| < N$ do
3     If there exists a file $f$ such that $f$ is present in OFF's cache, but no $i$-encoding
        of $f$ is present in ON's cache, where $i < w_u(f)$, then pick $f$
      Otherwise, pick a file $f$ such that $f$ is not present in ON's cache
4     append $\langle f \rangle$ to $\sigma$
5     OFF executes the request for $f$: if $f$ is not present in OFF's cache, then OFF
        adds $h_i(f)$, where $i$ is to be specified later (Line 16)
6     ON executes the request for $f$
7     while there exists a file $f'$ such that ON has stored the $i$-encoding of $f'$ and
          $w_\ell(f') \leq i < w_u(f')$ do
8       $mid = \lfloor (w_\ell(f') + w_u(f'))/2 \rfloor$
9       if $mid \leq i < w_u(f')$
10        $w_u(f') := mid$
11      else
12        $w_\ell(f') := mid$
13      fi
14    od
15    while there exists a file $f'$ such that $w_u(f') = w_\ell(f')$ do
16      OFF chooses the $w_\ell(f')$-encoding of $f'$, that is, in Line 5, $i$ is set to $w_\ell(f')$
17      $w_u(f') := m$; and $w_\ell(f') := 0$
18      OFF deletes the encoding of $f'$
19    od
20  od
```

Figure 3.5: The adversarial request generating algorithm to construct a uniform-encode instance of the compression caching problem. Here, $N$ is the number of requests to be generated.

*cost at least $c \cdot \log m$ to execute the request.*

*Proof.* In Lines 3 and 4, if ADV generates a request for a file $f$ such that $f$ is present in OFF's cache, but for any index $i < w_u(f)$, no $i$-encoding of $f$ is present in ON's cache, then the cost incurred by ON is at least $\log m$ times that incurred by OFF to execute the request.

Otherwise, if ADV generates a request for a file $f$ such that $f$ is in OFF's cache but not in ON's cache, then ON incurs miss penalty $p(f)$, and so the cost incurred by ON is at least $\log m$ times that incurred by OFF to execute the request. □

**Lemma 3.5.2.** *For any file $f$ and any epoch of $f$, the total miss penalty incurred by* OFF *on $f$ is $p(f)$ and the total encode cost of* OFF *on $f$ is encode.*

*Proof.* To execute the first request for $f$ in the epoch, OFF incurs the miss penalty $p(f)$ and adds an encoding of $f$ (Line 5). Since OFF removes the encoding of $f$ at the end of the epoch, in the rest of the epoch, OFF does not incur any miss penalty or encode cost on $f$. □

**Lemma 3.5.3.** *For any file $f$ and any epoch of $f$, if $w_u(f) - w_\ell(f) = \frac{m}{2^k}$, then the encode cost of* ON *on $f$ is at least $k$ times encode during the epoch.*

*Proof.* For any index $i$, if ON adds the $i$-encoding of $f$ to its cache such that $w_\ell(f) \le i < w_u(f)$, ADV reduces $(w_u(f) - w_\ell(f))$ by exactly half. Hence,

$w_u(f) - w_\ell(f)$ reduces to $\frac{m}{2^k}$ in the epoch only after addition of $k$ different encodings of $f$ by ON. □

**Lemma 3.5.4.** *For any file $f$ and any complete epoch of $f$, the total encode cost incurred by* ON *on $f$ in the epoch is at least* $\log m$ *times that by* OFF *on $f$ in the epoch.*

*Proof.* From Lemma 3.5.2, in any epoch of $f$, the total encode cost of OFF on $f$ is *encode*. On the other hand, from Lemma 3.5.3, the total encode cost of ON on $f$ is at least $\log m$ times *encode*. □

**Lemma 3.5.5.** *The total number of incomplete epochs is at most $k$, where $k$ is the number of $(m-1)$-encodings of files that can fit in* ON*'s cache.*

*Proof.* The lemma follows from the fact that the maximum number of encodings of files ON can have in the cache is $k$. □

**Theorem 6.** *Any deterministic online algorithm with an arbitrary factor capacity blowup is $\Omega(\log m)$-competitive for any uniform-encode instance of the compression caching problem.*

*Proof.* Let $n$ be the number of complete epochs and $k$ be the number of incomplete epochs. Let $D$ be the total decode cost of OFF.

From Lemma 3.5.2, the total cost of OFF is at most $(n + k) \cdot (p + \mathit{encode}) + D$.

By Lemma 3.5.1 and 3.5.4, the total cost of ON is at least $np + kp + n \cdot \mathit{encode} \cdot \log m + D \log m$.

Hence, the ratio of the total cost of ON and that of OFF is at least

$$\begin{aligned}
& \frac{np + kp + n \cdot \mathit{encode} \cdot \log m + D \log m}{(n+k)\cdot(p + \mathit{encode}) + D} \\
= \; & \frac{n + k + (n + \frac{D}{\mathit{encode}}) \log m}{n + k + \frac{D}{\mathit{encode}}} \\
= \; & \frac{\frac{n+k}{n+\frac{D}{\mathit{encode}}} + \log m}{\frac{k}{n+\frac{D}{\mathit{encode}}} + 1} \\
= \; & \Omega(\log m)
\end{aligned}$$

(The first equality follows since $p(f) = \mathit{encode}$. The last equality follows from Lemma 3.5.5 and the fact that by generating a sufficiently large number of requests, $n + \frac{D}{\mathit{encode}}$ can be made much larger than $k$.) □

### 3.5.2 An Upper Bound

In this section we present $O(m)$-competitive online algorithm ON with $O(m)$ factor capacity blowup for any uniform-encode instance $I$ of the compression caching problem, where $m = \mathit{numindex}(I)$.

As in Section 3.3.1, by preprocessing, we can arrange the encodings of the files in such a way that sizes are decreasing and decode costs are increasing. In other words, after preprocessing, for any file $f$ and index $i < m-1$, $\mathit{size}(i+1, f) < \mathit{size}(i, f)$, and $\mathit{decode}(i+1, f) > \mathit{decode}(i, f)$. Recall that for any file

$f$, $decode(m-1, f) < p(f)$.

#### 3.5.2.1 Algorithm

For any uniform-encode instance $I = (\sigma, m, k)$, the online algorithm ON is given a $2bm$ factor capacity blowup, where $b$ is at least $1+\epsilon$ for some constant $\epsilon > 0$. We divide ON's cache into $2m$ blocks, denoted $i$-*left* and $i$-*right*, $0 \leq i < m$, such that the capacity of each block is $bk$. For any index $i$, $i$-*left* keeps only the $i$-encodings of files, and $i$-*right* keeps only the 0-encodings of files. For any file $f$ and index $i$, we maintain an associated value $charge(f, i)$. Roughly, whenever the cost incurred in miss penalties or decode costs on a file $f$ exceeds *encode*, then ON adds an encoding of the file that is cheaper in terms of the access cost than the current encoding (if any) of $f$.

The complete description of algorithm ON is given in Figure 3.6.

#### 3.5.2.2 Analysis

**Lemma 3.5.6.** *Consider any uniform-encode instance* $I = (\sigma, m, k)$. *Let* $T = \text{ON}(\sigma, m, k)$. *Then,* $cost(T)$ *is at most twice that of the total increase in* $\sum_{f,i} charge(f, i)$ *during the execution of* ON.

*Proof.* From the description of the algorithm, if ON incurs a decode cost or miss penalty $c$, then the increase in $\sum_{f,i} charge(f, i)$ is at least $c$. Further, for any file $f$ and index $i$, ON adds an encoding $h_i(f)$ only when $charge(f, i)$ is at least *encode*. Hence, the lemma follows. $\square$

For any trace $T$, any file $f$, and any interval $(i, j)$ such that $j \le |T|$, we define $\mathit{cost}_f(T, (i, j))$ as $\mathit{cost}_f(T', S)$, where $T'$ is equal to the subsequence of $T$ starting from $T_i$ and ending after $T_{j-1}$, and $S$ is equal to *configuration*$(T_{i-1})$. We define $\mathit{cost}(T, (i, j))$ as the sum, over all files $f$, of $\mathit{cost}_f(T, (i, j))$.

For any trace $T$, any file $f$, any index $\ell$, and any interval $(i, j)$ such that $0 \le i$ and $j \le |T|$, *added*$(T, f, \ell, (i, j))$ holds if, for some $j'$ such that $i \le j' < j$, one of the following conditions hold: (1) $j' = 0$ and, for some $\ell' < \ell$, $\ell'$-encoding of $f$ is present in *configuration*$(T_0)$; (2) for all $\ell' \le \ell$, $\ell'$-encoding of $f$ is not present in $T_{j'-1}$, while for some $\ell'' \le \ell$, $\ell''$-encoding of $f$ is present in *configuration*$(T_{j'})$.

For any trace $T$, any file $f$, any index $\ell$, and any interval $(i, j)$ such that $0 \le i$ and $j \le |T|$, *deleted*$(T, f, \ell, (i, j))$ holds if, for some $j' > 0$ such that $i \le j' < j$, for all $\ell' < \ell$, the $\ell'$-encoding of $f$ is not present in *configuration*$(T_{j'})$ while for some $\ell'' < \ell$, the $\ell''$-encoding of $f$ is present in *configuration*$(T_{(j'-1)})$.

For any trace $T$, any file $f$, any index $\ell$, and any interval $(i, j)$ such that $0 \le i$ and $j \le |T|$, *throughout*$(T, f, \ell, (i, j))$ holds if $i > 0$ and for all $j'$ in $[i - 1, j]$, there exists some $\ell' < \ell$ such that $\ell'$-encoding of $f$ is present in *configuration*$(T_{j'})$.

**Lemma 3.5.7.** *Consider any uniform-encode instance $I = (\sigma, m, k)$. Let $T$ be any trace such that reqseq$(T) = \sigma$ and space$(T) \le k$. Let $T' = \mathrm{ON}(\sigma, m, 2bmk)$,* *where* $b > (1+\epsilon)$ *for some constant* $\epsilon$ *greater than* $0$*. Then, during the execution of* ON*, the total increase in* $\sum_{f,i}$ *charge*$(f,i)$ *is* $O(m) \cdot$ *cost*$(T)$*.*

*Proof.* Consider the execution of $T' = \mathrm{ON}(\sigma, m, 2bmk)$. For any index $\ell$, we define an *$\ell$-superepoch* (respectively, *$\ell$-epoch*) as follows. The first $\ell$-superepoch (resp., $\ell$-epoch) starts at the beginning of the request sequence $\sigma$. Any $\ell$-superepoch (resp., $\ell$-epoch) ends if the request sequence $\sigma$ ends or just before a request such that, during the execution of the request, *$\ell$-left* (resp., *$\ell$-right*) is flushed in Line 30 (resp., Line 17 or Line 30) in Figure 3.6. The next $\ell$-superepoch (resp., $\ell$-epoch) starts when the previous $\ell$-superepoch (resp., $\ell$-epoch) ends. Any $\ell$-superepoch is specified by an interval $(i,j)$ such that the superepoch starts just before $\sigma_i$ and ends just after $\sigma_{j-1}$. For any index $\ell$ and any $\ell$-superepoch $(i,j)$, we define *increase*(*charge*$(f,\ell), (i,j))$ as the total increase in *charge*$(f,\ell)$ during superepoch $(i,j)$.

The lemma is immediate from the following claim.

Claim: For any index $\ell$, the sum, over all files $f$ and all $\ell$-superepochs $(i,j)$, of *increase*(*charge*$(f,\ell), (i,j))$ is at most $O(1)$ times the sum, over all $\ell$-superepochs $(i,j)$, of *cost*$(T,(i,j))$.

The proof of the claim is as follows. In the following we fix an $\ell$-superepoch $(i,j)$.

An $\ell$-epoch is defined to be *complete* if the epoch ends due to the execution of Line 30; otherwise, the epoch is *incomplete*. An $\ell$-epoch $(i',j')$ is *enclosed* in the superepoch $(i,j)$ if $i \le i'$ and $j' \le j$. Let $k_\ell$ be the maximum number of $\ell$-encodings of files that can fit in any block. Let the number of complete $\ell$-epochs enclosed in the superepoch be $n$. Note that $n$ is at least 1 since $k_\ell$ is at least $k_0$. Also, since the superepoch ends when an incomplete epoch ends, the number of enclosed incomplete $\ell$-epochs is at most 1.

First, we show that $\sum_f \mathit{increase}(\mathit{charge}(f,\ell),(i,j))$ is $\Theta(n \cdot k_0 \cdot \mathit{decode}(\ell + 1))$. Consider any complete $\ell$-epoch $(i',j')$ enclosed in the superepoch. For any file $f$, since ON increases $\mathit{charge}(f,\ell)$ by $\mathit{decode}(\ell+1)$ when ON adds $f$ to $\ell$-right, and the $\ell$-epoch ends after addition of $k_0$ files to *$\ell$-right*, the sum over all $f$, of $\mathit{increase}(\mathit{charge}(f,\ell),(i',j'))$ is at most $k_0 \cdot \mathit{decode}(\ell+1)$. Since the number of incomplete $\ell$-epochs enclosed in the superepoch is at most 1, the following inequalities hold.

$$\begin{aligned} n \cdot k_0 \cdot \mathit{decode}(\ell+1) &\leq \sum_f \mathit{increase}(\mathit{charge}(f,\ell),(i,j)) && (3.1)\\ &\leq (n+1)\cdot k_0 \cdot \mathit{decode}(\ell+1) && (3.2)\end{aligned}$$

Next we show that $\mathit{cost}(T,(i,j))$ is $\Omega(n \cdot k_0 \cdot \mathit{decode}(\ell+1))$.

For any file $f$, ON adds the $\ell$-encoding of $f$ to $\ell$-left when $\mathit{charge}(f,i)$ is at least *encode*. Hence,

$$\mathit{increase}(\mathit{charge}(f,\ell),(i,j)) < \mathit{encode} + \mathit{decode}(\ell+1) \qquad (3.3)$$

For any file $f$ such that $\mathit{added}(T,f,\ell,(i,j))$ holds, $\mathit{cost}_f(T,(i,j))$ is at least $\mathit{encode}+\mathit{decode}(\ell+1)$, which is more than $\mathit{increase}(\mathit{charge}(f,\ell),(i,j))$, as argued above in Equation 3.3. Note that there could be at most one deletion per addition of any encoding. For any file $f$ such that *deleted*$(T, f, \ell, (i, j))$ holds, we can charge addition twice, and attribute *encode* + *decode*$(\ell + 1)$ to deletion, which is more than *increase*(*charge*$(f, \ell), (i, j))$, as argued above in Equation 3.3. For any file $f$ with $x$ requests during the superepoch, the value of *increase*(*charge*$(f, \ell), (i, j))$ is at most $x \cdot$ *decode*$(\ell + 1)$ since on each request *charge*$(f, \ell)$ increases by at most *decode*$(\ell + 1)$. For any file $f$, if *added*$(T, f, \ell, (i, j))$ does not hold, or *deleted*$(T, f, \ell, (i, j))$ does not hold, or *throughout*$(T, f, \ell, (i, j))$ does not hold, then $cost_f(T, (i, j))$ is at least $x \cdot$ *decode*$(\ell + 1)$, where $x$ is the number of requests for $f$ during the superepoch, and the latter quantity is at least *increase*(*charge*$(f, \ell), (i, j))$. Let $X$ be a set of files $f$ such that *throughout*$(T, f, \ell, (i, j))$ holds. Then the following inequality holds.

$$cost(T, (i, j)) \;\; \geq \;\; \sum_{f \notin X} increase(charge(f, \ell), (i, j)) \tag{3.4}$$

Since any $\ell$-superepoch ends after addition of $k_\ell$ files to $\ell$-*left* during the superepoch, the following holds.

$$\sum_{f} increase(charge(f, \ell), (i, j)) \geq k_\ell \cdot (encode + decode(\ell + 1)) \tag{3.5}$$

From Equations 3.2 and 3.5,

$$(n + 1) \cdot k_0 \cdot decode(\ell + 1) \geq k_\ell \cdot (encode + decode(\ell + 1)) \tag{3.6}$$

Let $X$ be a set of files $f$ such that *throughout*$(T, f, \ell, (i, j))$ holds. Since *space*$(T) \leq k$ and the capacity of any block is $bk$, the following equation holds.

$$|X| \leq \frac{k_\ell}{b} \tag{3.7}$$

From the above,

$$\begin{aligned}
cost(T,(i,j)) &\geq \sum_{f \notin X} increase(charge(f,\ell),(i,j)) \\
&\geq \sum_{\forall f} increase(charge(f,\ell),(i,j)) \\
&\qquad - \sum_{f \in X} increase(charge(f,\ell),(i,j)) \\
&\geq n \cdot k_0 \cdot decode(\ell+1) - \\
&\qquad - \sum_{f \in X} increase(charge(f,\ell),(i,j)) \\
&\geq n \cdot k_0 \cdot decode(\ell+1) - \frac{k_\ell}{b}(encode + decode(\ell+1)) \\
&\geq n \cdot k_0 \cdot decode(\ell+1) - \frac{1}{b}(n+1) \cdot k_0 \cdot decode(\ell+1) \\
&\geq \Omega(n \cdot k_0 \cdot decode(\ell+1))
\end{aligned}$$

(In the above equation, the first inequality follows from Equation 3.4. The second inequality is straightforward. The third inequality follows from Equation 3.1. The fourth inequality follows from Equation 3.3 and Equation 3.7. The fifth inequality follows from Equation 3.5. The sixth inequality follows from the fact that $b > 1 + \epsilon$, and by setting $\epsilon$ appropriately.)

Summing over all $\ell$-superepochs $(i,j)$, the claim follows. The lemma is immediate from the claim. □

**Lemma 3.5.8.** *Consider any uniform-encode instance $I = (\sigma, m, k)$. Let $T$ be any trace such that $reqseq(T) = \sigma$ and $space(T) \leq k$. Let $T' = \mathrm{ON}(\sigma, m, 2bmk)$, where $b > (1+\epsilon)$ for some constant $\epsilon$ greater than 0. Then $cost(T') = O(m) \cdot cost(T)$.*

*Proof.* Follows from Lemmas 3.5.6 and 3.5.7. □

The following theorem is immediate from Lemma 3.5.8.

**Theorem 7.** *For any uniform-encode instance $I$ of the compression caching problem, there exists an online algorithm that is is $O(m)$-competitive with $O(m)$ factor capacity blowup, where $m = numindex(I)$.*

```
ON($\sigma, m, k$)
begin
  $T := \emptyset$
  {Initially, for any file $f'$ and index $i$, $charge(f', i) := 0$}
  On an access for a file $f$
  if for any index $i$, $f$ is present in $i$-right then
    serve $f$
  else
    if $f$ is not present in the cache then
      $charge(f, m-1) := charge(f, m-1) + p(f)$
      $i := m$
    else
      let $i$ be the smallest index such that $f$ is in $i$-left
    fi
    foreach index $j < i$ do
      if $j < m-1$ then
        $charge(f, j) := charge(f, j) + decode(j+1)$ fi
      if $j$-right is full then flush $j$-right
      bring $f$ to $j$-right
      if $charge(f, j) \geq encode$ then
        $add(f, j)$ fi
    od
  fi
  Let $S$ be the set of encodings present in the cache
  $T := T \circ S$
  return $T$
end
$add$(file $f'$, int $j$)
begin
  if $j$-left is full then
    flush $j$-left and $j$-right
    for any file $f$, $charge(f, j) := 0$
  fi
  add $h_j(f')$ to $j$-left
  $charge(f', j) := 0$
end
```

Figure 3.6: An online algorithm for the uniform-encode compression caching problem.

# Chapter 4

# Online Aggregation over Trees

## 4.1 Introduction

Information aggregation is a basic building block in many large-scale distributed applications such as system management [25, 41], service placement [24, 44], file location [12], grid resource monitoring [18], network monitoring [29], and collecting readings from sensors [32]. Certain generic aggregation frameworks [18, 37, 45] proposed for building such distributed applications allow scalable information aggregation by forming tree-like structures with machines as nodes, and by using an aggregation function at each node to summarize the information from the nodes in the associated subtree.

Some of the existing aggregation frameworks use strategies optimized for certain workloads. For example, in MDS-2 [18], the information is aggregated only on reads, and no aggregation is performed on writes. This kind of strategy performs well for write-dominated workloads, but suffers from unnecessary latency or imprecision on read-dominated workloads. On the other hand, Astrolabe [37] employs the other extreme form of strategy in which, on a write at a node $u$ in the tree, each node $v$ on the path from $u$ to the root node recomputes the aggregate value for the subtree rooted at node $v$, and the new aggregate values are propagated to all the nodes. This kind of strategy performs well for read-dominated workloads, but consumes high bandwidth when applied to write-dominated workloads. Furthermore, instead of these two extreme forms of workloads, the workload may fluctuate and different nodes may exhibit activity at different times. Therefore, a natural question to ask is whether one can design an adaptive aggregation strategy that works well for varying workloads.

SDIMS [45] proposes a hierarchical aggregation framework with a flexible API that allows applications to control the update propagation, and hence, the aggregation aggressiveness of the system. Though SDIMS exposes such flexibility to applications, it requires applications to know the read and write access patterns a priori to choose an appropriate strategy (see our discussion on related work for further details). Thus, SDIMS leaves an open question of how to adapt the aggregation strategy in an online manner as the workload fluctuates.

In this work, we design an online aggregation algorithm, and show that the total number of messages required to execute a given set of requests is within a constant factor of the minimum number of messages required to execute the requests. We give the complete algorithm description in the abstract protocol notation [26], and also believe that our algorithm is practical.

**Broader Perspective**. The ever increasing complexity of developing large-scale distributed applications motivates a research agenda based on the identification of key distributed primitives, and the design of reusable modules for such primitives. To promote reuse, these modules should be "self-tuning", that is, should provide near optimal performance under a wide range of operating conditions. As indicated earlier, aggregation is useful in many applications. In the present we design a distributed protocol for aggregation that provides good performance guarantees under any operating conditions. Our focus on tree networks is not limiting since many large-scale distributed applications tend to be hierarchical (tree-like) in nature for scalability. If the network is not a tree, one can use standard techniques to build a spanning tree. For example, in SDIMS [45], nodes are arranged in a distributed hash table (DHT), and trees embedded in the DHT are used for the aggregation; these trees are automatically repaired in the face of failures. The present work can be viewed as a case study within the broader research agenda alluded to above. The techniques developed here may find application in the design of self-tuning modules for other primitives.

**Problem Formulation**. In order to describe our results we next present a brief description of the problem formulation; see Section 4.2 for a detailed description. We consider a distributed network with nodes arranged in an unrooted tree and each node having a local value. We formulate the aggregation problem as the problem of aggregating values (e.g., computing min, max, sum, or average) from all the nodes to the requesting nodes in the presence of writes. The goal is to minimize the total number of messages exchanged.

The main challenges are to define acceptable aggregate values in the presence of concurrent requests, and to design algorithms with good performance that produce acceptable aggregate values. We define the acceptability of aggregate values in terms of certain consistency guarantees. There is a spectrum of solutions that trade off between consistency and performance. We introduce a mechanism that uses the concept of leases for aggregation algorithms. Any aggregation algorithm that uses this mechanism is called a lease-based aggregation algorithm. The notion of a lease used in our mechanism is a generalization of that used in SDIMS [45].

**Results**. We evaluate lease-based aggregation algorithms in terms of consistency and performance. In terms of consistency, we generalize the notions of strict and causal consistency, traditionally defined for distributed shared memory [40, Chapter 6], for the aggregation problem. We show that any lease-based aggregation algorithm provides strict consistency for sequential executions, and causal consistency for concurrent executions.

In terms of performance, we analyze lease-based algorithms in the framework of competitive analysis [39]. As is typical in the competitive analysis of distributed algorithms [6, 7], we focus on sequential executions. In this chapter we present an online lease-based aggregation algorithm RWW which, for sequential executions, is $\frac{5}{2}$-competitive against an optimal offline lease-based aggregation algorithm. We use a potential function argument to show this result. We also show that the result is tight by providing a matching lower bound. Further, we show that, for sequential executions, RWW is 5-competitive against an optimal offline algorithm that provides strict consistency.

The three main contributions of the work are as follows. First, we design an online aggregation algorithm and show that our algorithm achieves a good competitive ratio for sequential executions. Second, we define the notion of causal consistency for the aggregation problem. Third, we show that our algorithm satisfies the definition of causal consistency for concurrent executions.

An interesting highlight of the techniques is the reduction of the analysis to reasoning about a pair of neighboring nodes. This reduction allows us to formulate a linear program of small size, independent of tree size, for the analysis.

**Related Work**. Various aggregation frameworks have been proposed in the literature such as SDIMS [45], Astrolabe [37], and MDS [18]. SDIMS is a hierarchical aggregation framework that utilizes DHT trees to aggregate values. SDIMS provides a flexible API that allows applications to decide how far to propagate updates to the aggregate value due to writes. In particular, SDIMS supports *Update-local*, *Update-all*, and *Update-up* strategies. In the Update-local strategy, a write affects only the local value. In the Update-all strategy, on a write, the new aggregate value is propagated to all the nodes. In the Update-up strategy, on a write, the new aggregate value is propagated to the root node of the hierarchy. Astrolabe is an information management system that builds a single logical aggregation tree over a given set of nodes. Astrolabe propagates all updates to the aggregate value due to writes to all the nodes, and hence, allows all reads to be satisfied locally. MDS-2 also forms a spanning tree over all nodes. MDS-2 does not propagate updates on writes, and each request for an aggregate value requires all nodes to be contacted.

There are some similarities between our lease-based aggregation algorithm and prior caching work. Here we describe some of the most relevant work. In CUP [38], Roussopoulos and Baker propose a *second-chance* algorithm for caching objects along the routing path. The algorithm removes a cached object after two consecutive updates are propagated to remote locations due to writes on that object at the source. The second-chance algorithm has been shown to provide good performance in an experimental evaluation. In a work related to distributed file allocation [7], Awerbuch et al. present a replication algorithm for a general network. In their algorithm, on a read, the requested object is replicated along a path from the destination to the requesting node. On a write, all copies are deleted except the one at the writing node. Awerbuch et al. show that their distributed algorithm achieves a poly-logarithmic competitive ratio for the distributed caching problem against an optimal centralized offline algorithm.

The concept of time-based leases has been proposed in the literature as a way to maintain consistency between a cached copy and the source. Such leases have been applied in many distributed applications such as replicated file systems [27] and web caching [20].

Ahamad et al. [3] gave the formal definition of causal consistency for a distributed message passing system. The key difference between their setup and ours is in reading one value compared to aggregating values from all the nodes.

There are several efforts to deal with numerical error in the aggregate value such as [11, 34]. However, to the best of our knowledge, no prior work yields a competitive online algorithm for the aggregation problem, and no prior work addresses the issue of ordering semantics in concurrent executions. In [11], Bawa et al. define a semantic for various scenarios such as approximate aggregation in a faulty environment, and called this semantic *approximate single-site validity*. They design algorithms that provide such a semantic, and evaluate their algorithms experimentally. In [34], Olston and Widom consider a single level hierarchy and propose a new class of replication systems, called TRAPP, that allows the user to control the tradeoff between precision (numerical error) and communication overhead.

**Organization**. In Section 4.2 we introduce various definitions. In Section 4.3 we give an informal description of our algorithm and analysis. In Section 4.4 we define the class of lease-based aggregation algorithms, and establish certain properties of such algorithms. In Section 4.5 we present our online lease-based aggregation algorithm RWW, and establish bounds on the competitive ratio of RWW for sequential executions. In Section 4.6 we define the notion of a causally consistent aggregation algorithm, and establish that any lease-based algorithm, including RWW, is causally consistent.

## 4.2 Preliminaries

Assume that we are given a finite set of nodes (i.e., machines) arranged in a tree network $T$ with reliable FIFO communication channels between neighboring nodes. We are also given an aggregation operator $\oplus$ that is commutative, associative, and has an identity element 0. We find it convenient to write an expression of the form $x \oplus y \oplus z$ as $\oplus(x, y, z)$. For the sake of concreteness, throughout this chapter, we assume that the local value associated with each node is a real value, and the domain of $\oplus$ is also real.

The *aggregate value* over a set of nodes is defined as $\oplus$ computed over the local values of all nodes in the set. That is, the aggregate value over a set of nodes $\{v_1, \ldots, v_k\}$ is $\oplus(v_1.val, \ldots, v_k.val)$, where $v_i.val$ is the local value of the node $v_i$. The *global aggregate value* is defined as the aggregate value over all nodes in the tree $T$.

A request is a tuple (*node*, *op*, *arg*, *retval*), where *node* is the node where the request is initiated, *op* is the type of the request, either *combine* or *write*, *arg* is the argument of the request (if any), and *retval* is the return value of the request (if any). To execute a *write* request, an aggregation algorithm takes the argument of the request and updates the local value at the requesting node. To execute a *combine* request, an aggregation algorithm returns a value. Note that this definition admits the trivial algorithm that returns 0 on any *combine* request. We define certain correctness criteria for aggregation algorithms later in the chapter. Roughly speaking, the returned value on a *combine* request corresponds to the global aggregate value.

The *aggregation problem* is to execute a given sequence of requests with the goal of minimizing the total number of messages exchanged among nodes. For any aggregation algorithm $A$, and any request sequence $\tau$, we define $C_A(\tau)$ as the total number of messages exchanged among nodes during the execution of $\tau$ by $A$. An online aggregation algorithm $A$ is $c$-competitive if for all request sequences $\tau$ and an optimal offline aggregation algorithm $B$, $C_A(\tau) \leq c \cdot C_B(\tau)$ [13, Chapter 1].

We say that $T$ is in a quiescent state if (1) there is no pending request at any node; (2) there is no message in transit across any edge; and (3) no message is sent until the next request is initiated. In short, $T$ is in a quiescent state if there is no activity in $T$ until the next request is initiated.

In a sequential execution of a request, the request is initiated in a quiescent state and is completed when $T$ reaches another quiescent state. In a sequential execution of a request sequence $\sigma$, every request $q$ in $\sigma$ is executed sequentially. In a concurrent execution of a request sequence, a new request can be initiated and executed while another request is being executed. We refer to the aggregation problem in which the given request sequence is executed sequentially as the *sequential aggregation problem*.

The aggregation function, denoted $f$, is defined over a set of real values or over a set of write requests. For a set $A$ of real values $x_1, \ldots, x_m$, $f(A)$ is defined as $\oplus(x_1, \ldots, x_m)$. For a set $A$ of write requests $q_1, \ldots, q_m$, $f(A)$ is defined as $\oplus(q_1.arg, \ldots, q_m.arg)$.

For any request $q$ in a request sequence $\sigma$, let $A(\sigma, q)$ be the set of the most recent writes preceding $q$ in $\sigma$ corresponding to each of the nodes in $T$. We say that an aggregation algorithm provides *strict consistency* in executing $\sigma$ if any *combine* request $q$ in $\sigma$ returns $f(A(\sigma, q))$ as the global aggregate value at $q.node$. Note that this definition of strict consistency for an aggregation algorithm is a generalization of the traditional definition of strict consistency for distributed shared memory systems (for further details, see [40, Chapter 6]). We define an aggregation algorithm to be *nice* if the algorithm provides strict consistency for sequential executions.

The set of all nodes in tree $T$ is denoted by *nodes*$(T)$. For any edge $(u, v)$ in $T$, removal of $(u, v)$ yields two trees, *subtree*$(u, v)$ is defined to be one of the trees that contains $u$.

For any request sequence $\sigma$ and any ordered pair of neighboring nodes $(u, v)$, we define $\sigma(u, v)$ as follows: (1) $\sigma(u, v)$ is a subsequence of $\sigma$; (2) for any *write* request $q$ in $\sigma$ such that $q.node$ is in *subtree*$(u, v)$, $q$ is in $\sigma(u, v)$; and (3) for any *combine* request $q$ in $\sigma$ such that $q.node$ is in *subtree*$(v, u)$, $q$ is in $\sigma(u, v)$.

## 4.3 Informal Overview

In this section we present an informal overview of our algorithm and analysis.

Recall that on a combine request at a node $u$, $u$ returns a value.

Roughly speaking, the value corresponds to the global aggregate value. In order to do that, $u$ contacts other nodes and collects the local values from all other nodes. Note that we can minimize the number of messages by performing aggregation at intermediate nodes, also referred as in-network aggregation.

However, for a combine-dominated workload, one may wish to propagate an updated local value on a write request to minimize the number of messages exchanged on a combine. On the other hand, for a write-dominated workload, such propagation tends to be wasteful. In order to facilitate adaptation of how many messages to send on a combine request versus a write request, we propose a lease mechanism. Here, we illustrate our lease mechanism for just two nodes $u$ and $v$ connected by an edge, and a scenario in which combine requests are initiated at $v$ and write requests are initiated at $u$. (See Section 4.4 for a complete description of the mechanism.)

If the lease from $u$ to $v$ is present, then on a write request at $u$, $u$ propagates the new local value to $v$ by sending an update message. Hence, in the presence of this lease, a combine request at $v$ is executed locally. On the other hand, if the lease from $u$ to $v$ is not present, then on a combine request at $v$, a probe message is sent from $v$ to $u$. As a result, a response message containing the local value at $u$ is sent from $u$ to $v$. Further, in this case, a write request at $u$ is executed locally. Note that in a combine-dominated scenario, presence of the lease is beneficial. However, in a write-dominated scenario, $v$ may receive many updates while $v$ is not initiating any request. In that case, $v$ can break the lease by sending a release message to $u$.

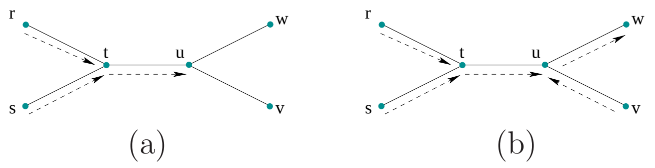

Figure 4.1: An example tree network.

In order to make the lease mechanism work for a tree network in a desirable way, we enforce two lease invariants. Consider the tree network in Figure 4.1 as an example. The presence of a lease on an edge is denoted by a dotted line. To motivate the first invariant, consider a combine request $q$ at node $w$ with leases as in Figure 4.1(a). During the execution of $q$, $w$ sends messages and collects the local values from all the other nodes. If the lease from $t$ to $u$ is present, then $u$ need not send any message to $t$. However, this works only if $t$ has leases from $r$ and $s$. Our first invariant requires that the lease from $t$ to $u$ is not set unless $t$ has leases from all the other neighboring nodes. Our second invariant requires that the lease from $t$ to $u$ cannot be broken if $u$ has given a lease to any other neighboring node, say node $w$ in Figure 4.1(b).

Given this lease mechanism, an aggregation algorithm can adapt how far an updated value should be propagated on a write request by setting and breaking leases appropriately. The next question is how to set and break the leases dynamically in an optimal manner. We answer this question by providing an online lease-based aggregation algorithm RWW (see Section 4.5). Roughly, RWW works as follows. For an edge $(u, v)$, RWW sets the lease from $u$ to $v$ during the execution of a combine request at any node in $subtree(v, u)$, and breaks the lease after two consecutive write requests at any node in *subtree*$(u, v)$. Using a potential function argument, we show that RWW is $\frac{5}{2}$-competitive against any offline lease-based algorithm for sequential executions. We also show that this bound is tight by providing a lower bound argument. Further, we show that RWW is 5-competitive against any offline algorithm that provides strict consistency for sequential executions.

With respect to consistency guarantees, we show that any lease-based aggregation algorithm provides strict consistency for sequential executions. For concurrent executions, it is difficult to provide strict or sequential consistency. Causal consistency is considered to be the next weaker consistency model for the distributed shared memory environment [40, Chapter 6]. At first, it is not clear how to generalize the causal consistency definitions for the aggregation problem.

We define causal consistency for the aggregation problem and show that any lease-based algorithm provides causal consistency for concurrent executions (see Section 4.6). First, we introduce a new type of ghost request *gather* to associate a combine request with a set of write requests. The concept of a gather request is similar to the way of associating a read request with a unique write request in analyzing distributed shared memory [3, 33]. Second, we define causal ordering among gather and write requests. Third, we extend the lease-based mechanism by adding ghost variables and ghost actions. Finally, we use an invariant style proof technique to show that any lease-based algorithm provides causal consistency. First, we show that a ghost log maintained at each node, containing gather and write requests, respects causal ordering among requests. Second, we show that there is one-to-one correspondence between gather and combine requests, such that the return value of a combine request is same as aggregation function computed over the set of write requests returned by the corresponding gather request.

## 4.4 Lease-Based Algorithms

In Section 4.3 we gave a high level description of an aggregation mechanism based on the concept of leases. See Figure 4.2 for the formal description of this mechanism. The underlined function calls represent stubs for policy decisions related to lease setting and breaking. Any policy function has read-only access to the mechanism variables. Throughout the remainder of this chapter, any aggregation algorithm that uses this mechanism and defines the policy functions is said to be *lease-based*.

The status of the leases for an edge $(u, v)$ is given by two boolean variables $u.taken[v]$ and $u.granted[v]$. Informally, Node $u$ believes that the lease from $v$ to $u$ is set if and only if $u.taken[v]$ holds. Also, $u$ believes that the lease from $u$ to $v$ is set if and only if $u.granted[v]$ holds. The local value at $u$ is stored in $u.val$. For each neighbor $v_i$ of $u$, $u.aval[v_i]$ represents the aggregate value computed over the set of nodes in $subtree(v_i, u)$. The following kinds of messages are sent by a lease-based algorithm: *probe*, *response*, *update*, and *release*.

The variable *sntupdates* is a set of tuples, where each tuple represents

```
      $node\ u$
      var $taken[]$ : array$[v_1,\ldots,v_k]$ of boolean;
        $granted[]$ : array$[v_1,\ldots,v_k]$ of boolean;
        $aval[]$ : array$[v_1,\ldots,v_k]$ of real;  $val$ : real;
        $uaw[]$ : array$[v_1,\ldots,v_k]$ of set {int};
        $pndg$ : set $\{node\}$;
        $snt[]$ : array$[v_1,\ldots,v_k]$ of set $\{node\}$;
        $upcntr$ : int; $sntupdates$ : set $\{\{node,$ int, int$\}\}$;
      init $val := 0; uaw := \emptyset; pndg := \emptyset;\ upcntr := 0;$
        $sntupdates := \emptyset; \forall v \in nbrs(),\ taken[v] :=$ false;
        $granted[v] :=$ false; $aval[v] := 0; snt[v] := \emptyset;$
      begin
$T_1$     true $\rightarrow \{combine\}$
 1          $\underline{oncombine}(u)$;
 2          foreach $v \in tkn()$ do
 3            $uaw[v] := \emptyset$; od
 4          if $u \notin pndg \rightarrow$
 5            if $nbrs() \setminus tkn() = \emptyset \rightarrow$
 6              return $gval()$;
 7            $\Box$ $nbrs() \setminus tkn() \neq \emptyset \rightarrow$
 8              $sendprobes(u)$;
 9              $snt[u] := nbrs() \setminus tkn()$; fi fi
$T_2$     true $\rightarrow \{write\ q\}$
 1          $val := q.arg$;
 2          if $grntd() \neq \emptyset\ \rightarrow$
 3            $id := newid()$;
 4            $forwardupdates(u, id)$; fi
$T_3$     $\Box$ rcv $probe()$ from $w \rightarrow$
 1          $\underline{probercvd}(w)$;
 2          foreach $v \in tkn() \setminus \{w\}$ do
 3            $uaw[v] := \emptyset$; od
 4          if $w \notin pndg \rightarrow$
 5            if $nbrs() \setminus \{tkn() \cup \{w\}\} = \emptyset \rightarrow$
 6              $sendresponse(w)$;
 7            $\Box$ $nbrs() \setminus \{tkn() \cup \{w\}\} \neq \emptyset \rightarrow$
 8              $sendprobes(w)$;
 9              $snt[w] := nbrs() \setminus \{tkn() \cup \{w\}\}$; fi fi
```

```
$T_4$     $\Box$ rcv $response(x, flag)$ from $w\ \rightarrow$
 1          $\underline{responsercvd}(flag, w)$;
 2          $aval[w] := x$;
 3          $taken[w] := flag$;
 4          foreach $v \in pndg$ do
 5            $snt[v] := snt[v] \setminus \{w\}$;
 6            if $snt[v] = \emptyset \rightarrow$
 7              $pndg := pndg \setminus \{v\}$;
 8              if $v = u \rightarrow$
 9                return $gval()$;
10              $\Box$ $v \neq u \rightarrow$
11                $sendresponse(v)$; fi fi od
$T_5$     $\Box$ rcv $update(x, id)$ from $w\ \rightarrow$
 1          $\underline{updatercvd}(w)$;
 2          $aval[w] := x$;
 3          $uaw[w] := uaw[w] \cup id$;
 4          if $grntd() \setminus \{w\} \neq \emptyset \rightarrow$
 5            $nid = newid()$;
 6            $sntupdates := sntupdates \cup \{(w, id, nid)\}$;
 7            $forwardupdates(w, nid)$;
 8          $\Box$ $grntd() \setminus \{w\} = \emptyset \rightarrow$
 9            $forwardrelease()$; fi
$T_6$     $\Box$ rcv $release(S)$ from $w \rightarrow$
 1          $\underline{releasercvd}(w)$;
 2          $granted[w] :=$ false;
 3          $onrelease(w, S)$;
      end
```

Figure 4.2: The mechanism for any lease-based aggregation algorithm. Nodes $\{v_1,\ldots,v_k\}$ refer to the neighbors of node $u$.

```
procedure sendprobes(node w)
  $pndg := pndg \cup \{w\}$;
  foreach $v \in nbrs() \setminus \{tkn() \cup sntprobes() \cup \{w\}\}$ do
    send $probe()$ to $v$; od

procedure forwardupdates(node w, int id)
  foreach $v \in grntd() \setminus \{w\}$ do
    send $update(subval(v), id)$ to $v$; od

procedure sendresponse(node w)
  if $(nbrs() \setminus \{tkn() \cup \{w\}\} = \emptyset) \rightarrow$
    $granted[w] := \underline{setlease}(w)$; fi
  send $response(subval(w), granted[w])$ to $w$;

boolean isgoodforrelease(node w)
  return $(grntd() \setminus \{w\} = \emptyset)$;

procedure onrelease(node w, set S)
  Let $id$ be the smallest id in $S$;
  foreach $v \in tkn() \setminus \{w\}$ do
    Let $A$ be the set of tuples $\alpha$ in $sntupdates$
      such that $\alpha.node = v$ and $\alpha.sntid \geq id$;
    Let $\beta$ be a tuple in $A$
      such that $\beta.rcvid \leq \alpha.rcvid$, for all $\alpha$ in $A$;
    Let $S'$ be the set of ids in $uaw[v]$ with ids $\geq \beta.rcvid$;
    $uaw[v] := S'$;
    if $isgoodforrelease(v) \rightarrow$
      $\underline{releasepolicy}(v)$;
    fi
  od
  $forwardrelease()$;
```

```
procedure forwardrelease()
  foreach $v \in tkn()$ do
    if $isgoodforrelease(v) \rightarrow$
      if $taken[v] \wedge \underline{breaklease}(v) \rightarrow$
        $taken[v] :=$ false;
        send $release(uaw[v])$ to $v$;
        $uaw[v] := \emptyset$; fi fi od

int newid()
  $upcntr := upcntr + 1$;
  return $upcntr$;

real gval()
  $x := val$;
  foreach $v \in nbrs()$ do
    $x := f(x, aval[v])$; od
  return $x$;

real subval(node w)
  $x := val$;
  foreach $v \in nbrs() \setminus \{w\}$ do
    $x := f(x, aval[v])$; od
  return $x$;

set nbrs()
  return the set of neighboring nodes;
set tkn()
  return $\{v \mid v \in nbrs() \wedge taken[v] = \textbf{true}\}$;
set grntd()
  return $\{v \mid v \in nbrs() \wedge granted[v] = \textbf{true}\}$;
set sntprobes()
  return $\{snt[v_1] \cup \cdots \cup snt[v_k]\}$;
```

Figure 4.3: Procedures used in the mechanism for any lease-based algorithm. Nodes $\{v_1, \ldots, v_k\}$ refer to the neighbors of node $u$.

forwarded *update* messages corresponding to a received *update* message. Each tuple consists of three elements, *node*, *rcvid*, and *sntid*. The first element, *node*, identifies the node from which the *update* message is received. The second element, *rcvid*, is the identifier of the received *update* message, and the last element, *sntid*, is the identifier of the corresponding sent *update* messages.

Informally, for any node $u$, a lease from a node $u$ to its neighboring node $v$ works as follows. If $u.\mathit{granted}[v]$ holds, then on a *write* request at any node in $\mathit{subtree}(u, v)$, $u$ propagates the new aggregate value to $v$ by sending an *update* message. To break the lease (that is, to falsify $u.\mathit{granted}[v]$), a $\mathit{release}()$ message is sent from $v$ to $u$. On the other hand, if $u.\mathit{granted}[v]$ does not hold, then on a *combine* request at any node in $\mathit{subtree}(v, u)$, a $\mathit{probe}()$ message is sent from $v$ to $u$. As a result, a *response* message is sent from $u$ to $v$.

### 4.4.1 Properties of any Lease-Based Algorithm for Sequential Executions

For any quiescent state $Q$, we define a *lease graph* $G(Q)$ as a directed graph with nodes as the nodes in $T$, and for any edge $(u, v)$ in $T$ such that $u.\mathit{granted}[v]$ holds, there is a directed edge $(u, v)$ in $G(Q)$. For any two distinct nodes $u$ and $v$, we define the $u$-parent of $v$ as the parent of $v$ in $T$ rooted at $u$.

**Lemma 4.4.1.** *For a sequential execution of a request sequence, in any quiescent state, for any two neighboring nodes $u$ and $v$, $u.\mathit{taken}[v] = v.\mathit{granted}[u]$.*

*Proof.* Consider any node $v$ in $u.\mathit{nbrs}()$. Variable $u.\mathit{taken}[v]$ can be set to **true** from **false** only in Line 3 of $T_4$ if the *flag* in the received *response* message is

**true**. However, while sending the *response* message from $v$ to $u$ with *flag* set to **true**, $v.granted[u]$ is set to **true** in *sendresponse*().

While sending a *release* message from $u$ to $v$, $u.taken[v]$ is falsified in *forwardrelease*(). However, on receiving the *release* message at $v$, $v.granted[u]$ is falsified in Line 2 of $T_6$. □

**Lemma 4.4.2.** *For a sequential execution of a request sequence, in any quiescent state, for any node $u$ and any node $v$ in $u.nbrs()$, if $u.granted[v]$ then, for all nodes $w$ in $u.nbrs() \setminus \{v\}$, $u.taken[w]$ holds.*

*Proof.* By inspection of the code, $u.granted[v]$ can be set to **true** only in the procedure *sendresponse*(). By inspection of the code of *sendresponse*(), $u.granted[v]$ can be set to **true** only if $u.nbrs() \setminus \{u.tkn() \cup \{v\}\} = \emptyset$. That is, $u.granted[v]$ can be set to **true** only if, for all nodes $w$ in $u.nbrs() \setminus \{v\}$, $u.taken[w]$ holds.

Further, by inspection of the code, $u.taken[w]$ is set to **false** only in the procedure *forwardrelease*(). By inspection of the code of *forwardrelease*(), $u.taken[w]$ can be set to **false** only if, for all nodes $v$ in $u.nbrs() \setminus \{w\}$, $u.granted[v]$ is **false**. That is, for any node $v$ in $u.nbrs()$, if $u.granted[v]$ holds then, for any node $w$ in $u.nbrs() \setminus \{v\}$, $u.taken[w]$ is not falsified. □

**Lemma 4.4.3.** *Consider a sequential execution of a request sequence $\sigma$ by a lease-based algorithm. For any combine request $q$ in $\sigma$, initiated at node $u$ in a quiescent state $Q$, let $A$ be the set of nodes $v$ such that $v.granted[w]$ does* *not hold in $Q$, where $w$ is the $u$-parent of $v$. In $Q$, for any node $v$ in $T$, if $v.pndg = \emptyset$ and for any node $w$ in $v.nbrs()$, $v.snt[w] = \emptyset$, then, during the execution of $q$, the following conditions hold: (1) $|A|$ probe messages are sent, and any node $v$ in $A$ receives a probe message from the $u$-parent of $v$; (2) $|A|$ response messages are sent, and any node $v$ in $A$ sends a response message to the $u$-parent of $v$; (3) no update or release messages are sent.*

*Proof.* We prove part (1) by induction on the length of the path from $u$ to an arbitrary node $v$ in $A$.

Base case (path length 1). By inspection of the code of $T_1$, *probe* messages are sent to all nodes in $u.nbrs() \setminus \{u.tkn() \cup u.sntprobes() \cup \{u\}\}$. Since in the quiescent state $Q$, for any node $v$ in $T$ and any node $w$ in $v.nbrs()$, $v.snt[w] = \emptyset$, $u.sntprobes() = \emptyset$. Hence, a *probe* message is sent to any node $v$ in $u.nbrs()$ such that $u.taken[v]$ does not hold. By Lemma 4.4.1, in $Q$, $u.taken[v] = v.granted[u]$. Hence, any node $v$ in $A$ such that $v$ is in $u.nbrs()$ and $v.granted[u]$ does not hold, receives a *probe* message from $u$.

Induction hypothesis. Any node $v$ in $A$ such that the length of the path from $u$ to $v$ is $i$ receives a *probe* message from the $u$-parent of $v$.

Induction step. Consider a node $v$ in $A$ such that the length of the path from $u$ to $v$ is $(i + 1)$. Let the $u$-parent of $v$ be $w$. By the definition of $A$, $v.granted[w]$ does not hold in $Q$. Hence, by Lemmas 4.4.1 and 4.4.2, $w.granted[u$-parent of $w]$ does not hold in $Q$. Thus, $w$ is in $A$, and by induction hypothesis $w$ receives a *probe* message from $w'$. By inspection of the code of

$T_3$, $w$ sends a *probe* message to any node $w'$ in $w.\mathit{nbrs}()$ such that $w.\mathit{taken}[w']$ does not hold. Since $w.\mathit{taken}[v]$ does not hold and the communication channels are reliable, $v$ receives a *probe* message from $w$, the $u$-parent of $v$.

From the above arguments, during the execution of $q$, at least $|A|$ *probe* messages are sent. By inspection of the code, no node $v$ in $A \cup \{u\}$ sends a *probe* message to a node in $v.\mathit{tkn}() \setminus \{u\text{-parent of } v\}$. Hence, it is straightforward to see that no node in $\mathit{nodes}(T) \setminus A$ receives a *probe* message. Hence, during the execution of $q$, only $|A|$ *probe* messages are sent.

We prove part (2) by reverse induction on the length of the path from $u$ to any node $v$ in $A$. Let the maximum length of the path from $u$ to any node $v$ in $A$ be $l$.

Base case. Consider a node $v$ in $A$ such that the length of the path from $u$ to $v$ is $l$. By part (1), $v$ receives a *probe* message from $w$, the $u$-parent of $v$. In the quiescent state $Q$, let $B$ be $v.\mathit{nbrs}() \setminus \{v.\mathit{tkn}() \cup \{u\text{-parent of } v\}\}$. By Lemma 4.4.1, $B$ must be $\emptyset$; otherwise, there would be a node in $A$ for which the length of the path from $u$ is equal to $l + 1$. By inspection of the code of $T_3$, if $B$ is empty, then $v$ sends back a *response* message to $w$.

Induction hypothesis. Let $v$ be a node in $A$ for which the length of path from $u$ is equal to $i$, and assume that $v$ sends a *response* message to its $u$-parent.

Induction step. Consider a node $v$ in $A$ such that the length of the path from $u$ to $v$ is $i - 1$. Since $v$ is in $A$, $i - 1$ is greater than 0. In $Q$, let $B$

be $v.nbrs() \setminus \{v.tkn() \cup \{u\text{-parent of } v\}\}$.

By part (1), $v$ receives a *probe* message from the $u$-parent of $v$. In order to prove part (2), we can assume that, in $Q$, $v.pndg = \emptyset$ and for any node $w$ in $v.nbrs()$, $v.snt[w] = \emptyset$. Hence, in $Q$, $v.sntprobes()$ is empty.

By inspection of the code of $T_3$, if $B$ is empty, then $v$ sends a *response* message back to the $u$-parent of $v$. Hence, the induction step succeeds.

Otherwise, $v$ sends *probe* messages to each of the nodes in $B$, and sets $v.pndg = \{u\text{-parent of } v\}$ and $v.snt[u\text{-parent of } v] = B$. Since we are dealing with sequential execution, no node initiates any request during the execution of $q$. Hence, $v$ does not initiate any request or receive a *probe* message during the execution of $q$. Hence, $v.pndg \leq 1$.

By Lemma 4.4.1 and the definition of $A$, any node in $B$ is also present in $A$. Further, the length of the path from $u$ to any node in $B$ is $i$. Hence, by the induction hypothesis, any node $w$ in $B$ sends a *response* message to $v$. By inspection of the code of $T_4$, on receiving the *response* message, $v$ removes $w$ from $v.snt[u\text{-parent of } v]$. If $v.snt[u\text{-parent of } v]$ becomes empty, then $v$ sets $v.pndg = \emptyset$, and sends a *response* message to the $u$-parent of $v$. Hence, the induction step succeeds.

(3) Follows from inspection of the code. □

**Lemma 4.4.4.** *For any sequential execution of a request sequence $\sigma$, in any quiescent state, for any node $u$, the following conditions hold: (1) $u.pndg = \emptyset$; (2) for any node $v$ in $u.nbrs()$, $u.snt[v] = \emptyset$.*

*Proof.* We prove the claim by induction on the number of requests executed.

Base case: Initially, for any node $v$, $v.pndg = \emptyset$ and for any node $w$ in $v.nbrs()$, $v.snt[w] = \emptyset$.

Induction hypothesis: In the quiescent state $Q$ just after execution of $i$ requests, for any node $v$, $v.pndg = \emptyset$ and for any node $w$ in $v.nbrs()$, $v.snt[w] = \emptyset$.

Induction step: Consider the execution of the $(i+1)$st request $q$ initiated in $Q$. If $q$ is a *write* request, then by inspection of the code, no *probe* or *response* message is generated. Hence, for any node $v$, $v.pndg$ and any node $w$ in $v.nbrs()$, $v.snt[w]$ is not modified. Therefore, the execution of $(i + 1)$st request preserves the claim of the lemma.

Otherwise, $q$ is a *combine* request, say at $u$. Consider execution of $q$. Let $A$ be the set of nodes $v$ such that $v.granted[w]$ does not hold at $Q$, where $w$ denotes the $u$-parent of $v$.

By the induction hypothesis, in $Q$, for any node $v$, $v.pndg = \emptyset$ and for any node $w$ in $v.nbrs()$, $v.snt[w] = \emptyset$.

First, consider any node $v$ in $nodes(T) \setminus \{A \cup \{u\}\}$. By inspection of the code, for any node $v$, $v.pndg$ and for any node $w$ in $v.nbrs()$, $v.snt[w]$ can be modified only in $T_1$ (on a *combine* request at $v$), in $T_3$ (on receiving a *probe* message), or in $T_4$ (on receiving a *response* message). In a sequential execution of $\sigma$, $v$ does not initiate any request during the execution of $q$. By Lemma 4.4.3, during the execution of $q$, any node in $A$ receives a *probe* message, and only $|A|$ *probe* messages are sent. Hence, $v$ does not receive any *probe* message during the execution of $q$. By the definition of $A$, the $u$-parent of any node in $A$ is in $A \cup \{u\}$. By Lemma 4.4.3, during the execution of $q$, $|A|$ *response* messages are generated and any node in $A$ sends a *response* message to the $u$-parent of the node. Hence, $v$ does not receive any *response* message during the execution of $q$. Hence, during the execution of $q$, $v.pndg$ remains $\emptyset$, and for any node $w$ in $v.nbrs()$, $v.snt[w]$ remains $\emptyset$.

Second, consider $v = u$. By inspection of the code of $T_1$, if $u.nbrs() \setminus u.tkn() = \emptyset$, then $u$ returns $gval()$, and so, $u.pndg$ remains $\emptyset$, and for any node $w$ in $u.nbrs()$, $u.snt[w]$ remains $\emptyset$. Further, by Lemma 4.4.1 and Lemma 4.4.2, $|A| = \emptyset$. Hence, from the arguments in the previous paragraph, the induction step succeeds, and the lemma follows.

Otherwise, $u.nbrs() \setminus u.tkn() \neq \emptyset$. Then, since $u.sntprobes() = \emptyset$ by the induction hypothesis, $u$ sends a *probe* message to each node in the set $u.nbrs() \setminus u.tkn()$, and node $u$ adds $u$ to $u.pndg$ and sets $u.snt[u] = nodes.nbrs() \setminus u.tkn()$. Since, in a sequential execution, a new request can be generated only in a quiescent state, no node generates a request until $q$ is completed. Hence, $u$ does not generate a request until $q$ is completed, and by Lemma 4.4.3, $u$ does not receive a *probe* message from any node. Therefore, $|u.pndg| \leq 1$. By the definition of $A$, any node $w$ in $u.nbrs() \setminus u.tkn()$ is also in $A$. By Lemma 4.4.3, $w$ sends back a *response* message to $u$. By inspection of the code of $T_4$, on receiving the *response* message, $u$ removes $w$ from $u.snt[u]$. When $u.snt[u] = \emptyset$, that is, $u$ has received *response* messages from all the nodes to which $u$ has sent a *probe* message, then $u$ sets $u.pndg = \emptyset$ and returns $gval()$.

Finally, consider any node $v$ in $A$. By Lemma 4.4.3, $v$ receives a *probe* message from the $u$-parent of $v$, say $w$. Let $C$ be $v.nbrs() \setminus \{v.tkn() \cup \{w\}\}$. By inspection of the code of $T_3$, if $C = \emptyset$, then $v$ sends a *response* message to $w$, and $v.pndg$ remains $\emptyset$, and for any node $w'$ in $v.nbrs()$, $v.snt[w']$ remains remains $\emptyset$.

Otherwise, $C \neq \emptyset$. Then, since $v.sntprobes() = \emptyset$, $v$ sends a *probe* message to each node in $C$. By inspection of the code of $T_3$, while sending a *probe* messages, $v$ adds $w$ to $v.pndg$ and sets $v.snt[w] = C$. As argued in the preceding paragraph, in a sequential execution, $|v.pndg| \leq 1$. By Lemma 4.4.3, any node $w'$ in $C$ sends back a *response* message to $v$. By inspection of the code of $T_4$, on receiving the *response* message, $v$ removes $w'$ from $v.snt[v]$. When $v.snt[w] = \emptyset$, that is, $v$ has received *response* messages from all nodes in $C$, then $w$ sets $v.pndg = \emptyset$ and sends a *response* message back to $w$.

Hence, after execution of $q$, for any node $v$ in $A$, $v.pndg = \emptyset$ and for any node $w$ in $v.nbrs()$, $v.snt[w] = \emptyset$. □

**Lemma 4.4.5.** *Consider a sequential execution of a request sequence $\sigma$ by a lease-based algorithm. For any write request $q$ in $\sigma$ initiated at node $u$ in a quiescent state $Q$, let $A$ be the set of nodes in $T$ reachable from $u$ in $G(Q)$. Then, during the execution of $q$, the following conditions hold: (1) any node $v$ in $A$ receives an update message from the $u$-parent of $v$; (2) $|A|$ update messages are sent; (3) no probe or response messages are sent.*

*Proof.* We prove (1) by induction on the length of the path from $u$ to any node $v$ in $A$.

Base case (path length 1). By inspection of the code of $T_2$, *update* messages are sent to all nodes in $u.grntd()$. That is, an *update* is sent to each node $v$ in $A$ such that the length of the path from $u$ to $v$ is 1.

Induction hypothesis. Any node $v$ in $A$ such that the length of the path from $u$ to $v$ is $i$ receives an *update* message from the $u$-parent of $v$.

Induction step. Consider a node $v$ in $A$ such that the length of the path from $u$ to $v$ is $(i + 1)$. By the induction hypothesis, the $u$-parent of $v$, say $w$, receives an *update* message. By the definition of $A$, $w.granted[v]$ holds. By inspection of the code of $T_5$, $w$ sends an *update* message to $v$. Since the communication channels are reliable, $v$ receives an *update* message from $w$, the $u$-parent of $v$.

Proof of (2) is as follows. From the above arguments, at least $|A|$ *update* messages are sent. By inspection of the code, no node $v$ in $A \cup \{u\}$ sends an *update* message to a node in $v.nbrs() \setminus \{v.grntd() \cup \{u\text{-parent of } v\}\}$. Hence, it is straightforward to see that no node $v$ in $nodes(T) \setminus A$ receives an *update* message. Hence, during the execution of $q$, only $|A|$ *update* messages are sent.

From inspection of the code, (3) follows. □

**Lemma 4.4.6.** *For any node $u$, $u.granted[v]$ is set to* **true** *only while sending a response message to $v$ with flag set to* **true**.

*Proof.* For any node $u$, $u.granted[v]$ can be set to **true** only in the *sendresponse* procedure. By inspection of the code, the lemma follows. $\square$

**Lemma 4.4.7.** *For any node $u$, $u.granted[v]$ is set to* **false** *only on receiving a release message from $v$.*

*Proof.* Follows from inspection of the code. $\square$

**Lemma 4.4.8.** *Consider a sequential execution of a request sequence $\sigma$ by a lease-based algorithm, and any two neighboring nodes $u$ and $v$.*

1. *Let a combine request $q$ in $\sigma(u, v)$ be initiated in a quiescent state $Q$. If $u.granted[v]$ does not hold in $Q$, then in the execution of $q$, the following conditions hold: (i) a probe message is sent from $v$ to $u$; (ii) a response message is sent from $u$ to $v$; (iii) $u.granted[v]$ can be set to* **true** *while sending the response message from $v$ to $u$; and (iv) no update or release messages are sent. Otherwise, if $u.granted[v]$ holds, then in the execution of $q$, no messages are exchanged between $u$ and $v$.*

2. *Let a write request $q$ in $\sigma(u, v)$ be initiated in a quiescent state $Q$. If $u.granted[v]$ does not hold in $Q$, then in the execution of $q$, no messages are exchanged between $u$ and $v$. Otherwise, if $u.granted[v]$ holds in $Q$, then in the execution of $q$, the following conditions hold: (i) an update message is sent from $u$ to $v$; (ii) a release message from $v$ to $u$ can be sent; (iii) on receiving the release message at $u$, $u.granted[v]$ is set to* **false***; (iv) no probe or response messages are sent.*

3. *Let a write request $q$ in $\sigma(v, u)$ be initiated in a quiescent state $Q$. If $u.granted[v]$ holds in $Q$, then in the execution of $q$, a release message can be sent from $v$ to $u$, and on receiving the release message at $u$, $u.granted[v]$ is set to* **false**.

4. *In the execution of a combine request in $\sigma(v, u)$, $u.granted[v]$ is not affected.*

*Proof.* Part (1) follows from Lemmas 4.4.3, 4.4.4, and 4.4.6. Part (2) follows from Lemmas 4.4.5, 4.4.7, and inspection of the code. Part (3) follows from Lemma 4.4.7 and inspection of the code. Part (4) follows from Lemmas 4.4.3, 4.4.4, and 4.4.6. □

| $u.granted[v]$ in $Q$ | Request $q$ in $\sigma(u, v)$ | $u.granted[v]$ in $Q'$ | Cost |
|---|---|---|---|
| **false** | R | **false** | 2 |
| **false** | R | **true** | 2 |
| **false** | W | **false** | 0 |
| **false** | N | **false** | 0 |
| **true** | R | **true** | 0 |
| **true** | W | **false** | 2 |
| **true** | W | **true** | 1 |
| **true** | N | **false** | 1 |
| **true** | N | **true** | 0 |

Figure 4.4: For any lease-based aggregation algorithm in executing any request $q$ in $\sigma(u, v)$, and for any two neighboring nodes $u$ and $v$, possible changes in the value of $u.granted[v]$ and costs incurred are shown in this figure. Here, $q$ is initiated in the quiescent state $Q$ and completed in the quiescent state $Q'$. A *release* message sent during the execution of a *write* request in $\sigma(v, u)$ is associated with a *noop* (N) request.

Lemma 4.4.8 is summarized in Figure 4.4. A *release* message sent during the execution of a *write* request in $\sigma(v, u)$ is associated with a *noop* (N) request in this figure.

For any node $u$, we define predicates $I_1(u)$, $I_2(u)$, and $I_3(u)$ as follows: (1) predicate $I_1(u)$ holds if, for the most recent *write* request $q$ at $u$, $u.val$ is equal to $q.arg$; (2) predicate $I_2(u)$ holds if, for any *update* or *response* message $m$ from any neighboring node $v$ to $u$, $m.x$ is equal to $f(A)$, where $A$ is the set of most recent write requests at each of the nodes in $subtree(v, u)$; and (3) predicate $I_3(u)$ holds if, for any quiescent state $Q$ and any node $v$ in $u.tkn()$, $u.aval[v]$ is equal to $f(A(v))$, where $A(v)$ is the set of the most recent *write* request at each of the nodes in $subtree(v, u)$. Let $I(u)$ be $I_1(u) \wedge I_2(u) \wedge I_3(u)$.

**Lemma 4.4.9.** *Consider a sequential execution of a request sequence $\sigma$ by a lease-based algorithm. For any node $u$, if $I_1(u)$ and $I_3(u)$ hold just before an update message $m$ is sent from $u$ to some node $v$ in $u.nbrs()$, then $m.x = f(A)$, where $A$ is the set of the most recent write requests at each of the nodes in $subtree(u, v)$.*

*Proof.* By Lemma 4.4.2, for any node $v$ in $u.nbrs()$, if $u.granted[v]$ holds, then for all nodes $w$ in $u.nbrs() \setminus \{v\}$, $u.taken[w]$ holds.

For any node $w$ in $u.nbrs()$, let $A(w)$ be the set of the most recent *write* requests preceding $q$ in $\sigma$ at each of the nodes in $subtree(w, u)$. By $I_3(u)$, if $u.taken[w]$ holds, then $u.aval[w] = f(A(w))$.

By inspection of the code, for any node $v$ in $u.grntd()$, an *update* message $m$ is sent to $v$ with $m.x = u.subval(v)$. Let $\{w_1, \ldots, w_k\}$ be $u.nbrs() \setminus \{v\}$, and let $B$ be the set of the most recent *write* requests at each of the nodes in *subtree*$(u, v)$. We have

$$\begin{aligned} m.x &= \mathit{subval}(v) \\ &= f(u.\mathit{val}, \mathit{aval}[w_1], \ldots, \mathit{aval}[w_k] \\ &= f(q.\mathit{arg}, f(A(w_1)), \ldots, f(A(w_k))) \\ &= f(B) \end{aligned} \tag{4.1}$$

In the above equation, the second equality follows from the definition of function *subval*(). The third equality follows from $I_1(u)$ and $I_3(u)$. The last equality follows from the fact that $\mathit{subtree}(u, v) = \{u\} \cup \mathit{subtree}(w_1, u) \cup \cdots \cup \mathit{subtree}(w_k, u)$. □

**Lemma 4.4.10.** *Consider a sequential execution of a request sequence $\sigma$ by a lease-based algorithm. For any node $u$, $I(u)$ is an invariant.*

*Proof.* Initially, there are no *write* requests at $u$, no messages in transit, and $u.\mathit{tkn}()$ is empty. Hence, $I(u)$ holds. Thus it is sufficient to check that every action preserves $I(u)$. In the following we present the reasons why every action preserves $I(u)$.

$\{I(u)\}T_1\{I(u)\}$. All the conjuncts, $I_1(u)$, $I_2(u)$, and $I_3(u)$ are not affected.

$\{I(u)\}T_2\{I(u)\}$. Let a *write* request $q$ be initiated in some quiescent state $Q$. In the execution of $T_2$, $I_1(u)$ is only affected in Line 1. By inspection of the code, Line 1 preserves $I_1(u)$. During the execution of $T_2$, $I_3(u)$ is not affected. If $u.grntd() \neq \emptyset$ in the quiescent state $Q$, then $I_2(u)$ is affected in the procedure *forwardupdates*(), invoked in Line 4. By Lemma 4.4.9, $I_2(u)$ is preserved in Line 4.

Therefore, $I_1(u) \wedge I_2(u) \wedge I_3(u)$ is preserved in the execution of $T_2$.

$\{I(u)\}T_3\{I(u)\}$. By inspection of the code, $I_1(u)$ and $I_3(u)$ are not affected. Predicate $I_2(u)$ is affected only in the procedure *sendresponse*(), invoked in Line 6 to send a *response* message $m$ to $w$. However, Line 6 is executed only if $u.nbrs() \setminus \{u.tkn() \cup \{w\}\}$ is empty. By $I_3(u)$, for any node $v$ in $u.nbrs()$, if $u.taken[v]$, then $u.aval[v] = f(A)$, where $A$ is the set of the most recent *write* requests at each of the nodes in *subtree*$(v, u)$. As in the proof of Lemma 4.4.9, $m.x = f(B)$, where $B$ is the set of the most recent *write* requests at each of the nodes in *subtree*$(u, w)$.

$\{I(u)\}T_4\{I(u)\}$. Predicate $I_1(u)$ is not affected in $T_4$. In $T_4$, $I_3(u)$ is affected in Line 2 and $I_2(u)$ is affected in the *sendresponse*() procedure, which is invoked in Line 11.

In the following, for any node $w'$ in $u.nbrs()$, let $B(w')$ be the set of the most recent *write* requests at each of the nodes in *subtree*$(w, u)$.

Since $I_2(u)$ holds for the received *response* message, after execution of Line 2, $u.aval[w] = f(B)$, where $B(w)$. Hence, $I_3(u)$ holds in the execution of Line 2.

To argue that $I_2(u)$ holds in Line 11, we show that just before the execution of Line 11, for each node $w'$ in $u.nbrs() \setminus \{v\}$, $u.aval[w'] = f(B(w'))$.

By Lemmas 4.4.3 and 4.4.5, a *response* message from $w$ is received during the execution of a *combine* request, say $q$. We can assume that $q.node \neq u$, since Line 11 is executed only if $q.node \neq u$.

From Lemma 4.4.3, $u$ is the $q.node$-parent of $w$ and $v$ is the $q.node$-parent of $u$. Let $q$ be initiated in the quiescent state $Q$. In quiescent state $Q$, let $A$ be the set of nodes $u.nbrs() \setminus \{u.tkn() \cup \{v\}\}$.

Again by Lemma 4.4.3, during the execution of $q$, $u$ sends a *probe* message to each of the nodes in $A$ and receives a *response* message from each of them. For each of the received *response* message from $w$, as argued above, after execution of Line 2, $u.aval[w] = f(B(w))$. By inspection of the code of $T_3$, while sending *probe* messages, $u$ sets $u.snt[v] = A$. By inspection of the code of $T_4$, on receiving a *response* message from a node $w$, $w$ is removed from $u.snt[v]$. Hence, Line 11 is executed only when $u$ has received *response* messages from all nodes in $A$. Hence, just before execution of Line 11, for each of the node $w'$ in $A$, $u.aval[w'] = B(w')$. By $I_2$, for each node $w'$ in $u.tkn()$, $u.aval[w'] = B(w')$. Hence, just before the execution of Line 11, for each node $w'$ in $u.nbrs \setminus \{v\}$, $u.aval[w'] = B(w')$. Hence, as in the proof of Lemma 4.4.9, for the *response* message $m$ sent to $v$, $m.x = f(C)$, where $C$ is the set of the most recent *write* requests at each of the nodes in $subtree(u, v)$.

$\{I(u)\}T_5\{I(u)\}$. Predicate $I_1(u)$ is not affected in the execution of $T_5$.

Predicate $I_3(u)$ is affected only in Line 2. Let $A$ be the set of the most recent *write* requests at each of the nodes in $subtree(w, u)$. By $I_2(u)$, $m.x = f(A)$. After Line 2, $u.aval[w] = f(A)$. Hence, $I_3(u)$ is preserved in Line 2.

If $u.grntd() \neq \emptyset$ in quiescent state $Q$, then $I_2(u)$ is affected in the procedure *forwardupdates*(), invoked in Line 7. By Lemma 4.4.9, $I_2(u)$ is preserved in Line 7.

Therefore, $I_1(u) \wedge I_2(u) \wedge I_3(u)$ is preserved in the execution of $T_5$.

$\{I(u)\}T_6\{I(u)\}$. Predicates $I_1(u)$, $I_2(u)$, and $I_3(u)$ are not affected. Hence, $I(u)$ is preserved. □

**Lemma 4.4.11.** *Any lease-based aggregation algorithm is nice.*

*Proof.* Consider a *combine* request $q$ at a node $u$. Let $q$ be initiated in a quiescent state $Q$. From Lemmas 4.4.1 and 4.4.3, during the execution of $q$, $u$ receives *response* messages from all neighboring nodes $v$ such that $u.taken[v]$ does not hold in $Q$. From Lemma 4.4.10, $I_1(node) \wedge I_2(u) \wedge I_3(u)$ is an invariant. Hence, the return value of the *combine* request, which is $u.gval()$, is $f(A)$, where $A$ is the set of the most recent *write* requests at each of the nodes in $nodes(T)$. Hence, the lemma follows. □

From Lemma 4.4.11, any lease-based aggregation algorithm provides strict consistency in a sequential execution.

## 4.5 Competitive Analysis Results for Sequential Executions

```
var $lt$ : array[$v_1 \ldots v_k$] of int;
  $granted$ : array[$v_1 \ldots v_k$] of boolean;

procedure oncombine()
  foreach $v \in tkn()$ do
    $lt[v] := 2$; od
procedure probercvd(node $w$)
  foreach $v \in tkn() \setminus \{w\}$ do
    $lt[v] := 2$; od
boolean setlease(node $w$)
  $lg[w] :=$ true;
  return true;
```

```
procedure responsercvd(boolean $flag$, node $w$)
  if $flag \wedge (taken[w] =$ false$) \rightarrow$
    $lt[w] := 2$; fi
procedure updatercvd(node $w$)
  if $(grntd() \setminus \{w\} = \emptyset) \wedge lt[w] > 0 \rightarrow$
    $lt[w] := lt[w] - 1$; fi
procedure releasepolicy(node $v$)
  $lt[v] := max(0, lt[v] - |uaw[v]|)$;
procedure releasercvd(node $w$)
  $lg[w] :=$ false;
boolean breaklease(node $w$)
  return($lt[w] = 0$);
```

Figure 4.5: Policies for RWW.

We define RWW as an online lease-based aggregation algorithm that follows the policies shown in Figure 4.5 for setting or breaking a lease.

Informally, RWW works as follows. For any edge $(u, v)$, RWW sets the lease from $u$ to $v$ during the execution of a *combine* request at a node in the $subtree(v, u)$, and breaks the lease after two consecutive *write* requests at nodes in $subtree(u, v)$.

### 4.5.1 Properties of RWW

For positive integers $a$ and $b$, an online lease-based algorithm $A$ is in the class of $(a, b)$-*algorithms* if, in a sequential execution of any request sequence $\sigma$ by $A$, for any edge $(u, v)$, $A$ satisfies the following conditions: (1) if $u.granted[v]$ is **false**, then it is set to **true** after $a$ consecutive *combine* requests in $\sigma(u, v)$; (2) if $u.granted[v]$ is **true**, then it is set to **false** after $b$ consecutive *write* requests in $\sigma(u, v)$.

For any ordered pair of neighboring nodes $u$ and $v$, we define *type*$(u, v)$ messages as the following kinds of messages exchanged between $u$ and $v$: (1) *probe* messages from $v$ to $u$; (2) *response* messages from $u$ to $v$; (3) *update* messages from $u$ to $v$; (4) *release* messages from $v$ to $u$. For a lease-based algorithm $A$ and a request sequence $\sigma$, we define $C_A(\sigma, u, v)$ as the number of *type*$(u, v)$ messages sent during the execution of $\sigma$ by $A$.

**Lemma 4.5.1.** *Consider a sequential execution of a request sequence $\sigma$ by* RWW *and two neighboring nodes $u$ and $v$. Then, during the execution of a request from $\sigma(v, u)$, $u.granted[v]$ is not affected.*

*Proof.* First, consider the execution of a *combine* request in $\sigma(v, u)$. By Lemmas 4.4.3 and 4.4.4, no *update* or *release* messages are sent. Further, no *response* messages from $u$ to $v$ are sent. Hence, $u.granted[v]$ is not affected during the execution of a *combine* request in $\sigma(v, u)$.

Second, consider the execution of a *write* request in $\sigma(v, u)$. By Lemma 4.4.5, no *probe* or *response* messages are sent. Further, no *update* message from $u$ to $v$ is sent. By inspection of the code of RWW, a *release* message from $v$ to $u$ can only be sent during the execution of a *write* request in $\sigma(u, v)$. Hence, $u.granted[v]$ is not affected during the execution of a *write* request in $\sigma(v, u)$. □

Let $I_4(u)$ be the following predicate. For any node $v$ in $u.nbrs()$, (1) if $u.taken[v]$ does not hold, then $u.uaw[v] = \emptyset$; (2) if $u.taken[v]$ holds and $u.grntd() \setminus \{v\} = \emptyset$, then $(u.lt[v] + |u.uaw[v]| = 2) \wedge u.lt[v] > 0$; (3) if $u.taken[v]$ holds and $u.grntd() \setminus \{v\} \neq \emptyset$, then $u.lt[v] = 2$.

**Lemma 4.5.2.** *Consider a sequential execution of a request sequence by* RWW*. For any node u, $I_4(u)$ is an invariant.*

*Proof.* Initially, for any node $v$ in $u.nbrs()$, $u.taken[v]$ does not hold and $u.uaw[v] = \emptyset$. Hence, $I_4(u)$ holds initially. Thus it is sufficient to check that every action preserves $I_4(u)$. In the following we present the reasons why every action preserves $I_4(u)$.

$\{I_4(u)\}T_1\{I_4(u)\}$. For any node $v$ in $u.tkn()$, $u.lt[v]$ is set to 2 in the *oncombine*() procedure and $u.uaw[v]$ is set to $\emptyset$ in Line 3. Hence, $I_4(u)$ is preserved.

$\{I_4(u)\}T_2\{I_4(u)\}$. Predicate $I_4(u)$ is not affected.

$\{I_4(u)\}T_3\{I_4(u)\}$. For any node $v$ in $u.tkn() \setminus \{w\}$, $u.lt[v]$ is set to 2 in *probercvd*() procedure, and $u.uaw[v]$ is set to $\emptyset$ in Line 3. Hence, $I_4(u)$ is preserved.

$\{I_4(u)\}T_4\{I_4(u)\}$. By Lemma 4.4.3, a *response* message is received from $w$ as a result of an earlier *probe* message sent to $w$ during execution of a *combine* request, say $q$. By Lemma 4.4.3 again, in the quiescent state $Q$ in which $q$ is initiated, $u.taken[w]$ does not hold. Hence, if $I_4(u)$ holds before execution of $T_4$, then $u.uaw[w]$ is empty.

If *flag* is **true**, then $u.lt[w]$ is set to 2 in *responsercvd*() procedure, and $u.taken[w]$ is set to **true** in Line 3. Since $u.uaw[w]$ remains empty, $I_4(u)$ holds after execution of $T_4$.

$\{I_4(u)\}T_5\{I_4(u)\}$. By Lemmas 4.4.1 and 4.4.5, $u$ receives an *update* message from $w$ if and only if $u.taken[w]$ holds.

If $u.grntd() \setminus \{w\} = \emptyset$, then $u.lt[w]$ is decremented by one in the *updatercvd*() procedure. Otherwise, $u.lt[w]$ is not affected. In Line 3, $|uaw[w]|$ is incremented by 1. Hence, if $u.lt[w]$ remains greater than 0, then $I_4(u)$ is preserved.

If $u.lt[w]$ is decremented to 0, then a *release* message is sent to $w$ in the *forwardrelease*() procedure invoked in Line 9. In the *forwardrelease*() procedure, $u.taken[w]$ is set to **false**, and $u.uaw[w]$ is set to $\emptyset$. Hence, $I_4(u)$ is preserved.

$\{I_4(u)\}T_6\{I_4(u)\}$. Fix an arbitrary node $v$ in $u.nbrs() \setminus \{w\}$.

By inspection of the code, if $u.grntd() \setminus \{v\} \neq \emptyset$, then $u.lt[v]$ is not affected. Hence, $I_4(u)$ is preserved in the execution of $T_6$.

Now we argue that, if $u.grntd() \setminus \{v\} = \emptyset$, then $I_4(u)$ is also preserved.

First, we argue that $|S| = 2$. By inspection of the code, a *release* message from node $w$ to $u$, containing $w.uaw[u]$, is sent only in the *forwardrelease*() procedure. Since a *release* message is sent only if $w.breaklease(u)$ returns **true**, $w.lt[u]$ is 0 while sending the *release* message. Since $I_4(u)$ holds before the execution of $T_6$, $|S| = 2$.

Second, we argue that in the *onrelease*() procedure, the number of tuples $\alpha$ in *sntupdates* with $\alpha.sntid$ greater than or equal to the smallest *id* in $S$ is at most 2. From inspection of the code, the following conditions holds:

(1) identifiers of all received *update* messages at node $w$ from $u$ are added to $w.uaw[u]$; (2) identifiers of sent *update* messages from $u$ are in increasing order; (3) an identifier is not removed from the middle in $w.uaw[u]$, that is, identifiers in $w.uaw[u]$ are contiguous; (4) on receiving an *update* message, the identifier of the forwarded *update* message to node $w$ is added to *sntupdates*. Hence, $S$ contains the identifiers of the last two *update* messages sent to $w$ from $u$, that is, $S$ contains the two highest identifiers of the *update* messages sent to $w$. Since $S$ may contain the identifiers corresponding to the *update* messages due to *write* requests at $u$, the number of tuples $\alpha$ in *sntupdates* with $\alpha.sntid$ greater or equal to the smallest id in $S$ is at most 2.

Third, because of above arguments, $|A|$ is at most 2, where $A$ is as defined in *onrelease*() procedure.

Fourth, we argue that $|S'|$ is at most 2. Identifiers of the received *update* messages are in increasing order. Before receiving the *release* message, $u.granted[w]$ holds. On receiving an *update* message from $v$, the identifier of the received *update* message is added to $u.uaw[v]$. Since $u.granted[w]$ holds, on receiving an *update* with *id*, an *update* message is sent to $w$ with *nid*, and a tuple $\{v, id, nid\}$ is added *sntupdates*. Hence, the size of the set of identifiers in $u.uaw[v]$ (i.e., $|S'|$) with identifiers $\geq \beta.rcvid$, where $\beta$ is as defined in the *onrelease*() procedure, is at most 2.

Finally, we argue that $|u.uaw[v]| + u.lt[v] = 2$. Since, before receiving the *release* message, $u.granted[w]$ and $I_4(u)$ hold, $u.lt[v]$ is equal to 2 just before the invocation of the *releasepolicy*() procedure. In the *releasepolicy*()

procedure, $u.lt[v]$ is set to $u.lt[v] - |u.uaw[v]|$. Hence, after the execution of the *releasepolicy*() procedure, $|u.uaw[v]| + u.lt[v] = 2$.

If $u.lt[v]$ becomes 0, then in the *forwardrelease*() procedure, $u.tkn[v]$ is set **false**, $u.uaw[v]$ is set to $\emptyset$, and a *release* message is sent to $v$.

Hence, $I_4(u)$ is preserved in the execution of $T_6$. $\square$

**Lemma 4.5.3.** *Consider a sequential execution of a request sequence $\sigma$ by* RWW *and any two neighboring nodes $u$ and $v$. The following conditions hold: (1) in the quiescent state after execution of a combine request in $\sigma(u, v)$, $u.granted[v]$ holds; (2) in the quiescent state after execution of two consecutive write requests in $\sigma(u, v)$, $u.granted[v]$ does not hold.*

*Proof.* The proof of (1) is as follows. Let the the *combine* request $q$ be initiated in the quiescent state $Q$ and completed in the quiescent state $Q'$.

If $u.granted[v]$ holds in $Q$, then no $type(u, v)$ messages are sent during the execution of $q$, and so $u.granted[v]$ holds in $Q'$.

If $u.granted[v]$ does not hold in $Q$, then by Lemma 4.4.3, during the execution of $q$, a *probe* message is sent from $v$ to $u$ and a *response* message is sent from $u$ to $v$. By inspection of the code of *sendresponse*() procedure, RWW's *setlease*() procedure is invoked. By inspection of the code of RWW, *setlease*() procedure always returns **true**, and so $u.granted[v]$ is set to **true**. Hence, after execution of $q$, $u.granted[v]$ holds.

The proof of (2) is as follows. Let the two consecutive *write* requests be $q_1$ and $q_2$, initiated in quiescent states $Q$ and $Q'$, respectively. Let $q_2$ be completed in quiescent state $Q''$.

By Lemma 4.4.5, if $u.granted[v]$ does not hold in $Q$, then during the execution of $q_1$, no $type(u, v)$ messages are exchanged between $u$ and $v$. Hence, $u.granted[v]$ is not affected, and remains **false** in $Q'$ and $Q''$.

Otherwise, if $u.granted[v]$ holds in $Q$, then without loss of generality we can assume that the request preceding $q_1$ in $\sigma(u, v)$ is a *combine* request $q$.

Since, by Lemma 4.5.1, any request in $\sigma(v, u)$ does not affect $u.granted[v]$, without loss of generality we can also assume that there are no requests in $\sigma(v, u)$ that lie between $q_1$ and $q_2$ in $\sigma$.

By part (1), in quiescent state $Q$, there is a path from $u$ to $q.node$ (say $w$) in the lease graph $G(Q)$. Further, in $Q$, $w.uaw[u\text{-parent of } w]$ is empty and $w.lt[u\text{-parent of } w]$ is 0. By Lemma 4.4.5, $w$ receives an *update* message during the execution of $q_1$. By inspection of the code of $T_5$, $w.taken[u\text{-parent of } w]$ holds in $Q'$. Hence, by Lemma 4.4.1 and 4.4.2, $u.granted[v]$ holds in $Q'$.

It is sufficient to show that during the execution of $q_2$, a *release* message is sent from $v$ to $u$, falsifying $u.granted[v]$.

Let $A$ be the set of reachable nodes in the lease graph $G(Q')$ from $u$ following the edge $(u, v)$. Let $id(q_1, w)$ be the *id* of the *update* message received at $w$ during the execution of $q_1$.

First, we show that the following properties hold. Fix an arbitrary node $w$ in $A$. Property (1): Node $w$ receives an *update* message during the execution of $q_1$. Property (2): In quiescent state $Q'$, $w.uaw[u\text{-parent of } w]$ contains $id(q_1, w)$. Property (3): In quiescent state $Q'$, if $w.grntd() \setminus \{u\text{-parent of } w\}$ is empty, $|w.uaw[u\text{-parent of } w]| = 1$ and $w.lt[u\text{-parent of } w] = 1$.

The proof of Property (1) is as follows. By Lemma 4.4.5, no *probe* or *response* messages are sent during the execution of $q_1$. By inspection of the code, an edge is added in the lease graph only while sending and receiving a *response* message. Hence, if an edge is present in the lease graph $G(Q')$, then the edge is also present in the lease graph $G(Q)$. Hence, by Lemma 4.4.5, each node in $A$ receives an *update* message during the execution of $q_1$.

The proof of Property (2) is as follows. From Property (1) and Lemma 4.4.5, $w$ receives an *update* message from the $u$-parent of $w$. From inspection of the code of $T_5$, $id(q_1, w)$ is added to $w.uaw[u\text{-parent of } w]$. In quiescent state $Q'$, since the identifiers of *update* messages sent from the $u$-parent of $w$ to $w$ are in increasing order, and since $q_1$ is the latest *write* request, $id(q_1, w)$ is the highest identifier in $w.uaw[u\text{-parent of } w]$. Hence, $w.uaw[u\text{-parent of } w]$ contains $id(q_1, w)$.

The proof of Property (3) is as follows. Without loss of generality assume that $w.grntd() \setminus \{u\text{-parent of } w\}$ is empty. By Property (2), in quiescent state $Q'$, $|w.uaw[u\text{-parent of } w]| > 0$.

By inspection of the code, $w.lt[u\text{-parent of } w] > 0$. Hence, by Lemma 4.5.2, $|w.uaw[u\text{-parent of } w]| \leq 2$.

We use proof by contradiction to show that $|w.uaw[u\text{-parent of } w]| \neq 2$.

Assume that $|w.\mathit{uaw}[u\text{-parent of } w]| = 2$ in $Q'$. By Lemma 4.5.2 and inspection of the code of $T_5$ and $T_6$, if the set $w.\mathit{grntd}() \setminus \{u\text{-parent of } w\}$ is empty and $|w.\mathit{uaw}[u\text{-parent of } w]| = 2$, then $w.\mathit{lt}[u\text{-parent of } w]$ is 0 in $Q'$. Hence, $w$ must send a *release* message to the $u$-parent of $w$ and set $w.\mathit{taken}[u\text{-parent of } w]$ to **false** during the execution of $q_1$. But $w$ is in $A$, yielding a contradiction.

Therefore, $|w.\mathit{uaw}[u\text{-parent of } w]| = 1$, and by Lemma 4.5.2, Property (3) follows.

Second, we show the desired result by establishing that every node $w$ in $A$, including $v$, sends a *release* message to the $u$-parent of $w$ containing the set $\{\mathit{id}(q_1, w), \mathit{id}(q_2, w)\}$. We prove this claim by reverse induction on the length of the path from $u$ to some node in $A$. Let the maximum length of the path from $u$ to any node in $A$ be $l$.

Base case. Consider a node $w$ in $A$ such that the length of the path from $u$ to $w$ is $l$. By definition of $A$, $w.\mathit{grntd}() \setminus \{u\text{-parent of } w\}$ is empty. By Properties 2 and 3, $w.\mathit{uaw}[u\text{-parent of } w] = \{\mathit{id}(q_1, w)\}$ and $w.\mathit{lt}[u\text{-parent of } w] = 1$.

By Lemmas 4.4.1 and 4.4.2, $w$ is reachable from $q_2.\mathit{node}$ in the lease graph $G(Q')$. Hence, by Lemma 4.4.5, during the execution of $q_2$, $w$ receives an *update* message from the $u$-parent of $w$.

By inspection of the code of $T_5$, the *updatercvd*() procedure of RWW is invoked. In the *updatercvd*() procedure, $w.\mathit{lt}[u\text{-parent of } w]$ is set to 0. By inspection of the code of $T_5$, *forwardrelease*() procedure is invoked. By inspection of the code of RWW, *breaklease*() returns **true**. Hence, $w.\mathit{granted}[u\text{-parent of } w]$

is set to **false** and a *release* message is sent to the $u$-parent of $w$ containing $\{id(q_1, w), id(q_2, w)\}$.

Induction hypothesis. Let $w$ be a node in $A$ such that the length of the path from $u$ to $w$ is $i$, where $i > 1$. Then, $w$ sends a *release* message to the $u$-parent of $w$ containing $\{id(q_1, w), id(q_2, w)\}$.

Induction step. Consider a node $w$ in $A$ such that the length of the path from $u$ to $w$ is $i - 1$. As argued in the base case, during the execution of $q_2$, $w$ receives an *update* message from the $u$-parent of $w$.

By Property (2) and the above arguments, $w.uaw[u\text{-parent of } w]$ contains $id(q_1, w)$ and $id(q_2, w)$. By the induction hypothesis, for each node $w'$ in $w.nbrs()$ such that $w$ is $u$-parent of $w'$, $w$ receives a *release* message from $w'$.

By inspection of the code of $T_6$, after receiving a *release* message from all nodes $w'$ such that $w.granted[w']$ in $Q'$, $w$ sets $w.lt[u\text{-parent of } w]$ to 0, and sends a *release* message to the $u$-parent of $w$ containing $\{id(q_1, w), id(q_2, w)\}$.

Therefore, during the execution of $q_2$, a *release* message is sent from $v$ to $u$, falsifying $u.granted[v]$. □

**Lemma 4.5.4.** *The algorithm* RWW *is a* $(1, 2)$*-algorithm.*

*Proof.* Follows from Lemma 4.5.3. □

### 4.5.2 Competitive Ratio of RWW

In this section we show that RWW is $\frac{5}{2}$-competitive against an optimal offline lease-based algorithm OFF for the sequential aggregation problem (see Theorem 8). We also show that RWW is 5-competitive against an optimal nice offline algorithm for the sequential aggregation problem (see Theorem 9). Further, we show that, for any lease-based aggregation algorithm $A$, there exist a request sequence $\sigma$ and an offline algorithm such that, in a sequential execution of $\sigma$, the cost of $A$ is at least $\frac{5}{2}$ times that of the offline algorithm (see Theorem 10).

**Lemma 4.5.5.** *In a sequential execution of a request sequence $\sigma$, for any two neighboring nodes $u$ and $v$, $C_{\text{RWW}}(\sigma, u, v) = C_{\text{RWW}}(\sigma(u, v), u, v)$.*

*Proof.* Follows from Lemmas 4.4.8 and 4.5.1. □

**Lemma 4.5.6.** *Consider a sequential execution of a request sequence $\sigma$ by a lease-based algorithm $A$. For any two neighboring nodes $u$ and $v$, the total number of messages exchanged between $u$ and $v$ in executing $\sigma$ is the sum of $C_A(\sigma, u, v)$ and $C_A(\sigma, v, u)$.*

*Proof.* Follows from the definitions of $C_A(\sigma, u, v)$ and $C_A(\sigma, v, u)$. □

Consider a sequential execution of an arbitrary request sequence $\sigma$ by RWW. For any quiescent state $Q$, and for any ordered pair of neighboring nodes $(u, v)$, we define the configuration of RWW, denoted $F_{\text{RWW}}(u, v)$, as follows: (1) if $Q$ is the initial quiescent state, then $F_{\text{RWW}}(u, v)$ is 0; (2) if the

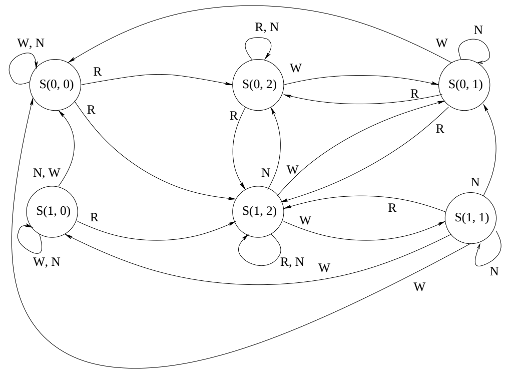

Figure 4.6: States and state transitions for any pair of nodes $(u, v)$ in executing requests from $\sigma'(u, v)$ (defined in Lemma 4.5.8).

last completed request in $\sigma(u, v)$ is a *combine* request, then $F_{\text{RWW}}(u, v)$ is 2; (3) if the last two completed requests in $\sigma(u, v)$ are a *combine* request followed by a *write* request, then $F_{\text{RWW}}(u, v)$ is 1; (4) if the last two completed requests in $\sigma(u, v)$ are *write* requests, then $F_{\text{RWW}}(u, v)$ is 0.

For any quiescent state $Q$ and ordered pair of neighboring nodes $(u, v)$, we define the configuration of OFF $F_{\text{OFF}}(u, v)$ to be 1 if $u.granted[v]$ holds, and 0 otherwise.

**Lemma 4.5.7.** *Consider a sequential execution of a request sequence $\sigma$ by* RWW*. For any quiescent state $Q$, and for any ordered pair of neighboring nodes $(u, v)$, $F_{\text{RWW}}(u, v)$ is greater than 0 if and only if $u.granted[v]$ holds.*

*Proof.* Follows from Lemmas 4.5.1 and 4.5.3. $\square$

**Lemma 4.5.8.** *Consider a sequential execution of a request sequence $\sigma$ by* RWW *and* OFF*. For any two neighboring nodes $u$ and $v$, $C_{\text{RWW}}(\sigma, u, v)$ is at*

$$
\begin{array}{lclcclr}
\text{minimize} & : & c & & & & \\
\Phi(0,2) & - & \Phi(0,0) & + & 2 & \leq & 2\cdot c \\
\Phi(1,2) & - & \Phi(0,0) & + & 2 & \leq & 2\cdot c \\
\Phi(0,0) & - & \Phi(0,0) & & & \leq & 0 \\
\Phi(1,2) & - & \Phi(1,0) & + & 2 & \leq & 0 \\
\Phi(0,0) & - & \Phi(1,0) & & & \leq & 2\cdot c \\
\Phi(1,0) & - & \Phi(1,0) & & & \leq & c \\
\Phi(0,0) & - & \Phi(1,0) & & & \leq & c \\
\Phi(0,2) & - & \Phi(0,2) & & & \leq & 2\cdot c \\
\Phi(1,2) & - & \Phi(0,2) & & & \leq & 2\cdot c \\
\Phi(0,1) & - & \Phi(0,2) & + & 1 & \leq & 0 \\
\Phi(1,2) & - & \Phi(1,2) & & & \leq & 0 \\
\Phi(0,1) & - & \Phi(1,2) & + & 1 & \leq & 2\cdot c \\
\Phi(1,1) & - & \Phi(1,2) & + & 1 & \leq & c \\
\Phi(0,2) & - & \Phi(1,2) & & & \leq & c \\
\Phi(0,2) & - & \Phi(0,1) & & & \leq & 2\cdot c \\
\Phi(1,2) & - & \Phi(0,1) & & & \leq & 2\cdot c \\
\Phi(0,0) & - & \Phi(0,1) & + & 2 & \leq & 0 \\
\Phi(1,2) & - & \Phi(1,1) & & & \leq & 0 \\
\Phi(0,0) & - & \Phi(1,1) & + & 2 & \leq & 2\cdot c \\
\Phi(1,0) & - & \Phi(1,1) & + & 2 & \leq & c \\
\Phi(0,1) & - & \Phi(1,1) & & & \leq & c
\end{array}
$$

Figure 4.7: LP formulation of the costs associated with the state transitions.

*most* $\frac{5}{2}$ *times* $C_{\text{OFF}}(\sigma, u, v)$.

*Proof.* Once a request $q$ in $\sigma$ is initiated in a quiescent state, without loss of generality, we assume that RWW executes $q$, and then OFF executes $q$.

We construct a new request sequence $\sigma'(u, v)$ from $\sigma(u, v)$ as follows: (1) insert a *noop* request in the beginning and at the end of $\sigma(u, v)$; (2) insert a *noop* request between every pair of successive requests in $\sigma(u, v)$.

In the rest of the proof, first, for both RWW and OFF, we argue that we can charge each of the *type*$(u, v)$ messages to a request in $\sigma'(u, v)$. Then, to prove the lemma, we use potential function arguments to show that $C_{\text{RWW}}(\sigma'(u, v), u, v)$ is at most $\frac{5}{2}$ times $C_{\text{OFF}}(\sigma'(u, v), u, v)$.

For RWW, from Lemma 4.5.5, the following equality holds.

$$C_{\text{RWW}}(\sigma, u, v) = C_{\text{RWW}}(\sigma(u, v), u, v)$$

For RWW, we do not charge any message to a *noop* request in $\sigma'(u, v)$. Hence, we have, $C_{\text{RWW}}(\sigma, u, v) = C_{\text{RWW}}(\sigma'(u, v), u, v)$.

For OFF, from Lemma 4.4.3, during the execution of a *combine* request in $\sigma(v, u)$, no *type*$(u, v)$ messages are sent. Also from Lemma 4.4.5 and part 3 of Lemma 4.4.8, during the execution of a *write* request in $\sigma(v, u)$ by OFF, only a *release* message from $v$ to $u$ can be sent. Consider a *type*$(u, v)$ *release* message $m$ sent during the execution of a *write* request $q$ in $\sigma(v, u)$ by OFF. On receiving $m$, $u.granted[v]$ is falsified. From Lemmas 4.4.3, 4.4.5, 4.4.6, and parts 3 and 4 of Lemma 4.4.8, $u.granted[v]$ is not set to **true** before executing another *combine* request in $\sigma(u, v)$. Hence, at most one *type*$(u, v)$ *release* message can be associated with a *noop* request. Thus, we can associate all *type*$(u, v)$ messages with a request in $\sigma'(u, v)$.

Therefore, in comparing $C_{\mathrm{RWW}}(\sigma, u, v)$ and $C_{\mathrm{OFF}}(\sigma, u, v)$, we can restrict our attention to messages sent in executing requests in $\sigma'(u, v)$ .

For the ordered pair $(u, v)$, in Figure 4.6, we show a state diagram depicting possible changes in $F_{\mathrm{RWW}}(u, v)$ and $F_{\mathrm{OFF}}(u, v)$ in executing a request from $\sigma'(u, v)$. In the state diagram, a state labeled $S(x, y)$ represent a state of the algorithms in which $F_{\mathrm{OFF}}(u, v)$ is $x$ and $F_{\mathrm{RWW}}(u, v)$ is $y$. Observe that the change in $F_{\mathrm{RWW}}(u, v)$ in executing a request is deterministic as specified by the algorithm in Figure 4.5. On the other hand, the change in $F_{\mathrm{OFF}}(u, v)$ in executing a request is not known in advance. Hence, more than one possible changes in $F_{\mathrm{OFF}}(u, v)$ in executing a request are depicted by non-deterministic state transitions. Recall that the cost of processing a request in a particular configuration for any lease-based algorithm is given in Figure 4.4.

We define a potential function $\Phi(x, y)$ as a mapping from a state $S(x, y)$ to a positive real number. The amortized cost of any transition is defined as the sum of the change in potential $\Delta(\Phi)$ and the cost of RWW in the transition. For any transition, we write that the amortized cost is at most $c$ times the cost of OFF in the transition, where $c$ is a constant factor to be determined. We solve these inequalities by formulating a linear program with an objective function to minimize $c$ (see Figure 4.7). By solving the linear program, we get $c = \frac{5}{2}$, $\Phi(0, 0) = 0$, $\Phi(0, 1) = 2$, $\Phi(0, 2) = 3$, $\Phi(1, 0) = \frac{5}{2}$, $\Phi(1, 1) = 2$, and

$\Phi(1,2) = \frac{1}{2}$.

Hence, for any state transition due to the execution of a request $q$ from $\sigma'(u,v)$, the amortized cost is at most $\frac{5}{2}$ times the cost of OFF in the execution of $q$. Recall that, in the initial quiescent state, $F_{\text{RWW}}(u,v)$ and $F_{\text{OFF}}(u,v)$ are 0, and the potential for any state is non negative. Therefore, in execution of $\sigma'(u,v)$, the total cost of RWW is at most $\frac{5}{2}$ times that of OFF. That is, $C_{\text{RWW}}(\sigma,u,v)$ is at most $\frac{5}{2}$ times $C_{\text{OFF}}(\sigma,u,v)$. □

**Theorem 8.** *Algorithm* RWW *is* $\frac{5}{2}$*-competitive with respect to any lease-based algorithm for the sequential aggregation problem.*

*Proof.* From Lemma 4.5.8, in a sequential execution of a request sequence $\sigma$, for any two neighboring nodes $u$ and $v$, $C_{\text{RWW}}(\sigma,u,v)$ is at most $\frac{5}{2}$ times $C_{\text{OFF}}(\sigma,u,v)$. By symmetry, $C_{\text{RWW}}(\sigma,v,u)$ is at most $\frac{5}{2}$ times $C_{\text{OFF}}(\sigma,v,u)$. Hence, the total number of messages exchanged between $u$ and $v$ in the execution of $\sigma$ by RWW is at most $\frac{5}{2}$ times that of OFF. Summing over all the pairs of neighboring nodes, we find that $C_{\text{RWW}}(\sigma)$ is at most $\frac{5}{2}$ times $C_{\text{OFF}}(\sigma)$. Hence, the theorem follows. □

**Theorem 9.** *Algorithm* RWW *is* 5*-competitive with respect to any nice algorithm for the sequential aggregation problem.*

*Proof sketch:* Let $\text{OPT}_\text{N}$ be an optimal nice algorithm for the sequential aggregation problem. Consider any pair of neighboring nodes $(u,v)$. We compare the cost of RWW and $\mathrm{OPT_N}$ in executing request sequences $\sigma(u,v)$ and $\sigma(v,u)$ separately.

First, consider the execution of requests in $\sigma(u,v)$. We define an *epoch* as follows. The first epoch starts at the beginning of the request sequence. An epoch ends with a *write* to *combine* transition in $\sigma(u,v)$, and a new epoch starts at the same instant. By the definition of a nice algorithm, $\mathrm{OPT_N}$ provides strict consistency for the sequential execution problem. Hence, $\mathrm{OPT_N}$ sends at least one message in any epoch. We are able to show that the algorithm RWW sends at most 5 messages in any epoch (follows from Lemma 4.5.3). Summing over all the epochs, we get that the cost of RWW in executing $\sigma(u,v)$ is at most 5 times that of $\mathrm{OPT_N}$. By symmetry, the cost of RWW in executing $\sigma(v,u)$ is at most 5 times that of $\mathrm{OPT_N}$. By summing over all pair of neighboring nodes, the desired result follows. □

**Theorem 10.** *For any lease-based algorithm A, there exists a request sequence $\sigma$ and an offline algorithm such that the cost A in executing $\sigma$ is at least $\frac{5}{2}$ times that of the offline algorithm.*

*Proof sketch:* We give an adversarial request generating argument to sketch the desired result. Consider an example of a tree consisting of just two nodes $u$ and $v$ such that there is an edge between $u$ and $v$. The adversarial request generating algorithm ADV is as follows. The algorithm ADV generates $a$ *combine* requests at $v$ such that there is a lease from $u$ to $v$ after execution of the $a$th request. And then, ADV generates $b$ *write* requests at $u$ such that there is no lease from $u$ to $v$ after execution of the $b$th request. Using potential function arguments, we can show that, for a sufficient long request sequence $\sigma$ generated by ADV, the cost of $A$ in executing $\sigma$ is at least $\frac{5}{2}$ times that of an optimal offline algorithm. $\square$

## 4.6 Consistency Results for Concurrent Executions

In this section we generalize the traditional definition of causal consistency [3] for the aggregation problem, and show that any lease-based aggregation algorithm is causally consistent. As mentioned earlier, the key difference between the consistency problem formulation in [3] and our consistency problem formulation is in reading one value compared to aggregating values from all nodes.

### 4.6.1 Definitions

**Request**. In this section we find it convenient to extend the definition of a request from Section 4.2 as follows. In the extended definition, a request $q$ is a tuple $(node, op, arg, retval, index)$, where (1) *node* is the node where the request is initiated; (2) *op* is the type of the request, *combine*, *gather*, or *write*; (3) *arg* is the argument of the request (if any); (4) *retval* is the return value of the request (if any); and (5) *index* is the number of requests that are generated at $q.node$ and completed before $q$ is completed.

An aggregation algorithm executes *write* and *combine* requests as described in Section 4.2. To execute a *gather* request, an aggregation algorithm returns a set $A$ of pairs of the form $(\mathit{node}, \mathit{index})$ such that (1) for each node $u$ in $T$, there is a tuple $(u, i)$ in $A$, where $i \geq -1$; (2) for any tuple $(u, i)$ in $A$, if $i \geq 0$, then there is a *write* request $q$ such that $q.\mathit{node} = u$ and $q.\mathit{index} = i$; and (3) $|A|$ is equal to the number of nodes in $T$.

**Miscellaneous**. In this section, we extend the definition of function $f$ from Section 4.2 as follows. In the extended definition, $f$ can also take a set of pairs $A$ of the form $(\mathit{node}, \mathit{index})$ as an argument. For a set of pairs $A$ of the form $(u, \mathit{index})$, $f(A)$ is defined to be $f(B)$, where $B$ is a set of *write* requests constructed as follows: for any tuple $(u, i)$ in $A$ with $i \geq 0$, include the *write* request $q$ in $B$ such that $q.\mathit{node} = u$ and $q.\mathit{index} = i$.

A *combine-write* sequence (set) is a sequence (set) of requests containing only *combine* and *write* requests. A *gather-write* sequence (set) is a sequence (set) of requests containing only *gather* and *write* requests. Let $A$ be a set of requests. Then, $\mathit{pruned}(A, u)$ is a subset of $A$ such that, for any request $q$ in $A$, $q$ is in $\mathit{pruned}(A, u)$ if and only if $q.\mathit{op} = \mathit{write}$ or $q.\mathit{node} = u$.

For any sequence of requests $S$ and any request $q$ in $S$, we define $\mathit{recentwrites}(S, q)$ as a set of pairs such that the size of $\mathit{recentwrites}(S, q)$ is equal to the number of nodes in $T$, and for any node $u$ in $T$, the following conditions hold: (1) if $q'$ is the most recent *write* request at $u$ preceding $q$ in $S$, then $(u, q'.\mathit{index})$ is in $\mathit{recentwrites}(S, q)$; (2) if there is no *write* request at $u$ preceding $q$ in $S$, in which case, $(u, -1)$ is in $\mathit{recentwrites}(S, q)$.

Let $A$ be a gather-write set, and $S$ be a linear sequence of all the requests in $A$. Then, $S$ is called a *serialization* of $A$ if for any *gather* request $q$ in $S$, $q.\mathit{retval} = \mathit{recentwrites}(S, q)$.

For any two request sequences $\tau$ and $\tau'$, $\tau - \tau'$ is defined to be the subsequence of $\tau$ containing all requests $q$ in $\tau$ such that $q$ is not present in $\tau'$. For any two request sequences $\tau$ and $\tau'$, $\tau.\tau'$ is defined to be $\tau$ appended by $\tau'$.

**Compatibility**. Let $q_1$ be a *combine* or *write* request and $q_2$ be a *gather* or *write* request. Then, $q_1$ and $q_2$ are *compatible* if the following conditions hold: (1) $q_1.\mathit{op} = \mathit{write}$ and $q_1 = q_2$; or (2) $q_1.\mathit{op} = \mathit{combine}$, $q_2.\mathit{op} = \mathit{gather}$, $q_1.\mathit{retval} = f(q_2.\mathit{retval})$, and the *node*, *arg*, and *index* fields are equal for $q_1$ and $q_2$. A combine-write sequence $\tau$ and a gather-write sequence $\tau'$ are compatible if the following conditions hold: (1) $\tau$ and $\tau'$ are of equal length; (2) for all indices $i$, $\tau(i)$ and $\tau'(i)$ are compatible. Let $A$ be a combine-write set and $B$ be a gather-write set. Then, $A$ and $B$ are compatible if for any node $u$ in $T$ there exists a linear sequence $S$ of all requests in $\mathit{pruned}(A, u)$, and a linear sequence $S'$ of all requests in $\mathit{pruned}(B, u)$, such that $S$ and $S'$ are compatible.

**Causal Consistency**. We define a notion of *causal ordering* ($\rightsquigarrow$) between any two requests $q_1$ and $q_2$ in a gather-write execution-history $A$ as follows. First, $q_1 \overset{1}{\rightsquigarrow} q_2$ if at least one of the following two conditions hold: (1) $q_1.\mathit{node} = q_2.\mathit{node}$ and $q_1.\mathit{index} < q_2.\mathit{index}$; (2) $q_1$ is a write request, $q_2$ is a *gather* request, and $q_2$ returns $(q_1.\mathit{node}, q_1.\mathit{index})$ in $q_2.\mathit{retval}$. Second, $q_1 \overset{i+1}{\rightsquigarrow} q_2$ if there exists a request $q'$ such that $q_1 \overset{i}{\rightsquigarrow} q' \overset{1}{\rightsquigarrow} q_2$. Finally, $q_1 \rightsquigarrow q_2$ $q_1 \overset{i}{\rightsquigarrow} q_2$, for some $i$.

The execution-history of an aggregation algorithm is defined as the set of all requests executed by the algorithm. A gather-write execution-history $A$ is *causally consistent* if, for any node $u$ in $T$, there exists a serialization $S$ of $pruned(A, u)$ such that $S$ respects the causal ordering $\rightsquigarrow$ among all the requests in $pruned(A, u)$. A combine-write execution-history $A$ is causally consistent if there exists a gather-write execution-history $B$ such that $A$ and $B$ are compatible and $B$ is causally consistent.

### 4.6.2 Algorithm

In Figure 4.9, we present the mechanism for any lease-based aggregation algorithm with *ghost actions* (in curly braces). The ghost actions are included for the purpose of our analysis.

For any node $u$, $u.log$ is a ghost variable. For any node $u$, $u.wlog$ is a subsequence of $u.log$ containing all the *write* requests in $u.log$.

Initially, for any node $u$, $u.val := 0$, $u.uaw := \emptyset$, $u.pndg := \emptyset$, $u.upcntr := 0$, $u.sntupdates := \emptyset$. For each node $v$ in $u.nbrs()$, $u.taken[v] :=$ **false**, $u.granted[v] :=$ **false**, $u.aval[v] := 0$, $u.snt[v] := \emptyset$, and $u.log$ is empty.

Function $request(combine)$ generates and returns a *combine* request $q'$ as follows: $q'.node = u$, $q'.op = combine$, $q'.arg = \emptyset$, $q'.retval = gval()$, and $q'.index$ is one greater than the number of completed requests at $u$. Function $request(write, q)$ generates and returns a *write* request $q'$ as follows: $q'.node =$

```
         $node\ u$
         var $taken$ : array$[v_1 \ldots v_k]$ of boolean;
           $granted$ : array$[v_1 \ldots v_k]$ of boolean;
           $aval$ : array$[v_1 \ldots v_k]$ of real;    $val$ : real;
           $uaw[]$ : array$[v_1, \ldots, v_k]$ of set {int};
           $pndg$ : set $\{node\}$;
           $snt[]$ : array$[v_1, \ldots, v_k]$ of set $\{node\}$;
           $upcntr$ : int; $sntupdates$ : set $\{\{node, \textbf{int}, \textbf{int}\}\}$;
         begin
$T_1$        true $\rightarrow \{combine\ q\}$
 1             $\underline{oncombine}(u)$;
 2             foreach $v \in tkn()$ do
 3               $uaw[v] := \emptyset$; od
 4             if $u \notin pndg \rightarrow$
 5               if $nbrs() \setminus tkn() = \emptyset \rightarrow$
 6                 {append $request(combine)$ to $log$};
 7                 return $gval()$;
 8               $\square$ $nbrs() \setminus tkn() \neq \emptyset \rightarrow$
 9                 $sendprobes(u)$;
10                 $snt[u] := nbrs() \setminus tkn()$; fi fi
$T_2$        true $\rightarrow \{write\ q\}$
 1             $val := q.arg$; {append $request(write, q)$ to $log$}
 2             if $grntd() \neq \emptyset$ $\rightarrow$
 3               $id := newid()$;
 4               $forwardupdates(u, id)$; fi
$T_3$        $\square$ rcv $probe()$ from $w \rightarrow$
 1             $\underline{probercvd}(w)$;
 2             foreach $v \in tkn() \setminus \{w\}$ do
 3               $uaw[v] := \emptyset$; od
 4             if $w \notin pndg \rightarrow$
 5               if $nbrs() \setminus \{tkn() \cup \{w\}\} = \emptyset \rightarrow$
 6                 $sendresponse(w)$;
 7               $\square$ $nbrs() \setminus \{tkn() \cup \{w\}\} \neq \emptyset \rightarrow$
 8                 $sendprobes(w)$;
 9                 $snt[w] := nbrs() \setminus \{tkn() \cup \{w\}\}$; fi fi
```

```
$T_4$        $\square$ rcv $response(x, flag)$ from $w$ $\rightarrow$
             {rcv $response(wlog_w, flag)$ from $w$} $\rightarrow$
 1             $\underline{responsercvd}(flag, w)$;
 2             $\underline{aval}[w] := x$; $\{log := log.(wlog_w - log)\}$;
 3             $taken[w] := flag$;
 4             foreach $v \in pndg$ do
 5               $snt[v] := snt[v] \setminus \{w\}$;
 6               if $snt[v] = \emptyset \rightarrow$
 7                 $pndg := pndg \setminus \{v\}$;
 8                 if $v = u \rightarrow$
 9                   {append $request(combine)$ to $log$};
10                   return $gval()$;
11                 $\square$ $v \neq u \rightarrow$
12                   $sendresponse(v)$; fi fi od
$T_5$        $\square$ rcv $update(x, id)$ from $w$ $\rightarrow$
             {rcv $update(wlog_w, id)$ from $w$ } $\rightarrow$
 1             $\underline{updatercvd}(w)$;
 2             $\underline{aval}[w] := x$; $\{log := log.(wlog_w - log)\}$;
 3             $uaw[w] := uaw[w] \cup id$;
 4             if $grntd() \setminus \{w\} \neq \emptyset \rightarrow$
 5               $nid = newid()$;
 6               $sntupdates := sntupdates \cup \{w, id, nid\}$;
 7               $forwardupdates(w, nid)$;
 8             $\square$ $grntd() \setminus \{w\} = \emptyset \rightarrow$
 9               $forwardrelease()$; fi
$T_6$        $\square$ rcv $release(S)$ from $w \rightarrow$
 1             $\underline{releasercvd}(w)$;
 2             $granted[w] :=$ false;
 3             $onrelease(w, S)$;
           end
```

Figure 4.8: The mechanism for any lease-based aggregation algorithm with ghost actions. Nodes $\{v_1, \ldots, v_k\}$ refer to the neighbors of node $u$.

```
procedure $sendprobes(node\ w)$
  $pndg := pndg \cup \{w\}$;
  foreach $v \in nbrs() \setminus \{tkn() \cup snt \cup \{w\}\}$ do
    send $probe()$ to $v$; od

procedure $forwardupdates(node\ w, int\ id)$
  foreach $v \in grntd() \setminus \{w\}$ do
    send $update(subval(v), id)$ to $v$;
    {send $update(wlog, id)$ to $v$}; od

procedure $sendresponse(node\ w)$
  if $(nbrs() \setminus \{tkn() \cup \{w\}\} = \emptyset) \rightarrow$
    $granted[w] := \underline{setlease}(w)$; fi
  send $response(subval(w), granted[w])$ to $w$;
  {send $response(wlog, granted[w])$ to $w$; }

boolean $isgoodforrelease(node\ w)$
  return $(grntd() \setminus \{w\} = \emptyset)$;

procedure $onrelease(node\ w,$ set $S)$
  Let $id$ is the smallest id in $S$;
  foreach $v \in tkn() \setminus \{w\}$ do
    Let $A$ be the set of tuples $\alpha$ in $sntupdates$
      such that $\alpha.node = v$ and $\alpha.sntid \geq id$;
    Let $\beta$ be a tuple in $A$
      such that $\beta.rcvid \leq \alpha.rcvid$, for all $\alpha$ in $A$;
    Let $S'$ be the set of ids in $uaw[v]$ with ids $\geq \beta.rcvid$;
    $uaw[v] := S'$;
    if $isgoodforrelease(v) \rightarrow$
      $\underline{releasepolicy}(v)$;
    fi
  od
  $forwardrelease()$;
```

```
procedure $forwardrelease()$
  foreach $v \in tkn()$ do
    if $isgoodforrelease(v) \rightarrow$
      if $taken[v] \wedge \underline{breaklease}(v) \rightarrow$
        $taken[v] :=$ false;
        send $release(uaw[v])$ to $v$;
        $uaw[v] := \emptyset$; fi fi od

int $newid()$
  $upcntr := upcntr + 1$;
  return $upcntr$;

real $gval()$
  $x := val$;
  foreach $v \in nbrs()$ do
    $x := f(x, aval[v])$; od
  return $x$;

real $subval(node\ w)$
  $x := val$;
  foreach $v \in nbrs() \setminus \{w\}$ do
    $x := f(x, aval[v])$; od
  return $x$;

set $nbrs()$
  return the set of neighboring nodes;
set $tkn()$
  return $\{v \mid v \in nbrs() \wedge taken[v] =$ true$\}$;
set $grntd()$
  return $\{v \mid v \in nbrs() \wedge granted[v] =$ true$\}$;
```

Figure 4.9: Procedures used in the mechanism for any lease-based algorithm with ghost actions. Nodes $\{v_1, \ldots, v_k\}$ refer to the neighbors of node $u$.

$u$, $q'.op = \mathit{write}$, $q'.\mathit{arg} = q.\mathit{arg}$, $q'.\mathit{retval} = \emptyset$, and $q'.\mathit{index}$ is one greater than the number of completed requests at $u$.

### 4.6.3 Analysis

For each node $u$ in $T$, we construct a gather-write sequence $u.\mathit{gwlog}$ from $u.\mathit{log}$ as follows: (1) if $u.\mathit{log}(i)$ is a *write* request then $u.\mathit{gwlog}(i) = u.\mathit{log}(i)$; (2) if $u.\mathit{log}(i)$ is a *combine* $q_1$, then $u.\mathit{gwlog}(i)$ is a *gather* $q_2$ such that $q_2.\mathit{node} = q_1.\mathit{node}$, $q_2.op = \mathit{gather}$, $q_2.\mathit{index} = q_1.\mathit{index}$, and $q_2.\mathit{retval} = \mathit{recentwrites}(u.\mathit{log}, q_1)$.

For each node $u$ in $T$, we construct $u.\mathit{log}'$ and $u.\mathit{gwlog}'$ from $u.\mathit{log}$ and $u.\mathit{gwlog}$ as follows. First, initialize $u.\mathit{log}'$ to $u.\mathit{log}$, and $u.\mathit{gwlog}'$ to $u.\mathit{gwlog}$. Then, for each node $v$ in $T$ except $u$, repeat the following steps: (1) $u.\mathit{log}' = u.\mathit{log}'.(v.\mathit{wlog} - u.\mathit{log}')$; (2) $u.\mathit{gwlog}' = u.\mathit{gwlog}'.(v.\mathit{wlog} - u.\mathit{gwlog}')$.

For any set of nodes $A$ and a request sequence $\tau$, $\mathit{recent}(A, \tau)$ returns a set of $|A|$ pairs such that, for any node $u$ in $A$, the following conditions hold: (1) if $q'$ is the most recent *write* request at $u$ in $\tau$, then $(u, q'.\mathit{index})$ is in $\mathit{recent}(\tau, q)$; (2) if there is no *write* request at $u$ in $\tau$, then $(u, -1)$ is in $\mathit{recent}(S, q)$.

For a set of nodes $A$, a real value $x$, and a request sequence $\tau$, we define $\mathit{corresponds}(A, x, \tau)$ to be **true** if $x = f(\mathit{recent}(A, \tau))$.

For a set of nodes $A$ and a request sequence $\tau$, $\mathit{projectwrites}(A, \tau)$ returns the subsequence of $\tau$ containing all of the *write* requests at any node in $A$.

For request sequences $\tau$ and $\tau'$, *prefix*$(\tau, \tau')$ is defined to be **true** if $\tau'$ is a prefix of $\tau$.

**Lemma 4.6.1.** *For any update or response message $m$ from a node $v$ to a neighboring node $u$, let $S$ be $v.wlog$ after $m$ has been sent. Then, prefix$(S, m.wlog)$ holds.*

*Proof.* By inspection of the code (*forwardupdates*() and *sendresponse*() procedures), $m.wlog = v.wlog$ when $m$ is sent. Since $v.wlog$ grows only at the end, the lemma follows. □

**Lemma 4.6.2.** *For any two update or response messages $m_1$ and $m_2$ sent from any node $v$ to any neighboring node $u$ such that $m_2$ is sent after $m_1$, prefix$(m_2.wlog, m_1.wlog)$ holds.*

*Proof.* By Lemma 4.6.1, $m_1.wlog$ is a prefix of $v.wlog$ after $m_1$ has been sent. By inspection of the code (*forwardupdates*() and *sendresponse*()), $m_2.wlog = v.wlog$ when $m_2$ is sent. Hence, the lemma follows. □

**Lemma 4.6.3.** *Just before the execution of $T_4$ (resp., $T_5$) at $u$, on receiving a response message (resp., an update message) $m$ sent from $v$, let sequence $\tau$ be projectwrites$(A, m.wlog)$ and sequence $\tau'$ be projectwrites$(A, u.log)$, where $A = subtree(v, u)$. Then the following conditions hold: (1) prefix$(\tau, \tau')$ holds; (2) projectwrites$(nodes(T) \setminus A, m.wlog - u.log)$ is empty.*

*Proof.* We prove (1) by induction on the number of *update* or *response* messages from $v$ to $u$.

Base case. Since $v.\mathit{granted}[u]$ does not hold initially, the first message of interest is a *response* message $m$. Since $u$ receives any *write* requests in $A$ only from $v$, $\tau'$ is empty. Hence, $\mathit{prefix}(\tau, \tau')$ holds.

Induction hypothesis. Just before receiving the $n$th message $m$ at node $u$, $\mathit{projectwrites}(A, u.\mathit{log})$ is prefix of $\mathit{projectwrites}(A, m.\mathit{wlog})$.

Induction step. Since the communication channels are FIFO, the $(n+1)$st *update* or *response* message $m$ reaches $u$ after the $n$th message $m'$. By the induction hypothesis, just before receiving $m'$, $\mathit{projectwrites}(A, u.\mathit{log})$ is prefix of $\mathit{projectwrites}(A, m'.\mathit{wlog})$. In line 2 of $T_4$ (resp., $T_5$), $u.\mathit{log} = u.\mathit{log}.(m'.\mathit{wlog} - u.\mathit{log})$, that is, all *write* requests in $m'.\mathit{wlog}$ not present in $u.\mathit{log}$ are appended to $u.\mathit{log}$. Hence, $\mathit{projectwrites}(A, u.\mathit{log}) = \mathit{projectwrites}(A, m'.\mathit{wlog})$ after execution of Line 2 of $T_4$ (resp., $T_5$).

By Lemma 4.6.2, $m'.\mathit{wlog}$ is a prefix of $m.\mathit{wlog}$. Hence, just before receiving $m$, $\mathit{projectwrites}(A, u.\mathit{log})$ is a prefix of $\mathit{projectwrites}(A, m.\mathit{wlog})$.

We prove (2) as follows. Let $B$ be $\mathit{nodes}(T) \setminus A$. By Lemmas 4.6.1, 4.6.2, and condition (1), at any instant $\mathit{projectwrites}(B, v.\mathit{log})$ is a prefix of $\mathit{projectwrites}(B, u.\mathit{log})$. By Lemma 4.6.1, $m.\mathit{wlog}$ is a prefix of $v.\mathit{wlog}$ after $m$ has been sent. Hence, just before receiving $m$, $\mathit{projectwrites}(B, m.\mathit{wlog})$ is a prefix of $\mathit{projectwrites}(B, u.\mathit{log})$. Therefore, $\mathit{projectwrites}(B, m.\mathit{wlog} - u.\mathit{log})$

is empty. □

For any node $u$, we define predicates $I_1(u)$, $I_2(u)$, and $I_3(u)$ as follows: (1) predicate $I_1(u)$ holds if $\mathit{corresponds}(A, u.\mathit{gval}(), u.\mathit{log})$ holds, where $A$ is the set of all nodes in $T$; (2) predicate $I_2(u)$ holds if, for any *update* or *response* message $m$ from $u$ to any node $v$ in $u.\mathit{nbrs}()$, $\mathit{corresponds}(A, m.x, m.\mathit{wlog})$ holds, where $A$ is the set of all nodes in $\mathit{subtree}(u, v)$; and (3) predicate $I_3(u)$ holds if, for any node $v$ in $u.\mathit{nbrs}()$, $\mathit{corresponds}(A, u.\mathit{aval}[v], u.\mathit{log})$ holds, where $A$ is the set of all nodes in $\mathit{subtree}(v, u)$. Let $I(u)$ be $I_1(u) \wedge I_2(u) \wedge I_3(u)$.

**Lemma 4.6.4.** *For any node $u$, if $I_1(u)$ and $I_3(u)$ hold just before an update or a response message $m$ is sent from $u$ to any node $v$ in $u.\mathit{nbrs}()$, then $\mathit{corresponds}(A, m.x, m.\mathit{wlog})$, where $A = \mathit{subtree}(u, v)$.*

*Proof.* Initially, $u.\mathit{val}$ is 0 and $u.\mathit{log}$ is empty. Hence, initially,

$$u.\mathit{val} \quad = \quad f(\mathit{recent}(\{u\}, u.\mathit{log})) \tag{4.2}$$

The only line of code that modifies $u.\mathit{val}$ is Line 1 of $T_2$. This line preserves Equation (4.2). Hence, Equation (4.2) holds just before sending any *update* or *response* message.

In the following equation, let $\{v_1, \ldots, v_k\} = u.\mathit{nbrs}() \setminus \{v\}$ and $S_i =$

*subtree*$(v_i, u)$.

$$\begin{aligned} m.x &= u.subval(v) \\ &= f(u.val, u.aval[v_1], \ldots, u.aval[v_k]) \\ &= f(f(recent(\{u\}, u.log)), f(recent(S_1, u.log)), \ldots, f(recent(S_k, u.log))) \\ &= f(recent(\{u\} \cup S_1 \cup \cdots \cup S_k), u.log) \\ &= f(recent(A, u.log)) \\ &= f(recent(A, m.wlog)) \qquad (4.3) \end{aligned}$$

The first equality above follows from the algorithm. The second equality follows from the definition of *subval*$(v)$. The third equality follows from $I_3(u)$ and Equation (4.2). The fourth and fifth equalities follows from the fact that $\{u\}, S_1, \ldots, S_k$ are disjoint sets of nodes and their union is *subtree*$(T, u, v)$. The last equality follows from the fact that $m.wlog = wlog$ and $recent(A, log) = recent(A, wlog)$.

Hence, the lemma follows. □

**Lemma 4.6.5.** *For any node $u$, $I(u)$ is an invariant.*

*Proof.* Initially, for any node $u$, $u.gval()$ is 0 and $u.log$ is empty. Hence, $I_1(u)$ holds. There are no *update* or *response* messages. Hence, $I_2(u)$ holds. For any node $v$ in $u.nbrs()$, $u.aval[v]$ is 0 and $u.log$ is empty. Hence, $I_3(u)$ holds. Therefore, $I(u)$ holds initially. Thus it is sufficient to check that every action preserves $I(u)$. In the following we present the reasons why every action preserves $I(u)$.

$\{I(u)\}T_1\{I(u)\}$. In the execution of $T_1$, for any node $v$ in $u.nbrs()$, $u.aval[v]$ and $u.val$ remain unchanged. No *update* or *response* messages are generated in execution of $T_1$. No *write* request is added to $u.log$. Hence, $I_1(u)$, $I_2(u)$, and $I_3(u)$ are not affected in execution of $T_1$.

$\{I(u)\}T_2\{I(u)\}$. In the execution of $T_2$, only part of the code affecting $I_1(u)$ is Line 1. Note that Line 1 does not affect $I_2(u)$ and $I_3(u)$. In the following equation, let $\{v_1, \ldots, v_k\} = u.nbrs()$ and $S_i = subtree(T, v_i, u)$.

$$\begin{aligned} f(u.aval[v_1], \ldots, u.aval[v_k]) &= f(f(recent(S_1, u.log)), \ldots, f(recent(S_k, u.log))) \\ &= f(recent(S_1, u.log) \cup \cdots \cup recent(S_k, u.log)) \\ &= f(recent(S_1 \cup \cdots \cup S_k, u.log) \\ &= f(recent(nodes(T) \setminus \{u\}, u.log)) \qquad (4.4) \end{aligned}$$

The first equality above follows from $I_3(u)$. The second equality follows from the fact that $S_1, \ldots, S_k$ are disjoint sets of nodes.

Let $q$ be the *write* request appended to $u.log$ in Line 1. After Line 1, $val$ is $q.arg$, and $\{q\}$ is $recent(\{u\}, log)$. Hence, after Line 1,

$$u.val = f(recent(\{u\}, u.log)) \qquad (4.5)$$

Therefore, after Line 1,

$$
\begin{aligned}
u.\mathit{gval}() &= f(u.\mathit{val}, u.\mathit{aval}[v_1], \ldots, u.\mathit{aval}[v_k]) \\
&= f(u.\mathit{val}, f(u.\mathit{aval}[v_1], \ldots, u.\mathit{aval}[v_k])) \\
&= f(f(\mathit{recent}(\{u\}, u.\mathit{log})), f(\mathit{recent}(\mathit{nodes}(T) \setminus \{u\}, u.\mathit{log})) \\
&= f(\mathit{recent}(\{u\}, u.\mathit{log}) \cup \mathit{recent}(\mathit{nodes}(T) \setminus \{u\}, u.\mathit{log})) \\
&= f(\mathit{recent}(\mathit{nodes}(T), u.\mathit{log})) \qquad (4.6)
\end{aligned}
$$

The first equality above follows from the definition of $u.\mathit{gval}()$. The second equality follows from the associativity property of $f$. The third equality follows from Equations (4.4) and (4.5).

Hence, $\mathit{corresponds}(\mathit{nodes}(T), u.\mathit{gval}(), u.\mathit{log})$ holds after Line 1. That is, $I_1(u)$ holds after Line 1. Therefore, for each line of the code in $T_2$, if $I_1(u) \wedge I_2(u) \wedge I_3(u)$ holds before the execution of the line, then $I_1(u)$ holds after execution of the line.

In the execution of $T_2$, the only part of the code affecting $I_2(u)$ is the invocation of procedure *forwardupdates*() in Line 4. By Lemma 4.6.4, $I_2(u)$ holds after Line 4. Therefore, for each line of code in $T_2$, if $I_1(u) \wedge I_2(u) \wedge I_3(u)$ holds before the execution of the line, then $I_2(u)$ holds after execution of the line.

In $T_2$, $I_3(u)$ is not affected.

$\{I(u)\}T_3\{I(u)\}$. Predicates $I_1(u)$ and $I_3(u)$ are not affected in the execution of $T_3$. The only part of the code that affects $I_2(u)$ is the invocation of procedure *sendresponse*() in Line 6. By Lemma 4.6.4, $I_2(\mathit{node})$ holds after Line 6.

$\{I(u)\}T_4\{I(u)\}$. The only lines that affect $I(u)$ are Lines 2 and 12. Line 2 does not affect $I_2(u)$, but affects $I_1(u)$ and $I_3(u)$ since the line modifies $u.\mathit{aval}[w]$ and $u.\mathit{log}$. First we show that $I_3(u)$ is preserved in Line 2, and so $I_1(u)$ is also preserved.

Let $m$ be the *response* message received and $A$ be the set of nodes in $\mathit{subtree}(w, u)$. By part (1) of Lemma 4.6.3, after the execution of Line 2, $u.\mathit{aval}[w] = m.x$ and $\mathit{recent}(A, u.\mathit{log}) = \mathit{recent}(A, m.\mathit{wlog})$. Hence, by $I_2(u)$, $u.\mathit{aval}[w] = f(\mathit{recent}(A, u.\mathit{log}))$.

By part (2) of Lemma 4.6.3, for all $v$ in $u.\mathit{nbrs}()\backslash\{w\}$, $\mathit{recent}(B, u.\mathit{log})$ is not affected, where $B = \mathit{subtree}(v, u)$, and so, $\mathit{corresponds}(B, u.\mathit{aval}[v], u.\mathit{log})$ remains unchanged. By the preceding paragraph, after the execution of Line 2, $u.\mathit{aval}[w] = f(\mathit{recent}(A, u.\mathit{log}))$, where $A$ be the set of nodes in $\mathit{subtree}(w, u)$. Therefore, $I_3(u)$ is preserved in Line 2, and hence, preserved in the execution of $T_4$.

By part (2) of Lemma 4.6.3, $\mathit{recent}(\{u\}, u.\mathit{log})$ is not affected. Therefore, $I_1(u)$ is preserved in Line 2, and hence is preserved in the execution of $T_4$.

Line 12 only affects $I_2(u)$. By Lemma 4.6.4, $I_2(u)$ holds in Line 12.

Therefore, $I_1(u) \wedge I_2(u) \wedge I_3(u)$ is preserved in the execution of $T_4$.

$\{I(u)\}T_5\{I(u)\}$. The only lines that affect $I(u)$ are Lines 2 and 7. Line

2 does not affect $I_2(u)$, but affects $I_1(u)$ and $I_3(u)$. Line 7 affects only $I_2(u)$.

By part (2) of Lemma 4.6.3, $\mathit{recent}(\{u\}, u.\mathit{log})$ is not affected in Line 2. Therefore, $I_1(u)$ is preserved in Line 2, and hence is preserved in the execution of $T_5$.

Let $m$ be the *update* message received and $A$ be the set of nodes in $\mathit{subtree}(w, u)$. By part (1) of Lemma 4.6.3, after the execution of Line 2, $u.\mathit{aval}[w] = m.x$ and $\mathit{recent}(A, u.\mathit{log}) = \mathit{recent}(A, m.\mathit{wlog})$. Hence, by $I_2(u)$, $u.\mathit{aval}[w] = f(\mathit{recent}(A, u.\mathit{log}))$.

By part (2) of Lemma 4.6.3, for any node $v$ in the set $u.\mathit{nbrs}() \setminus \{w\}$, $\mathit{recent}(B, u.\mathit{log})$ is not affected, where $B = \mathit{subtree}(v, u)$, and so the value of $\mathit{corresponds}(B, u.\mathit{aval}[v], u.\mathit{log})$ remains unchanged. Hence, along with the arguments in the preceding paragraph, $I_3(u)$ is preserved in Line 2, and hence is preserved in the execution of $T_5$.

Line 7 affects only $I_2(u)$. By Lemma 4.6.4, $I_2(u)$ holds in Line 7.

Therefore, $I_1(u) \wedge I_2(u) \wedge I_3(u)$ is preserved in the execution of $T_5$.

$\{I(u)\}T_6\{I(u)\}$. In the execution of $T_6$, $I_1(u)$, $I_2(u)$, and $I_3(u)$ are not affected. Hence, $I(u)$ is preserved in the execution of $T_6$. $\square$

For a request sequence $\tau$ and a request $q$, $\mathit{index}(\tau, q)$ returns the index of $q$ in $\tau$ if present, and returns $-1$ otherwise. For any request sequence $\tau$, and requests $q_1$ and $q_2$ in $\tau$, the predicate $\mathit{precedes}(\tau, q_1, q_2)$ holds if $\mathit{index}(\tau, q_1) < \mathit{index}(\tau, q_2)$.

**Lemma 4.6.6.** *Let $q_1$ and $q_2$ be any gather or write requests such that $q_1.node = q_2.node$ and $q_1.index < q_2.index$. Then, $q_1$ and $q_2$ belong to $q_1.node.gwlog$, and $precedes(q_1.node.gwlog, q_1, q_2)$.*

*Proof.* From the given condition, requests $q_1$ and $q_2$ belong to $q_1.node.log$ and $precedes(q_1.node.log, q_1, q_2)$. By the construction of *gwlog*, the lemma follows. □

**Lemma 4.6.7.** *Let $u$ and $v$ be distinct nodes and let $q_1$ and $q_2$ be write requests in $v.gwlog$ such that $q_2.node = v$, $precedes(v.gwlog, q_1, q_2)$, and $q_2$ belongs to $u.gwlog$. Then, $q_1$ belongs to $u.gwlog$ and $precedes(u.gwlog, q_1, q_2)$.*

*Proof.* By induction on the length $l$ of the path from $v$ to $u$.

Base case. $l = 1$, that is, $u$ and $v$ are neighboring nodes. Let $u$ receive $q_2$ in an *update* or a *response* message $m$, that is, $q_2$ belongs to $m.wlog$ and $q_2$ does not belong to $u.log$ just before $u$ receives $m$. By inspection of the code, $m.wlog = v.wlog$. Hence, just before $m$ is sent, $q_2$ belongs to $v.log$. Since $precedes(v.log, q_1, q_2)$, $precedes(m.wlog, q_1, q_2)$. If $q_1$ is in $u.log$ just before receiving $m$, then on receiving $m$, $q_2$ belongs to $u.log$, so $precedes(u.gwlog, q_1, q_2)$. Otherwise, on receiving $m$, $u.log = u.log.(u.log - m.wlog_w)$ and $precedes(u.log, q_1, q_2)$. Hence, by the construction of $u.gwlog$, $precedes(u.gwlog, q_1, q_2)$.

Induction hypothesis. Assume that the lemma holds for $l - 1$, where $l \geq 2$.

Induction step. Let $w$ be the node such that $w$ belongs to $u.nbrs()$ and $v$ belongs to $subtree(T, w, u)$. Let $u$ receive $q_2$ from $w$ in an *update* or a *response* message $m$. By inspection of the code, $q_2$ belongs to $w.log$, and so, by construction of $w.gwlog$, $q_2$ also belongs to $w.gwlog$. By the induction hypothesis and by the construction of $w.gwlog$, $q_1$ belongs to $w.log$ and $precedes(w.log, q_1, q_2)$ when $m$ is sent. Since $m.wlog = w.wlog$ when $m$ is sent, $q_1$ belongs to $m.wlog$ and $precedes(m.log, q_1, q_2)$. As in the base case, regardless of whether $q_1$ belongs to $u.log$ just before receiving $m$, $q_1$ belongs to $u.log$ and $precedes(u.log, q_1, q_2)$ on receiving $m$. Hence, by construction of $u.gwlog$, $precedes(u.gwlog, q_1, q_2)$. □

**Lemma 4.6.8.** *Let $q_1$ and $q_2$ be gather requests such that $q_1.node \neq q_2.node$, and $q_1 \overset{i}{\rightsquigarrow} q_2$ for some integer $i$, where $i > 1$. Then, there is a write request $q'$ such that $q'.node = q_1.node$ and $q_1 \overset{j}{\rightsquigarrow} q' \overset{i-j}{\rightsquigarrow} q_2$ for some integer $j$, where $1 \leq j < i$.*

*Proof.* By contradiction. Assume that there is no such *write* request at $q_1.node$. Let $q_1 \overset{j'}{\rightsquigarrow} q' \overset{1}{\rightsquigarrow} q'' \overset{i-j'-1}{\rightsquigarrow} q_2$ such that $q''$ is the first request in this chain that is not at $q_1.node$. That is, in this chain, $q_1, \ldots, q'$ are at $q.node$. We can find such a request ($q''$) since $q_2.node \neq q_1.node$. By the definition of causal ordering, $q' \overset{1}{\rightsquigarrow} q''$ if $q'$ is a *write* request and $q''$ is a *gather* request, which contradicts the assumption. Hence, the contradiction. Therefore, the lemma follows. □

**Lemma 4.6.9.** *For any node $u$, let $q_i$ be a request such that $(q_i.op = write) \vee (q_i.op = gather \wedge q_i.node = u)$, for $i$ in $\{1, 2\}$. Further assume that $q_1 \rightsquigarrow q_2$ and*

*$q_2$ belongs to $u.gwlog$. Then, $q_1$ belongs to $u.gwlog$ and $precedes(u.gwlog, q_1, q_2)$.*

*Proof.* We prove the lemma by induction on the positive integer $i$ such that $q_1 \overset{i}{\leadsto} q_2$.

Base case: $q_1 \overset{1}{\leadsto} q_2$. There are two cases, depending on $q_1 \overset{1}{\leadsto} q_2$ due to rule (1) or rule (2).

If $q_1 \overset{1}{\leadsto} q_2$ by rule (1), then $q_1.node = q_2.node$ and $q_1.index < q_2.index$. There are two subcases: ($a$) $u = q_1.node$; ($b$) $u \neq q_1.node$. In subcase ($a$), by Lemma 4.6.6, $q_1$ and $q_2$ belong to $u.gwlog$, and $precedes(u.gwlog, q_1, q_2)$. In subcase ($b$), let $v$ be $q_1.node$. By Lemma 4.6.6, $precedes(v.gwlog, q_1, q_2)$. Since $u \neq v$, $q_1$ and $q_2$ are *write* requests. Since $q_2$ belongs to $u.gwlog$, by Lemma 4.6.7, $q_1$ is in $u.gwlog$ and $precedes(u.gwlog, q_1, q_2)$ holds.

If $q_1 \overset{1}{\leadsto} q_2$ by rule (2), then $q_1$ is a *write* request and $q_2$ is a *gather* request such that $q_2$ returns $(q_1.node, q_1.index)$ in $q_2.retval$. Since $q_2$ returns $(q_1.node, q_1.index)$, $q_1$ is in $u.log$ and $precedes(u.log, q_1, q_2)$. By the construction of $u.gwlog$, $q_1$ is in $u.gwlog$ and $precedes(u.gwlog, q_1, q_2)$.

Induction hypothesis: Suppose that $q_1 \overset{i}{\leadsto} q_2$, for some positive integer $i$. Then, request $q_1$ belongs to $u.gwlog$ and $precedes(u.gwlog, q_1, q_2)$.

Induction step: Suppose that $q_1 \overset{i}{\leadsto} q' \overset{1}{\leadsto} q_2$, where $i > 0$. We consider the two cases separately.

First we consider the case where either $(q'.op = write)$ or $q'.op = gather \wedge q'.node = u$. By the induction hypothesis, $q'$ belongs to $u.gwlog$

and $\mathit{precedes}(u.\mathit{gwlog}, q', q_2)$. Also by the induction hypothesis, $q_1$ belongs to $u.\mathit{gwlog}$ and $\mathit{precedes}(u.\mathit{gwlog}, q_1, q')$. Hence, $q_1$ belongs to $u.\mathit{gwlog}$, and $\mathit{precedes}(u.\mathit{gwlog}, q_1, q_2)$.

Second we consider the case where $q'.\mathit{op} = \mathit{gather}$ and $q'.\mathit{node} \neq u$. Let $q'.\mathit{node}$ be $v$. Since $q'.\mathit{op} = \mathit{gather}$, $q' \overset{1}{\rightsquigarrow} q_2$ holds by rule (1), that is, $q_2.\mathit{node} = v$ and $q'.\mathit{index} < q_2.\mathit{index}$. Since $v \neq u$, $q_2$ is a *write* request. By Lemma 4.6.6, $\mathit{precedes}(v.\mathit{gwlog}, q', q_2)$. Now consider the following two possible subcases for $q_1$: (a) $q_1.\mathit{op} = \mathit{write}$; (b) $q_1.\mathit{op} = \mathit{gather} \wedge q_1.\mathit{node} = u$. On subcase (a), by the induction hypothesis, $q_1$ belongs to $v.\mathit{gwlog}$ and $\mathit{precedes}(v.\mathit{gwlog}, q_1, q')$. From the argument given above, $q_1$ and $q_2$ belong to $v.\mathit{gwlog}$ and $\mathit{precedes}(v.\mathit{gwlog}, q_1, q_2)$. By Lemma 4.6.7, $q_1$ belongs to $u.\mathit{gwlog}$ and $\mathit{precedes}(u.\mathit{gwlog}, q_1, q_2)$.

Now consider subcase (b). Since $q_1.\mathit{node} \neq q'.\mathit{node}$, $q_1 \overset{i}{\rightsquigarrow} q'$, where $i > 1$, and $q_1$ and $q'$ are *gather* requests. By Lemma 4.6.8, there is a *write* request $q''$ such that $q''.\mathit{node} = u$ and $q_1 \overset{j}{\rightsquigarrow} q'' \overset{i-j}{\rightsquigarrow} q'$, for some $j$ such that $1 \leq j < i$. By the induction hypothesis, $q''$ belongs to $v.\mathit{gwlog}$ and $\mathit{precedes}(v.\mathit{gwlog}, q'', q')$. Hence, from the arguments given above, $\mathit{precedes}(v.\mathit{gwlog}, q'', q_2)$. Since $q''$ and $q_2$ are *write* requests, $q_2.\mathit{node} = v$, $q_2$ belongs to $u.\mathit{gwlog}$, and predicate $\mathit{precedes}(v.\mathit{gwlog}, q'', q_2)$ holds, by Lemma 4.6.7, $\mathit{precedes}(u.\mathit{gwlog}, q'', q_2)$. From the arguments given above, $q''$ belongs to $u.\mathit{gwlog}$ and $q_1 \overset{j}{\rightsquigarrow} q''$ for some $j \geq 1$. Hence, by the induction hypothesis, $\mathit{precedes}(u.\mathit{gwlog}, q_1, q'')$. From the arguments given above, it follows that, $q_1$ belongs to $u.\mathit{gwlog}$ and $\mathit{precedes}(u.\mathit{gwlog}, q_1, q_2)$. □

**Lemma 4.6.10.** *For any node $u$, $u.gwlog'$ respects the causal ordering among the requests in $u.gwlog'$.*

*Proof.* We prove this lemma by induction on the number of iterations in the construction of $u.gwlog'$. For the base case, by Lemma 4.6.9, $u.gwlog$ respects the causal ordering among the requests in $u.gwlog$. In each iteration of the construction, the additional requests are added at the end of $u.gwlog'$. By Lemma 4.6.9, this step preserves the causal ordering among the requests in $u.gwlog'$. □

**Lemma 4.6.11.** *For any node $u$, $u.log'$ and $u.gwlog'$ are compatible.*

*Proof.* We prove this lemma by induction on the number of iterations in the construction of $u.log'$ and $u.gwlog'$. For the base case, we need to show that $u.log$ and $u.gwlog$ are compatible. Consider any *combine* request $q$ at node $u$. By Lemma 4.6.5, $I(u)$ is an invariant, so $corresponds(nodes(T), q.retval, u.log)$, where $q.retval = u.gval()$. Hence, by the construction of $u.gwlog$, for any *combine* request in $u.log$, there is a compatible *gather* request in $u.gwlog$, and $u.log$ and $u.gwlog$ are compatible.

In each iteration of the construction of $u.log'$ and $node.gwlog'$, by the base case and the induction hypothesis, additional requests appended to both of the request sequences are mutually compatible. Hence, $u.log'$ and $u.gwlog'$ are compatible. □

**Theorem 11.** *Let set $A$ be the execution-history of any lease-based algorithm $A$. Then, $A$ is causally consistent.*

*Proof.* Consider any node $u$ in $T$. By construction, $u.gwlog'$ is a serialization of all requests in $u.gwlog'$. From this observation and Lemma 4.6.10, $u.gwlog'$ is causally consistent. By construction, $u.log'$ contains all requests in $pruned(A, u)$. By Lemma 4.6.11, $u.log'$ and $u.gwlog'$ are compatible. Hence, $A$ is causally consistent. □

## 4.7 Experimental Results

In this section we present experimental results comparing the performance of algorithm RWW to that of certain static strategies under a wide range of operating conditions. Our performance metric is the cost of communication, which is measured in terms of the average number of messages incurred in executing a given request sequence. We compare the performance of RWW to that of the update-all and update-none static lease-based algorithms. In the update-all algorithm, leases on all edges are set, and the leases are kept that way throughout the processing of a given request sequence. Therefore, during the execution of a write request, update-all algorithm results in sending an update message along each edge. On the other hand, in the update-none algorithm, no lease is set on any edge, and on a write request at a node, the local value of the node is updated.

We simulate 127 nodes arranged in a complete binary tree. We use summation as our aggregation operator. Our evaluation shows that algorithm RWW adapts well with varying workload.

### 4.7.1 Sequential Execution

In Figure 4.10, we plot the performances of RWW, update-all and update-none for sequential executions of different request sequences with varying combine-write ratio. Each request sequence consists of 10000 requests and is generated as follows. For a combine-write ratio $r$, we generate a real random number uniformly distributed in the range $[0, 1]$, and if the number is less than $r$, then we generate a *combine* request at a node uniformly and randomly selected from the set of all nodes. We repeat the above process 10000 times to generate a request sequence with the desired combine-write ratio.

In Figure 4.10, we vary the combine-write ratio from 0 to 1 with a step size of 0.1. For each combine-write ratio, the performance of each of the algorithm is base on the runs over 10 different request sequences with the same combine-write ratio. We observe that RWW algorithm adapts well to different combine-write ratios.

### 4.7.2 Concurrent Execution

In this section we compare the performance of RWW, update-all, and update-none for concurrent execution of request sequences. For our experiments, we construct trees and request sequences as follows.

For each of the experiments in Figures 4.11 to 4.17, we generate a

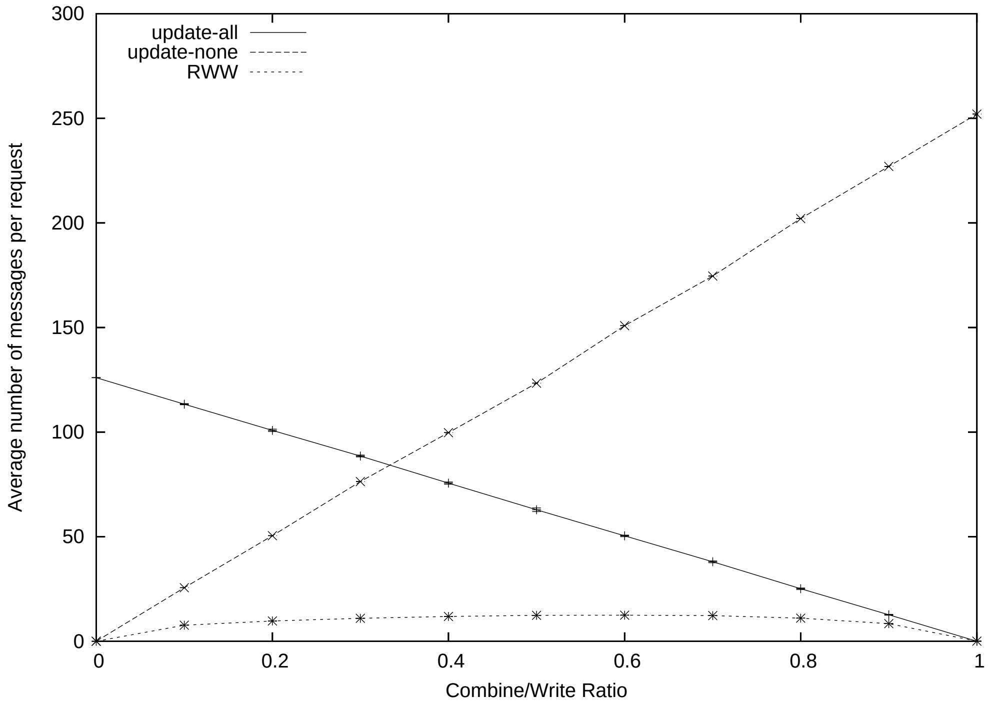

Figure 4.10: Sequential execution.

complete binary tree consisting of 127 nodes. For each edge, we generate a Poisson-distributed delay along the edge. In Figures 4.11 to 4.17, we vary the Poisson distribution parameter $\lambda$ as indicated.

Each request sequence consists of 10000 requests and is generated as follows. For a combine-write ratio $r$, we generate a uniformly distributed value in the range $[0, 1]$, and if this value is less than $r$, then we generate a *combine* request at a node uniformly and randomly selected from the set of all nodes. We repeat the above process 10000 times to generate a request sequence with the desired combine-write ratio. To determine the interval between any two requests, we generate a random integer uniformly distributed in the range of $[0, 64]$.

Our experiments (Figure 4.11 to Figure 4.17) show that algorithm RWW performs well in executing request sequences with different combine-write ratios.

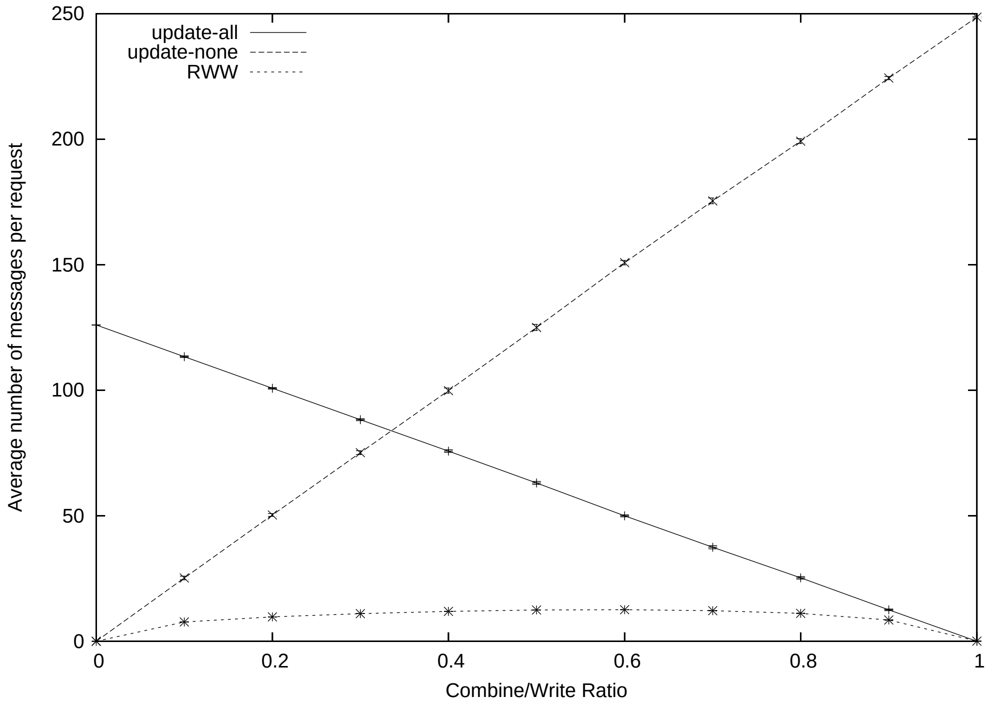

Figure 4.11: Concurrent execution with $\lambda = 0$.

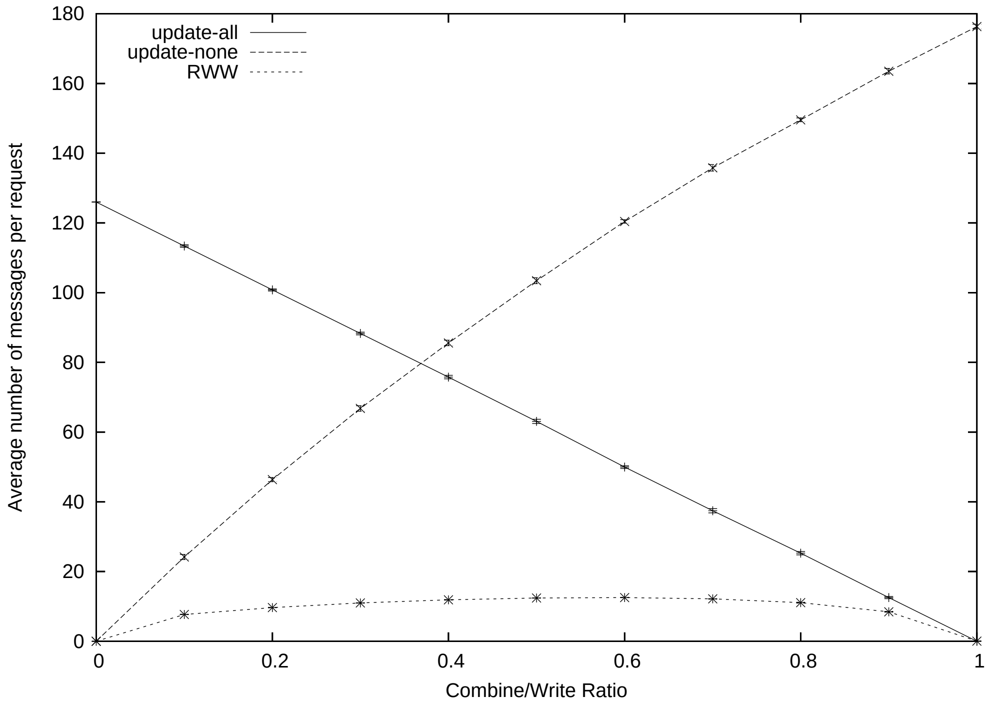

Figure 4.12: Concurrent execution with $\lambda = 1$.

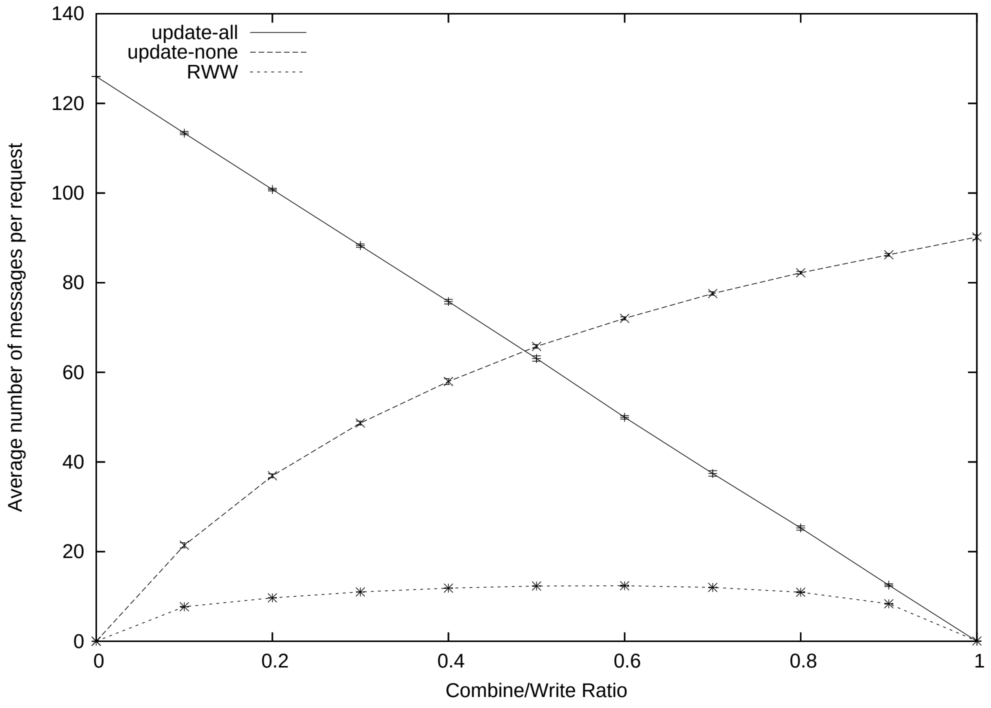

Figure 4.13: Concurrent execution with $\lambda = 5$.

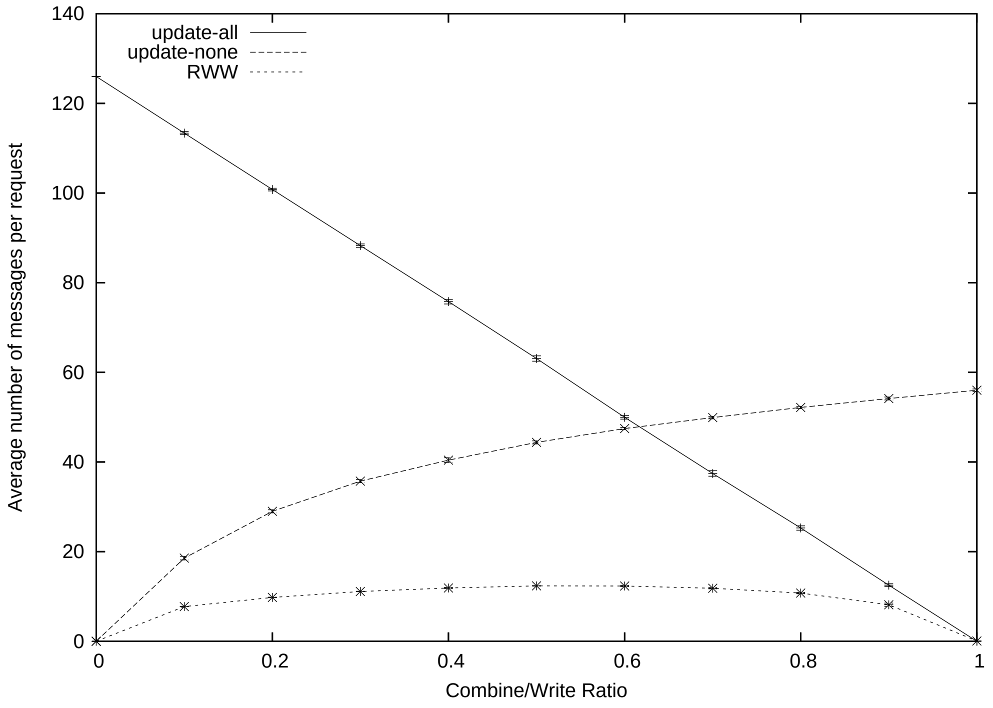

Figure 4.14: Concurrent execution with $\lambda = 10$.

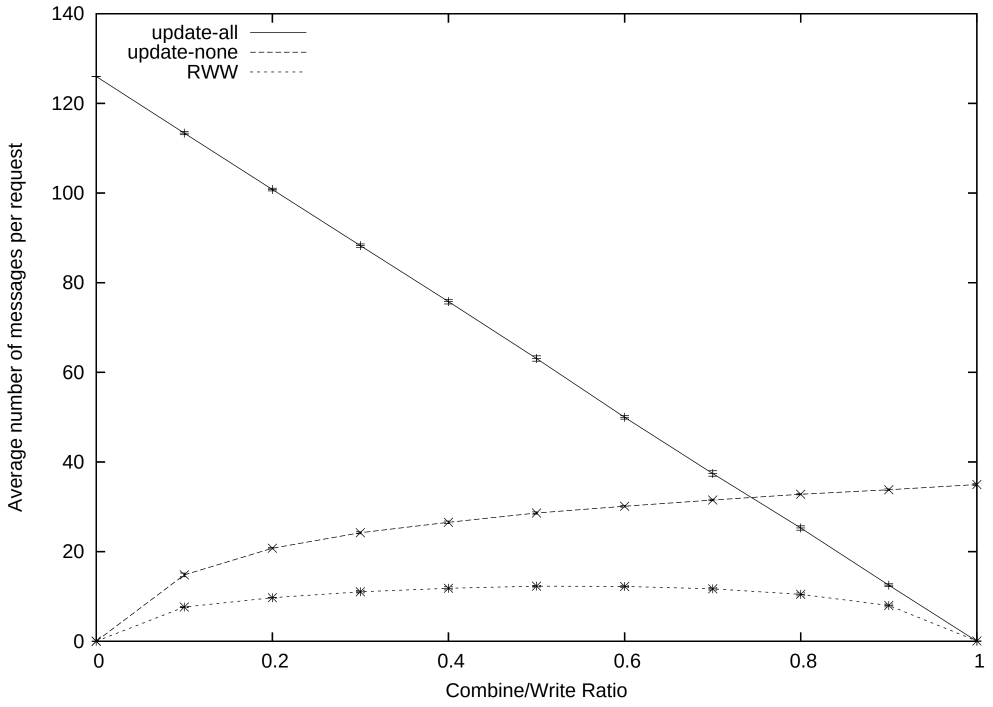

Figure 4.15: Concurrent execution with $\lambda = 20$.

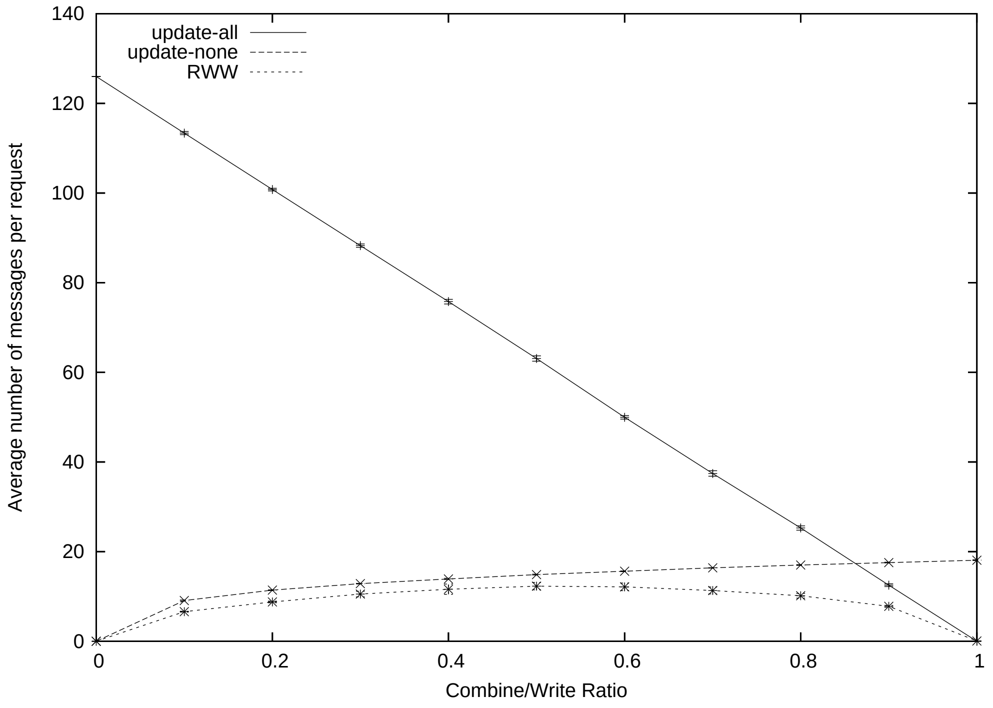

Figure 4.16: Concurrent execution with $\lambda = 50$.

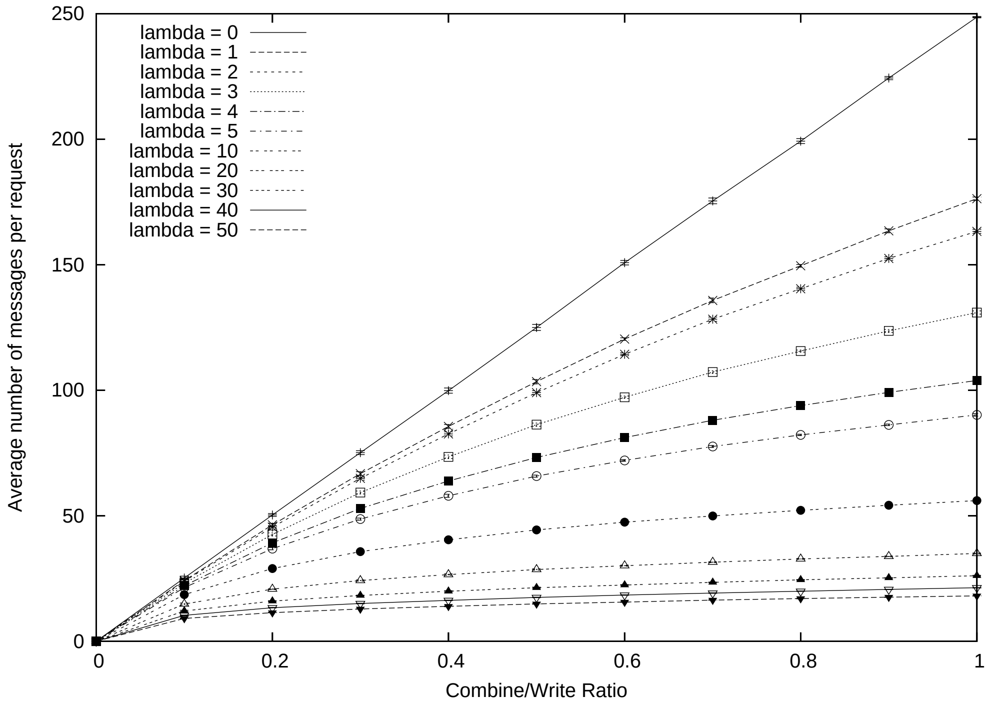

Figure 4.17: Concurrent execution with no leases.

# Chapter 5

# Conclusion

We conclude this dissertation with some open problems for future research.

In the realm of cooperative caching, an open problem for further research is to tighten the bounds for an online Hierarchical Cooperative Caching algorithm with cache capacity blowup $O(d^{1-\epsilon})$, where $d$ is the depth of the given hierarchy.

In the realm of compression caching, there are multiple directions for future research. First, it would be interesting to generalize the upper bound algorithms presented in Chapter 3 to achieve $O(m)$-competitiveness with $O(m)$ factor capacity blowup for a richer set of the problems in the class of compression caching. Second, an important direction is to consider the design of the distributed storage system as alluded in Section 3.3, and collectively address the existence of different file formats and machines distributed over the network.

In the realm of aggregation, an interesting open problem is as follows. In many emerging networks like peer-to-peer and sensor networks, nodes in the network participate in routing, forwarding, querying, and aggregating data.

In Chapter 4, we study the aggregation problem and designed an aggregation protocol that adapts aggregation aggressiveness in a given tree network. There is a tradeoff between performance and accuracy in the aggregation problem [17, 34, 35], where accuracy can be measured in various ways, for example, in terms of consistency, staleness, and numerical error. In Chapter 4, our notion of accuracy is defined in terms of consistency. In some applications like network monitoring and sensor networks, one can tolerate numerical error and staleness in order to minimize the communication overhead.

One formulation of the hierarchical approximate aggregation problem is as follows. Assume that we are given a set of nodes arranged in a hierarchy, and that each node periodically updates its local value. Also assume that we are given an aggregation function (e.g., sum or average) that is hierarchically computable. To simplify the present discussion, let us assume that the execution is sequential, in the sense that an update is initiated only in a quiescent state of the network. Given an error budget $\Delta$, we wish to ensure that after each successive update is processed, the root node knows an interval of width $\Delta$ that contains the actual aggregate value. The goal is to minimize the total number of messages used in order to maintain this invariant.

One open problem is whether one could design and analyze an efficient distributed protocol for online hierarchical approximate aggregation.

One natural approach to the above problem is as follows. Each node maintains an interval containing the aggregate value for the subtree rooted at that node. The width of the interval at the root is at most $\Delta$, and the sum of the interval widths of the children of any node $u$ is at most the width of the interval of $u$. In effect, at any time, each node has an error window — determined by the width of its interval — and it distributes all or part of this window to its children. Intuitively, the distribution of error should favor children with more update-related activity; for example, if the hierarchy rooted at some child $v$ of $u$ does not produce any updates over a long period of time, then $u$ is likely to assign a small error window to $v$.

In Chapter 4, we allow concurrent requests and generalize the traditional causal consistency definitions for the aggregation problem. Similarly, for the hierarchical approximate aggregation problem one could allow concurrent updates and define appropriate semantics for the protocol. In terms of performance, one could analyze the protocol in the framework of competitive analysis. As is typical in the competitive analysis of distributed algorithms [6, 7], one could focus on performance bounds for sequential executions.

# Bibliography

[1] Akamai Technologies, Inc. `http://www.akamai.com`, 2007.

[2] B. Abali, M. Banikazemi, X. Shen, H. Franke, D. E. Poff, and T. B. Smith. Hardware compressed main memory: Operating system support and performance evaluation. *IEEE Transactions on Computers*, 50:1219–1233, 2001.

[3] M. Ahamad, G. Neiger, J. E. Burns, P. Kohli, and P. W. Hutto. Causal memory: Definitions, implementation, and programming. *Distributed Computing*, 9:37–49, 1995.

[4] A. R. Alameldeen and D. A. Wood. Adaptive cache compression for high-performance processors. In *Proceedings of the 31st Annual International Symposium on Computer Architecture*, pages 212–223, June 2004.

[5] T. E. Anderson, M. D. Dahlin, J. N. Neefe, D. A. Patterson, D. S. Rosselli, and R. Y. Wang. Serverless network file systems. In *Proceedings of the 15th Symposium on Operating Systems Principles*, pages 109–126, December 1995.

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

[7] B. Awerbuch, Y. Bartal, and A. Fiat. Competitive distributed file allocation. *Information and Computation*, 185:1–40, 2003.

[8] Y. Bartal. Probabilistic approximation of metric spaces and its algorithmic applications. In *Proceedings of the 37th Annual IEEE Symposium on Foundations of Computer Science*, pages 184–193, October 1996.

[9] Y. Bartal. Distributed paging. In A. Fiat and G. J. Woeginger, editors, *The 1996 Dagstuhl Workshop on Online Algorithms*, volume 1442 of *Lecture Notes in Computer Science*, pages 97–117. Springer, 1998.

[10] Y. Bartal. On approximating arbitrary metrics by tree metrics. In *Proceedings of the 30th Annual ACM Symposium on Theory of Computing*, pages 161–168, May 1998.

[11] M. Bawa, A. Gionis, H. Garcia-Molina, and R. Motwani. The price of validity in dynamic networks. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, pages 515–526, June 2004.

[12] N. Belaramani, M. Dahlin, L. Gao, A. Nayate, A. Venkataramani, P. Yalagandula, and J. Zheng. PRACTI replication. In *USENIX Symposium on Networked Systems Design and Implementation*, May 2006.

[13] A. Borodin and R. El-Yaniv. *Online computation and competitive analysis.* Cambridge University Press, Cambridge, 1998.

[14] C. M. Bowman, P. B. Danzig, D. R. Hardy, U. Manber, and M. F. Schwartz. The Harvest information discovery and access system. *Computer Networks and ISDN Systems*, 28:119–125, 1995.

[15] P. Cao and S. Irani. Cost-aware WWW proxy caching algorithms. In *Proceedings of the 1st Usenix Symposium on Internet Technologies and Systems*, pages 193–206, December 1997.

[16] A. Chankhunthod, P. B. Danzig, C. Neerdaels, M. F. Schwartz, and K. J. Worrell. A hierarchical internet object cache. In *USENIX Annual Technical Conference*, pages 153–164, 1996.

[17] G. Cormode and M. N. Garofalakis. Efficient strategies for continuous distributed tracking tasks. *IEEE Data Engineering Bulletin*, 28:33–39, 2005.

[18] K. Czajkowski, C. Kesselman, S. Fitzgerald, and I. T. Foster. Grid information services for distributed resource sharing. In *Proceedings of the 10th IEEE International Symposium on High Performance Distributed Computing*, pages 181–194, August 2001.

[19] M. D. Dahlin, R. Y. Wang, T. E. Anderson, and D. A. Patterson. Cooperative caching: Using remote client memory to improve file system performance. In *Proceedings of the First Symposium on Operating Systems Design and Implementation*, pages 267–280, November 1994.

[20] V. Duvvuri, P. Shenoy, and R. Tewari. Adaptive Leases: A strong consistency mechanism for the world wide web. *IEEE Transactions on Knowledge and Data Engineering*, 15:1266–1276, 2003.

[21] J. Fakcharoenphol, S. Rao, and K. Talwar. A tight bound on approximating arbitrary metrics by tree metrics. In *Proceedings of the 35th Annual ACM Symposium on Theory of Computing*, pages 448–455, June 2003.

[22] M. J. Fischer, N. A. Lynch, and M. S. Paterson. Impossibility of distributed consensus with one faulty process. *Journal of the ACM*, 32:374–382, 1985.

[23] M. Freedman, E. Freudenthal, and D. Mazieres. Democratizing content publication with coral. In *Proceedings of the 1st Annual ACM Symposium on Networked Systems Design and Implementation*, pages 239–252, March 2004.

[24] Y. Fu, J. Chase, B. Chun, S. Schwab, and A. Vahdat. SHARP: An architecture for secure resource peering. In *Proceedings of the 19th ACM Symposium on Operating Systems Principles*, pages 133–148, October 2003.

[25] Ganglia: Distributed monitoring and execution system. `http://ganglia.sourceforge.net`, 2007.

[26] M. G. Gouda. *Elements of Network Protocol Design.* John Wiley & Sons, New York, 1998.

[27] C. Gray and D. Cheriton. Leases: An efficient fault-tolerant mechanism for distributed file cache consistency. In *Proceedings of the 12th ACM Symposium on Operating Systems Principles*, pages 202–210, December 1989.

[28] E. G. Hallnor and S. K. Reinhardt. A unified compressed memory hierarchy. In *Proceedings of the 11th International Symposium on High-Performance Computer Architecture*, pages 201–212, February 2005.

[29] N. Jain, P. Yalagandula, M. Dahlin, and Y. Zhang. INSIGHT: A distributed monitoring system for tracking continuous queries. In *Work-in-Progress Session at SOSP 2005*, pages 23–26, October 2005.

[30] M. R. Korupolu, C. G. Plaxton, and R. Rajaraman. Placement algorithms for hierarchical cooperative caching. *Journal of Algorithms*, 38:260–302, 2001.

[31] X. Li, C. G. Plaxton, M. Tiwari, and A. Venkataramani. Online hierarchical cooperative caching. *Theory of Computing System*, 39:851–874, 2006.

[32] S. Madden, M. J. Franklin, J. M. Hellerstein, and W. Hong. TAG: A tiny aggregation service for ad-hoc sensor networks. In *Proceedings of the 5th Symposium on Operating Systems Design and Implementation*, December 2002.

[33] J. Misra. Axioms for memory access in asynchronous hardware systems. *ACM Transactions on Programming Languages and Systems*, 8:142–153, 1986.

[34] C. Olston and J. Widom. Offering a precision-performance tradeoff for aggregation queries over replicated data. In *Proceedings of the 26th International Conference on Very Large Data Bases*, pages 144–155, September 2000.

[35] C. Olston and J. Widom. Efficient monitoring and querying of distributed, dynamic data via approximate replication. *IEEE Data Engineering Bulletin*, 28:11–18, 2005.

[36] C. G. Plaxton, M. Tiwari, and P. Yalagandula. Online aggregation over trees. In *Proceedings of the 21st IEEE International Parallel and Distributed Processing Symposium*, March 2007.

[37] R. Renesse, K. P. Birman, and W. Vogels. Astrolabe: A robust and scalable technology for distributed system monitoring, management, and data mining. *ACM Transactions on Computer Systems*, 21:164–206, 2003.

[38] M. Roussopoulos and M. Baker. CUP: Controlled update propagation in peer-to-peer networks. In *USENIX Annual Technical Conference*, pages 167–180, June 2003.

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

[40] A. S. Tanenbaum. *Distributed Operating Systems*. Prentice-Hall, 1995.

[41] M. Wawrzoniak, L. Peterson, and T. Roscoe. Sophia: An information plane for networked systems. In *Proceedings of the 2nd Workshop on Hot Topics in Networks*, November 2003.

[42] D. Wessels. Squid Internet object cache. Available at URL http://squid.nlanr.net/squid, January 1998.

[43] D. Wessels and K. Claffy. RFC 2187: Application of Internet Cache Protocol, 1997.

[44] R. Wolski, N. Spring, and J. Hayes. The network weather service: A distributed resource performance forecasting service for metacomputing. *Journal of Future Generation Computing Systems*, 15:757–768, 1999.

[45] P. Yalagandula and M. Dahlin. A scalable distributed information management system. In *Proceedings of the ACM SIGCOMM Conference*, pages 379–390, August 2004.

[46] N. E. Young. On-line file caching. *Algorithmica*, 33:371–383, 2002.

[47] L. Zhang, S. Floyd, and V. Jacobson. Adaptive Web Caching. In *Proceedings of the 1997 NLANR Web Cache Workshop*, 1997.

# Vita

Mitul Tiwari was born in Raipur, India to Saroj Tiwari and Dinesh Chandra Tiwari. He completed his schooling from Senior Secondary School Sector X, Bhilai, India, in April 1997. He received the Bachelor of Technology degree in Computer Science and Engineering from the Indian Institute of Technology at Bombay in May 2001. Thereafter, he joined the graduate program in Computer Science at the University of Texas at Austin, and received his Master of Sciences degree in December 2003. He was employed as an intern at Microsoft, Washington, and Google, California, during the summers of 2003 and 2004 respectively. He was awarded the MCD graduate fellowship, in August 2001, by the University of Texas at Austin. Earlier, during his high school studies in 1997, he was awarded a gold medal in Physics by the Indian Association of Physics Teachers.

Permanent address: 15 Samata Colony, Raipur
Chattisgarh, India

This dissertation was typeset with LaTeX[†] by the author.

[†]LaTeX is a document preparation system developed by Leslie Lamport as a special version of Donald Knuth's TeX Program.