H-0213 December 7, 2003
Other

IBM Research Report

Hierarchical Storage-Reliability and Object-Based Storage

Ami Tavory, Vladimir Dreizin, Shmuel Gal
IBM Research Division
Haifa Research Laboratory
Haifa 31905, Israel

Meir Feder
Department of EE-Systems
Tel-Aviv University
Israel

Research Division
Almaden - Austin - Beijing - Haifa - India - T. J. Watson - Tokyo - Zurich

Hierarchical Storage-Reliability and Object-Based Storage

Ami Tavory†, Vladimir Dreizin†, Shmuel Gal‡, and Meir Feder§
IBM Haifa Research Laboratories
Email:{tavory,dreizin,gals}@il.ibm.com, meir@eng.tau.ac.il

Abstract— **Networked storage-systems typically use codes to protect data from failures. In this work we deal with aspects of *hierarchical reliability* and *object-based* storage schemes. A hierarchical-reliability scheme maintains devices employing different coding schemes, and transits data between these groups based on activity level. Object-based storage utilizes basic storage-devices storing variable-sized objects, as opposed to blocks.**

The use of hierarchical object-based storage, requires a policy to transition data between different levels. We discuss such a policy for a two-level system, and show how to extend it for a multiple-level system.

We then show an information-theoretic interpretation for the need for hierarchical reliability, and show implications for the necessity of hierarchical reliability schemes as the growth-gap between storage-devices' capacity and IO-bandwidth continues.

I. Introduction

Storage systems typically use some form of an *ECC* (error-correcting code), *e.g.*, RAID [1], in order to protect data blocks from device failures. Most storage systems employ the same coding technique to protect all data blocks. This might be improved upon in some ways.

It was first observed in [2] that highly-active data should be protected using high-performance codes, while relatively inactive data should be protected using cost-efficient codes, yielding a hierarchical reliability scheme. In a hierarchical reliability storage-system, various levels are maintained. The levels protect data according to importance, but using schemes of different IO-performance. Based on the activity levels of data, they should be constantly transited to the appropriate level. Naturally, a cybernetic storage system should have some automated policy to perform this, similar to a paging policy in a cache system.

There are some differences between this problem and that of a cache paging problem. In most paging problems, different components are maintained, some of which are effectively faster than others. This might be due to their composition from differen types of hardware, (*e.g.*, RAM versus disk drives), or due to their differing locations in respect to a client (*e.g.* network caching). The interesting point from [2] is that a paging-like setting occurs because of a *choice* to handle data, stored in similar components, in different ways. We show why this choice is more like a necessity, in Section **??**.

† Also at the Dept. of EE-Systems, Tel-Aviv University.
‡ Also at the Dept. of Statistics, University of Haifa.
§ At the Dept. of EE-Systems, Tel-Aviv University.

It should also be noted that the performance metrics of a transition policy differ substantially than those in a classic paging problem. We define this in detail in Section II.

The use of *storage objects*, as opposed to blocks, can also assist in improving storage reliability. A storage object is a logical entity composed of data considered by its creator to be related. The use of objects allows to exploit access correlation and user-defined attributes [3]. In the context of reliability, it is natural to protect data in object level, since this complies with the expected usage correlation.

The granularity of objects allows also the assignment of a *UEP* (unequal error-protection) attribute to data, determining the extent to which it should protected. In practice, data are not all important to the same extent [4]. Less important data can be protected using lower-redundancy schemes than used for more important data. In the settings of storage reliability, it might seem that the benefits of UEP are primarily in conserving storage space. We show in Section II that the benefits are primarily in improving the competitiveness of the transition policy.

Related Work. Reliability in storage systems was originally studied in the context of small-capacity systems [5], [6], [1], and in conjunction with performance improvement via parallelism, *e.g.*, RAID. The schemes were later extended in some directions. Concatenated codes were studied, *e.g.*, two-dimensional codes [7], and new RAID levels [8]. Questions on coding-group placement within devices were studied, *e.g.*, various distributed striping and sparing techniques [9], [10], [11], [12], [13], [14]). Effects of physical device-topologies were studied [15]. Storage reliability via coding was extend in the direction of disaster recovery as well [16].

The important idea of hierarchical protection of data based on data activity was shown in work on HP-AutoRAID [17], a work to which ours is an extension. Differential coding based on data activity was studied in the context of very large, concrete systems [18], [19], [20].

The important concept of online competitiveness for the analysis of cache paging-algorithms, has been found quite some time ago [21], [22]. There has been much research in evolving this concept, *e.g.*, limiting the power of offline adversaries [23], [**?**], extending the setting of the problem to varying access sizes and costs [24], [25], [26], and using more realistic models of hardware [**?**], [**?**].

Our Contribution. In this work we show an algo-

rithm for transiting data between levels of a hierarchical-reliability storage-system, a problem which we show differs from that of classic cache-paging. This algorithm takes into account the differing sizes and costs associated with variable-sized data which need be protected according to different protection-requirements. We also show the non-competitiveness of any reliability scheme which ignores different protection-requirements. We show how to extend a two-level transition policy into a multi-level transition policy, using a *binary decomposability* analysis, which is a simple observation generalizing previous results obtained through complicated analysis.

We also give an information-theoretic interpretation for the requirement of hierarchical protection. We show lower bounds on the number of devices required, the average load of reliability-related operations per device, the average coding-group size, and the average update penalty. These bounds hold regardless of specific codes, data-transition policies, and device-replacement policies. Although these bounds are weak in the sense that they only prove the eventual necessity of multi-levelled reliability schemes as the growth-gap between device capacity and IO-bandwidth continues, we hope that they will lead into insight allowing to determine the number of levels required for practical application.

Paper Organization. The remainder of this paper is organized as follows. In Section II we show a level-transition algorithm. In Section III we show a justification for the necessity of hierarchical reliability.

II. Data Transition between Levels

A hierarchical reliability scheme composed of increasing-sized levels requires a data transition policy. Ideally, the high performance (and smaller) levels should contain the data required by the users. The challenges here are similar to those of hierarchical-memory data-caching, but are complicated by what we will show are time changing costs.

In this section we show a data transition policy. The setting we assume is relatively restricted. We consider a two-level system, in which data are accessed only via the higher level. We consider policies in which blocks are transferred in entire-object granularity, only entire-object modification takes place, and objects are evicted to a lower level consecutively and to uncorrelated lower-level placements. This corresponds to a system in which inter-object correlation is total, and cross-object correlation is non-existent. We defer a fuller solution to future work.

The section is organized as follows. In Subsection II-A we define precisely the transition costs. In Subsection II-B we show a competitive migration policy. In Subsection II-C we analyze some common solutions in use.

A. Performance Costs in Hierarchy levels

We first precisely define the transition costs and comparison measures.

We consider two levels composed of sets of blocks, S_0 and S_1. They are the higher and lower levels, respectively. The size S_0 is $|S_0| = k$; the size of S_1 obeys $|S_1| \gg |S_0|$. The two levels contain objects. We denote the set of objects they hold by L_0 and L_1, respectively. In general, $L_0 \bigcap L_1 \neq \emptyset$.

The policy handles a sequence of M requests $\underline{\rho} = [\rho_1, \ldots, \rho_M]$. Each request ρ_j is a pair (e_j, t_j). The entry e_j identifies the pertinent object. The entry t_j is either R or W, depending on whether the request is of type read or type write.

The cost of each operation depends on objects' locations and modification state. The cost incurred by a request (e_j, R) is 0 if $e_j \in L_0$; otherwise, the cost is denoted by $f^R_{e_j}$. If an unmodified object e is deleted from L_0, then the deletion cost is 0; otherwise the cost is denoted by f^W_e. The number of blocks required by an unmodified $e \in L_0$ is denoted by $\left|e^R\right|$. If the object is modified, an additional $\left|e^W\right|$ blocks of redundancy are required. We deal with fixed rate codes, and so for any two objects e_i and e_j of the same priority,

$$\frac{\left|e^R_i\right|}{\left|e^R_i\right| + \left|e^W_i\right|} = \frac{\left|e^R_j\right|}{\left|e^R_j\right| + \left|e^W_j\right|}, \tag{1}$$

regardless of the sizes of e_i and e_j.

The data transition problem above has much similarity to the problem of memory-hierarchy online paging.

It is well known [27] that in such settings, absolute performance measures for an algorithm are meaningless. We briefly review two meaningful comparison measures.

Let $\mathcal{A}$ be an *online* paging algorithm, *i.e.*, its response to $\underline{\rho}[j]$ does *not* depend on $\underline{\rho}[j']$ for any $j' \gneq j$. Let $\mathcal{A}(k)$ denote an instance of it for which $|L_0| = k$. *E.g.*, $\mathcal{A}$ is the *LRU* (least recently used) algorithm, and $\mathcal{A}(k)$ is LRU maintaining k items. We denote the cost incurred by $\mathcal{A}(k)$ on $\underline{\rho}$ by $f^{\mathcal{A}(k)}\left(\underline{\rho}\right)$. To assess how relatively good is $f^{\mathcal{A}(k)}\left(\underline{\rho}\right)$, we require the following two costs [23]:

- The off-line cost- let $\mathcal{O}$ denote the optimal *off-line* algorithm (*i.e.*, with advance knowledge of $\underline{\rho}$). Let $\mathcal{O}(h)$ denote its instance when $|L_0| = h$ ($h \leq k$). The cost incurred by this instance due to $\underline{\rho}$, is the off-line cost, $f^{\mathcal{O}(h)}\left(\underline{\rho}\right)$.
- The un-cached cost- let $f_{\underline{\rho}[j]=(e_j,t_j)}$ denote the *un-cached* cost of the jth operation, *i.e.*,

$$f_{\underline{\rho}[j]} = \begin{cases} f_{e^R_j} & , \quad t_j = R \\ f_{e^W_j} & , \quad t_j = W \end{cases} . \tag{2}$$

The un-cached cost of the sequence is $\sum_{i=1}^{M} f_{\underline{\rho}[j]}$.

The following definition [21] defines the competitiveness of an on-line algorithm relative to an optimal off-line algorithm.

Definition 1: An algorithm $\mathcal{A}$ is $\alpha = \alpha(h,k)$ competitive, if there is a constant $\gamma = \gamma(h,k)$, such that for any request sequence $\underline{\rho}$,

$$\mathbf{E}\left[f^{\mathcal{A}(k)}\left(\underline{\rho}\right)\right] \leq \alpha \cdot f^{\mathcal{O}(h)}\left(\underline{\rho}\right) + \gamma. \tag{3}$$

The competitiveness coefficient of $\mathcal{A}$, $\alpha_{\mathcal{A},h,k}$, is the infinimum of any such α which satisfies (3), *i.e.*,

$$\alpha_{\mathcal{A},h,k} = \inf_{\alpha} \exists_{\gamma=\gamma(h,k)} \forall_{\underline{\rho}} \mathbf{E}\left[f^{\mathcal{A}(k)}\left(\underline{\rho}\right)\right] \leq \alpha \cdot f^{\mathcal{O}(h)}\left(\underline{\rho}\right) + \gamma. \quad (4)$$

In the above, our definition differs from that of [23], by the additive γ element.

Subsequently, a modified definition of competitiveness, *loose competitiveness* [28], [23], was created. The new version has advantages in its not allowing $\underline{\rho}$ to be too closely tailored to k, and limiting the effect of any $\underline{\rho}$ for which the absolute cost is too low. The following is a modified version of loose competitiveness.

Definition 2: An algorithm $\mathcal{A}$ is $\hat{\alpha} = \hat{\alpha}(\epsilon, \delta, k)$-loosely-competitive, if there is a constant $\gamma = \gamma(k)$, such that for any request sequence $\underline{\rho}$, at least $(1-\delta)k$ of the values $k' \in [k]$ satisfy

$$\mathbf{E}\left[f^{\mathcal{A}(k')}\left(\underline{\rho}\right)\right] \leq \max\left\{\alpha \cdot f^{\mathcal{O}(k')}\left(\underline{\rho}\right), \epsilon \cdot \sum_{\rho \in \underline{\rho}} f\left(\rho\right)\right\} + \gamma. \quad (5)$$

The (ϵ, δ, k)-loose-competitiveness coefficient of $\mathcal{A}$, $\hat{\alpha}_{\mathcal{A},\epsilon,\delta,k}$, is the infinimum of any such $\hat{\alpha}$ which satisfies (5), *i.e.*,

$$\hat{\alpha}_{\mathcal{A},\epsilon,\delta,k} = \inf_{\alpha} \exists_{\gamma=\gamma(k)} \forall_{\underline{\rho}} \exists_{K' \in [(]k), |(|K') \geq (1-\delta)\cdot k} \; k' \subseteq K' \Rightarrow \mathbf{E}\left[f^{\mathcal{A}(k')}\left(\underline{\rho}\right)\right] \leq \max\left\{\alpha \cdot f^{\mathcal{O}(k')}\left(\underline{\rho}\right), \epsilon \cdot \sum_{\rho \in \underline{\rho}} f\left(\rho\right)\right\} + \gamma. \quad (6)$$

Definition 3: An algorithm $\mathcal{A}$ is (ϵ, δ)-loosely $\tilde{\alpha}$-competitive, if for any k, except for a finite number of ks, $\mathcal{A}$ is $\hat{\alpha} = \hat{\alpha}(\epsilon, \delta, k)$-loosely competitive, for $\hat{\alpha} \leq \tilde{\alpha}$.

In the above, our definition differs from that of [23], by requiring an algorithm to be $\hat{\alpha}(\epsilon, \delta, k)$-loosely competitive for *almost all* k.

Competitive paging algorithms have previously been studied for the case of multi-level memory hierarchies. This case differs from ours. In a memory hierarchy, the size an object requiring being a constant; in our case modified objects require redundancy. In a memory hierarchy, the cost is, in general, dominated by object retrieval; in our case, modified objects incur an eviction cost, while unmodified objects do not. This is aggravated by UEP. For the memory hierarchy case, algorithms for uniform-size uniform-cost objects were studied [22], [21], [26], algorithms for uniform-size arbitrary-cost objects were studied [25], [26], and algorithms for arbitrary-size and differing-costs were studied [29], [24], [23].

B. A Competitive Algorithm

In this subsection we describe *M-Landlord*, which is a modification of an algorithm from [23]. The act of writing modified data to a lower level is known as *destage*; the act of deleting unmodified from a higher level is known as *demote*. The algorithm works performing a continuing series of destage and demote operations, based on a space-per-cost object assessment. For brevity, we will refer in equations to this algorithm as $\mathcal{LL}^{\mathcal{M}}$.

The main result we prove is the following.

Theorem 1: Fix ϵ and δ, and let $\delta' \gneq 0$ be an arbitrarily small constant.

1. $\mathcal{LL}^{\mathcal{M}}$ is $(\epsilon, \delta + \delta')$-loosely $\frac{1-\ln(\epsilon)}{\delta}e$-competitive.
2. $\mathcal{LL}^{\mathcal{M}}$ has computational complexity $O(1)$ for operations which do not access the lower level, and computational complexity $O(\log(k))$, for operations which do.

The subsection is organized as follows. In Sub-subsection II-B.1 we describe the algorithm. In Sub-subsection II-B.2 we study a related hierarchical-memory algorithm, in order to analyze the $\mathcal{LL}^{\mathcal{M}}$ competitiveness. In Sub-subsection II-B.3 we prove the loose-competitiveness and computational-complexity properties of $\mathcal{LL}^{\mathcal{M}}$.

B.1 M-Landlord

In this sub-subsection we describe the algorithm $\mathcal{LL}^{\mathcal{M}}$. This is a modification of the Landlord algorithm [23], extended to the case where some objects are modified, and with lower computational complexity (at a cost to the generality of the original algorithm). Essentially, the algorithm maintains a space-per-cost estimate of each object. Based on this ratio, it decides on the next destage or demote operation used for freeing space. Algorithms 1 and 2 show the algorithm in pseudo-code.

The algorithm maintains the following global variable and array:

$$\begin{aligned} LL &= \text{set of objects in } L_0, \\ c &= \text{a real value describing “credit history”}. \end{aligned} \quad (7)$$

The algorithm maintains the following object-specific arrays and heap-based PQ (priority queues) [30]. Let $e \in LL$ be a an object, then:

$$\begin{aligned} \underline{m} &= \text{an array s.t. } \underline{m}(e) = \mathbf{T} \Leftrightarrow \text{e is modified}, \\ \underline{c} &= \text{an array s.t. } \underline{c}(e) = \text{object credit},, \\ H^R &= \text{a PQ s.t. } \underline{m}[e] = \mathbf{F} \Leftrightarrow e \in H^r, \\ H_i^W &= \text{a PQ s.t. } \underline{m}[e] = \mathbf{T} \bigwedge \mu\left(e\right) = i \Leftrightarrow e \in H_i^W. \end{aligned} \quad (8)$$

An object e is ordered within its PQ by $\underline{c}[e]$. Initially, $c = 0$.

Algorithm 1 shows a high-level description of the algorithm. Let e_j be the object accessed at step j. If e_j need not be retrieved and is not newly modified, no state updates need be done (lines 1 to 5). Space is allocated (by calling Algorithm 2), and the object's queue membership is updated (lines 6 to 11). The object is retrieved if it is not already in place (lines 12 to 20). A modification to the object results in updating the internal data structures (lines 21 to 26).

In all cases, the object's credit, $\underline{c}[e_j]$ is updated according to the credit history c and the operation type t_j. The object's cost-per-space is efficiently stored relative to objects in its class, by means of the appropriate PQ.

The algorithm maintains the following invariants on objects' credit,

Invariant 1:

$$\forall_{e \in LL} \left(\underline{c}[e] \leq f_e^R \right) \bigvee \left(\underline{m}[e] = \mathbf{T} \bigwedge \underline{c}[e] \leq f_e^W \right), \quad (9)$$
$$\forall_{e \in LL} \underline{c}[e] \geq 0,$$

and the following invariants on object PQs' location,

Invariant 2:

$$\forall_{e \in LL} \forall_{i \in [d]} e \in H^R \Rightarrow e \notin H_i^W, \quad (10)$$
$$\forall_{e \in LL} \forall_{i \in [d]} e \in H_i^W \Rightarrow e \notin H^R \bigwedge e \notin H_j^w \ (j \neq i).$$

Algorithm 1 M-Landlord($\underline{\rho}[j]$)

Handles a request $\underline{\rho}[j] = (e_j, t_j)$.

```
/* Check if ρ[j] does not modify objects' state.*/
if e_j ∈ LL ⋀ (t_j = W ⇒ m[e_j] = T) then
    Access e_j
    return
end if
/* Evict space needed */
M-Landlord-Evict(ρ[j])
/* Remove modified object from read PQ.*/
if e_j ∈ LL ⋀ t_j = W then
    PQ-Remove(H^R, e_j)
end if
/* Check if object should be retrieved.*/
if e_j ∉ LL then
    Retrieve e_j
    LL ← LL ⋃ {e_j}
    if t_j = R then
        c[e_j] ← c + f^R_{e_j} / |e^R_j|
        PQ-Insert(H^R, e_j)
    end if
end if
/* Check if object must be marked as modified.*/
if t_j = W then
    c[e_j] ← c (1 + |e^R_j| / |e^W_j|) + f^W_{e_j} / |e^W_j|
    m[e_j] ← T
    PQ-Insert(H^W_{μ(e_j)}, e_j)
end if
Access e_j
```

Algorithm 2 shows how space is cleared. First, the space needed for eviction is calculated (lines 1 to 6). The algorithm loops until at least that amount has been evicted (lines 7 to 31). First, the credit history is updated (lines 9 to 16). By comparing an objects' credit to the credit history, some objects are possibly demoted (lines 22 to 21), and for each of the priorities, some are possibly destaged (lines 22 to 30).

Algorithm 2 M-Landlord-Evict($\underline{\rho}[j]$)

Clears space for a request $\underline{\rho}[j] = (e_j, t_j)$.

```
Require: e_j ∉ LL ⋁ (t_j = W ⋀ m[e_j] = F)
1:  /* s indicates the space which need be cleared.*/
2:  s ← (t_j = R)? |e^R_j| : |e^W_j|
3:  /* Check if modifying an unmodified existing object.*/
4:  if t_j = W ⋀ e_j ∈ LL ⋀ m[e_J] = F then
5:      s ← s − |e^R_j|
6:  end if
7:  /* Loop until enough space has been cleared.*/
8:  while |LL| ≥ k − s do
9:      /* δ^R, δ^W_1, ..., δ^W_d = credit-history changes.*/
10:     δ^R ← c[PQ-Min(H^R)] − c
11:     for i ∈ [d] do
12:         δ^W[i] ← c[PQ-Min(H^W_i)] / (1 + |e^R_i| / |e^W_i|) − c
13:     end for
14:     δ ← min{δ^R, δ^W[1], ..., δ^W[d]}
15:     /* Update the credit history.*/
16:     c ← c + δ
17:     /* Demote some objects.*/
18:     while c[e = PQ-Min(H^R)] = c do
19:         LL ← LL \ {e}
20:         PQ-Remove(H^R)
21:     end while
22:     /* Destage some objects.*/
23:     for i ∈ [d] do
24:         while c[e = PQ-Min(H^W_i)] = c (1 + |e^R| / |e^W|) do
25:             PQ-Remove(H^W_i)
26:             m[e] = F
27:             c[e] ← c + f_{e^R} / |e^R|
28:             PQ-Insert(H^R, e)
29:         end while
30:     end for
31: end while
```

B.2 RW-Landlord

In this subsection we describe the algorithm RW-Landlord, which we will use for the competitiveness analysis of M-Landlord. It is difficult to analyze M-Landlord directly, because of the time-varying costs and sizes of objects. For brevity in mathematical expressions, we will interchange RW-Landlord with $\mathcal{LL}^{\mathcal{RW}}$.

RW-Landlord operates in a standard caching-problem setting (*i.e.*, without object modification). We relate the setting of RW-Landlord to that of M-Landlord as follows. For any object e in M-Landlord, we consider a *read object* e^R, and a *write object* e^W. The retrieval cost of e^R is f_{e^R}, the retrieval cost of e; the retrieval cost of e^W is f_{e^W}, the eviction cost of e. The size of e^R is the size of e; the size

of e^W is the size of the redundancy of e. We conceptually partition LL, the cache of $\mathcal{LL}^{\mathcal{RW}}$, into sub-caches LL^R and LL^W, containing read and write objects respectively. *I.e.*, $LL = LL^R \dot{\bigcup} LL^W$, where

$$\begin{aligned} e^R \in LL &\Rightarrow e^R \in LL^R, \\ e^W \in LL &\Rightarrow e^W \in LL^W. \end{aligned} \tag{11}$$

When studying the competitiveness of RW-Landlord, we similarly partition the cache of $\mathcal{O}$ to $O = O^R \dot{\bigcup} O^W$ as well.

We differentiate the setting from that of a classical caching problem, by requiring the following invariants on object containment:

Invariant 3:

$$\begin{aligned} \forall_{e^W} e^W \in LL^W &\Rightarrow e^R \in LL^R \\ \forall_{e^W} e^W \in O^W &\Rightarrow e^R \in O^R, \end{aligned} \tag{12}$$

and the requirement that servicing a write request entails that both read object and write objects are in the cache, *i.e.*,

Invariant 4:

$$\underline{\rho}[j] = (e_j, W) \Rightarrow \left(e_j^R \in LL^R \bigwedge e_j^W \in LL^W\right). \tag{13}$$

Specifically, we must take care that the algorithm does not evict a read object before the corresponding write object, and that a read object is not evicted in order to make place for its corresponding write object. The former would be equivalent to evicting an object while retaining its redundancy. The latter would be equivalent to evicting an object in order to make place for its redundancy.

We will show that RW-Landlord has the corresponding invariants to that of M-Landlord's Invariant 1:

Invariant 5:

$$\begin{aligned} \forall_{e^R \in LL^R} \underline{c}\left[e^R\right] &\leq f_{e^R}, \\ \forall_{e^W \in LL^W} \underline{c}\left[e^W\right] &\leq f_{e^W}, \\ \forall_{e \in LL} \underline{c}[e] &\geq 0 \end{aligned} \tag{14}$$

Algorithm 3 shows a high level description of the algorithm. In structure, it is quite similar to that of Algorithm 1. There are some differences. Note that a request $\underline{\rho}[j]$ is now explicitly to e_j^R or to e_j^W, instead of specifying an object and an access type. The credit of each object is maintained directly, instead of the use of the credit history variable c in Algorithm 1. Access to objects is done via Algorithm 4, to make explicit Invariance 4.

Algorithm 5 shows how space is cleared for a request. First, the space needed for eviction is calculated (lines 1 to 6). The algorithm loops until at least that amount has been evicted (lines 8 to 31), while maintaining Invariant 3. In each iteration, credit is decreased from all objects (line 11). Objects whose credit is 0, are evicted (line 28).

To ensure that invariant 5 is maintained, credit transference is used. When clearing space for a write object, the credit of the corresponding read object is raised to the maximum (line 14). In addition, all write objects transfer credit to their read objects (lines 16 to 23).

Algorithm 3 RW-Landlord($\underline{\rho}[j]$)

Handles a request $\underline{\rho}[j] = e_j^{t_j}$.

```
/* Check if ρ[j] does not modify objects' state.*/
if e_j^R ∈ LL^R ⋀ (t_j = W ⇒ e_j^W ∈ LL^W) then
    RW-Landlord-Service-Request(ρ[j])
    return
end if
/* Evict space needed */
RW-Landlord-Evict(ρ[j])
/* Check if read object must be retrieved.*/
if e_j^R ∉ LL^R then
    Retrieve e_j^R
    LL^R ← LL^R ⋃̇ {e_j^R}
    c[e_j^R] ← f_{e_j^R}
end if
/* Check if write object must be retrieved.*/
if t_j = W then
    Retrieve e_j^W
    LL^W ← LL^W ⋃̇ {e_j^W}
    c[e_j^R] ← f_{e_j^R}
    c[e_j^W] ← f_{e_j^W}
end if
RW-Landlord-Service-Request(ρ[j])
```

Algorithm 4 RW-Landlord-Service-Request($\underline{\rho}[j]$)

Handles an existing-object request $\underline{\rho}[j] = e_j^{t_j}$.

Require: $e_j^R \in LL^R \bigwedge (t_j = W \Rightarrow e_j^W \in LL^W)$

```
/* For reads, access and update a single object.*/
if (t_j = R) then
    access e_j^R
    /* For write, access and update two objects.*/
else
    access e_j^R and e_j^W
end if
```

We now analyze the competitiveness of the algorithm.

Theorem 2: $\mathcal{LL}^{\mathcal{RW}}$ is $\frac{k}{h-k+1}$ competitive.

To analyze $\mathcal{LL}^{\mathcal{RW}}$, we use a potential function from [23].

Algorithm 5 RW-Landlord-Evict($\underline{\rho}[j]$)

Clears space for a request $\underline{\rho}[j] = e_j^{t_j}$.

Require: $e_j^R \notin LL^R \bigvee \left(t_j = W \bigwedge e_j^W \notin LL^W\right)$

1: /* s indicates the space which need be cleared.*/
2: $s \leftarrow (t_j = R)? \left|e_j^R\right| : \left|e_j^W\right|$
3: **if** $t_j = W \bigwedge e_j^R \notin LL^R$ **then**
4: /* Increment s if a read object is needed.*/
5: $s \leftarrow s + \left|e_j^R\right|$
6: **end if**
7: /* Evict space while maintaining Invariant 3.*/
8: **while** $|LL| \geq k - s$ **do**
9: $\delta \leftarrow \min\left\{\min_{e^R \in LL^R} \frac{\underline{c}\left[e^R\right]}{\left|e^R\right|}, \min_{e^W \in LL^W} \frac{\underline{c}\left[e^W\right]}{\left|e^R\right|+\left|e^W\right|}\right\}$
10: **for** $e \in LL$ **do**
11: $\underline{c}[e] \leftarrow \underline{c}[e] - \delta\,|e|$
12: **if** $t_j = W \bigwedge e_j^R \in LL^R$ **then**
13: /* Update corresponding read-object, if in cache.*/
14: $\underline{c}\left[e_j^R\right] \leftarrow f_{e_j^R}$
15: **end if**
16: **for** $e^R \in LL^R$ **do**
17: **if** $e^W \in LL^W$ **then**
18: /* Transfer credit from write objects to read objects.*/
19: $\delta' \leftarrow \min\left\{\delta\left|e^R\right|, \delta\left|e^W\right|\right\}$
20: $\underline{c}\left[e^R\right] \leftarrow \underline{c}\left[e^R\right] + \delta'$
21: $\underline{c}\left[e^W\right] \leftarrow \underline{c}\left[e^W\right] - \delta'$
22: **end if**
23: **end for**
24: **end for**
25: **for** $e \in LL$ **do**
26: **if** $\underline{c}[e] = 0$ **then**
27: /* Evict objects with 0 credit.*/
28: $LL \leftarrow LL \setminus \{e\}$
29: **end if**
30: **end for**
31: **end while**

Definition 4: Define the potential function

$$\Phi = (h-1) \sum_{e'^R \in LL^R} \begin{cases} \underline{c}\left[e'^R\right] + \underline{c}\left[e'^W\right] & , e'^W \in LL^W \\ \underline{c}\left[e'^R\right] & , e'^W \notin LL^W \end{cases} + k \sum_{e'^R \in O^R} \begin{cases} f_{e'^R} + f_{e'^W} & , e'^W \in O^W \\ f_{e'^R} & , e'^W \notin O^W \end{cases} - k \sum_{e'^R \in O^R} \begin{cases} \underline{c}\left[e'^R\right] + \underline{c}\left[e'^W\right] & , e'^W \in O^W \\ \underline{c}\left[e'^R\right] & , e'^W \notin O^W \end{cases} \tag{15}$$

The following lemma proves Theorem 2.

Lemma 1: Following any series of actions by $\mathcal{LL}^{\mathcal{RW}}$ and $\mathcal{O}$, $\Phi \geq 0$. The following four operations affect Φ:

- $\mathcal{O}$ retrieves e^R: Φ increases by at most kf_{e^R}.
- $\mathcal{O}$ retrieves e^W: Φ increases by at most kf_{e^W}.
- $\mathcal{LL}^{\mathcal{RW}}$ retrieves e^R: Φ decreases by at least $(k-h+1)f_{e^R}$.
- $\mathcal{LL}^{\mathcal{RW}}$ retrieves e^W: Φ decreases by at least $(k-h+1)f_{e^W}$.

No other action by $\mathcal{O}$ or $\mathcal{LL}^{\mathcal{RW}}$ increases Φ.

The proof of Lemma 1 is similar in many points to that in [23]. We focus mainly on the points in which it differs due to Invariants 3, 4, and 5.

Proof: We analyze the effect of the steps taken by $\mathcal{O}$ and $\mathcal{LL}^{\mathcal{RW}}$ on Φ.

- $\mathcal{O}$ evicts $e \in O$: Since Invariant 5 is maintained (see Algorithm 3), Φ cannot increase.
- $\mathcal{O}$ retrieves $e \in O$: In this case $\mathcal{O}$ pays f_e. Since Invariant 5 ($\forall_{e \in LL} \underline{c}[e] \geq 0$) is maintained (see Algorithm 3), then Φ increases by at most kf_e.
- $\mathcal{LL}^{\mathcal{RW}}$ transfers credit from $\underline{c}\left[e^W\right]$ to $\underline{c}\left[e^R\right]$ (lines 21 and 20): In this case $e_j^R \in LL^R$ and $e_j^W \in LL^W$. Rewriting Φ, we have

$$\Phi = (h-1)\left(\underline{c}\left[e^R\right] + \underline{c}\left[e^W\right]\right) + (h-1) \sum_{e'^R \in LL^R \setminus \{e^R\}} \begin{cases} \underline{c}\left[e'^R\right] + \underline{c}\left[e'^W\right] & , e'^W \in LL^W \\ \underline{c}\left[e'^R\right] & , e'^W \notin LL^W \end{cases} + k \begin{cases} f_{e^R} + f_{e^W} - \left(\underline{c}\left[e^R\right] + \underline{c}\left[e^W\right]\right) & , e^W \in O^W \\ f_{e^R} - \underline{c}\left[e^R\right] & , e^R \in O^R,\ e^W \notin O^W \\ 0 & , e^R \notin O^R \end{cases} + k \sum_{e'^R \in O^R \setminus \{e^R\}} \begin{cases} f_{e'^R} + f_{e'^W} & , e'^W \in O^W \\ f_{e'^R} & , e'^W \notin O^W \end{cases} - k \sum_{e'^R \in O^R \setminus \{e^R\}} \begin{cases} \underline{c}\left[e'^R\right] + \underline{c}\left[e'^W\right] & , e'^W \in O^W \\ \underline{c}\left[e'^R\right] & , e'^W \notin O^W \end{cases}, \tag{16}$$

which shows that Φ cannot increase.

- $\mathcal{LL}^{\mathcal{RW}}$ increments $\underline{c}\left[e^R\right]$ to f_{e^R} (line 14): We can assume in this case that $e^R \in O^R$, and therefore the increase to Φ is $(f_{e^R} - \underline{c}\left[e^R\right])(h-1-k) \leq 0$.
- $\mathcal{LL}^{\mathcal{RW}}$ decreases uniformly $\underline{c}[e']$ for all $e' \in LL$ (line 11): This occurs in one of three cases: either there is an $e^R \in O^R \setminus LL^R$, or there is an $e^W \in O^W \setminus LL^W$, or both of the previous. We prove the first case (which is the only case in [23]). The other two cases are similar. Letting $O'^R = O^R \bigcap LL^R$, and, $O'^W = O^W \bigcap LL^W$ we have

$$\Phi \overset{(a)}{=} (h-1) \sum_{e'^R \in LL^R} \begin{cases} \underline{c}\left[e'^R\right] + \underline{c}\left[e'^W\right] & , e'^W \in LL^W \\ \underline{c}\left[e'^R\right] & , e'^W \notin LL^W \end{cases} + k \sum_{e'^R \in O^R} \begin{cases} f_{e'^R} + f_{e'^W} & , e'^W \in O^W \\ f_{e'^R} & , e'^W \notin O^W \end{cases} - k \sum_{e'^R \in O'^R} \begin{cases} \underline{c}\left[e'^R\right] + \underline{c}\left[e'^W\right] & , e'^W \in O'^W \\ \underline{c}\left[e'^R\right] & , e'^W \notin O'^W \end{cases}. \tag{17}$$

(where (a) follows from the fact that for $e' \notin LL$, $\underline{c}[e'] = 0$. Since $e^R \in O \setminus LL$, $|O \bigcap LL| \leq h - \left|e^R\right|$. Since there is no

room for e^R in LL, $|LL| \geq k - \left|e^R\right| + 1$. It follows that the decrease to Φ is at least $(h-1)(k-\left|e^R\right|+1) - k(h-\left|e^R\right|) = (1-\left|e^R\right|)(h-1-k) \geq 0$.

- $\mathcal{LL}^{\mathcal{RW}}$ evicts an object e (line 28): Since this happens when $\underline{c}[e] = 0$ (line 26), Φ does not change.
- $\mathcal{LL}^{\mathcal{RW}}$ retrieves e^R (line 11) and sets its credit to $\underline{c}\left[e^R\right]$ (line 12): In this case, $\mathcal{LL}^{\mathcal{RW}}$ pays a cost of f_{e^R}. Since the retrieval is done in response to a request for e^R or e^W, we can assume that $e_j^R \in O^R$. The increment to the credit therefore increments Φ by $(h-1+k)f_e$, or equivalently, decrease it by $(k-h+1)f_e$.
- $\mathcal{LL}^{\mathcal{RW}}$ retrieves e^W (line 17) and sets its credit to $\underline{c}\left[e^W\right]$ (line 19): This is similar to the previous case.

■

B.3 Loose Competitiveness of M-Landlord

The following lemma shows that $\mathcal{LL}^{\mathcal{M}}$ and $\mathcal{LL}^{\mathcal{RW}}$ work similarly.

Lemma 2: Assume at the starting point $\mathcal{LL}^{\mathcal{M}}$ is operating on an empty level, and $\mathcal{LL}^{\mathcal{RW}}$ is operating on an empty cache. Let $\underline{\rho}^{\mathcal{M}}$ and $\underline{\rho}^{\mathcal{RW}}$ be request sequences to $\mathcal{LL}^{\mathcal{M}}$ and $\mathcal{LL}^{\mathcal{RW}}$, respectively, s.t. at any request $j \in [M]$,

$$\underline{\rho}^{\mathcal{M}}[j] = (e_j, t_j) \Leftrightarrow \underline{\rho}^{\mathcal{RW}}[j] = e_j^{t_j} \tag{18}$$

Then following the request,

- $\mathcal{LL}^{\mathcal{M}}$ contains an unmodified object $e \leftrightarrow \mathcal{LL}^{\mathcal{RW}}$ contains e^R and does not contain e^W.
- $\mathcal{LL}^{\mathcal{M}}$ contains a modified object $e \Leftrightarrow \mathcal{LL}^{\mathcal{RW}}$ contains both e^R and e^W.
- $\mathcal{LL}^{\mathcal{M}}$ does not contain an object $e \Leftrightarrow \mathcal{LL}^{\mathcal{RW}}$ contains neither e^R nor e^W.

Proof: We first show the relation between credit variables maintained by the algorithms $\mathcal{LL}^{\mathcal{M}}$ and $\mathcal{LL}^{\mathcal{RW}}$.

If LL contains modified e_j, then

$$\underline{c}\left[e_j^W\right] = \left(\underline{c}[e_j] - c\left(1 + \frac{\left|e_j^R\right|}{\left|e_j^W\right|}\right)\right)\left|e_j^W\right| \tag{19}$$

and

$$\underline{c}\left[e_j^R\right] = f_{e_j^R}. \tag{20}$$

If LL contains unmodified e_j, then

$$\underline{c}\left[e_j^R\right] = \left(\underline{c}[e_j] - c\right)\left|e_j^R\right|. \tag{21}$$

and there is no e_j^W object.

In other words, the actual credit of an element is given by (19) if it is modified and by (21) if it is unmodified.

if e_j is modified then (20) clearly holds, since its read and write objects exist, and hence whenever $\underline{c}\left[e_j^R\right]$ is decreased in line 11 of Algorithm RW-Landlord-Evict, it is increased back to the initial value in lines 16-21.

Initially (when an element is inserted to priority queues in Algorithm $\mathcal{LL}^{\mathcal{M}}$) (19) and (21) are clearly true (see lines 18 and 25 in Algorithm M-Landlord and line 28 in Algorithm M-Landlord-Evict).

The value of element's credit is effectively decreased in line 16 in Algorithm M-Landlord-Evict (by increasing the counter). If e_j is unmodified, the credit of e_j^R is decreased by $\delta\left|e_j^R\right|$ in line 11 in Algorithm RW-Landlord-Evict, while c is decreased by δ in line 16 in Algorithm M-Landlord-Evict. Hence, Equation 21 still holds. If e_j is modified, the credit of e_j^W is decreased by $\delta\left|e_j^W\right|$ in line 11 and additionally by $\delta\left|e_j^R\right|$ in line 21 in Algorithm RW-Landlord-Evict, while c is increased by δ in line 16 in Algorithm M-Landlord-Evict. Hence, Equation 19 still holds.

Finally, we note that by Equation 21, condition for removal of e_j from LL in line 20 in M-Landlord-Evict corresponds to a condition $f_{e^R} = 0$ in line 26 in RW–Landlord-Evict. Similarly, by Equation 19, condition for marking e_j as unmodified in line 26 in M-Landlord-Evict corresponds to a condition $f_{e^W} = 0$ (for removal of write object) in line 26 in RW–Landlord-Evict. ■

By Theorem 2, $\mathcal{LL}^{\mathcal{RW}}$ is $\frac{k}{k-h+1}$ competitive. We show that this implies the competitiveness of $\mathcal{LL}^{\mathcal{M}}$ as well.

Theorem 3: $\mathcal{LL}^{\mathcal{M}}$ is $\frac{k}{k-h+1}$ competitive.

Proof: Let

$$\gamma'(h,k) = \max_{e^W} f_{e^W}\frac{h}{|e^W|}. \tag{22}$$

We then have

$$\begin{aligned} & f^{\mathcal{LL}^{\mathcal{M}}(k)}\left(\underline{\rho}^{\mathcal{M}}\right) \overset{(a)}{\leq} \\ & \quad f^{\mathcal{LL}^{\mathcal{RW}}(k)}\left(\underline{\rho}^{\mathcal{RW}}\right) \overset{(b)}{\leq} \\ & \quad \frac{k}{k-h+1} f^{\mathcal{O}^{\mathcal{RW}}(h)}\left(\underline{\rho}^{\mathcal{RW}}\right) \overset{(c)}{\leq} \\ & \quad \frac{k}{k-h+1} f^{\mathcal{O}^{\mathcal{M}}(h)}\left(\underline{\rho}^{\mathcal{M}}\right) + \frac{k}{k-h+1}\gamma'(h,k) \end{aligned} \tag{23}$$

where in the above, (a) follows from the fact that $\mathcal{LL}^{\mathcal{M}}$ "simulates" the actions of $\mathcal{LL}^{\mathcal{RW}}$, and so its cost is not higher than that of $\mathcal{LL}^{\mathcal{RW}}$, (b) follows from Theorem 2, and (c) follows from the fact that if an algorithm $\mathcal{O}'^{\mathcal{RW}}$ for the cache problem would "simulate" the actions of $\mathcal{O}^{\mathcal{M}}$, then

$$f^{\mathcal{O}'^{\mathcal{RW}}(k)}\left(\underline{\rho}^{\mathcal{RW}}\right) \leq f^{\mathcal{O}^{\mathcal{M}}(h)}\left(\underline{\rho}^{\mathcal{M}}\right) + \gamma'(h,k), \tag{24}$$

since for each write object retrieved, $\mathcal{O}^{\mathcal{M}}$ might decide not to flush it to the lower level, but the number of objects for which this is true is bounded by a function of h, and not of $\underline{\rho}$.

From (23) we have that $\mathcal{LL}^{\mathcal{RW}}$ is competitive according to Definition 1, with

$$\begin{aligned} \alpha(h,k) &= \frac{k}{k-h+1}, \\ \gamma(h,k) &= \frac{k}{k-h+1}\gamma'(h,k). \end{aligned} \tag{25}$$

■

The following proof of claim 1 in Theorem 1 follows [23], but differ in the definitions of loose competitive, and of the accompanying constants.

Lemma 3: $\mathcal{LL}^{\mathcal{M}}$ is (ϵ,δ)-loosely $\frac{(1-\ln(\epsilon))e}{\delta}$-competitive.

Proof: For fixed ϵ,δ, and k, let $b \gneq 0$ be a constant, and define

$$\eta = \eta(b,k,\epsilon,\delta) = \frac{k}{b}\epsilon^{-\frac{b}{\delta k-b}}, \tag{26}$$

For a fixed $\underline{\rho}$, let B be a set of s *bad* values consisting of any j for which

$$f^{\mathcal{LL}^{\mathcal{M}}(j)}\left(\underline{\rho}\right) \geq \max\left\{\alpha\cdot f^{\mathcal{O}(j)}\left(\underline{\rho}\right), \epsilon\sum_{i=1}^{M} f_{\underline{\rho}[i]}\right\} + \gamma(j). \tag{27}$$

Specifically, let $B=\{k_1,\ldots,k_s\}$. W.l.o.g., let $k_1\leq\cdots\leq k_s\leq k$. Define the subset $B'=\{k'_1,\ldots,k'_{s'}\}\subseteq B$, by $k'_i = k_{\lceil ib\rceil}$, for $i\in\left[\left[\frac{s}{\lceil b\rceil}\right]\right]$. Note that for $i\in[s']$, $k'_i-k'_{i-1}\geq b$, and that $s'\geq\frac{s}{b+1}-1$. We then have

$$\begin{aligned} & f^{\mathcal{LL}^{\mathcal{M}}(k'_i)}\left(\underline{\rho}\right) \overset{(a)}{\leq} \\ & \frac{k'_i}{k'_i-k'_{i-1}+1} f^{\mathcal{O}(k'_{i-1})}\left(\underline{\rho}\right) \overset{(b)}{\leq} \\ & \frac{k'_i}{\eta(k'_i-k'_{i-1}+1)}\cdot \\ & \left(f^{\mathcal{LL}^{\mathcal{M}}(k'_{i-1})}\left(\underline{\rho}\right) - \gamma(k'_i-k'_{i-1}+1,k'_{i-1})\right) \overset{(c)}{\leq} \\ & \epsilon^{\frac{b}{\delta k-b}} f^{\mathcal{LL}^{\mathcal{M}}(k'_{i-1})}\left(\underline{\rho}\right), \end{aligned} \tag{28}$$

where, in the above, (a) follows from Theorem 3, (b) follows from the fact that k'_{i-1} is bad and (27), and (c) follows from the fact that $\frac{k'_i}{\eta(k'_i-k'_{i-1}+1)}\leq\frac{k}{\eta b}$ and (26).

Inductively,

$$f^{\mathcal{LL}^{\mathcal{M}}(k'_{s'})}\left(\underline{\rho}\right) \leq \left(\epsilon^{\frac{b}{\delta k-b}}\right)^{s'}\cdot f^{\mathcal{LL}^{\mathcal{M}}(k'_0)}\left(\underline{\rho}\right). \tag{29}$$

We also have

$$\epsilon\cdot f^{\mathcal{LL}^{\mathcal{M}}(k'_0)}\left(\underline{\rho}\right) \overset{(a)}{\leq} \epsilon\cdot\sum_{i=1}^{M} f_{\underline{\rho}[i]} \overset{(b)}{\leq} < f^{\mathcal{LL}^{\mathcal{M}}(k'_{s'})}\left(\underline{\rho}\right), \tag{30}$$

where (a) follows from the fact that the cost of an algorithm cannot be larger than the sequence request, and (b) follows from (27).

It follows from (29) and (30) that $\epsilon\leq\left(\epsilon^{\frac{b}{\delta k-b}}\right)^{s'}$, from which it follows that $s'\leq\frac{\delta k-b}{b}$, and therefore

$$s\leq\delta\frac{b+1}{b}k. \tag{31}$$

From the definition of the set B, and (31), we have that $\mathcal{LL}^{\mathcal{M}}$ is $(\epsilon,\delta\frac{b+1}{b},k)$-loosely $\frac{k}{b}\epsilon^{-\frac{b}{\delta k-b}}$-competitive, for $b\gneq 0$.

For any k, setting $b=\frac{\delta k}{1-\ln(\epsilon)}$ in (26), yields $\eta=\frac{1-\ln(\epsilon)}{\delta}e$. For all but a finite number of k,

$$\delta\frac{b+1}{b} = \delta\left(1+\frac{1}{\frac{\delta k}{1-\ln(\epsilon)}}\right)\leq\delta+\delta'. \tag{32}$$

■

C. Comparison to Some Common Solutions

Present systems commonly employ two types of solutions for the eviction problem in Subsection II-A. In the first, *maximal protection*, all data is protected to the maximal requirement. In the second, *static partitioning* devices, or devices' blocks, are statically partitioned s.t. each partition element is dedicated to elements of a single priority. In both cases, an eviction policy for uniform priority is used (either according to the highest priority, or within each partition element). It is clear that these solutions are not efficient in terms of storage efficiency (in particular the first one). We show in this subsection that, in addition, they are not competitive in any definition in Subsection II-A.

Theorem 4: Let d be a number of priorities. Let $\mathcal{A}_m$ be any eviction algorithm based on maximal protection (while the ratio between the highest priority space consumption per one element to the lowest priority one is $\frac{\nu_d}{\nu_1}$) , and $\mathcal{A}_s$ be any eviction algorithm based on static partitioning, where either or both of $\mathcal{A}_m$ and $\mathcal{A}_s$ can be random algorithms.

1. For any h,k s.t. $h\gneq\frac{k}{d}$, $\alpha_{\mathcal{A}_s,h,k}=\infty$.
2. For any $\delta>\frac{1}{d}$, $\epsilon<1-\frac{1}{d\delta}$ and k, $\hat{\alpha}_(\mathcal{A}_s,\epsilon,\delta,k)=\infty$.
3. For any h,k s.t. $h\gneq\frac{k}{\frac{\nu_d}{\nu_1}}$, $\alpha_{\mathcal{A}_m,h,k}=\infty$.
4. For any $\delta>\frac{1}{d}$, $\epsilon<1-\frac{1}{\delta\frac{\nu_d}{\nu_1}}$ and k, $\hat{\alpha}_(\mathcal{A}_m,\epsilon,\delta,k)=\infty$.

We first prove claim 1 in Theorem 4.

Proof: Let $E=\left\{e_1,\ldots,e_{\frac{k}{d}+1}\right\}$ be arbitrary elements of priority j. Consider the request sequence $\underline{\rho}=\rho_1,\ldots,\rho_M$ s.t. $\rho_i=(e_{i \mod (\frac{k}{d}+1)},R)$. Clearly, the cost of the optimal algorithm on the request sequence is $f^{\mathcal{O}(h)}\left(\underline{\rho}\right)=\sum_{i\in\left\{\frac{k}{d}+1\right\}} f_{e_i}$. Dividing the request series into rounds, each of size $\frac{k}{d}$, it is easy to see that in each round, $\mathcal{A}_s$ incurs a cost of at least $\min_{i\in\left[\frac{k}{d}+1\right]} f_{e_i}$. Clearly,

$$\frac{f^{\mathcal{A}_s(k)}\left(\underline{\rho}\right)}{f^{\mathcal{O}(h)}\left(\underline{\rho}\right)} \geq \frac{\sum_{j=0}^{\frac{M}{\frac{k}{d}+1}}\min_{i\in\left[\frac{k}{d}+1\right]} f_{e_i}}{\sum_{i\in\left\{\frac{k}{d}+1\right\}} f_{e_i}} \overset{M\to\infty}{\longrightarrow} \infty. \tag{33}$$

The proof now follows by Yao's Minimax principle [27]. ■ The proof of claim 3 in Theorem 4 is similar.

We now prove claim 2 in Theorem 4.

Proof: Let $\gamma\in(1,d)$ be a real number. Let $\mathcal{A}_s^{det}$ be the best online paging deterministic algorithm, which uses static partitioning. Clearly, for each cache size $k'\leq k$ there is some j such that Algorithm $\mathcal{A}_s^{det}$ allocates for priority j at most $\frac{k}{d}$ space.

Set $E=\left\{e_1,\ldots,e_{\gamma\frac{k}{d}}\right\}$ be arbitrary elements of priority j (having cost f_e each one). The read sequence $\underline{\rho}$ of length M

is chosen in the following way: $\rho[i]$ is chosen uniformly from all the items of E. Then for cache sizes k' s.t. $\gamma\frac{k}{d} \leq k' \leq k$ we have

$$\mathbf{E}\left[f^{\mathcal{A}_s^{det}(k')}\left(\underline{\rho}\right)\right] \geq M\frac{\gamma-1}{\gamma}f_e, \tag{34}$$

$$\sum_{i=1}^{M} f_{\rho[i]} \leq Mf_e, \tag{35}$$

$$f^{\mathcal{O}(k')}\left(\underline{\rho}\right) \leq \gamma\frac{k}{d}f_e. \tag{36}$$

Note that (36) holds if $k' \geq \gamma\frac{k}{d}$. Hence,

$$\frac{\mathbf{E}\left[f^{\mathcal{A}_s^{det}(k')}\left(\underline{\rho}\right)\right]}{f^{\mathcal{O}(k')}\left(\underline{\rho}\right)} \geq \frac{M\frac{\gamma-1}{\gamma}}{\gamma\frac{k}{d}} \overset{M\to\infty}{\longrightarrow} \infty \tag{37}$$

for all $\gamma\frac{k}{d} \leq k' \leq k$.

$$\frac{\mathbf{E}\left[f^{\mathcal{A}_s^{det}(k')}\left(\underline{\rho}\right)\right]}{\sum_{i=1}^{M} f_{\rho[i]}} \geq \frac{\gamma-1}{\gamma} = \epsilon. \tag{38}$$

Hence, for $\epsilon < \frac{\gamma-1}{\gamma}$ and $\delta > \frac{\gamma}{d}$, the algorithm $\mathcal{A}_s^{det}$ is not (ϵ, δ, k) loosely-competitive. In other words, the algorithm $\mathcal{A}_s^{det}$ is not (ϵ, δ, k) loosely-competitive for $\delta > \frac{1}{d}$ and $\epsilon < 1 - \frac{1}{d\delta}$. By theorem **??**, the algorithm $\mathcal{A}_s$ is not loosely-competitive too. ■

The proof of claim 4 in Theorem 4 is similar.

III. The Necessity of Hierarchical Reliability

In this section we deal with bounds on storage-system reliability as the devices' capacity grows. These bounds are absolute limits, in the sense that they cannot be exceeded by any choice of codes, disk-replacement policy, or data-migration policy. The results in this section justify the necessity of a hierarchical reliability system. For brevity, we omit much of the details of the proofs.

We consider a system S composed of n storage devices. Each device is composed of c *blocks* containing b consecutive bits, and has a read/write bandwidth of r blocks per time unit. We model the devices' failure laws as being distributed i.i.d. exponentially with mean $\frac{1}{\lambda}$ [1]. The failure of a device corresponds to the erasure of all data on it. devices have i.i.d. (independently and identically distributed) entire-device failure laws. A *block erasure* is a detected corrupted block, and occurs with probability P_E. A *block substitution* is an undetected corrupt block, and occurs with probability P_S. If a block is substituted, we assume each bit has been flipped with probability P_s. *E.g.*, $P_s = \frac{1}{2}$ corresponds to a random earlier version of block contents. The bounds in this section hold even for cases in which $P_E = P_S = 0$.

The amount of user data stored in the entire system is c_{us}. The *MTTF* (mean time to failure) of a component C is the mean of t_C^f. The *MTTDL* (mean time to data loss) of a component C is the mean time until the failure of a component in C will cause data within it to be irreversibly lost. The *system MTTDL* is the MTTDL of S, the entire system. W.l.o.g., we consider a uniform error-protection requirement. Let ν be the ratio between the system MTTDL and the time to sequentially read a single device. We consider systems in which the system MTTDL (equivalently ν) can be made arbitrarily high as c_{us} and $|S|$ grow large. The *storage rate* of the system is $\rho = \frac{c_{\text{us}}}{|S|\cdot c} \leq 1$ (some of the related work term this the *storage efficiency*). The maximal amount of data which can be stored with such ν is $c_{\max}^R(S) = c_{\max}^R(S, \nu)$.

The main idea of the proof is as follows. We will first show that the maximal amount of data which can be stored in the system, $c_{\max}^R(S)$, is determined by the *effective volume* of the system and by the *block capacity* of each of its blocks. Using this, we will show in two ways that an increase in the amount of user data, c_{us}, must result in an increase in the amount of devices, n: the effective volume of the system does not increase beyond a certain point of increase in c, and the average amount of recovery-related operations grows at least linearly with c. For performance reasons, the blocks in a storage system are usually not coded en-mass in a single coding group. Rather, the blocks are partitioned into smaller groups, and the blocks of each group are protected by some code (*e.g.*, in a mirroring scheme, the blocks are partitioned into groups of size 2). We will show that an increase in n must entail an increase in the average redundancy of each group. This can be shown to lead to an increase in the amount of blocks that need be accessed when a data-block is written or modified.

We first define the *effective volume* and *block capacity*, and show how they limit the effective amount of user data which can be reliably stored in the system.

Definition 5: (Effective Volume and Block Capacity) The *effective volume* is defined as

$$\hat{c}(S) = r \max_{c_1,\ldots,c_n}\left\{\sum_{i=1}^{n} c_i \mid \exists_{C_1,\ldots,C_n}\mathbf{P}\left(\bigwedge_{i=1}^{n} t_{C_i}^f \geq c_i\right) \geq \frac{1}{\nu}\right\}. \tag{39}$$

The *block capacity* is defined as

$$\begin{aligned}\hat{b}(S) = {} & b(1-P_E) + H(P_E) + P_E\log(P_E) + \\ & \sum_{i=1}^{b}\left(\binom{2^b}{i}(1-P_E)P_SP_s^i(1-P_s)^{b-i}\cdot\right. \\ & \left.\log\left(\binom{2^b}{i}(1-P_E)P_SP_s^i(1-P_s)^{b-i}\right)\right) + \\ & (1-P_E)\left(P_S(1-P_s)^b + (1-P_S)\right)\cdot \\ & \log\left((1-P_E)\left(P_S(1-P_s)^b + (1-P_S)\right)\right).\end{aligned} \tag{40}$$

Theorem 5: For any system S,

$$c_{\max}^R(S) \leq \hat{c}(S)\,\hat{b}(S). \tag{41}$$

To prove Theorem 5, consider *epochs*, each taking $\frac{c}{r}$ time, and divided into *steps* of duration $\frac{1}{r}$. We consider an epoch starting at time 1. We define the following sets of devices:

$$\begin{aligned} \mathcal{X} &= \text{set of system devices at time 1,} \quad (42)\\ \mathcal{Y} &= \text{set of replacement devices inserted by time} \frac{c}{r}. \end{aligned}$$

We also define the following sets of blocks:

$$\begin{aligned} X(i) &= \text{set of distinct } \mathcal{X} \text{ blocks read at time } i, \quad (43)\\ \hat{X}(i) &= \text{set of distinct } \mathcal{X} \text{ blocks written at time } i,\\ Y(i) &= \text{set of distinct } \mathcal{Y} \text{ blocks written at time } i,\\ X_{us} &= \text{set of user data at time 1,}\\ X_r &= \text{set of recovered data at time } \frac{c}{r}. \end{aligned}$$

For any k' and k'', we define the sets

$$\begin{aligned} &\langle X(i)\rangle_{i=k'}^{k''} = \quad (44)\\ &\qquad \{X(i) \mid i = k', k'+1, \ldots, k''\},\\ &\left\langle X(i), \hat{X}(i), Y(i)\right\rangle_{i=k'}^{k''} =\\ &\qquad \left\{X(i), \hat{X}(i), Y(i) \mid i = k', k'+1, \ldots, k''\right\}. \end{aligned}$$

By manipulating the definition of conditional entropy, and using the data-processing inequality [31], it is easy to show that the information pertinent for recovery written at step j, is contained in the information read and written up to j, and the information read in j. *I.e.*, for any[1] j

$$\begin{aligned} &H\left(X_{us} \middle| \langle X(i)\rangle_{i=\frac{c}{r}-j}^{\frac{c}{r}}, \left\langle X(i), \hat{X}(i), Y(i)\right\rangle_{i=1}^{\frac{c}{r}-j-1}\right) = \quad (45)\\ &\quad H\left(X_{us} \middle| \langle X(i)\rangle_{i=\frac{c}{r}-j-1}^{\frac{c}{r}}, \left\langle X(i), \hat{X}(i), Y(i)\right\rangle_{i=1}^{\frac{c}{r}-j-2}\right). \end{aligned}$$

Let $H(\cdot|\cdot, c'_S)$ to denote the conditional entropy when a total of c'_S distinct blocks were read from $\mathcal{X}$ during the epoch. Then for any c'_S,

$$\begin{aligned} &H(X_{us}|X_r, c'_S) \quad (46)\\ &\overset{(a)}{\leq} H\left(X_{us} \middle| \left\langle X(i), \hat{X}(i), Y(i)\right\rangle_{i=1}^{\frac{c}{r}}, c'_S\right)\\ &\overset{(b)}{=} H\left(X_{us} \middle| X(\frac{c}{r}), \left\langle X(i), \hat{X}(i), Y(i)\right\rangle_{i=1}^{\frac{c}{r}-1}, c'_S\right)\\ &\overset{(c)}{=} H\left(X_{us} \middle| \langle X(i)\rangle_{i=\frac{c}{r}-1}^{\frac{c}{r}}, \left\langle X(i), \hat{X}(i), Y(i)\right\rangle_{i=1}^{\frac{c}{r}-2}, c'_S\right)\\ &\vdots\\ &\overset{(d)}{=} H\left(X_{us} \middle| \langle X(i)\rangle_{i=\frac{c}{r}-j}^{\frac{c}{r}}, \left\langle X(i), \hat{X}(i), Y(i)\right\rangle_{i=1}^{\frac{c}{r}-j-1}, c'_S\right)\\ &\vdots\\ &\overset{(e)}{=} H\left(X_{us} \middle| \bigcup_{i=1}^{\frac{c}{r}} X(i), c'_S\right)\\ &\overset{(f)}{\leq} c'_s \hat{b}(S). \end{aligned}$$

[1]We use $H(\cdot)$ to denote the binary-entropy function

In the above, inequality (a) follows from the fact that $X_{us} \to \left\langle X(i), \hat{X}(i), Y(i)\right\rangle_{i=1}^{\frac{c}{r}} \to X_r$ is a Markov chain, and so the data processing inequality applies, and (b) through (e) follow from further manipulating the conditional entropies and applying (45). Inequality (f) follows from Shannon's channel-capacity theorem. A user sending c'_s consecutive blocks over a communication channel suffering from the given block-erasure and block-substitution probabilities, can send only $c'_s\hat{b}(S)$ information. Now, if $c'_S \gneq \hat{c}_S$, then with probability $\frac{1}{\nu}$, data is lost. It follows that the expected number of epochs until data loss is ν.

The proof of Theorem 5 gives the following interpretation. The effective volume of the system in an epoch, is determined by the number of blocks, which it is "reasonable to assume" will be encountered when all the system devices are scanned in parallel. To achieve the system capacity, each block read should effectively contain user data. For the system to remain reliable, this should be the case for the next epoch as well. It follows that $\langle Y(i)\rangle_{i=k'}^{k''}$ is the only set of blocks which need be written during an epoch. To achieve capacity, all realistic shapes with the given area should suffice to recover the user data. For a large number of devices, this is not difficult. We return to this point in Subsection **??**.

Assume that each device capacity, c, grows very large, while r remains relatively the same. Using the idea of effective volume, it is easy to show the limit of the effect on the total amount of data which can be stored reliably. It follows that for scalable systems, a large n must be considered.

Theorem 6: for a system S with n devices, the effective capacity is bounded by

$$\begin{aligned} &c^R_{\max}(S) \leq \quad (47)\\ &\quad 2nr\left(1 - \left(\frac{1}{2}\right)^{\frac{\frac{c}{r}}{\frac{\ln(2)}{\lambda}}}\right)(1+o(1))\hat{b}(S)\\ &\quad \overset{c\to\infty}{\longrightarrow} 2nr\hat{b}(S)(1+o(1)). \end{aligned}$$

By the properties of the exponential distribution,

$$R_C\left(t + \frac{\ln(2)}{\lambda} \mid t\right) \leq \frac{1}{2}. \quad (48)$$

By the law of large number, with probability one,

$$\begin{aligned} &\left|\left\{C \in S \mid t_C \geq i\frac{\ln(2)}{\lambda}\right\}\right| \leq \quad (49)\\ &\quad \frac{1}{2^{i-1}} n(1+o(1)), \end{aligned}$$

and so, with probability one,

$$\langle X(i)\rangle_{i=1}^{\frac{c}{r}} \leq \tag{50}$$
$$r \sum_{i=0}^{\frac{\frac{c}{r}}{\frac{\ln(2)}{\lambda}}} \left|\left\{C \in S \mid t_C \geq i\frac{\ln(2)}{\lambda}\right\}\right|$$
$$2nr\left(1-\left(\frac{1}{2}\right)^{\frac{\frac{c}{r}}{\frac{\ln(2)}{\lambda}}}\right)(1+o(1)).$$

The above idea closely resembles the analysis of the *skip-list* data structure [32]. Although this data structure can theoretically have an infinite amount of elements in its levels (corresponding here to an infinite c), the total number of levels is bounded, with high probability, by the number of items in its lowest level (corresponding here to the bounded effective volume).

The same result can be shown by considering the average number of recovery operations. For a fixed number of devices, an increase in c causes an increase in recovery activity. When the average recovery activity is higher than afforded by the devices, data is lost.

Theorem 7: Let the n devices contain ρnc blocks of user data. The average number of recovery IOs per storage device per time unit is at least

$$\rho c(1-o(1)). \tag{51}$$

Fix $\rho' \lneq \rho$. Let $t' = \frac{(1-\rho')\cdot n}{\lambda}$. By the Poisson approximation, at time t', the number of original devices which have not failed is $\rho' n(1+o(1))$. As shown in Theorem 5, the effective volume of the original devices at t' is only

$$\langle X(i)\rangle_{i=t'}^{t'+1+\frac{c}{r}} \leq \rho' nc(1+o(1)). \tag{52}$$

It follows that up to time t', the average number of recovery IOs per original storage device per time unit is at least

$$\frac{(\rho-\rho')cn(1-o(1))}{n} \xrightarrow{\rho'\to 0} \rho c(1-o(1)). \tag{53}$$

The two previous points show that as c_{US} grows large enough, so must n. For performance reasons, the blocks in a large storage system are usually not coded en-mass in a single coding group. Rather, the blocks are partitioned into smaller groups, and the blocks of each group are protected by some code (*e.g.*, in a mirroring scheme, the blocks are partitioned into groups of size 2). We now show that as n increases, so must the average redundancy in the coding groups.

Theorem 8: Let S be a system in which the data is protected in the following manner. The data is divided into (at most $\frac{cn}{k}$) groups, each of size k and resiliency k'. Let the fraction of blocks used in S be β. Assume the failure of a device is detected after t_d time units. Then the system MTTDL is bounded by

$$t_d\left(1+\frac{1}{1-p_r^{\frac{\beta n-\frac{k}{c-1}}{k(k-1)}}}\right) \xrightarrow{n\to\infty} t_d, \tag{54}$$

where

$$p_r = 1-\sum_{i=0}^{k'}\binom{k}{i}\left(1-e^{-\lambda t_d}\right)^i\left(e^{-\lambda t_d}\right)^{k-i}. \tag{55}$$

The proof follows by considering a system S' of devices, composed of disjoint sub-groups, each of size k and resiliency k'. The time to data loss in S' is a Bernoulli process, parameterized by the number of sub-groups. It is possible to show that any system using coding groups of size k and resiliency k', inherently contains a subsystem similar to S'. The manner in which data groups are laid out, can at most minimize the number of disjoint sub-groups which need be considered, but cannot eradicate them altogether.

It was observed in [33] that groups with resiliency k' require k' updates per write; if this were not the case, a write, followed by the $k'' \lneq k'$ updates, would be susceptible to $k''+1$ failures. Combining this with Theorem 8, we obtain the following.

Theorem 9: For any fixed k', ρ and large enough n, the number of IO updates per modification is larger than k'.

Consider any coding scheme which is used for all data in the system. The above points show that as c_{US} grows large, the performance viewed by users must degrade. This can be avoided by presenting the users with a relatively-small group of devices containing their active data, and using high-performance (and therefore high redundancy) codes to protect these data. Since active data changes with time, the rest of the system must also display reasonable performance. This leads to the necessity of hierarchical reliability.

IV. Conclusions and Future Work

V. Acknowledgements

Thanks to Dana Ron and David Burshtein of the Dept. of EE-Systems at Tel-Aviv University, and Alain Azagury, Michael Factor, Kalman Meth, Julian Satran, and Dafna Sheinwald of IBM Haifa Research Laboratories, for useful discussions.

References

[1] G. A. Gibson, *Redundant Disk Arrays: Reliable Parallel Secondary Storage*, ser. ACM Distinguished Dissertations. Cambridge, MA: MIT Press, 1992.

[2] J. Wilkes, R. Golding, C. Staelin, and T. Sullivan, "The HP AutoRAID hierarchical storage system," *ACM Transactions on Computer Systems*, vol. 14, no. 1, pp. 108–136, Feb. 1996.

[3] A. Azagury, V. Dreizin, M. Factor, E. Henis, D. Naor, N. Rinetzky, O. Rodeh, J. Satran, A. Tavory, and L. Yerushalmi, "Towards an object store," in *In Proceedings of the 20th IEEE / 11th NASA Goddard Conference on Mass Storage Systems and Technologies*, 2003, pp. 165–176.

[4] A. Albanese, J. Blomer, J. Edmonds, M. Luby, and M. Sudan, "Priority encoding transmission," in *Proceedings: 35th Annual Symposium on Foundations of Computer Science, Santa Fe, New Mexico*. IEEE Computer Society Press, Nov. 1994, pp. 604–612.

[5] J. A. Katzman, "System architecture for nonstop computing," in *14th IEEE Computer Society International Conference (COMPCON)*, Mar. 1977, pp. 77–80.

[6] D. A. Patterson, G. Gibson, and R. H. Katz, "A case for redundant arrays of inexpensive disks (raid)," in *Proceedings of the ACM Conference on Management of Data (SIGMOD)*, June 1988, pp. 109–116.

[7] Q. M. Malluhi and W. E. Johnston, "Coding for high availability of a distributed-parallel storage system," *IEEE Trans. Parallel Distrib. Syst.*, vol. 9, no. 12, pp. 1237–1252, Dec. 1998.

[8] Q. Xin, E. Miller, D. Long, S. Brandt, T. Schwarz, and W. Litwin, "Reliability mechanisms for very large storage systems," in *In Proceedings of the 20th IEEE / 11th NASA Goddard Conference on Mass Storage Systems and Technologies*, Apr. 2003, pp. 146–156.

[9] M. Holland, G. A. Gibson, and D. P. Siewiorek, "Architectures and algorithms for on-line failure recovery in redundant disk arrays," *Journal of Distributed and Parallel Databases*, vol. 2, no. 3, pp. 295–335, July 1994.

[10] J. Menon and D. Mattson, "Distributed sparing in disk arrays," in *Proceedings of the COMPCOM Conference*, Feb. 1992, pp. 410–421.

[11] J. M. Menon and R. L. Mattson, "Comparison of sparing alternatives for disk arrays," in *Proceedings the 19th Annual International Symposium on Computer Architecture,ACM SIGARCH*, Gold Coast, Australia, May 1992, pp. 318–329.

[12] A. Thomasian and J. Menon, "RAID5 performance with distributed sparing," *IEEE Transactions on Parallel and Distributed Systems*, vol. 8, no. 6, pp. 640–657, June 1997.

[13] X. Wu, J. Li, and H. Kameda, "Reliable analysis of disk array organizations by considering uncorrectable bit errors," in *Proceedings of The 16th Symposium on Reliable Distributed Systems (SRDS '97)*. Washington - Brussels - Tokyo: IEEE, Oct. 1997, pp. 2–9.

[14] J. Chandy and A. L. N. Reddy, "Failure evaluation of disk array organizations," in *Proceedings of the 13th International Conference on Distributed Computing Systems*. Pittsburgh, PA: IEEE Computer Society Press, May 1993, pp. 319–327.

[15] R. G. S. Kirkpatrick, W. Wilcke and H. Huels, "Percolation in dense storage arrays," in *Physica A*, vol. 314, Nov. 2002, pp. 220–229.

[16] F. Chang, M. Ji, S.-T. Leung, J. MacCormick, S. Perl, and L. Zhang, "Myriad: Cost-effective disaster tolerance," in *Proceedings of the FAST '02 Conference on File and Storage Technologies (FAST-02)*. Berkeley, CA: USENIX Association, Jan. 28–30 2002, pp. 103–116.

[17] J. Wilkes, R. Golding, C. Staelin, and T. Sullivan, "The HP AutoRAID hierarchical storage system," in *High Performance Mass Storage and Parallel I/O: Technologies and Applications*, H. Jin, T. Cortes, and R. Buyya, Eds. New York, NY: IEEE Computer Society Press and Wiley, 2001, pp. 90–106.

[18] J. Kubiatowicz, D. Bindel, P. Eaton, Y. Chen, D. Geels, R. Gummadi, S. Rhea, W. Weimer, C. Wells, H. Weatherspoon, and B. Zhao, "OceanStore: An architecture for global-scale persistent storage," *ACM SIGPLAN Notices*, vol. 35, no. 11, pp. 190–201, Nov. 2000.

[19] D. Bindel, Y. Chen, P. Eaton, D. Geels, R. Gummadi, S. Rhea, H. Weatherspoon, W. Weimer, C. Wells, B. Zhao, and J. Kubiatowicz, "Oceanstore: An extremely wide-area storage system," in *Proceedings of the Nine International Symposium on Architectural Support for Programming Languages and Operating Systems (ASPLOS IX)*, Nov. 2000.

[20] H. Weatherspoon, M. Delco, and S. Zhuang, "Typhoon: An archival system for tolerating high degrees of file server failure," 1999, available through http://www.cs.berkeley.edu/~hweather/Typhoon/TyphoonReport.html.

[21] D. D. Sleator and R. E. Tarjan, "Amortized efficiency of list update and paging rules," *Comm. ACM*, vol. 28, no. 2, pp. 202–208, Feb. 1985.

[22] A. Fiat, R. M. Karp, M. Luby, L. A. McGeoch, D. D. Sleator, and N. E. Young, "Competitive paging algorithms," *Journal of Algorithms*, vol. 12, no. 4, pp. 685–699, Dec. 1991.

[23] Y. Neal, "On-line file caching," in *Proceedings of the 9th Annual ACM-SIAM Symposium on Discrete Algorithms*, Jan. 28–30 1998, pp. 82–86.

[24] S. Irani, "Page replacement with multi-size pages and applications to web caching," *Algorithmica*, vol. 33, no. 3, pp. 384–409, July 2002.

[25] M. Chrobak, H. Karloff, T. Payne, and S. Vishwanathan, "New results on server problems," *SIAM Journal on Discrete Mathematics*, vol. 4, no. 2, pp. 172–181, May 1991.

[26] N. E. Young, "The k-server dual and loose competitiveness for paging," *Algorithmica*, vol. 11, no. 6, pp. 525–541, june 1994.

[27] R. Motwani and P. Raghavan, *Randomized Algorithms*. Cambridge University Press.

[28] L. A. McGeoch and D. D. Sleator, "A strongly competitive randomized paging algorithms," *Algorithmica*, vol. 6, pp. 816–825, 1991.

[29] S. Albers, S. Arora, and S. Khana, "Page replacmenet for general caching problems," in *Proceedings of the 10th Annual ACM-SIAM Symposium on Discrete Algorithms*, 1999, pp. 31–40.

[30] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein, *Introduction to Algorithms*, 2nd ed. MIT Press, 2001.

[31] T. M. Cover and J. A. Thomas, *Elements of Information Theory*, ser. Wiley Series in Telecommunications. New York, NY, USA: John Wiley & Sons, 1991.

[32] K. Mulmuley, *Computational Geometry, An Introduction Through Randomized Algorithms*. Prentice-Hall, Inc., New-Jersey.

[33] Y. Chee, Colbourn, and Ling, "Asymptotically optimal erasure-resilient codes for large disk arrays," *DAMATH: Discrete Applied Mathematics and Combinatorial Operations Research and Computer Science*, vol. 102, no. 1-2, pp. 3–36, 2000.