Variations on the Theme of Caching

by

Cristian Gaspar

A thesis

presented to the University of Waterloo

in fulfilment of the

thesis requirement for the degree of

Master of Mathematics

in

Computer Science

Waterloo, Ontario, Canada, 2005

©Cristian Gaspar 2005

Author's Declaration for Electronic Submission of a Thesis

I hereby declare that I am the sole author of this thesis. This is a true copy of the thesis, including any required final revisions, as accepted by my examiners.

I understand that my thesis may be made electronically available to the public.

Abstract

This thesis is concerned with caching algorithms. We investigate three variations of the caching problem: web caching in the Torng framework, relative competitiveness and caching with request reordering.

In the first variation we define different cost models involving page sizes and page costs. We also present the Torng cost framework introduced by Torng in [29]. We analyze the competitive ratio of online deterministic marking algorithms in the Bit cost model combined with the Torng framework. We show that given some specific restrictions on the set of possible request sequences, any marking algorithm is 2-competitive.

The second variation consists in using the relative competitiveness ratio on an access graph as a complexity measure. We use the concept of access graphs introduced by Borodin et al. [11] to define our own concept of relative competitive ratio. We demonstrate results regarding the relative competitiveness of two cache eviction policies in both the basic and the Torng framework combined with the Classical cost model.

The third variation is caching with request reordering. Two reordering models are defined. We prove some results about the value of a move and number of orderings, then demonstrate results about the approximation factor and competitive ratio of offline and online reordering schemes, respectively.

Acknowledgements

I am very grateful to my supervisor, Prof. Ian Munro, for his time, support, and help, for always being available to discuss research ideas and results, for his insight and guidance that made this thesis possible. Thank you, Ian!

I am also grateful to Spyros Angelopoulos for all his comments and insights, for his patience in discussing my research results, and for his suggestions on how to improve them. $Eu\chi\alpha\rho\acute{\imath}\sigma\tau\acute{\omega}$, Spyros!

I also thank my readers, Prof. Alex López-Ortiz and Prof. Jeremy Barbay, for their comments and suggestions on how to make this thesis more readable. Thank you for your help!

My graduate study and research at UW was supported by a NSERC Scholarship and a President's Graduate Scholarship for which I would like to thank the Natural Sciences and Engineering Research Council of Canada and the University of Waterloo.

I thank God and my family: Mihail, Mariana, Anca, and Diana for helping and supporting me for the duration of my Masters program. Extra special thanks go to my best friends, Melodie and Cristian, who encouraged me and boosted my morale during the writing of this thesis.

Contents

1 Introduction **1**

1.1 Thesis Overview 3

1.2 Terminology 4

1.3 Classical Caching 7

1.3.1 Deterministic Online Eviction Policies 8

1.3.2 Randomized Online Eviction Policies 9

2 Web Caching and the Torng Framework **11**

2.1 Introduction 11

2.2 Torng Framework 13

2.2.1 Introduction 13

2.2.2 Previous Work 15

2.2.3 New Results for the BIT Model with Non-Optional Caching 16

3 Relative competitiveness **22**

3.1 Introduction 22

3.2 New Results for the CLASSICAL Model with Non-Optional Caching 24

4 Demand Caching with Request Reordering **37**

4.1 Introduction 37

4.2 Reordering Models 42

4.3 Previous Work in the FEDER Model 44

4.4 Overlapping Moves and Ignoring Backward Moves 46

4.5 Value of a Move and a Property of Effective Moves 51

4.6 Number of Orderings 59

4.7 BIDIRECTIONAL Model Results 61

4.8 UNIDIRECTIONAL Model Results 65

4.8.1 Introduction 65

4.8.2 A Reordering Scheme for the General Case 65

4.8.3 Optimal Ordering for a Special Case 73

4.8.4 Online Reordering Schemes 94

4.8.5 Conservative Online Reordering Schemes 99

4.8.6 Online Reordering Schemes with Look-Ahead 100

5 Conclusions **104**

5.1 Summary of Results 104

5.2 Open Problems 106

A Appendix **107**

A.1 Miscellaneous Results 107

A.2 Notation 112

A.3 Definitions 115

Bibliography **122**

List of Figures

3.1 Graphs G and G^* (the cycle C_{max} is indicated by bold edges) 26

3.2 Graph G constructed for $c(G, EDO, LRU)$. 32

4.1 A forward move m. 38

4.2 Simulation of a backward move by forward moves. 48

4.3 A typical partition when $r = 3$. $S_1 = \{m_1, m_2, m_3\}$. $S_2 = \{m_1', m_2', m_3'\}$. . . . 66

4.4 *LFD* serves σ_1. Faults are indicated by dashes. Arrows indicate the moves that *OPT* performs. Initial *LFD* cache configuration: $\{1, 2, \ldots k\}$. Final *LFD* cache configuration: $\{1, 2, \ldots k, k+1\} \setminus \{x\}$ where x is one of 1, 2, ... k. . . 68

4.5 *LFD* serves σ_1'. Faults are indicated by dashes. Initial *LFD* cache configuration: $\{1, 2, \ldots k\}$. Final *LFD* cache configuration: $\{2, \ldots k, k+1\}$. 69

4.6 *LFD* serves σ_2. Faults are indicated by dashes. Arrows indicate the moves that *OPT* performs. Initial *LFD* cache configuration: $\{2, \ldots k, k+1\}$. Final *LFD* cache configuration: $\{2, \ldots k, k+1, k+2\} \setminus \{x\}$ where x is one of 2, ... k, k+1. 70

4.7 *LFD* serves σ_2. Faults are indicated by dashes. Initial *LFD* cache configuration: $\{2 \ldots k, k+1\}$. Final *LFD* cache configuration: $\{2, \ldots k, k+2\}$. 71

4.8 *LFD* serves σ_3. Faults are indicated by dashes. Arrows indicate the moves that *OPT* performs. Initial *LFD* cache configuration: $\{2, \ldots k, k+2\}$. Final *LFD* cache configuration: $\{1, 2, \ldots k, k+2\} \setminus \{x\}$ where x is one of 2, 3, ... k, k+2. 71

4.9 *LFD* serves σ'_3. Faults are indicated by dashes. Initial *LFD* cache configuration: $\{2, \ldots k, k+2\}$. Final *LFD* cache configuration: $\{1, 2, \ldots k\}$. 72

4.10 Type ① moves: m_1 and m'_1. Type ② move: m_2. Type ③ move: m_3 74

4.11 Type I move: $m_1 = (5, 3)$. Type II moves: $m_2 = (5, 3)$, $m'_2 = (5, 3)$. Type III moves: $m_3 = (4, 1)$, $m'_3 = (4, 2)$ 78

4.12 Cases (1.5) and (1.6) in the $C_1[j]$ formula 82

4.13 Cases (2.4) and (2.5) in the $C_2[j]$ formula 83

4.14 Cases (3.8), (3.9), and (3.10) in the $C_3[j]$ formula 85

Chapter 1

Introduction

Caching is ubiquitous in the world of computer hardware and software, it is crucial to efficient performance in many aspects of computation including operating systems, databases, and search engines. In this thesis we focus our attention on the mathematical modelling of the caching problem that is defined as follows, simplifying a multi-level memory hierarchy to two levels. We are given access to two types of memory: slow but large memory (secondary storage) and fast but small memory (the cache). The cache can store a fixed number of memory units which are a subset of the secondary storage memory units. We regard each memory unit as a **page** and will use this term through the rest of this thesis. The access time ratio between the secondary storage and the cache can be as large as five orders of magnitude. Hennessy [17] reports a ratio between $7,000$ and $150,000$. It is easy to see the improvement if a page is read from cache instead of secondary storage. The processing of a sequence of page requests is done in the following manner: if the page is already in cache,

no extra work needs to be done. We call this case a **hit**. If the page is not in cache, then it is brought in. This case is a **miss**, also referred to as a **fault** and the page on which the fault occurs is called a **faulting page**. Processing a request has a different cost depending on whether the request is a fault or a hit. The total processing cost is the sum of all fault costs and hit costs. Note that in practice the hit cost is very small in comparison to the fault cost, this will be reflected in our mathematical model. In the case of a fault, a number of pages might be evicted from cache so that the faulting page can be brought into cache. We call this event an **eviction**. Given a fixed request sequence, the factor that determines its total processing cost is the decision of which pages to evict (the **cache eviction policy**). For the rest of this thesis the term **eviction policy** refers to a cache eviction policy. The goal of a caching problem is to minimize the total cost incurred while processing the given request sequence.

There is a multitude of variations of caching models given all possibilities regarding hit costs, fault costs, page sizes, reordering of requests, and complexity measures. This thesis focuses on two aspects that have not previously been studied in literature, namely **request reordering** [3, 14] in Chapter 4 (see Definition 4.8) and **relative competitiveness** in Chapter 3 (see Definitions 3.2 and 3.3). We also discuss a variation on the well studied **Torng framework** [29] in Chapter 2 (see Definition 2.2).

1.1 Thesis Overview

This chapter continues with the definitions, terminology and other background to our results. It also presents the most commonly studied problem: **classical caching** (Definition 1.7). Finally we provide a description of all three variations analyzed in this thesis: web caching in the Torng framework, relative competitiveness and caching with request reordering.

Chapter 2 deals with the first major variation: a combination of the Torng framework and web caching variants. We define various cost models involving page sizes and page costs. We present the Torng framework which includes costs for hits as well in the total service cost as opposed to the **basic** framework in which only fault costs are counted in the total cost (Definition 2.2). We survey some of the results of Torng [29] and provide new results on the Torng framework in the Bit model where fault costs are proportional to page sizes.

In Chapter 3 we analyze the second variation: using the relative competitiveness ratio on an access graph as a complexity measure. We explain the concept of access graphs introduced by Borodin [11] and use it to define our own concept of relative competitive ratio. We prove results regarding the relative competitiveness of two cache eviction policies, *LRU* and *EDO*, in both the basic and the Torng framework for the Classical cost model.

In Chapter 4 we focus on the third variation: caching with request reordering. Two reordering models are defined. We survey the relevant results of Feder et al. [14] and Albers [3]. We first prove some important results regarding the value of a move and number of orderings. The main sections present results for the two reordering models regarding the competitiveness of offline and online caching schemes.

In Chapter 5 we present a summary of our results and a list of open problems for future research. The Appendix contains a few stand-alone results, followed by a list of terms and notation that are used in this thesis. We attempted to reference every definition and notation we have used, but whenever the reader is still in doubt we recommend checking the Appendix.

Note that in all chapters, the complexity measures being used are the approximation factor and the competitive ratio except for Chapter 3 that uses the relative competitive ratio. Results in Chapter 2 involve the Torng framework, results in Chapter 3 involve both the basic and Torng framework, and results in other chapters involve only the basic framework.

1.2 Terminology

In order to understand the results and their proofs in this thesis we now define the basic caching models and relevant terms: offline/online algorithms, c-competitiveness, runs, phases, etc.

We first provide some basic classifications of the types of caching.

Definition 1.1 *In* **demand caching** *a page can be brought into cache if and only if it is the currently faulting page. We call such an operation a* **demand admission**. *An eviction policy using this model is called a* **demand eviction policy**.

In **non-demand caching** *any page can be brought into cache at any time and the additional processing cost is equal to the fault cost of the page that was brought into cache. We call such an operation a* **non-demand admission**.

Note that in Observation 4.1 we show that any eviction policy can be simulated by a demand eviction policy. Consequently, all results in this thesis implicitly use the demand caching model.

Definition 1.2 *In* **optional caching** *a faulting page may or may not be brought into cache. In* **non-optional caching** *every faulting page must be brought into cache.*

In most results presented in this thesis the type of caching (optional or non-optional) is not mentioned. Whenever this occurs the implicit type is non-optional caching. Those results using optional caching will mention this explicitly in the statement of the result.

We continue by presenting terms that are relevant to the caching problem. An eviction policy is said to be **online** if it does not have any information about future requests. On the other hand an **offline** eviction policy is one that knows the complete request sequence in advance.

We present the concept of competitive ratio of an online eviction policy. Let $cost(\mathcal{E}(\sigma))$ denote the cost of processing σ using some eviction policy $\mathcal{E}$.

Definition 1.3 *Let σ be a request sequence, $\mathcal{E}$ be a deterministic online eviction policy and OPT be an optimal offline eviction policy. We say that $\mathcal{E}$ is* **c-competitive** *if*

$$\exists \text{ constant } b \text{ such that } \forall \sigma, cost(\mathcal{E}(\sigma)) \leq c \cdot cost(OPT(\sigma)) + b$$

The **deterministic competitive ratio** *of $\mathcal{E}$, denoted by $c_{\mathcal{E}}$, is the smallest value of c such that $\mathcal{E}$ is c-competitive.*

Definition 1.4 *Let σ be a request sequence, $\mathcal{E}$ be a randomized online eviction policy and OPT be an optimal offline eviction policy. Let $E(cost(\mathcal{E}(\sigma)))$ denote the expected cost of processing σ using eviction policy $\mathcal{E}$. The expectation is taken over the random choices made by $\mathcal{E}$. We say that $\mathcal{E}$ is* **c-competitive** *if*

$$\exists \text{ constant } b \text{ such that } \forall \sigma, E(cost(\mathcal{E}(\sigma))) \leq c \cdot cost(OPT(\sigma)) + b$$

The **randomized competitive ratio** *of $\mathcal{E}$, denoted by $c_{\mathcal{E}}$, is the smallest value of c such that $\mathcal{E}$ is c-competitive.*

We will use the term **competitive ratio** when it is clear from the context whether we are referring to the deterministic or to the randomized case.

We introduce the following notation: given a request sequence σ let $\boldsymbol{\sigma(i:j)}$ denote the subsequence of requests $\sigma(i)\ \sigma(i+1)\ \ldots\ \sigma(j)$ $(i \leq j)$.

Definition 1.5 *Let σ be a request sequence. We call subsequence $\sigma(i:j)$ a* **run** *if and only if it is a maximal subsequence of requests for a particular page; more precisely, for some page x*

(i) $\sigma(i:j) = x^{j-i+1}$ (i.e.: $j-i+1$ requests for x), and

(ii) $\sigma(i-1) \neq \sigma(i)$ and $\sigma(j) \neq \sigma(j+1)$

n will always denote the length of the request sequence ($n = |\sigma|$) and k will denote the cache size (in bytes). u will be the number of distinct pages in the universe of requests. Note

that when we use the term **subsequence** of σ we always mean a set of consecutive requests in σ.

Definition 1.6 *Let σ be a request sequence. The* **size** *of subsequence $\sigma(i:j)$ is the sum (in bytes) of sizes of all* **distinct** *pages requested in B. Given index $i > 0$, let $j \geq i$ be the maximum index such that the size of $\sigma(i:j)$ is less than or equal to k. We say that $\sigma(i:j)$ is the* **chunk** *starting at index i.*

We partition a request sequence σ into adjacent chunks as follows: let the first chunk in σ start at $\sigma(1)$ and j_1 be its end index. The second chunk starts at $\sigma(j_1+1)$; let j_2 be its end index. The third chunk starts at $\sigma(j_2+1)$; repeat until the last chunk is selected as the one whose end index is n. Note that, in this thesis, the term "chunk" refers to one of the chunks implied by the partition above.

If page sizes are not uniform then it is possible that any of the chunks to have total size strictly less than k. This does not affect the validity of our results in the non-classical cost models (see Definition 2.1). Even if all page size are equal to 1, it is possible that the last chunk has size strictly less than k.

1.3 Classical Caching

We now present the caching variation that has been most studied in previous literature. Recall the concept of non-optional caching introduced in Definition 1.2.

Definition 1.7 *The* **classical caching problem** *is non-optional caching with page size and page fault costs equal to* 1 *and hit costs equal to* 0*.*

In the classical caching problem, the total cost of processing a request sequence σ is exactly the number of faults that are encountered while serving σ and using a particular eviction policy. Note that Definition 1.7 implies that the cache can hold at most k pages at any time.

The classical caching problem has been extensively studied in [1, 8, 11, 16, 22, 25, 27, 28, 33]. We will present a short survey of the main results to introduce the reader to the world of theoretical offline/online caching results.

Belady [8] introduced an optimal offline eviction policy running in polynomial time called *Longest Forward Distance*(*LFD*). *LFD* always evicts the page in cache whose next request occurs furthest in the future.

1.3.1 Deterministic Online Eviction Policies

Sleator and Tarjan in [28] introduced a slight modification to the classical caching problem by having different cache sizes for the online and optimal offline eviction policies (k and h for the online and offline cache sizes respectively $h \leq k$). They proved in [28] that any deterministic online eviction policy has competitive ratio at least $\frac{k}{h-k+1}$. By taking $h = k$ we get the next important result.

Theorem 1.1 ([28]) *In the classical caching problem any deterministic online eviction policy has competitive ratio at least* k*.*

Two other well studied eviction policies are *Least Recently Used* (*LRU*) and *First In First Out* (*FIFO*). As their title suggest *LRU* evicts the page in cache whose last access time is smallest while *FIFO* evicts the page whose cache entry time is smallest.

Sleator and Tarjan [28] prove a matching upper bound on the competitive ratio of *LRU* and *FIFO*, so taking $h = k$ it follows that both eviction policies have competitive ratio at most k. By Theorem 1.1 we get the following result.

Theorem 1.2 ([28]) *In the classical caching problem LRU and FIFO attain a competitive ratio of k and are optimal among deterministic online eviction policies.*

1.3.2 Randomized Online Eviction Policies

Before we present the next lower bound result due to Fiat et al. [16], we need the following notation. We denote by H_k the ***k*-th harmonic number**: $H_k = \sum_{i=1}^{k} \frac{1}{i}$. For any $k \geq 1$, $\ln k < H_k \leq \ln k + 1$.

Fiat, Karp, Luby, McGeoch, Sleator and Tarjan give the following lower bound on randomized online eviction policies, for details we direct the reader to [16].

Theorem 1.3 ([16]) *In the classical caching problem every randomized online eviction policy has competitive ratio at least H_k.*

In [16] they also present the following upper bound result which shows that the *MARKING* eviction policy is optimal, up to a constant factor. *MARKING* uses the following phase concept: in the beginning of a phase all pages in cache are unmarked. When a page is

accessed for the first time in the current phase it is marked. If an eviction is necessary the eviction policy chooses uniformly at random one of the unmarked pages. If all pages are marked, then this is the end of a phase, all pages in cache are unmarked and the process resumes.

Theorem 1.4 ([16]) *In the classical caching problem the $MARKING$ eviction policy is $2H_k$-competitive.*

Achlioptas et al. [2] present a randomized online eviction policy *EQUITABLE* that is optimal. For details see [2].

Theorem 1.5 ([2]) *In the classical caching problem the $EQUITABLE$ eviction policy is H_k-competitive.*

Chapter 2

Web Caching and the Torng Framework

2.1 Introduction

One of the most active areas of caching research is web caching [4, 7, 12, 18, 19, 20, 31, 32]. Recall that, in classical caching, all pages have size equal to 1. In web caching, pages have different sizes; usually the range of page sizes varies significantly, from a few kilobytes for a text file to a few megabytes for an mp3 file or even to a few hundreds of megabytes for a video file.

When dealing with caching problems in the basic framework, two major parameters must be considered: page sizes and page fault costs (recall hit costs are 0 in the basic framework). Based on these two parameters, previous literature introduced the following seven models. In

the first model, the CLASSICAL model, fault costs and page sizes are equal to 1. This model has been extensively researched, for relevant papers see [1, 8, 11, 16, 22, 25, 27, 28, 33].

In the second model, fault costs vary and page sizes are equal. It is usually referred to as the WEIGHTED model. In the third model, fault costs are equal and page sizes vary. This model is called the FAULT model. Clearly, one can normalize the fault costs under the FAULT model and assume that they are equal to 1. Similarly, page sizes in the WEIGHTED model can be normalized so that they are equal to 1. In the fourth model (the GENERALIZED model), fault costs vary and page sizes vary as well. In order to restrict the last model a bit more, the BIT model was defined, where page sizes and faults costs vary and the fault cost is proportional to the size of the page the fault occurs on.

We now present two other hybrid models that were suggested by López-Ortiz in [24] and introduced by Au et al. in [6]. In the DUAL-SIZE model, the page sizes are equal to 1 or to a constant, L ($L > 1$). The fault costs under this model are based either on the FAULT model or on the BIT model. Another hybrid model is the REAL model in which page sizes vary and the fault cost is a linear function of the page size. To summarize the models presented above, we provide the next definition that was introduced in [6].

Definition 2.1 ([6]) (Caching Models)

CLASSICAL - *Uniform page size and uniform cost of fault).*

WEIGHTED - *Uniform page size and variable cost of fault.*

FAULT - *Variable page size and uniform cost of fault. [19]*

GENERALIZED - *Variable page size and variable cost of fault.*

BIT - *Variable page size and cost of fault equals p times the page size (p is a constant). [19]*

DUAL-SIZE - *Two page sizes and variable cost of fault. [24]*

REAL - *Variable page size and cost of a fault is* $w_1 \cdot$ pageSize $+w_2$

(w_1 and w_2 are positive constants). [24]

2.2 Torng Framework

2.2.1 Introduction

We now describe two cost frameworks that are to be used in combination with one of the cost models presented in Definition 2.1. Each cost framework will define specific hit costs and fault costs.

Recall, that in the classical caching problem, hit costs are 0 and fault costs are 1. We call this the **basic cost framework**. The assumption behind this cost framework is that cache memory is so fast compared to secondary storage, that reading a page from cache is "instantaneous". However this simplification that hit costs are 0 signifies that fault costs are basically infinite compared to the hit cost. However in practice this is not true. For example, the fault cost - hit cost ratio for paging is between $7,000$ and $150,000$ (see [17]).

Having observed this discrepancy, Torng introduced a new framework (**Torng framework**) in [29] for the Classical model. He considers hit costs equal to 1 and fault costs

equal to $(p+1)$: p for bringing the page into cache and 1 to access it. Since we will combine the Torng framework with the BIT model, in our results p denotes the fault penalty for 1 byte.

We now formalize the cost framework concept.

Definition 2.2 *The* **basic framework** *is one where hit costs are equal to* 0 *and fault costs are equal to* 1*. The* **Torng framework** *is one such that hit costs are equal to* 1*, and fault cost for a page x is equal to $c_f(x) = p \cdot s(x) + 1$ where $s(x)$ is the size in bytes of page x and p is the fault penalty for* 1 *byte.*

Definition 2.3 *The* **total cost** *of processing a request sequence σ using some eviction policy $\mathcal{E}$ is the sum of costs of all faults and hits that occur while $\mathcal{E}$ is serving σ.*

Observation 2.1 *A request sequence σ of length n is processed using an eviction policy $\mathcal{E}$. Let s be the total sum (in bytes) of sizes of pages that $\mathcal{E}$ faults on. In the Torng framework the total cost of processing σ using $\mathcal{E}$ is $cost(\mathcal{E}(\sigma)) = s \cdot p + n$.*

The competitive ratio for the Torng framework is defined exactly the same as for the basic framework (Definitions 1.3 and 1.4) but it uses the total cost as defined above.

Torng [29] also introduces the class of marking eviction policies.

Definition 2.4 ([29]) *A* **marking eviction policy** *is an eviction policy whose behaviour can be mimicked as follows. The eviction policy works in phases: at the beginning of a phase all pages in cache are unmarked. Each new page being requested is marked and only unmarked*

pages are evicted, if necessary. The phase ends when all pages in cache are marked. At this moment they are all unmarked and a new phase begins.

2.2.2 Previous Work

We now survey the online eviction policy results of Torng [29] in the Torng cost framework. All of his results are regarding the Torng framework combined with the CLASSICAL model (hit costs equal to 1 and fault costs equal to $p \cdot s(x) + 1 = p + 1$) and non-optional caching (Definition 1.2). Torng focuses on the class of online eviction policies called marking eviction policies (Definition 2.4).

Observe that *LRU* and *FIFO* are marking eviction policies therefore any lower bound results concerning marking eviction policies applies to both *LRU* and *FIFO*. Torng [29] proved that any marking eviction policy is $\frac{k(p+1)}{k+p} \approx min(k, p+1)$-competitive. Since p is a constant the result partially confirms the constant competitive ratio of deterministic online eviction policies like *LRU* that was observed in practice. This is in contrast to the basic framework where there is a non-constant lower bound of k on the competitive ratio (see Theorem 1.1). Furthermore Torng showed that $min(k, p+1)$ is also a lower bound on the competitive ratio of any deterministic eviction policy, so marking eviction policies are optimal among deterministic online eviction policies.

Recall the concept of a chunk (Definition 1.6). Torng also modelled the locality of reference by considering request sequences to have average chunk size (Definition 2.5) larger than $a \cdot k$. We call these request sequences "(a, k)-referenced request sequences" (see Defini-

tion 2.6). Torng [29] proved that, given this restriction, the competitive ratio of any marking eviction policy is less than $1 + \frac{p}{a}$. Therefore if a is comparable to p then the competitive ratio is a constant, matching previous *LRU* experimental results. In contrast to this, eviction policies like *Most Recently Used* (*MRU*), *Last In First Out* (*LIFO*) and *Least Frequently Used* (*LFU*) have lower bounds of $(p+1)$ on the competitive ratio even on severely restricted request sequences where the average chunk size is an arbitrarily large multiple of $k \cdot p$.

In the same paper [29] Torng also showed several results about randomized online eviction policies. The eviction policy *MARKING* (see Section 1.3) always evicts a page from the unmarked pages in cache. Torng proved that the competitive ratio of the *MARKING* eviction policy is $2H_p + 2 - 2\ln 2$ if $2\sqrt{e} \leq p \leq 2k$ or $2H_k$ if $p > 2k$ or $1 + \frac{p}{\sqrt{e}}$ if $p < 2\sqrt{e}$ (recall H_i is the i-th Harmonic number). He also showed that *MARKING* is optimal, within a constant, among randomized online eviction policies. The last relevant result of Torng is the following: if the set of possible request sequences contains only (a, k)-referenced request sequences, then the competitive ratio of *MARKING* is at most $2 + 2\ln\frac{p}{2a}$ if $\frac{p}{2k} \leq a \leq \frac{p}{2\sqrt{e}}$, at most $2H_k$ if $a < \frac{p}{2k}$, and at most $1 + \frac{p}{a\sqrt{e}}$ if $p > \frac{p}{2\sqrt{e}}$.

2.2.3 New Results for the BIT Model with Non-Optional Caching

We provide new results on the Torng framework in the Bit model with non-optional caching (Definition 1.2) where fault costs are proportional to page sizes.

Recall the definition of a chunk (see Definition 1.6).

Definition 2.5 *Let σ be a request sequence, n be its length and s_σ be the sum (in bytes) of sizes of all requests in σ. Let $B(\sigma, k)$ denote the number of chunks in σ. The* **average chunk size** *in σ is $L(\sigma, k) = \frac{s_\sigma}{B(\sigma,k)}$ and the* **average request size** *in σ is $L_r(\sigma) = \frac{s_\sigma}{n}$.*

It is possible for a chunk to have total size less than k, therefore $L(\sigma, k)$ can be smaller than k. Recall that k denotes the cache size (in bytes). Let s_{min} and s_{max} be the minimum and maximum sizes (in bytes) of a page in U, the universe of requests ($1 \leq s_{min} \leq s_{max} \leq k$). Using the definition above we immediately obtain the following observation.

Observation 2.2 *Let σ be a request sequence, $L(\sigma, k)$ be the average size of a chunk in σ, and $L_r(\sigma)$ be the average request size in σ. The following inequalities hold:*

$$L(\sigma, k) \geq k - s_{max} + 1 \tag{2.1}$$

$$s_{min} \leq L_r(\sigma) \leq s_{max} \tag{2.2}$$

We now present an upper bound on the competitive ratio of any marking eviction policy (Definition 2.4). Let $c_f(x)$ be the fault cost on page x and $s(x)$ be the size of page x. Recall that $c_f(x) = p \cdot s(x) + 1$ where p is a constant.

Theorem 2.1 *Let σ be any request sequence, $\mathcal{M}$ be any marking eviction policy and OPT be an offline optimal eviction policy. The following inequality holds in the* Bit *model combined with the Torng framework:*

$$\frac{cost(\mathcal{M}(\sigma))}{cost(OPT(\sigma))} \leq 1 + \frac{p(k - s_{min})}{\frac{L(\sigma,k)}{L_r(\sigma)} + s_{min}p}$$

Proof:

Observe that any marking eviction policy $\mathcal{M}$ pays at most the cost of one page fault on each distinct page in each chunk in σ. Let B be the current chunk and p_1, p_2, ..., p_{d_B} be all distinct pages that are requested in B. It follows that $\frac{k}{s_{max}} \leq d_B \leq \frac{k}{s_{min}}$. Let l_B be the total number of requests in B. The hit cost of $\mathcal{M}$ on B is $l_B - d_B$. The fault cost of $\mathcal{M}$ on B is $p\sum_{i=1}^{d_B} s(p_i) + d_B$. The total cost of $\mathcal{M}$ on B is $p\sum_{i=1}^{d_B} s(p_i) + l_B \leq p \cdot k + l_B$. Let $B(\sigma, k) = b$ and the chunks of σ be denoted by B_1, B_2, ..., B_b. Therefore $cost(\mathcal{M}(\sigma)) \leq pkb + \sum_{j=1}^{b} l_{B_j} = pkb + n$ (i).

The offline optimal eviction policy OPT has to fault at least once in each chunk B. Recall that $c_f(x) = ps(x) + 1$. Since both p and s_{min} are at least 1, the cost of a fault is strictly greater than the cost of a hit. Therefore the minimum cost of OPT on B is achieved when OPT faults only once on B and its cost is $p \cdot s(x) + l_B$ where x is the page OPT faults on. The total cost of OPT on B is at least $p \cdot s_{min} + l_B$. Summing over all chunks we get $cost(OPT(\sigma)) \geq ps_{min}b + \sum_{j=1}^{b} l_{B_j} = ps_{min}b + n$ (ii). By inequalities (i) and (ii):

$$\begin{aligned}\frac{cost(\mathcal{M}(\sigma))}{cost(OPT(\sigma))} &\leq \frac{n + ps_{min}b + p(k - s_{min})b}{n + ps_{min}b} = 1 + \frac{p(k - s_{min})b}{n + ps_{min}b} \\ &= 1 + \frac{p(k - s_{min})}{\frac{n+ps_{min}b}{b}} = 1 + \frac{p(k - s_{min})}{\frac{n}{b} + s_{min}p}\end{aligned}$$

Recall that by Definition 2.5 $b = B(\sigma, k) = \frac{s_\sigma}{L(\sigma,k)}$ and $L_r(\sigma) = \frac{s_\sigma}{n}$.

It follows that $\frac{cost(\mathcal{M}(\sigma))}{cost(OPT(\sigma))} \leq 1 + \frac{p(k - s_{min})}{\frac{nL(\sigma,k)}{s_\sigma} + s_{min}p} = 1 + \frac{p(k - s_{min})}{\frac{L(\sigma,k)}{L_r(\sigma)} + s_{min}p}$. ■

Theorem 2.2 *In the* BIT *model combined with the Torng framework the following holds: the competitive ratio of every marking eviction policy is less than* $\min\left\{\frac{k}{s_{min}}, 1 + s_{max}p\right\}$.

Proof:

Let $\mathcal{M}$ be a marking eviction policy.

By Theorem 2.1 we have that $\frac{cost(\mathcal{M}(\sigma))}{cost(OPT(\sigma))} \leq 1 + \frac{p(k - s_{min})}{\frac{L(\sigma,k)}{L_r(\sigma)} + s_{min}p}$, $\forall\sigma$.

By inequalities 2.1 and 2.2 we get that

$$\frac{cost(\mathcal{M}(\sigma))}{cost(OPT(\sigma))} \leq 1 + \frac{p(k - s_{min})}{\frac{k - s_{max} + 1}{s_{max}} + s_{min}p} = 1 + \frac{s_{max}p(k - s_{min})}{k - s_{max} + s_{min}s_{max}p + 1}, \forall\sigma \quad (1)$$

This gives us an upper bound on the competitive ratio $c_{\mathcal{M}}$ of every marking eviction policy $\mathcal{M}$. Since $k - s_{max} + 1 > 0$ it must be that $c_{\mathcal{M}} \leq 1 + \frac{s_{max}p(k-s_{min})}{s_{max}s_{min}p} = 1 + \frac{k-s_{min}}{s_{min}} = \frac{k}{s_{min}}$.

Now it remains to show that $c_{\mathcal{M}} \leq 1 + s_{max}p$. Start with inequality (1) above:

$$\frac{cost(\mathcal{M}(\sigma))}{cost(OPT(\sigma))} \leq 1 + \frac{s_{max}p(k - s_{min})}{k - s_{max} + s_{min}s_{max}p + 1} \leq 1 + \frac{s_{max}pk}{k - s_{max} + s_{min}s_{max}p + 1}$$

Since $s_{min} \geq 1$ and $p \geq 1$ it holds that $s_{min}s_{max}p - s_{max} + 1 > 0$.

Hence $\frac{cost\mathcal{M}(\sigma)}{costOPT(\sigma)} \leq 1 + \frac{s_{max}pk}{k} = 1 + s_{max}p$, $\forall\sigma$. The result follows. ■

Since p is a constant, if s_{max} is also constant, the above theorem implies that the competitive ratio of every marking eviction policy is constant. If s_{max} and s_{min} are equal to 1 then our result matches the previous result of Torng that the competitive ratio of M is less than $\min\{k, p + 1\}$.

We now introduce a restriction on the possible request sequences.

Definition 2.6 *Let σ be a request sequence that satisfies $L(\sigma, k) \geq ak$. We call σ an* **(a, k)-referenced request sequence**.

Theorem 2.3 *Let S be the set of all (a, k)-referenced request sequences. Let the model be the* Bit *model combined with the Torng framework. If the set of possible request sequences is S, then the competitive ratio of any marking eviction policy is less than*

$$\min\left\{\frac{k}{s_{min}}, 1 + \frac{s_{max}p}{a}\right\}$$

Proof:

Let $\mathcal{M}$ be a marking eviction policy. By $c_{\mathcal{M}}$ we denote the competitive ratio of $\mathcal{M}$ when the set of all possible request sequences is S.

The proof that $c_{\mathcal{M}} \leq \frac{k}{s_{min}}$ is the same as the one for the first part of Theorem 2.2. It remains to demonstrate that $c_{\mathcal{M}} \leq 1 + \frac{s_{max}p}{a}$.

By Theorem 2.1 we have that $\dfrac{cost(\mathcal{M}(\sigma))}{cost(OPT(\sigma))} \leq 1 + \dfrac{p(k - s_{min})}{\frac{L(\sigma,k)}{L_r(\sigma)} + s_{min}p}, \forall\sigma.$

By definition of (a, k)-referenced request sequences it follows that $L(\sigma, k) \geq a{\cdot}k$. Combine this with inequality 2.2 to obtain that

$$\frac{cost(\mathcal{M}(\sigma))}{cost(OPT(\sigma))} \leq 1 + \frac{p(k - s_{min})}{\frac{ak}{s_{max}} + s_{min}p} = 1 + \frac{s_{max}p(k - s_{min})}{ak + s_{min}s_{max}p}, \forall\sigma \in S$$

Since $s_{max}p > 0$ we obtain that $c_{\mathcal{M}} \leq 1 + \frac{s_{max}pk}{ak + s_{min}s_{max}p}$. Also $s_{min}s_{max}p > 0$ hence $c_{\mathcal{M}} \leq 1 + \frac{s_{max}pk}{ak} = 1 + \frac{s_{max}p}{a}$. The result follows. ■

We know that in [29] p is estimated to range between 0.5 to 150 with 10 being a reasonable value. Therefore, if $a \geq s_{max}$ in the previous theorem, then the upper bound on the competitiveness of $\mathcal{M}$ is about 11.

If $L(\sigma, k)$ is restricted even more, then we have an upper bound of 2 on the competitiveness of every marking eviction policy $\mathcal{M}$.

Theorem 2.4 *Let S be the set of all (a, k)-referenced request sequences σ that satisfy $a \geq pL_r(\sigma)$. Let the model be the* Bit *model combined with the Torng framework. If the set of possible request sequences is S, then the competitive ratio of any marking eviction policy is less than* 2.

Proof:

Let $\mathcal{M}$ be a marking eviction policy. By $c_{\mathcal{M}}$ we denote the competitive ratio of $\mathcal{M}$ when the set of all possible request sequences is S.

By Theorem 2.1 we have that $\dfrac{cost(\mathcal{M}(\sigma))}{cost(OPT(\sigma))} \leq 1 + \dfrac{p(k - s_{min})}{\frac{L(\sigma,k)}{L_r(\sigma)} + s_{min}p}$, $\forall\sigma$.

By definition of (a, k)-referenced request sequences, $L(\sigma, k) \geq ak$, so

$$\frac{cost(\mathcal{M}(\sigma))}{cost(OPT(\sigma))} \leq 1 + \frac{p(k - s_{min})}{\frac{ka}{L_r(\sigma)} + s_{min}p} = 1 + \frac{k - s_{min}}{\frac{ka}{pL_r(\sigma)} + s_{min}}$$

Since $a \geq pL_r(\sigma)$, $\forall\sigma \in S$ we obtain that $\dfrac{cost(\mathcal{M}(\sigma))}{cost(OPT(\sigma))} \leq 1 + \dfrac{k - s_{min}}{k + s_{min}} \leq 2$, $\forall\sigma \in S$.

The result follows. ■

Chapter 3

Relative competitiveness

3.1 Introduction

In this chapter we will prove results that use both total cost definitions: both the **basic** and the **Torng cost framework** (see Definition 2.2).

The underlying request model is that of the **access graph model** introduced by Borodin et al. in [11]. More precisely each page in the universe of requests has a corresponding vertex in the access graph G. Given two vertices v_p and v_q, $(v_p \rightarrow v_q)$ is a directed edge if q is contained in the set of pages that can be immediately requested after a request for page p. The undirected edge (v_p, v_q) is defined similarly; both pages p and q can be requested immediately one after the other.

Next we introduce the concept of a request sequence that is **consistent with an access graph**.

Definition 3.1 *Given an access graph G and a request sequence σ we say* **σ is consistent with G** *if σ can be obtained by walking on G and requesting the page associated with each of the vertices visited during this walk.*

The difference between the standard caching model and the access graph model is that the access graph model models a particular type of request sequences whereas the standard caching models assumes any request sequence is possible. Note that the standard caching model is represented by the complete access graph.

Given that k is the size of the cache, any graph with less than $(k+1)$ vertices is not interesting: for any eviction policy, once all pages are in its cache, it does not fault on any page anymore. Therefore in all results that follow we consider graphs with at least $(k+1)$ vertices.

We now present the relative competitive ratio of two eviction policies. Two definitions are provided: one for the standard caching model and another one for the access graph model.

Definition 3.2 *Let $\mathcal{E}_1$ and $\mathcal{E}_2$ be two eviction policies. $\mathcal{E}_1$ is* **c-competitive relative to $\mathcal{E}_2$** *if*

$$\exists \text{ constant } b \text{ such that } \forall \sigma, cost(\mathcal{E}_1(\sigma)) \leq c \cdot cost(\mathcal{E}_2(\sigma)) + b$$

The **competitive ratio of $\mathcal{E}_1$ relative to $\mathcal{E}_2$**, *denoted by $c(\mathcal{E}_1, \mathcal{E}_2)$ is the smallest value of c such that $\mathcal{E}_1$ is c-competitive relative to $\mathcal{E}_2$.*

Definition 3.3 *Let G be an access graph and S_G be the set of all request sequences that are consistent with G. Let $\mathcal{E}_1$ and $\mathcal{E}_2$ be two eviction policies. $\mathcal{E}_1$ is* **c-competitive on G**

relative to $\mathcal{E}_2$ *if*

$$\exists \text{ constant } b \text{ such that } \forall \sigma \in S_G, cost(\mathcal{E}_1(\sigma)) \leq c \cdot cost(\mathcal{E}_2(\sigma)) + b$$

The **competitive ratio of $\mathcal{E}_1$ on G relative to $\mathcal{E}_2$**, *denoted by* $c(G, \mathcal{E}_1, \mathcal{E}_2)$ *is the smallest value of* c *such that* $\mathcal{E}_1$ *is* c*-competitive on* G *relative to* $\mathcal{E}_2$.

This definition of relative competitive ratio has been introduced in [5]. It is different from the one introduced by Koutsoupias and Papadimitriou in [23]. We compare two particular eviction policies, whereas Koutsopias and Papadimitriou compare classes of eviction policies. The access graph model was studied in [11, 10, 21, 15, 13], but all results involve the common competitive ratio that compares the online eviction policy against the offline optimal. None of the previous literature mentions any results using the relative competitive ratio on an access graph as defined in Definition 3.3.

3.2 New Results for the CLASSICAL Model with Non-Optional Caching

We now prove a few basic results regarding the relative ratios of *Least Recently Used* (*LRU*) and *Evict Diametrically Opposed* (*EDO*) in the CLASSICAL model (Definition 2.1) with non-optional caching (Definition 1.2).

The concept of a **distance** between two vertices of a graph is presented next.

Definition 3.4 *Given an undirected and non-weighted graph G and two vertices v_p and v_q of G the* **distance $d(v_p, v_q)$** *is the length of the shortest path in G between v_p and v_q.*

EDO is the eviction policy that always evicts the page in cache that is the **most distant** from the current faulting page. Formally if the current faulting page is p then *EDO* evicts from its cache the page q that maximizes $d(v_p, v_q)$. *EDO* is similar to the *FAR* eviction policy introduced by Borodin et al. [11]. The difference is that *FAR* evicts from cache the page that is the most distant from a **set** of marked pages whereas *EDO* evicts the page that is the most distant from the faulting page.

Given a non-weighted, undirected and connected access graph G, let C_{max} be a cycle of G of maximum length. Using only edge deletions, we build its connected spanning subgraph G^* such that C_{max} is the only cycle in G^*.

Let the set of vertices in C_{max} be denoted by $V_{max} = V_1^{max} \cup V_2^{max}$ where V_1^{max} is the set of vertices on C_{max} that have degree 2 and V_2^{max} is the set of vertices on C_{max} whose degree is at least 3. We can say that each vertex in V_2^{max} has an external tree attached to it (external to C_{max}, rooted at a vertex in V_2^{max}). We refer to these external trees as **hanging trees**.

For example, see hanging tree $T = (V_T, E_T)$ in Figure 3.1 where

$$V_T = \{v_1, v_{22}, v_{23}, v_{24}, v_{25}, v_{26}, v_{27}, v_{28}, v_{29}\}$$

and

$$E_T = \{(v_1, v_{22}), (v_{22}, v_{23}), (v_{22}, v_{24}), (v_{24}, v_{25}), (v_{24}, v_{27}), (v_{25}, v_{26}), (v_{27}, v_{28}), (v_{27}, v_{29})\}$$

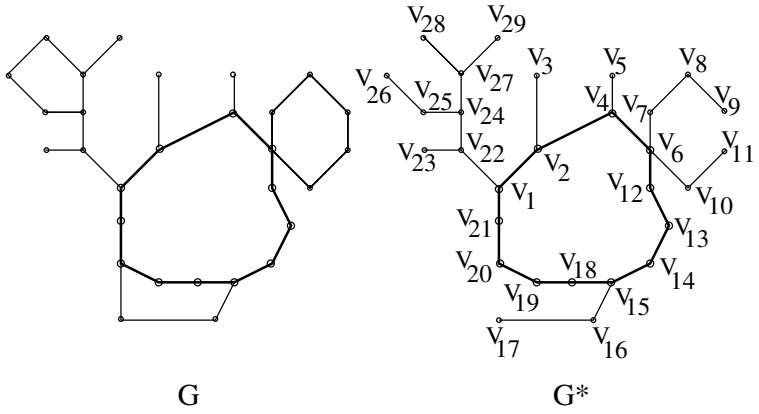

Figure 3.1: Graphs G and G^* (the cycle C_{max} is indicated by bold edges)

Let T_h be the set of all hanging trees in G^*. Let $l(T)$ denote the number of leaves of a rooted tree T. Let $L = \sum_{T \in T_h} l(T)$ and $N = |V_1^{max}|$. L and N depend on the way we break ties in selecting the maximum cycle and removing edges. For the results in this section we assume that the ties are broken such that $L + N$ is maximized to $M = \max_{L,N}(L + N)$.

The following result reveals a property of the pair of vertices whose distance is maximum among all vertex pairs in a G^*-type graph.

Lemma 3.1 *Let G be an undirected, non-weighted, and connected graph such that G contains exactly one cycle C. Let S_1 be the set of vertices of degree 1 in G. Let v_p and v_q be two vertices in G whose distance $d(v_p, v_q)$ is maximum among all pairs of vertices in G. If $|S_1| > 0$, then at least one of v_p and v_q is in S_1.*

Proof:

Use proof by contradiction; assume that neither v_p, nor v_q are in S_1. We distinguish several cases: either both v_p and v_q belong to C (case 1) or v_p is in C, v_q is in a hanging tree T_i, and v_q is not a leaf in T_i (case 2) or v_q is in C, v_p is in a hanging tree T_i, and v_p is not a leaf in T_i (case 3) or both v_p and v_q are vertices in two hanging trees $T_i = (V_i, E_i)$ and $T_j = (V_j, E_j)$ and neither v_p nor v_q are leaves (case 4).

Focus on case 1 where both v_p and v_q belong to C. Let the vertices of C be v_1, v_2, ..., v_c. Since C is the only cycle in G there are no edges $e = (v_i, v_j)$ in G such that $i, j \in \{1, 2, \ldots, c\}$ and $j \neq i+1$ except for edge (v_c, v_1). Hence there are no shortcuts between any two vertices v_g and v_h on C; there are only two paths between v_g and v_h, one going clockwise on C and the other one going counter-clockwise on C. Observe that $d(v_p, v_q) = \left\lfloor \frac{c}{2} \right\rfloor$. Furthermore, given any vertex v'_p on C, there exists a vertex v'_q on C such that $d(v'_p, v'_q) = d(v_p, v_q)$ due to cycle symmetry (i).

G has at least one vertex of degree 1 ($|S_1| > 0$) and C is the only cycle in G. Therefore there exists a subtree $T_i = (V_i, E_i)$ of G that is rooted at a vertex v_i on cycle C such that none of the edges in C are in E_i. T_i is a hanging tree on at least 2 vertices. Using the Breadth First Search algorithm we can identify a leaf w_i in T_i such that w_i is the vertex furthest from the root (w_i maximizes $d(v_i, w_i)$ over vertices in V_i).

By observation (i) above, there exists a vertex v'_q on C such that $d(v_i, v'_q) = d(v_p, v_q)$. Since T_i is a tree and C is the only cycle in G, it follows that $d(w_i, v'_q) = d(w_i, v_i) + d(v_i, v'_q) = d(w_i, v_i) + d(v_p, v_q) > d(v_p, v_q)$. We have a contradiction: $d(v_p, v_q)$ is not the maximum

distance in G.

Consider case 2: v_p is in C and v_q is in a hanging tree T_i rooted at some vertex v_i on C and v_q is not a leaf in T_i. There exists a leaf w_i in T_i such that $d(w_i, v_i)$ is the maximum distance from the root v_i to any vertex in T_i. Hence $d(w_i, v_i) > d(v_q, v_i)$. It holds that $d(w_i, v_p) = d(w_i, v_i) + d(v_i, v_p) > d(v_q, v_i) + d(v_i, v_p) = d(v_p, v_q)$. This contradicts the fact that $d(v_p, v_q)$ is maximum.

Focus on case 3 where v_q is in C and v_p is in a hanging tree T_i rooted at some vertex v_i on C and v_p is not a leaf in T_i. This case is identical to case 3, with v_p and v_q interchanged.

In case 4, both v_p and v_q are two vertices in two hanging trees $T_i = (V_i, E_i)$ and $T_j = (V_j, E_j)$ and none of v_p and v_q are leaves. T_i is rooted at v_i and T_j is rooted at v_j. There are two subcases: $T_i = T_j$ (4.1) and $T_i \neq T_j$ (4.2).

In case 4.1, $T_i = T_j$. Given any pair of vertices v and w in a tree, there exists only one path between those vertices. The distance between them is the length of this path. Let P denote the path in T_i between v_p and v_q. Since v_p is not a leaf, it must have at least one neighbour v_s such that v_s is not on P. It follows that $d(v_s, v_q) = 1 + d(v_p, v_q) > d(v_p, v_q)$ which contradicts the hypothesis that $d(v_p, v_q)$ is maximum.

Consider case 4.2 where $T_i \neq T_j$. Note that $d(v_p, v_q) = d(v_p, v_i) + d(v_i, v_j) + d(v_j, v_q)$. Let w_i and w_j be leaves in T_i and T_j such that they maximize the distances $d(v_i, w_i)$ over vertices in V_i and $d(v_j, w_j)$ over vertices in V_j. Hence $d(w_i, v_i) > d(v_p, v_i)$ and $d(w_j, v_j) > d(v_q, v_j)$. Therefore $d(w_i, w_j) = d(w_i, v_i) + d(v_i, v_j) + d(v_j, w_j) > d(v_p, v_i) + d(v_i, v_j) + d(v_j, v_q) = d(v_p, v_q)$. This contradicts the fact that $d(v_p, v_q)$ is maximum.

In all four cases, the hypothesis that $d(v_p, v_q)$ is maximum was contradicted. Our assumption is false, hence at least one of v_p and v_q is a leaf. The result follows. ■

Recall that T_h is the set of all hanging trees in G^* and $l(T)$ denotes the number of leaves of a rooted tree T. Also recall that $L = \sum_{T \in T_h} l(T)$, $N = |V_1^{max}|$, and $M = \max_{L,N}(L + N)$.

Theorem 3.1 *Let G be any access graph on $(k+1)$ vertices. In the basic framework (Definition 2.2) combined with the* Classical *model if $n \to \infty$, then the competitive ratio of LRU on G relative to EDO is at least $\frac{M}{2}$:*

$$c(G, LRU, EDO) \geq \frac{M}{2}$$

Proof:

Let C_{max} be a cycle in G of maximum length. Remove edges as necessary to obtain the connected spanning subgraph G^* of G such that C_{max} is the only cycle in G^*. It is obvious that G^* is planar, fix a planar embedding of it such that all the edges of G^* not in C_{max} are on the outside face of C_{max}. See Figure 3.1 for an example.

Let v_p and v_q be two vertices in G^* whose distance $d(v_p, v_q)$ is the largest among all pairs of vertices in G^*. Given the planar embedding of G^*, consider all request sequences that can be obtained by a clockwise traversal of G^*. Of all these request sequences, pick the one starting at v_p and denote it by σ. Let $\sigma' = \sigma^m$ (σ' is m repeated copies of σ). When serving the first copy of σ, both *LRU* and *EDO* fault on every request since the cache is initially

empty.

Consider the behaviour of *EDO* on each of the $(m-1)$ remaining copies of σ. *EDO* faults first on the request for p. Since $d(v_p, v_q)$ is maximum among all pairs of vertices in G, v_q is the vertex most distant from v_p. When *EDO* faults on p, it evicts page q. Recall the cache size is k and the number of distinct requests in σ is $k+1$. All requests occurring before the next request for q are hits. When q is requested, it is a fault. The most distant vertex from v_q is v_p, so *EDO* evicts p. All requests, before p is requested again, are hits. When p is requested again, the process repeats itself. Hence *EDO* faults only twice on each of the $(m-1)$ remaining copies of σ (i).

While serving the remaining $(m-1)$ copies of σ, *LRU* faults on only two types of pages. The first type are the pages corresponding to vertices in V_1^{max}; recall that V_1^{max} is the set of vertices on C_{max} that have degree 2. The second type are the pages corresponding to leaves of hanging trees. We consider two cases: $L = 0$ (case 1) and $L \neq 0$ (case 2).

Focus on case 1 where $\boldsymbol{L = 0}$. G^* contains no hanging trees, so it is a cycle of length $k+1$ ($G^* = C_{max}$). All vertices in C_{max} have degree 2, so $N = |V_1^{max}| = |C_{max}| = k+1 \neq 0$. $M = \max_{L,N}(L+N) = N = k+1$. Since G^* is a cycle of length $k+1$, when a fault occurs, *LRU* evicts the page that is requested next. *LRU* faults on each request of the remaining $(m-1)$ copies of σ. Therefore $cost(LRU(\sigma')) = m(k+1)$. By (i) we get that $cost(EDO(\sigma')) = 2(m-1)+k+1$. Hence

$$\lim_{m\to\infty} \frac{cost(LRU(\sigma'))}{cost(EDO(\sigma'))} = \lim_{m\to\infty} \frac{m(k+1)}{2(m-1)+k+1} = \frac{k+1}{2} = \frac{M}{2}$$

Consider case 2 where $\boldsymbol{L} \neq \mathbf{0}$. There exists at least one hanging tree in G^*, so there exists at least one vertex of degree 1 in G^*. By applying Lemma 3.1 one of v_p and v_q is a leaf in one of these hanging trees. Without loss of generality assume v_p is a leaf in some hanging tree T.

Recall that $\sigma(1) = p$, due to construction of σ. The last faulting request in the first copy of σ can be one of the following three: the last leaf v in T requested before v_p (case 2.1) or the last-requested leaf w in hanging tree T_0 traversed before T will be traversed (case 2.2) or the vertex x of degree 2 on C_{max} that is visited just before T is traversed (case 2.3). Case 2.1 occurs when v_p is not the first visited leaf in T, cases 2.2 and 2.3 occur when v_p is the first visited leaf in T. In case 2.2, $l(T_0) \geq 1$ and in case 2.3, $l(T_0) = 0$ (T_0 does not exist, the corresponding vertex x is in V_1^{max}). In all three cases (2.1, 2.2, and 2.3), the page that LRU evicts is p and LRU's next fault will be on the next request for p.

Let p_{next} be the least recently used page when the next request for p occurs. $v_{p_{next}}$ is either the next leaf (after v_p) in the traversal of T (case 2.4) or the first leaf in hanging tree T_1 traversed after T (case 2.5) or the vertex y of degree 2 on C_{max} that is visited just after T is traversed (case 2.6). Case 2.4 occurs when v_p is not the last visited leaf in T, cases 2.5 and 2.6 occur when v_p is the last visited leaf in T. In case 2.5, $l(T_1) \geq 1$ and in case 2.6, $l(T_1) = 0$ (T_1 does not exist, the corresponding vertex y is in V_1^{max}). In all three cases (2.4, 2.5, and 2.6), the page that LRU evicts is either a leaf in one of the hanging trees or a vertex in V_1^{max}. Repeating this argument yields our claim that LRU faults only on such requests.

In case 2 it holds that $cost(LRU(\sigma')) = (L + N)(m - 1) + k + 1$. By (i) we obtain that

$cost(EDO(\sigma')) = 2(m-1)+k+1$. Recall that $n \to \infty$, hence $m \to \infty$.

$$\lim_{m\to\infty} \frac{cost(LRU(\sigma'))}{cost(EDO(\sigma'))} = \lim_{m\to\infty} \frac{(L+N)(m-1)+k+1}{2(m-1)+k+1} = \frac{L+N}{2} = \frac{M}{2}$$

We conclude that $c(G, LRU, EDO) \geq \frac{M}{2}$. ■

Theorem 3.2 *In the basic framework combined with the* CLASSICAL *model there exists an access graph G such that if $n \to \infty$, then the competitive ratio of EDO on G relative to LRU is infinite:*

$$c(G, EDO, LRU) = \infty$$

Proof:

We give an access graph G and a walk σ on G such that $\lim_{n\to\infty} \frac{cost(EDO(\sigma))}{cost(LRU(\sigma))} = \infty$

Let $P = v_1 v_2 \dots v_k$ be a path of length k and v_{k+1} be a vertex not on P. Let G be the graph obtained by connecting v_{k+1} to $v_{\lceil k/2 \rceil}$ (see Figure 3.2).

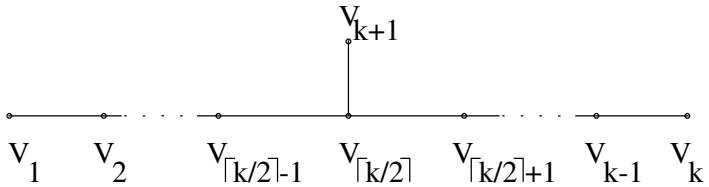

Figure 3.2: Graph G constructed for $c(G, EDO, LRU)$.

Let $\sigma_b = v_k v_{k-1} \dots v_{\lceil k/2 \rceil+1} v_{\lceil k/2 \rceil} v_{\lceil k/2 \rceil-1} \dots v_2 v_1 v_2 \dots v_{\lceil k/2 \rceil-1} v_{\lceil k/2 \rceil} v_{\lceil k/2 \rceil+1} \dots v_{k-1}$.

Set $\sigma = v_{k+1} v_{\lceil k/2 \rceil} v_{\lceil k/2 \rceil-1} \dots v_2 v_1 v_2 \dots v_{\lceil k/2 \rceil-1} v_{\lceil k/2 \rceil} v_{\lceil k/2 \rceil+1} \dots v_{k-1} (\sigma_b)^m$ (σ ends in m copies of σ_b).

We call the prefix $v_{k+1}v_{\lceil k/2\rceil}v_{\lceil k/2\rceil-1}\ldots v_2\ v_1v_2\ldots v_{\lceil k/2\rceil-1}v_{\lceil k/2\rceil}v_{\lceil k/2\rceil+1}\ldots v_{k-1}$ the **starting block** of σ.

On requests in the starting block *LRU*'s behaviour is the same as *EDO*'s behaviour. Their hits and faults match exactly: there are $\lceil k/2\rceil - 1$ hits and k faults. On the next request following the starting block (v_k) *LRU* evicts v_{k+1} which is never requested again in the future. Hence *LRU* never faults on the remaining requests. However when v_k is requested *EDO* evicts v_1 instead, then faults on v_1 and evicts v_k, then faults on v_k and evicts v_1 and so on. So *EDO* faults exactly twice on each of the σ_b subsequences. Therefore the cost of *LRU* on σ is $k+1$ and the cost of *EDO* is $k+2m$. $n = \lceil k/2\rceil + k - 1 + (2k-2)m$. Since $n \to \infty$, it must be that $m \to \infty$ as well.

$$\lim_{m\to\infty} \frac{cost(EDO(\sigma))}{cost(LRU(\sigma))} = \lim_{m\to\infty} \frac{2m+k}{k+1} = +\infty$$

Hence $c(G, EDO, LRU) = \infty$. ■

The next result follows immediately from the previous theorem since the CLASSICAL model is a subset of the BIT model.

Corollary 3.1 *In the basic framework combined with the* BIT *model there exists an access graph G such that if $n \to \infty$, then the competitive ratio of EDO on G relative to LRU is infinite:*

$$c(G, EDO, LRU) = \infty$$

Theorem 3.3 *In the Torng framework combined with the* CLASSICAL *model, for any access graph G with $(k+1)$ vertices, if $n \to \infty$, then the competitive ratio of LRU on G relative to EDO is larger than $1 + \frac{(L+N-2)p}{2p+2k}$:*

$$c(G, LRU, EDO) \geq 1 + \frac{(L+N-2)p}{2p+k+1}$$

Moreover, there exists an access graph G such that if $n \to \infty$, then the competitive ratio of EDO on G relative to LRU is larger than $1 + \frac{p}{k-1}$:

$$c(G, EDO, LRU) \geq 1 + \frac{p}{k-1}$$

Proof:

For the first statement we construct G^* from G. Using the same argument as in Theorem 3.1, we construct a request sequence σ' such that

$$\frac{cost(LRU(\sigma'))}{cost(EDO(\sigma'))} \geq \frac{n + ((L+N)(m-1) + k + 1)p}{n + (2(m-1) + (k+1))p}$$

We know that $\sigma' = \sigma^m$, so $n = m|\sigma|$, where $|\sigma|$ denotes the length of σ. This implies

$$\frac{cost(LRU(\sigma'))}{cost(EDO(\sigma'))} \geq \frac{m(|\sigma| + (L+N)p) + (k+1-L-N)p}{m(|\sigma| + 2p) + (k-1)p}$$

Since $n \to \infty$ it must be that $m \to \infty$. k, p, L and N are fixed, so we obtain

$$c(G, LRU, EDO) \geq \lim_{m\to\infty} \frac{m(|\sigma| + (L+N)p) + (k+1-L-N)p}{m(|\sigma| + 2p) + (k-1)p} = \frac{(L+N)p + |\sigma|}{2p + |\sigma|}$$

Then

$$c(G, LRU, EDO) \geq 1 + \frac{(L+N-2)p}{2p + |\sigma|} \quad (i)$$

We derive a formula for $|\sigma|$. Recall that σ is a clockwise traversal of G^* starting at some vertex v_p of degree 1 and ending at v_p's neighbour. For any vertex v let $deg(v)$ denote the degree of v in G^*. Each vertex v not on C_{max} is visited $deg(v)$ times, but each vertex v on C_{max} is visited only $(deg(v) - 1)$ times. Let V_{G^*} and E_{G^*} be the set of vertices and edges in G^*. Hence $|\sigma| = \sum_{v \in V_{G^*}} deg(v) - |C_{max}| = 2\,|E_{G^*}| - |C_{max}|$.

Recall T_h is the set of hanging trees in G^*. Observe that $|E_{G^*}| = \sum_{T \in T_h} |E_T| + |C_{max}|$. Since T is a tree it holds that $|E_T| = |V_T| - 1$, hence $|E_{G^*}| = \sum_{T \in T_h} (|V_T| - 1) + |C_{max}|$. Also the root of each tree T lies on C_{max} therefore $\sum_{T \in T_h} (|V_T| - 1) = |V_{G^*}| - |C_{max}|$. Combining the observations above we obtain that $|E_{G^*}| = |V_{G^*}| = k + 1$. Hence $|\sigma| = 2\,|V_{G^*}| - |C_{max}| = 2k + 2 - |C_{max}|$ (ii).

The minimum value of $|C_{max}|$ is obtained when G^* is a tree. In such a case, any edge can be considered to be C_{max} and $|C_{max}| = 2$. Substituting in formula (ii) we get $|\sigma| = 2k$. The maximum value of $|C_{max}|$ is obtained when G^* is the same as C_{max} (G^* contains no hanging trees). In this case $|\sigma| = 2k + 2 - (k + 1) = k + 1$. Notice how the two cases above give the maximum and minimum length of σ, hence $k + 1 \leq |\sigma| \leq 2k$. Combining this inequality

with (i) we obtain $c(G, LRU, EDO) \geq 1 + \frac{(L+N-2)p}{2p+2k}$ which proves the first statement of the theorem.

For the second statement we use the same construction for σ as in Theorem 3.2 and derive

$$\frac{cost(EDO(\sigma))}{cost(LRU(\sigma))} \geq \frac{n + (k + 2m)p}{n + (k + 1)p}$$

Substituting $n = \lceil k/2 \rceil + k - 1 + (2k - 2)m$ we get that

$$\frac{cost(EDO(\sigma))}{cost(LRU(\sigma))} \geq \frac{(2k - 2 + 2p)m + kp + \lceil k/2 \rceil + k - 1}{(2k - 2)m + (k + 1)p + \lceil k/2 \rceil + k - 1}$$

Since $n \to \infty$ we have $m \to \infty$. Since both k and p are fixed,

$$c(G, EDO, LRU) \geq \lim_{m \to \infty} \frac{cost(EDO(\sigma))}{cost(LRU(\sigma))} = \frac{k - 1 + p}{k - 1} = 1 + \frac{p}{k - 1}$$

This proves the second statement of the theorem. ■

Chapter 4

Demand Caching with Request Reordering

4.1 Introduction

It is often acceptable to alter the order of requests, within some bounds. The concept of caching with request reordering was proposed and analyzed by Feder et al. in [14] and by Albers in [3]. We consider two variations of the caching problem that are similar to the one mentioned above.

We allow the request sequence to be reorganized subject to a set of delay constraints. Reorganization is performed in steps; each step is called a **move**. Before defining the concept of a **move**, we introduce the following notation. Given two positive integers i and j such that $i < j$, let $[\![i, j]\!]$ denote the set $\{i, i+1, \ldots j-1, j\}$. Recall that given a request sequence

σ, $\sigma(i:j)$ denotes the subsequence $\sigma(i)\ \sigma(i+1)\ \ldots\ \sigma(j-1)\ \sigma(j)$.

Definition 4.1 *Let σ be any request sequence. A* **move** *$m = (j, i)$ applied to σ is an operation that relocates request $\sigma(j)$ to position i. Let the resulting request sequence be denoted by σ'. Move m is a* **forward move** *if $\sigma(j)$ is moved closer to the front of σ ($j > i$). Move m is a* **backward move** *if $\sigma(j)$ is moved closer to the back of σ ($j < i$). Its* **from-coordinate** *and* **to-coordinate** *are j and i respectively. If m is a forward move, the* **range** *of m is $R_m = [\![i, j]\!]$ and m shifts every request in $\sigma(i : j-1)$ backward by one position. I.e.: $\sigma'(1 : i-1) = \sigma(1 : i-1)$, $\sigma'(i) = \sigma(j)$, $\sigma'(i+1 : j) = \sigma(i : j-1)$, and $\sigma'(j+1 : n) = \sigma(j+1 : n)$. If m is a backward move, the* **range** *of m is $R_m = [\![j, i]\!]$ and m shifts every request in $\sigma(j+1 : i)$ forward by one position. I.e.: $\sigma'(1 : j-1) = \sigma(1 : j-1)$, $\sigma'(j : i-1) = \sigma(j+1 : i)$, $\sigma'(i) = \sigma(j)$, and $\sigma'(i+1 : n) = \sigma(i+1 : n)$.*

See Figure 4.1 for an illustration of a forward move. Recall that $\sigma(1)$ is the first request in σ and $\sigma(n)$ is the last one.

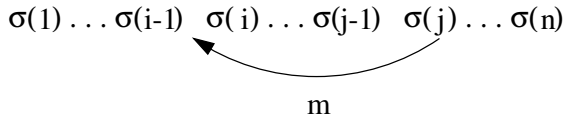

Figure 4.1: A forward move m.

We give an example to illustrate Definition 4.1. Let $\sigma = u\ y\ v\ w\ x\ z$. The forward move $m = (5, 3)$ relocates request for x from position 5 to position 3. The resulting request sequence is $\sigma' = u\ y\ x\ v\ w\ z$. The range of m is $R_m = [\![3, 5]\!]$ and its from-coordinate and to-coordinate are 5 and 3, respectively.

A **request reordering algorithm** or simply a **reordering algorithm** is an algorithm that performs a sequence of moves on a given request sequence σ_0 and obtains a permutation (final ordering) σ_f. We distinguish the process of reordering requests from that of serving them so each algorithm for this problem (**reordering scheme**) is a pair of a request reordering algorithm and an eviction policy. In the first phase the request reordering algorithm is applied to the initial request sequence σ_0 and a final ordering σ_f is obtained. In the second phase σ_f is served using the selected eviction policy. We now formalize the concepts of a reordering scheme and its cost.

Definition 4.2 *A* **reordering scheme** *is a pair* $(\mathcal{R}, \mathcal{E})$ *where* $\mathcal{R}$ *is a reordering algorithm and* $\mathcal{E}$ *is an eviction policy.*

I.e.: $(NoReordering, LFD)$ is a reordering scheme that does not apply any moves to the initial request sequence σ_0 ($\sigma_f = \sigma_0$) and serves σ_0 using the *Longest Forward Distance* (*LFD*) eviction policy.

Definition 4.3 *Let* σ *be a request sequence and* $(\mathcal{R}, \mathcal{E})$ *be a reordering scheme. Let* $\mathcal{R}(\sigma)$ *be the request sequence obtained by applying* $\mathcal{R}$ *to* σ*. The* **cost of reordering scheme** $(\mathcal{R}, \mathcal{E})$*, denoted by* $\boldsymbol{cost(\mathcal{E}(\mathcal{R}(\sigma)))}$*, is the cost of processing request sequence* $\mathcal{R}(\sigma)$ *using* $\mathcal{E}$ *as an eviction policy.*

We now introduce the concepts of **online/offline reordering algorithms** and **online/offline reordering schemes**.

Definition 4.4 *An* **offline reordering algorithm** *is a reordering algorithm that can observe all requests in the request sequence, move of any of them, and revoke any of its decisions. An* **online reordering algorithm** *is a reordering algorithm which cannot observe any requests past the current request, can move forward only the current request, and makes irrevocable decisions.*

Definition 4.5 *Given a reordering scheme* $(\mathcal{R}, LFD)$*, if* $\mathcal{R}$ *is an offline reordering algorithm then* $(\mathcal{R}, LFD)$ *is an* **offline reordering scheme**. *If* $\mathcal{R}$ *is online, then* $(\mathcal{R}, LFD)$ *is an* **online reordering scheme**.

Recall that Belady's result in [8] shows that LFD is an optimal eviction policy given a fixed request sequence. Our goal is to find a reordering algorithm $\mathcal{R}$ that minimizes $cost(LFD(\mathcal{R}(\sigma)))$, $\forall\sigma$ $(\mathcal{E} = LFD)$. We denote such an algorithm by OPT, call it an **optimal offline reordering algorithm** and call the reordering scheme (OPT, LFD) an **optimal offline reordering scheme**.

Note that we focus on deterministic offline and online reordering schemes. The **approximation factor** of a deterministic offline reordering scheme is presented next.

Definition 4.6 *Let* σ *be a request sequence,* $\mathcal{R}$ *be a deterministic offline reordering algorithm, and* OPT *be an optimal offline reordering algorithm. We say that the offline reordering scheme* $(\mathcal{R}, LFD)$ *is an* $\boldsymbol{\alpha}$**-approximation** *if*

$$cost(LFD(\mathcal{R}(\sigma))) \leq \alpha \cdot cost(LFD(OPT(\sigma))) + b, \forall\sigma \text{ and some constant } b$$

The **approximation factor** *of* $(\mathcal{R}, LFD)$*, denoted by* $\alpha(\mathcal{R}, LFD)$ *is the smallest value of* α *such that* $(\mathcal{R}, LFD)$ *is an* α*-approximation.*

We present the concept of a **competitive ratio** of a deterministic online reordering scheme.

Definition 4.7 *Let* σ *be a request sequence,* $\mathcal{R}$ *be a deterministic online reordering algorithm, and* OPT *be an optimal offline reordering algorithm. We say that the online reordering scheme* $(\mathcal{R}, LFD)$ *is* ***c*-competitive** *if*

$$cost(LFD(\mathcal{R}(\sigma))) \leq c \cdot cost(LFD(OPT(\sigma))) + b, \forall \sigma \text{ and some constant } b$$

The **competitive ratio** *of* $(\mathcal{R}, LFD)$*, denoted by* $c(\mathcal{R}, LFD)$ *is the smallest value of* c *such that* $(\mathcal{R}, LFD)$ *is* c*-competitive.*

Most of the results presented in Chapter 4 use Definitions 4.3, 4.6, and 4.7. The only exceptions are results in Sections 4.3 and 4.7 where we could not separate the analysis of the reordering algorithm and the eviction policy. Hence for all results mentioned in Sections 4.3 and 4.7 we decided to use the term **caching algorithm** instead of **reordering scheme**. The main difference between a reordering scheme and a caching algorithm is the way we analyze reordering and evictions: in a reordering scheme they are distinct phases (first the whole request sequence is reordered and afterwards it is served using an eviction policy), but in a caching algorithm they are intertwined (the request sequence is divided into batches and each batch has its own reordering and eviction phases, so the analysis alternates between

the reordering and eviction phases).

Note that our results concerning reordering schemes use *LFD* as an eviction policy, but our results concerning caching algorithms use other eviction policies. All results presented in Chapter 4 are for the CLASSICAL cost model combined with the basic framework (Definitions 2.1 and 2.2).

4.2 Reordering Models

We now present our BIDIRECTIONAL and UNIDIRECTIONAL **reordering models** and mention the similarities and differences between our BIDIRECTIONAL model and that of Feder et al. [14]. All three models constrain the "distance", r, that a request may be moved. The model of Feder et al. says that one request can be served before another if it occurs no later than r positions after that request in the original request sequence. This model, then, deals with the promptness of one request relative to any other. We introduce two models that deal with the absolute time that a request is served relative only to its position in the original request sequence. Under the BIDIRECTIONAL model a request must be served within r positions of its place in the initial request sequence. Under the UNIDIRECTIONAL model a request may be moved forward arbitrarily but may not be delayed more than r positions of its place in the initial request sequence.

In Chapter 4 by request $\overline{x} = (h, x)$ we mean the h-th request for page x in a request sequence σ_g. Note that we usually write $\overline{x}$ when referring to $\overline{x} = (h, x)$ and the index of $\overline{x}$ clarifies the exact position of this request. Let $\sigma_g^{-1}(\overline{x})$ denote the index of request $\overline{x}$ in

request sequence σ_g. Whenever required we use a more detailed notation for the same index: $\sigma_g^{-1}((h,x))$. For example let $\sigma_g = x\ y\ x\ z\ y\ v\ z\ x\ y\ x\ z\ y\ y$. The position of request $\overline{x} = (4,x)$ is $\sigma_g^{-1}(\overline{x}) = \sigma_g^{-1}((4,x)) = 10$.

We now present the three **reordering models** and the concept of a **reordering threshold**.

Definition 4.8 *Let $r \geq 1$ be a fixed parameter. Let σ_0 be the initial request sequence and σ_f be the final request sequence that a reordering algorithm produces. The constraints that any reordering algorithm must satisfy are:*

- FEDER *model ([14]): request $\overline{y}$ can be served before request $\overline{x}$ if $\sigma_0^{-1}(\overline{y}) - \sigma_0^{-1}(\overline{x}) < r$, $\forall$ requests $\overline{x}$ and $\overline{y}$ in σ_0*
- BIDIRECTIONAL *model: $|\sigma_f^{-1}(\overline{x}) - \sigma_0^{-1}(\overline{x})| \leq r$, $\forall$ request $\overline{x}$ in σ_0*
- UNIDIRECTIONAL *model: $\sigma_f^{-1}(\overline{x}) - \sigma_0^{-1}(\overline{x}) \leq r$, $\forall$ request $\overline{x}$ in σ_0*

In all three models the parameter r is called the **reordering threshold**.

Observe the following reordering model that is similar to the ones above: $0 \leq \sigma_f^{-1}(\overline{x}) - \sigma_0^{-1}(\overline{x}) \leq r$, $\forall$ request $\overline{x}$ in σ_0. For any final ordering σ_f of σ_0 ($\sigma_f \neq \sigma_0$), there exists some request $\overline{x}$ such that $\sigma_f^{-1}(\overline{x}) < \sigma_0^{-1}(\overline{x})$; this reordering model is useless, so we only analyze the 3 reordering models presented in Definition 4.8.

Note that the BIDIRECTIONAL model is similar to the FEDER model, but they are not equivalent. Observe that any two consecutive request blocks of length equal to r, the reorder-

ing threshold, can be exchanged without violating the constraints of the BIDIRECTIONAL model. However performing such an exchange is impossible in the FEDER model.

For instance, consider $\sigma_0 = x_1\ x_2\ \ldots\ x_{r-1}\ x_r\ x_{r+1}\ x_{r+2}\ \ldots\ x_{2r}$ and let $\sigma_f = x_{r+1}\ x_{r+2}\ \ldots\ x_{2r}\ x_1\ x_2\ \ldots\ x_{r-1}\ x_r$. On one hand, $\left|\sigma_f^{-1}(\overline{x}) - \sigma_0^{-1}(\overline{x})\right| = r$, $\forall$ requests $\overline{x}$ in σ_0, so the BIDIRECTIONAL model constraints are satisfied. On the other hand, the request for page x_{2r} is served before the request for page x_1 and this violates the constraints of the FEDER model.

4.3 Previous Work in the FEDER Model

This section focuses on the FEDER model and the results of Feder et al. [14] and Albers [3]. Recall u is the number of distinct pages in the universe of requests. All results that Feder et al. present in [14] are for non-optional caching (Definition 1.2). They show that when the cache size k is equal to 1 there is an optimal offline caching algorithm that runs in time $O(nr2^r)$ or $O(nr^{u+1})$. Observe that this runtime is polynomial if u is a constant or r is $O(\log n)$. Feder et al. [14] also present some results concerning online caching algorithms. Their definition of online caching algorithms is similar but is not equivalent to our definition of online reordering schemes with look-ahead of size equal to r, the reordering threshold. See Definitions 4.4, 4.5, and 4.20) for details on online reordering schemes. Feder et al.'s online caching algorithms **reorder** and **serve** requests **online**, while online reordering schemes **reorder** requests **online** but **serve** them **offline**. We note only the results of Feder et al. where k is arbitrary and where both the online and the optimal offline caching

algorithm are allowed to reorder the request sequence. For deterministic caching algorithms, the competitive ratio has a lower bound of k. They demonstrate that a modified version of *LRU* achieves a competitive ratio of $k + 2$ in the FEDER model. The competitive ratio of randomized caching algorithms is lower bounded by H_k. Recall the randomized *MARKING* eviction policy presented in Section 1.3. Feder et al. [14] prove that a modified version of *MARKING* attains a competitive ratio of $2H_k + 2$ in the FEDER model.

Albers [3] developed new results for the offline problem in the FEDER model with optional caching (Definition 1.2) for arbitrary k. She reduced the request reordering problem to that of computing the optimal reordering in individual batches of size equal to r, the reordering threshold. Albers [3] presented an offline caching algorithm *BMIN* for the CLASSICAL cost model. Let $b(x)$ denote the index of the batch where the next request for x occurs. If such a batch does not exist, then set $b(x) = \lfloor \frac{n}{r} \rfloor + 1$. *BMIN* serves σ in batches. Let S be the set of pages in *BMIN*'s cache before it starts processing the current batch B. The caching algorithm first serves all requests in B that are in S. For each remaining request for x that is a fault, if the cache is full, then *BMIN* computes $b_m = \min_{x' \in C} b(x')$ where C is set of pages that are currently in cache. If $b(x) = b_m$ then x is not brought into the cache. If $b(x) < b_m$ *BMIN* evicts a page y from cache such that $b(y) = b_m$. Albers [3] proved *BMIN* is a 2-approximation algorithm in the FEDER model.

The complexity of caching with request reordering is open in both the UNIDIRECTIONAL and BIDIRECTIONAL models. The brute force offline reordering scheme, which applies the *LFD* eviction policy to all possible orderings, takes exponential time. Even for r equal to 1

the number of possible orderings is exponential (see Theorems 4.3 and 4.4).

4.4 Overlapping Moves and Ignoring Backward Moves

Given a sequence of moves M and a request sequence σ, let σ_M denote the request sequence obtained by applying all moves in M to σ.

Definition 4.9 *Let σ a request sequence and M and M' be two sequences of moves. If $\sigma_M = \sigma_{M'}$ we say M and M' are* **equivalent**.

We now introduce notation for a decomposition of a sequence of moves. Let M be the sequence of moves m_1, m_2, ..., m_f. For some $g \leq f$, let M_1 be the sequence of moves m_1, m_2, ..., m_g, and M_2 be the sequence of moves m_{g+1}, m_{g+2}, ..., m_f. We write $M = M_1, M_2$ to show the decomposition of M into two sequences of moves M_1 and M_2. Note that this notation can be easily extended for the decomposition of M into h sequences of moves M_1, M_2, ..., M_h, where $h \leq f$.

Lemma 4.1 *For every sequence of moves M, there exists an equivalent sequence of moves M' such that all moves in M' are forward moves.*

Proof:

We show that any backward move in M can be replaced by a sequence of forward moves such that the resulting sequence of moves M' is equivalent to M. By repeating this argument, the result follows.

We now formalize the argument. Let $M = M_1, m, M_2$ be a sequence of moves and m be a backward move. We prove that there exists a sequence M_f of forward moves such that the sequence of moves $M' = M_1, M_f, M_2$ is equivalent to M.

For reasons of simplicity, we use an alternative definition and notation for a move. Let σ be any request sequence and x, y be two requests in σ. A **move** $m = (x, y)$ is an operation that relocates request x behind request y. Move m is a **forward move** if $\sigma^{-1}(x) > \sigma^{-1}(y)+1$ or a **backward move** if $\sigma^{-1}(x) < \sigma^{-1}(y)$. The rest of the definition and notation is identical to one used in Definition 4.1 where $i = \sigma^{-1}(y)$ and $j = \sigma^{-1}(x)$.

Let $m = (x, y)$ be a backward move in M ($M = M_1, m, M_2$) that relocates request x behind request y in σ_{M_1}. Let $b = \sigma_{M_1}^{-1}(y) - \sigma_{M_1}^{-1}(x)$. Note that m moves request for page x backward b positions. This is equivalent to b steps in each of which x is bumped backward one position. Each of these b steps is the result of performing a forward move in which the request behind x is moved in front of x. Let the requests that are moved forward one position be denoted by $x_1, x_2, \ldots, x_b = y$. Let z be the request requested immediately before x in σ_{M_1}. Consider the following b forward moves: $m_1 = (x_1, z)$ and $m_h = (x_h, x_{h-1})$, $\forall h \in \{2, 3, \ldots, b\}$. Let $M_f = \{m_h : 1 \leq h \leq b\}$.

Move m_1 relocates request x_1 behind request z. The second move m_2 relocates x_2 behind x_1, m_3 relocates x_3 behind x_2, etc., the last move m_b relocates $x_b = y$ behind x_{b-1}. For an illustration, see Figure 4.2.

Let M' be a sequence of moves such that $M' = M_1, M_f, M_2$. Observe that $\sigma_{M'} = \sigma_M$, hence M and M' are equivalent sequences of moves. ■

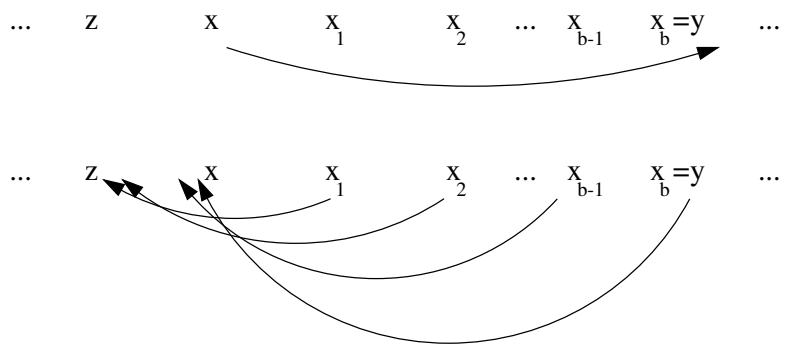

Figure 4.2: Simulation of a backward move by forward moves.

Given Lemma 4.1 we can assume that a request reordering algorithm performs only forward moves. Hence for the rest of this thesis the term "moves" refers to forward moves, unless explicitly stated otherwise.

We now introduce the concept of **valid** sequence of moves. Note that σ_g **satisfies the** UNIDIRECTIONAL **model constraints** if the UNIDIRECTIONAL model constraints are satisfied for $\sigma_f = \sigma_g$.

Definition 4.10 *Let M be a sequence of f moves applied to request sequence σ_0. Let σ_g be the request sequence obtained after the g-th move in M is applied to σ_{g-1} ($\forall g \in \{1, 2, \ldots, f\}$).*

M is **valid in the** UNIDIRECTIONAL **model** *if σ_g satisfies the* UNIDIRECTIONAL *model constraints, $\forall g \in \{1, 2, \ldots, f\}$.*

The concept of **overlapping moves** is presented next.

Definition 4.11 *Let σ be a request sequence. Let m_1 and m_2 be two moves that are applied to σ and $R_1 = [\![i_1, j_1]\!]$ and $R_2 = [\![i_2, j_2]\!]$ be their ranges. If $i_1 \leq i_2 \leq j_1$ or $i_2 \leq i_1 \leq j_2$, then m_1 and m_2 are* **overlapping** *moves.*

We now present a result concerning the maximum number of moves that pairwise overlap one another.

Theorem 4.1 *Let M be a sequence of moves which, applied to a request sequence σ_0, result in a request sequence σ_f that satisfies all constraints of the* UNIDIRECTIONAL *model.*

There exists a sequence of moves M' that is valid in the UNIDIRECTIONAL *model, is equivalent to M, and has the property that the maximum number of moves in M' that pairwise overlap one another is equal to r, the reordering threshold.*

Proof:

First we show how to construct a sequence of moves M' that is equivalent to M and is valid in the UNIDIRECTIONAL model. Consider the initial request sequence σ_0 and the final request sequence σ_f, both of length n. Let m_g be the g-th move we select to be in M'. Let σ_{g-1} be the request sequence to which m_g is applied and σ_g be the resulting request sequence. The selection process is such that the common prefix of σ_g and σ_f is strictly longer than the common prefix of σ_{g-1} and σ_f. By repeating this process, σ_0 is transformed into σ_f.

The selection proceeds as follows: at step g the current ordering is σ_{g-1}. Find the smallest index i such that $\sigma_{g-1}(i) \neq \sigma_f(i)$. Let j be the position of request $\sigma_f(i)$ in σ_{g-1}. Let σ_g be the request sequence obtained by applying move $m_g = (j, i)$ to σ_{g-1}. Since $\sigma_g(i) = \sigma_f(i)$, the common prefix of σ_g and σ_f is at least 1 longer than the common prefix of σ_{g-1} and σ_f (i).

If $\sigma_g = \sigma_f$, then stop. Otherwise proceed to the next step. By (i), this process is finite. Let M' be the sequence of moves that were selected. It is obvious that M' is equivalent to M.

Before we show that M' is valid in the UNIDIRECTIONAL model we present the concept of a **delay value**. Given a request sequence σ_0 and a ordering of it, σ_g, the delay value of a request $\overline{x}$ relative to σ_g is $d(\overline{x}, \sigma_g) = \sigma_g^{-1}(\overline{x}) - \sigma_0^{-1}(\overline{x})$. In words, the delay value is equal to the number of positions that $\overline{x}$ has been delayed in σ_g. Note that a negative delay value means that $\overline{x}$ has been advanced before its initial position in σ_0.

Let $m_g = (j, i)$ be any move in M'. Recall m_g is applied to σ_{g-1} and the resulting request sequence is denoted by σ_g. Observe that if the delay values of requests in $\sigma_{g-1}(i : n)$ are in non-increasing order, then the delay values of requests in $\sigma_g(i + 1 : n)$ are non-increasing order. This is true since requests in $\sigma_{g-1}(j + 1 : n) = \sigma_g(j + 1 : n)$ do not have their delay value changed due to m_g and requests in $\sigma_{g-1}(i : j - 1) = \sigma_g(i + 1 : j)$ have their delay value increased by 1 due to m_g. Since delay values of requests in initial request sequence σ_0 are in non-increasing order, by the invariant above it follows that for any move $m_g = (j, i)$ in M' the delay values of requests in $\sigma_g(i + 1 : n)$ are in non-increasing order.

We now prove that if σ_{g-1} satisfies the constraints of the UNIDIRECTIONAL model so does σ_g (ii). Since σ_0 satisfies these constraints by default, this invariant proves that M' is valid in the UNIDIRECTIONAL model.

Let σ_{g-1} satisfy the constraints of the UNIDIRECTIONAL model. Falsely assume that σ_g does not satisfy them. Since move m_g can only increase delay values by 1 and only for requests in $\sigma_{g-1}(i : j - 1)$, there exists some request $\overline{x}$ in $\sigma_g(i + 1 : j)$ such that $d(\overline{x}, \sigma_{g-1}) = r$

and $d(\overline{x}, \sigma_g) = r + 1$. Let request $\sigma_g(i+1)$ be denoted by $\overline{y}$. By the delay value invariant, $d(\overline{y}, \sigma_g) \geq d(\overline{x}, \sigma_g) \geq r + 1$. Due to the selection process, $\sigma_g(1 : i) = \sigma_f(1 : i)$, hence $\sigma_f^{-1}(\overline{y}) \geq \sigma_g^{-1}(\overline{y})$. Therefore $d(\overline{y}, \sigma_f) \geq d(\overline{y}, \sigma_g) \geq r + 1$. σ_f violates the UNIDIRECTIONAL model constraint, this is a contradiction. Hence our assumption was false, σ_g also satisfies the UNIDIRECTIONAL model constraints.

We now show that the maximum number of moves in M' that pairwise overlap one another is equal to r, the reordering threshold. Notice that, due to the selection process, the from-coordinates of moves in M' are in strictly increasing order (see Definition 4.1) (iii). Let $m_g = (j_g, i_g)$ and $m_h = (j_h, i_h)$ be any two moves in M' such that m_h is performed after m_g is applied. By (ii), $i_g < i_h$. By definition of a move, $i_h < j_h$, hence $j_h \geq i_g + 2$ (iv). By (iii), if m_g and m_h overlap, then there is at least one request in the range of m_g that has been bumped back two positions as a result of applying moves m_g and m_h. Observe that if there exist h moves in M' that pairwise overlap one another, then there exists a request that has been bumped back h positions. Since M' is valid in the UNIDIRECTIONAL model, no request was bumped back more than r positions, therefore the maximum number of moves in M' that pairwise overlap one another is r. ■

4.5 Value of a Move and a Property of Effective Moves

We now introduce the concept of **value of a move**.

Definition 4.12 *Let σ_{g+1} be the request sequence obtained by applying move m to request sequence σ_g. The* **value of a move** *m is*

$$v(m) = cost(LFD(\sigma_g)) - cost(LFD(\sigma_{g+1}))$$

Note that if m has a positive value, then the *LFD* cost will increase if m is performed. If m has a negative value, then the *LFD* cost will increase if m is performed. See the following definition for a classification of moves according to their value.

Definition 4.13 *Let m be a move. If $v(m) > 0$ then m is a* **positive move**. *If $v(m) < 0$ then m is a* **negative move**. *If $v(m) = 0$ then m is a* **neutral move**.

Borodin and El-Yaniv give the following statement as an exercise in [10] (Exercise 3.1). Note that they used the term page replacement algorithm instead of eviction policy, and that their definition of a demand eviction policy is slightly different from ours (see Definition 1.1).

Observation 4.1 ([10]) *Any eviction policy can be changed into a demand eviction policy without increasing the overall cost of processing any request sequence.*

Proof:

Let σ be any request sequence and $\mathcal{E}_1$ be some eviction policy. We construct an eviction policy $\mathcal{E}_2$ that uses one less non-demand admission than $\mathcal{E}_1$ and its cost satisfies $cost(\mathcal{E}_2(\sigma)) \leq cost(\mathcal{E}_1(\sigma))$. By repeating this construction, the result follows.

Suppose that, after serving some request $\sigma(i)$ for a page p, $\mathcal{E}_1$ brings into cache some page $q_1 \neq p$ and, if necessary, evicts some page $q_2 \neq q_1$ to make room for q_1. Recall this

is a non-demand admission (Definition 1.1). Let $\mathcal{E}_2$ be an eviction policy that is identical to $\mathcal{E}_1$ except that it does not bring in q_1 when $\sigma(i)$ is served. Note that $\mathcal{E}_2$ uses one less non-demand admission than $\mathcal{E}_1$.

Given an eviction policy $\mathcal{E}$ let $C_{\mathcal{E},h}$ be $\mathcal{E}$'s cache contents **just before** serving request $\sigma(h)$. Since $\mathcal{E}_2$ does not bring q_1 into cache and evict q_2 after serving $\sigma(i)$, we get that $C_{\mathcal{E}_1,i+1} \setminus \{q_1\} = C_{\mathcal{E}_2,i+1} \setminus \{q_2\}$ and up to position $i+1$ the cost incurred by $\mathcal{E}_2$ is one less than the cost incurred by $\mathcal{E}_1$ (i).

Since $\mathcal{E}_2$ evicts the same pages that $\mathcal{E}_1$ evicts, the only differences in cost and cache contents between the two eviction policies occur at those times when pages q_1 or q_2 are requested, brought into cache or evicted from cache.

Let the next request for q_1 and q_2 be at time $i_1 > i$ and $i_2 > i$ respectively ($\sigma(i_1) = q_1$ and $\sigma(i_2) = q_2$). We consider four cases: $\mathcal{E}_1$ evicts q_1 before its next request (case 1), $\mathcal{E}_1$ brings in q_2 by a non-demand admission (Definition 1.1) before its next request (case 2), $\mathcal{E}_1$ serves the next request to q_2 (case 3), and $\mathcal{E}_1$ serves the next request to q_1 (case 4).

Consider case 1, where $\mathcal{E}_1$ evicts q_1 to bring in some page q_3 at time i_3 ($i_3 < i_1$). At time i_3, $\mathcal{E}_2$ evict q_2 to bring in q_3. The costs of $\mathcal{E}_1$ and $\mathcal{E}_2$ increase by 1 each (ii). The cache contents of the two eviction policies are now the same and will remain identical while the rest of the request sequence is processed.

In case 2, $\mathcal{E}_1$ brings q_2 into cache and evicts some page q_3 at time i_3 ($i_3 < i_2 < i_1$). Since $i_3 < i_2$, $\sigma(i_3) \neq q_2$; $\mathcal{E}_1$ brings q_2 during a non-demand admission. $\mathcal{E}_2$ already has q_2 in cache, so instead it brings in q_1 and evicts q_3. The costs of $\mathcal{E}_1$ and $\mathcal{E}_2$ increase by 1 each (iii). Notice

that the cache contents of both eviction policies are identical while the rest of the request sequence is processed.

Consider case 3, where $\mathcal{E}_1$ serves request $\sigma(i_2) = q_2$ and evicts some page q_3 at time i_2 ($i_2 < i_1$), $\mathcal{E}_1$ faults on $\sigma(i_2) = q_2$ and evicts q_3 to bring in q_2. $\mathcal{E}_2$ already has q_2 in its cache, $\sigma(i_2)$ is a hit for it. The cost of $\mathcal{E}_2$ is unchanged, but the cost of $\mathcal{E}_1$ increases by 1 (iv). Note that $C_{\mathcal{E}_1,i_2+1} \setminus \{q_1\} = C_{\mathcal{E}_2,i_2+1} \setminus \{q_3\}$ (v). The cache contents of $\mathcal{E}_1$ and $\mathcal{E}_2$ still differ in only one pair.

In case 4, $\mathcal{E}_1$ serves request $\sigma(i_1) = q_1$ at time i_1. Request $\sigma(i_1) = q_1$ is a hit for $\mathcal{E}_1$. $\mathcal{E}_2$ faults on it, brings in q_1 and evicts from its cache the page by which it differs from $\mathcal{E}_1$'s cache (see equality (v) above). The cost of $\mathcal{E}_1$ remains the same, but the cost of $\mathcal{E}_2$ increases by 1 (vi). Observe that the contents of the two caches are now the same and will remain the same while the remaining requests are processed.

The analysis for all four cases still holds if the mismatch between the cache contents of $\mathcal{E}_1$ and $\mathcal{E}_2$ changes slightly as it does in case 3 (see equality (v)). There are three possibilities. If none of cases 1, 2, 3, and 4 occur, then by observation (i) we get that $cost(\mathcal{E}_2(\sigma)) \leq cost(\mathcal{E}_1(\sigma))$. If only case 3 occurs (one or more times), by observations (i) and (iv) $cost(\mathcal{E}_2(\sigma)) \leq cost(\mathcal{E}_1(\sigma))$. If case 3 occurs zero or more times and afterwards one of cases 1, 2, or 4 occurs once, by observations (i), (iv), (ii), (iv) and (vi) above we obtain that $cost(\mathcal{E}_2(\sigma)) \leq cost(\mathcal{E}_1(\sigma))$. Due to the observation made in the beginning of this proof, the result follows. ■

We present a result about the value of a move.

Theorem 4.2 *For any move m, the value of m satisfies the inequality $-2 \leq v(m) \leq 2$. Furthermore if m relocates the last request, then $v(m) \leq 1$.*

Proof:

Let $m = (j, i)$ be a move that is applied to some request sequence σ_g. Let σ_{g+1} be the resulting request sequence. m relocates request $\sigma_g(j) = x$ to position i in σ_g.

We construct an eviction policy *EMU* that is not necessarily a demand eviction policy. *EMU* that processes σ_g, but emulates the behaviour of *LFD* on σ_{g+1}. Let $C_{EMU,h}$ and $C_{LFD,h}$ be *EMU*'s and *LFD*'s cache contents **just before** serving the requests $\sigma_g(h)$ and $\sigma_{g+1}(h)$, respectively. Saying that *EMU* and *LFD*'s cache contents mismatch in pages x and y before serving the h-th request is equivalent to $C_{EMU,h} \setminus \{x\} = C_{LFD,h} \setminus \{y\}$.

We now describe the behaviour of *EMU*. *EMU* serves $\sigma_g(1 : i-1)$ by making the same evictions as *LFD* does when it serves $\sigma_{g+1}(1 : i-1)$. Therefore $C_{EMU,h} = C_{LFD,h}$, $\forall h \in \{1, 2, \ldots, i-1\}$. *LFD* is currently serving request $\sigma_{g+1}(i) = x$. Note that when *EMU* will serve the same request for x ($\sigma_g(j) = x$), *LFD* will not do anything since it has already served this request as request $\sigma_{g+1}(i) = x$.

If $x \in C_{LFD,i}$ then the request $\sigma_{g+1}(i) = x$ is a hit for *LFD*; $C_{LFD,i+1} = C_{LFD,i}$, hence $C_{EMU,i} = C_{LFD,i+1}$. If $x \notin C_{LFD,i}$ then the request $\sigma_{g+1}(i) = x$ is a fault for *LFD*. It evicts some page p, brings in page x, and its cost increases by 1. We force *EMU* to also evict p and bring in x, so its cost also increases by 1. Note that $C_{EMU,i} = C_{LFD,i+1}$.

EMU serves requests in $\sigma_g(i : j-1)$ by making the same evictions that *LFD* does while serving $\sigma_{g+1}(i+1 : j)$. Therefore $C_{EMU,j} = C_{LFD,j+1}$ and the costs incurred by *EMU* and *LFD* up to and including position $j-1$ and j, respectively, are equal.

If $x \in C_{EMU,j}$ then $\sigma_g(j) = x$ is a hit, $C_{EMU,j+1} = C_{LFD,j+1}$. If $x \notin C_{EMU,j}$ then $\sigma_g(j) = x$ is a fault. *EMU* evicts some page p and brings in x, so its cost increases by 1. Immediately afterwards, *EMU* evicts x and brings p back into cache so *EMU*'s cost increases by 1 again and $C_{EMU,j+1} = C_{LFD,j+1}$. Therefore the cost incurred by *EMU* up to and including position j is at most two larger than the cost of *LFD* up to and including position j (i).

In both cases, $C_{EMU,j+1} = C_{LFD,j+1}$. *EMU* serves requests in $\sigma_g(j+1 : n)$ by making the same evictions that *LFD* does while serving $\sigma_{g+1}(j+1 : n)$. Therefore the cost incurred by *EMU* while serving $\sigma_g(j+1 : n)$ is equal to the cost incurred by *LFD* while serving $\sigma_{g+1}(j+1 : n)$ (ii).

By (i) and (ii), $cost(EMU(\sigma_g)) \leq cost(LFD(\sigma_{g+1}))+2$. Since *LFD* is an optimal demand eviction policy, by Observation 4.1 we obtain $cost(LFD(\sigma_g)) \leq cost(EMU(\sigma_g))$. By the last two inequalities and the definition of the value of a move, it follows that $v(m) \leq 2$.

The proof that $v(m) \geq -2$ is extremely similar to the one above. The difference is that *EMU* will process σ_{g+1} and imitate the behaviour of *LFD* on σ_g such that $cost(EMU(\sigma_{g+1})) \leq cost(LFD(\sigma_g)) + 2$.

Let m relocate the last request in σ_g ($\sigma_g(n) = x$). The proof is identical to the one above, except that when *EMU* serves $\sigma_g(j) = x$ it does not evict x and bring p back into cache. The costs satisfy $cost(EMU(\sigma_g)) \leq cost(LFD(\sigma_{g+1})) + 1$, so $v(m) \leq 1$. ■

We now introduce the concept of an **ineffective move**.

Definition 4.14 *Let M be a sequence of moves and m be a move in M. We say move m is* **ineffective with respect to M** *if the cost of LFD is reduced by removing m from M:*

$$cost(LFD(\sigma_{M\setminus\{m\}})) \leq cost(LFD(\sigma_M))$$

Otherwise m is **effective with respect to M**.

For example, let $k = 1$, $\sigma = z\ x\ x\ y\ v\ v\ y\ x$, $M = m_1, m_2$, and $m = m_1$, where $m_1 = (4, 3)$ and $m_2 = (8, 4)$. Observe that $\sigma_{M\setminus\{m\}} = \sigma_{m_2} = z\ x\ x\ x\ y\ v\ v\ y$ and $\sigma_M = z\ x\ y\ x\ x\ v\ v\ y$. Since $cost(LFD(\sigma_{M\setminus\{m\}})) = 5 < cost(LFD(\sigma_M)) = 6$, m is a move that is *ineffective with respect to M.*

Observe that if m is ineffective with respect to M then a reordering algorithm $\mathcal{R}$ can choose to perform all moves in $M \setminus \{m\}$ instead of those in M, without increasing the LFD cost of processing the final ordering. By repeating this observation, we obtain a reordering algorithm that performs a sequence M' of effective moves such that the cost of processing $\sigma_{M'}$ using LFD is equal to or smaller than the cost of processing σ_M. This implies there exists an optimal reordering algorithm that performs only effective moves.

Lemma 4.2 *Let M be a sequence of moves, m be a move in M, and R_m be the range of m. If m is effective with respect to M, then LFD faults on at least one request in σ_M whose index is in R_m.*

Proof:

Let $\sigma_g = \sigma_{M\setminus\{m\}}$ and $\sigma_{g+1} = \sigma_M$. Let $m = (j, i)$ relocate request $\sigma_g(j) = x$ to position i in σ_g. Let σ_{g+1} denote the resulting request sequence. By definition $R_m = [\![i, j]\!]$.

We use a proof by contradiction. Assume that all requests whose indices are in R_m are hits when *LFD* serves σ_{g+1}. We show that

$$cost(LFD(\sigma_g)) \leq cost(LFD(\sigma_{g+1})) \tag{4.1}$$

so m is an ineffective move, which contradicts the hypothesis.

To prove 4.1 we describe an eviction policy *EMU* that emulates *LFD* but processes σ_g instead, such that

$$cost(EMU(\sigma_g)) = cost(LFD(\sigma_{g+1}))$$

Let $C_{EMU,h}$ and $C_{LFD,h}$ be *EMU*'s and *LFD*'s cache contents **just before** serving the requests $\sigma_g(i)$ and $\sigma_{g+1}(i+1)$, respectively. Up to and including position $i-1$ *EMU* evicts exactly the same pages that *LFD* evicts. Hence $C_{EMU,i} = C_{LFD,i}$ (i).

The requests in σ_{g+1} whose indices are in R_m are the requests in $\sigma_{g+1}(i : j)$. By our assumption, all of these requests are hits for *LFD*, so they are in $C_{LFD,i}$. By (i) these requests are also in $C_{EMU,i}$. Observe that $\sigma_{g+1}(i : j)$ contains the same requests as $\sigma_g(i : j)$. Therefore all requests in $\sigma_g(i : j)$ are hits for *EMU* as well.

Next it follows that $C_{EMU,j+1} = C_{LFD,j+1}$. *EMU* again serves requests past position j by evicting the same pages that *LFD* evicts. *EMU* faults on each of the requests that *LFD*

faults and it does not fault on any other ones. Hence $cost(EMU(\sigma_g)) = cost(LFD(\sigma_{g+1}))$.

Since $cost(LFD(\sigma_g)) \leq cost(EMU(\sigma_g))$, we get that $cost(LFD(\sigma_g)) \leq cost(LFD(\sigma_{g+1}))$ so $cost(LFD(\sigma_{M\setminus\{m\}})) \leq cost(LFD(\sigma_M))$. By Definition 4.14, m is ineffective with respect to M. This contradicts the hypothesis that m is effective with respect to M. Our assumption was wrong, so the result follows. ■

4.6 Number of Orderings

The complexity of caching with request reordering is open in both the UNIDIRECTIONAL and BIDIRECTIONAL models. Recall that Belady shows in [8] that *LFD* is an optimal offline eviction policy, given a fixed request sequence. *LFD* can be implemented to run in polynomial time and space (Observation A.1). However the brute force offline reordering scheme that applies *LFD* to all possible orderings runs in exponential time. Even for r equal to 1 the number of possible orderings is exponential (see Theorems 4.3 and 4.3).

Theorem 4.3 *Let the reordering threshold r be equal to* 1 *and let the initial request sequence be σ_0 of length n. The number of possible orderings of σ_0 in the* UNIDIRECTIONAL *model is* 2^{n-1}.

Proof:

We use strong induction on n. The theorem clearly holds for $n = 1$ or 2. The induction hypothesis is that the result holds for $n \leq k$. Consider a request sequence σ of length $k + 1$.

Now consider the orderings of σ in which request $\sigma(k+1)$ is in position j, $\forall j \in \{1, 2, \ldots, k+1\}$. By Theorem 4.1 we can assume that the maximum number of pairwise overlapping moves is r. Since r is 1, moves do not overlap. For $j = 1$, there is one ordering of this form. For $j > 1$, the number of these orderings is equal to the number of orderings of a request sequence of length $j-1$ since moves do not overlap. By the induction hypothesis, the number of such orderings is 2^{j-2}. Since these are all the possible orderings, there are $1 + \sum_{h=0}^{k-1} 2^h = 2^k$ orderings. Since the result holds $n = k+1$, by induction it holds for any value of n. ■

Theorem 4.4 *Let the reordering threshold r be equal to 1 and the initial request sequence be σ_0 of length n. The number of possible orderings of σ_0 in the* BIDIRECTIONAL *model is F_{n+1}, the (n+1)-st Fibonacci number. It is equal to $\frac{\phi^{n+1}}{\sqrt{5}}$ rounded to the nearest integer, where ϕ is the golden ratio.*

Proof:

Clearly the result holds for $n = 1$ and $n = 2$. We prove the property by induction on n. Fix $k \geq 2$ and make the induction hypothesis that the result holds for $n \leq k$. Consider the case of a request sequence σ of length $k+1$. Let $\overline{x}$ and $\overline{y}$ be the requests in positions $k+1$ and k in σ. By the inductive step there are F_{k+1} orderings in which $\overline{x}$ is still in position $k+1$ and a further F_k with $\overline{x}$ in position k and $\overline{y}$ in position $k+1$. Request $\overline{x}$ cannot be elsewhere, hence there is $F_k + F_{k+1} = F_{k+2}$ valid orderings of length $k+1$. By induction, the result holds for any value of n.

The second part of the theorem holds by Binet's Fibonacci Number Formula [30]. ■

4.7 BIDIRECTIONAL Model Results

We now turn our attention to the BIDIRECTIONAL model (Definition 4.8). Recall that this model simulates strong dependencies between requests such that the delay constraints are to move an element no more than r positions in both forward and backward directions where r is the reordering threshold (Definition 4.8).

Partition a request sequence into **batches** as follows: given request sequence σ_0 of length n, the ***i*-th batch** in σ_0 is the subsequence

$$\sigma_0(ir+1 : min\{ir+r, n\}), \forall i \in \{0, 1, 2, \ldots, \lfloor n/r \rfloor\}$$

All batches, except for possibly the last one, have size equal to r.

A **batch-processing algorithm** is a caching algorithm that can only perform moves whose from-coordinate and to-coordinate are within the same batch. Recall the concept of a caching algorithm that was introduced in Section 4.1. The terms caching algorithm and reordering scheme refer to the same concept; the difference is that for caching algorithms we do not have a separate analysis of the reordering algorithm and the eviction policy being used.

Theorem 2 by Albers in [3] implies that offline algorithm *BMIN* (Section 4.3) is an optimal

batch-processing algorithm in the FEDER model. A batch-processing algorithm can permute the r requests in a given batch in any possible order, without violating the constraints of either the BIDIRECTIONAL or FEDER models. Therefore *BMIN* is an optimal offline batch-processing algorithm in BIDIRECTIONAL model.

By slightly adapting the proof of Lemma 1 presented by Albers in [3] we prove that any caching algorithm can be transformed into a batch-processing algorithm whose cost is at most three times the cost of the initial caching algorithm.

Theorem 4.5 *Let our reordering model be the* BIDIRECTIONAL *model and the type of caching be* **optional caching** *(Definition 1.2). Let $\mathcal{A}$ be a caching algorithm that serves request sequence σ_0. There exists a batch-processing algorithm $\mathcal{B}$ that serves σ_0 such that its cost is smaller than 3 times $\mathcal{A}$'s cost:*

$$cost(\mathcal{B}(\sigma_0)) \leq 3 \cdot cost(\mathcal{A}(\sigma_0))$$

Proof:

Recall that given a request sequence σ_0 of length n, its i-th batch of requests is

$$\sigma_0(ir+1 : min\{ir+r, n\}), \forall i \in \{0, 1, 2, \ldots, \lfloor n/r \rfloor\}$$

Let B_i be the set of pages requested in batch i. Let A_i be the set of pages in $\mathcal{A}$'s cache at the beginning of batch i (just before request $\sigma_0(ir+1)$ is served). A_0 is defined to be the empty set and $A_{\lfloor n/r \rfloor+1}$ is the final cache configuration of $\mathcal{A}$.

Let $D_i = B_i \setminus (A_i \cup A_{i+1})$. Recall the definition of optional caching (Definition 1.2). It follows that a page in D_i is either not loaded into cache at all or it is loaded but then evicted before the end of the batch. $\mathcal{A}$ faults on those requests to pages in D_i and in $A_{i+1} \setminus A_i$. Therefore

$$cost(\mathcal{A}(\sigma_0)) \geq \sum_{i=0}^{\lfloor n/r \rfloor} (|D_i| + |A_{i+1} \setminus A_i|) \tag{4.2}$$

Algorithm $\mathcal{B}$ starts processing batch i by first serving the requests for pages in A_i that are requested in batch i. $\mathcal{B}$ incurs no cost on these requests. $\mathcal{B}$ then serves requests for pages in $A_{i+1} \setminus A_i$ by evicting pages in $A_i \setminus A_{i+1}$. Afterwards $\mathcal{B}$ serves the requests in D_i without bringing them into its cache. If $i > 0$, then $\mathcal{B}$ schedules requests to pages in $D_{i-1} \cup (A_{i-1} \setminus A_i)$ that are requested in batch i. Finally, if $i < \lfloor \frac{n}{r} \rfloor$, then $\mathcal{B}$ serves the requests for pages in $D_{i+1} \cup (A_{i+2} \setminus A_{i+1})$ that are requested in batch i.

By the definition of the BIDIRECTIONAL model, $\mathcal{A}$ could have moved requests in batch i to batch $i-1$ or to batch $i+1$. Therefore requests in batch i must be in $A_i \cup A_{i+1} \cup D_i$ or in $D_{i-1} \cup A_{i-1}$ or in $D_{i+1} \cup A_{i+2}$. Hence $\mathcal{B}$ serves all the requests in σ_0. It follows that

$$cost(\mathcal{B}(\sigma_0)) \leq \sum_{i=0}^{\lfloor n/r \rfloor} (|A_{i+1} \setminus A_i| + |A_{i+2} \setminus A_{i+1}| + |A_{i-1} \setminus A_i| + |D_{i-1}| + |D_i| + |D_{i+1}|) \tag{4.3}$$

$|D_{-1}| = 0$, $|D_{\lfloor n/r \rfloor + 1}| = 0$, $|A_{-1} \setminus A_0| = 0$, and $|A_{\lfloor n/r \rfloor + 2} \setminus A_{\lfloor n/r \rfloor + 1}| = 0$.

Observe that:

(i) $\sum_{i=0}^{\lfloor n/r \rfloor} |D_{i-1}| \leq \sum_{i=0}^{\lfloor n/r \rfloor} |D_i|$

(ii) $\sum_{i=0}^{\lfloor n/r \rfloor} |D_{i+1}| \leq \sum_{i=0}^{\lfloor n/r \rfloor} |D_i|$

(iii) $\sum_{i=0}^{\lfloor n/r \rfloor} |A_{i+2} \setminus A_{i+1}| \leq \sum_{i=0}^{\lfloor n/r \rfloor} |A_{i+1} \setminus A_i|$

We show that:

(iv) $\sum_{i=0}^{\lfloor n/r \rfloor} |A_{i-1} \setminus A_i| \leq \sum_{i=0}^{\lfloor n/r \rfloor} |A_{i+1} \setminus A_i|$

Let p be a page in $A_{i-1} \setminus A_i$. p gets evicted some time before the end of batch $i-1$. Let $j < i-1$ be the index of the batch when p was last brought into $A's$ cache. Therefore $p \in A_{j+1} \setminus A_j$. Hence for every page eviction that contributes 1 unit to the left hand side of (iv), there is an earlier page admission to cache that also contributes 1 unit to the right hand side of (iv). Totalling over all pages proves inequality (iv).

By inequalities (i),(ii), (iii), and (iv)

$$\sum_{i=0}^{\lfloor n/r \rfloor} |A_{i+1} \setminus A_i| + |A_{i+2} \setminus A_{i+1}| + |A_{i-1} \setminus A_i| + |D_{i-1}| + |D_i| + |D_{i+1}| \leq 3 \sum_{i=0}^{\lfloor n/r \rfloor} (|A_{i+1} \setminus A_i| + |D_i|) \tag{4.4}$$

By inequalities 4.2, 4.3, and 4.4, it follows that $cost(\mathcal{B}(\sigma_0)) \leq 3 \cdot cost(\mathcal{A}(\sigma_0))$. ■

Corollary 4.1 *Caching algorithm BMIN is a* 3*-approximation in the* BIDIRECTIONAL *model if optional caching (Definition 1.2) is used.*

The result follows immediately by our previous result (Theorem 4.5), since *BMIN* is an optimal batch-processing algorithm in the BIDIRECTIONAL model.

4.8 UNIDIRECTIONAL Model Results

4.8.1 Introduction

Recall that the UNIDIRECTIONAL model simulates weak dependencies between requests such that the delay constraints are to delay an element no more than r positions (Definition 4.8). Therefore no limits are placed on the forward direction, so that even the last request in the initial request sequence can in the first position in the final request sequence.

4.8.2 A Reordering Scheme for the General Case

No Reordering (NR) is an a reordering algorithm that performs no moves on the initial request sequence. We now analyze the approximation factor achieved by reordering scheme (NR, LFD) in the UNIDIRECTIONAL model.

We first prove an upper bound on the approximation factor of reordering scheme (NR, LFD).

Theorem 4.6 *In the* UNIDIRECTIONAL *model, the approximation factor of the reordering scheme* (NR, LFD) *is smaller than* $2r + 1$.

Proof:

Let σ_0 denote the initial request sequence. Let M be the sequence of moves that some optimal reordering scheme (OPT, LFD) performs. By Theorem 4.1 we can assume that M is valid (Definition 4.10) and that the maximum number of moves in M that pairwise overlap one another is equal to r, the reordering threshold.

Let S be the set of moves in M. Consider the moves in S as sorted in **increasing order** of their **from-coordinate** (Definition 4.1). Partition S into sets S_1, S_2, ... by the following process. Let $m_{1,1}$ be the first move in S, in left-to-right order. $S_1 = \{m \in S : m \text{ and } m_{1,1} \text{ are overlapping moves}\}$. By convention $m_{1,1} \in S_1$. The process continues: $m_{1,2}$ is the first move in $S \setminus S_1$ in left-to-right order and $S_2 = \{m \in S \setminus S_1 : m \text{ and } m_{1,2} \text{ are overlapping moves}\}$. At step i, let $m_{1,i}$ be the first move in set $S \setminus (\bigcup_{j=1}^{j=i-1} S_j)$, in left-to-right order. $S_i = \{m \in S \setminus (\bigcup_{j=1}^{j=i-1} S_j) : m \text{ and } m_{1,i} \text{ are overlapping moves}\}$. Note that $m_{1,i} \in S_i$. The partition process stops at step i if $S = (\bigcup_{j=1}^{j=i-1} S_j)$. Figure 4.3 shows a typical partition.

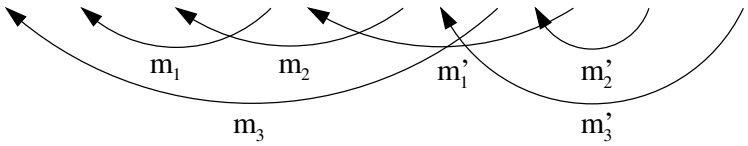

Figure 4.3: A typical partition when $r = 3$. $S_1 = \{m_1, m_2, m_3\}$. $S_2 = \{m_1', m_2', m_3'\}$.

Recall moves $m_{1,i}$ and $m_{1,i+1}$ are the first moves in left-to-right order in $S \setminus (\bigcup_{j=1}^{j=i-1} S_j)$ and $S \setminus (\bigcup_{j=1}^{j=i} S_j)$, respectively. Observe that the from-coordinate of $m_{1,i+1}$ is larger than the from-coordinate of $m_{1,i}$ (i). Since $m_{1,i+1} \notin S_i$, $m_{1,i+1}$ does not overlap $m_{1,i}$ (ii). Using (i) and (ii) above it follows that the to-coordinate of $m_{1,i+1}$ is larger than the from-coordinate of $m_{1,i}$. Therefore the ranges of $m_{1,i+1}$ and $m_{1,i}$ are disjoint and the range of $m_{1,i+1}$ is to the right of the range of $m_{1,i}$. Repeating this argument for increasing values of i it follows that for any i, j, $i \neq j$ the ranges of moves $m_{1,i}$ and $m_{1,j}$ are disjoint.

As mentioned before we can assume OPT performs only moves that are effective with respect to M (see Definition 4.14 and the observation following it). Hence we can apply

Lemma 4.2 to move $m_{1,i}$. *LFD* faults at least once in the range of $m_{1,i}$ in the final ordering σ_f. Since for any i, j, $i \neq j$ the ranges of moves $m_{1,i}$ and $m_{1,j}$ are disjoint, the cost of processing σ_f using *LFD* is larger or equal to the number of S_i sets (iii).

Recall that there are at most r moves that pairwise overlap in any part of σ. Observe $|S_i| \leq r$ and the number of S_i sets is larger or equal to $\frac{|S|}{r}$. By (iii) $cost(LFD(OPT(\sigma_0))) \geq \frac{|S|}{r}$ (iv). By Theorem 4.2 each move can reduce the *LFD* cost by at most 2. The cost of reordering scheme (NR, LFD) is equal to *LFD*'s cost on the initial request sequence. Therefore $cost(LFD(OPT(\sigma_0))) \geq cost(LFD(\sigma_0)) - 2|S| = cost(LFD(NR(\sigma_0))) - 2|S|$. Apply this inequality to get that:

$$\frac{cost(LFD(NR(\sigma_0)))}{cost(LFD(OPT(\sigma_0)))} \leq \frac{cost(LFD(OPT(\sigma_0))) + 2|S|}{cost(LFD(OPT(\sigma_0)))} = 1 + \frac{2|S|}{cost(LFD(OPT(\sigma_0)))}$$

By (iv)

$$\frac{cost(LFD(NR(\sigma_0)))}{cost(LFD(OPT(\sigma_0)))} \leq 2r + 1, \forall \sigma_0$$

which proves our result. ■

The next result shows a lower bound on the approximation factor of (NR, LFD).

Theorem 4.7 *In the* UNIDIRECTIONAL *model, the approximation factor of the reordering scheme* (NR, LFD) *is larger than* $2r + 1$.

Proof:

We create three request sequences σ_1, σ_2, and σ_3 such that *LFD*'s cache configuration

is the same at the end of request sequence $\sigma_1\sigma_2\sigma_3$ as it was in its beginning. We create an arbitrarily long request sequence σ_0 by repeating $\sigma_1\sigma_2\sigma_3$ such that

$$\lim_{n\to\infty} \frac{cost(LFD(NR(\sigma_0)))}{cost(LFD(OPT(\sigma_0)))} = 2r + 1$$

where (OPT, LFD) is an optimal reordering scheme.

Let σ_{b_1} be the request sequence 2 3 ... k 1 2 3 ... k $(k+1)$. Let $\sigma_1 = (k+1)(\sigma_{b_1})^r$, that is page $(k+1)$ followed by r copies of σ_{b_1}.

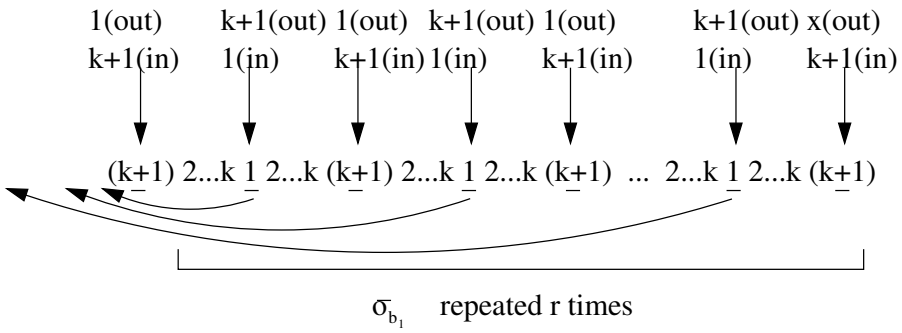

Figure 4.4: *LFD* serves σ_1. Faults are indicated by dashes. Arrows indicate the moves that *OPT* performs. Initial *LFD* cache configuration: $\{1, 2, \ldots k\}$. Final *LFD* cache configuration: $\{1, 2, \ldots k, k+1\} \setminus \{x\}$ where x is one of 1, 2, ... k.

Let pages 1, 2, ... and k be in *LFD*'s cache before it starts serving σ_1. See Figure 4.4 for details. Note that notation **x(in)** and **y(out)** on a fault means page x is brought into *LFD*'s cache and page y is evicted from *LFD*'s cache.

LFD faults on request $\sigma_1(1)$. In each of the r copies of σ_{b_1} *LFD* faults twice: once one the request for page 1 and once on the request for page $k+1$. See Figure 4.4 for more details. We get that $cost(LFD(NR(\sigma_1))) = 2r + 1$. *OPT* performs the following r moves on σ_1: for

each copy, the request for page 1 is moved to the first position. Let the resulting sequence be denoted by σ'_1 (Figure 4.5).

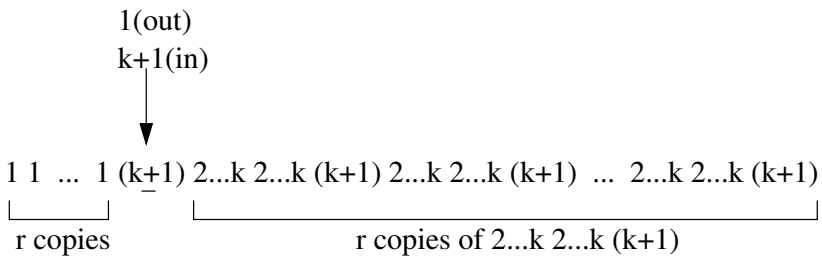

Figure 4.5: *LFD* serves σ'_1. Faults are indicated by dashes. Initial *LFD* cache configuration: $\{1, 2, \ldots k\}$. Final *LFD* cache configuration: $\{2, \ldots k, k+1\}$.

When serving σ'_1 *LFD* faults only on the first request for $k+1$. Observe that the r requests for page 1 that were moved to the front are now hits (1 is in cache at the beginning). The fault occurs when $k+1$ is first requested (Figure 4.5). *LFD* evicts 1 which is never requested again in the new request sequence. All other requests are for pages already in cache so *LFD* never faults again. It holds that $cost(LFD(OPT(\sigma_1))) = 1$ so $\frac{cost(LFD(NR(\sigma_1)))}{cost(LFD(OPT(\sigma_1)))} = 2r + 1$.

This result assumes the initial cache configuration is $\{1, 2, \ldots k\}$. Since we need an empty initial cache configuration, let $\sigma_{e_1} = 1\ 2\ \ldots\ \mathrm{k}\ \sigma_1$ and $\sigma'_{e_1} = 1\ 2\ \ldots\ \mathrm{k}\ \sigma'_1$. We obtain that $\frac{cost(LFD(NR(\sigma_{e_1})))}{cost(LFD(OPT(\sigma_{e_1})))} = \frac{2r+1+k}{k}$. Note that repeating the request pattern in σ_{e_1} with totally different sets of pages does not help; the final cost ratio will still be $\frac{2r+1+k}{k}$. To get a cost ratio of $2r + 1$ we introduce two other request sequences σ_2 and σ_3 to wrap around; in the following analysis, the *LFD* cache configuration at the end of σ_1 is the initial *LFD* cache configuration for σ_2, the *LFD* cache configuration at the end of σ_2 is the initial *LFD* cache configuration for σ_3 and the *LFD* cache configuration at the end of σ_3 is equal to $\{1, 2, \ldots k\}$

which is the initial *LFD* cache configuration for σ_1. Due to this wrap-around feature we first analyze the cost of *LFD* on σ_1, σ_2, and σ_3 separately, and combine these costs in the analysis of the final request sequence $\sigma_0 = 1\ 2\ \ldots\ k(\sigma_1\sigma_2\sigma_3)^m$.

We define σ_2. Let σ_{b_2} be the request sequence $2\ \ldots\ k\ (k+1)\ 2\ \ldots\ k\ (k+2)$ and let $\sigma_2 = (k+2)\ (\sigma_{b_2})^r$.

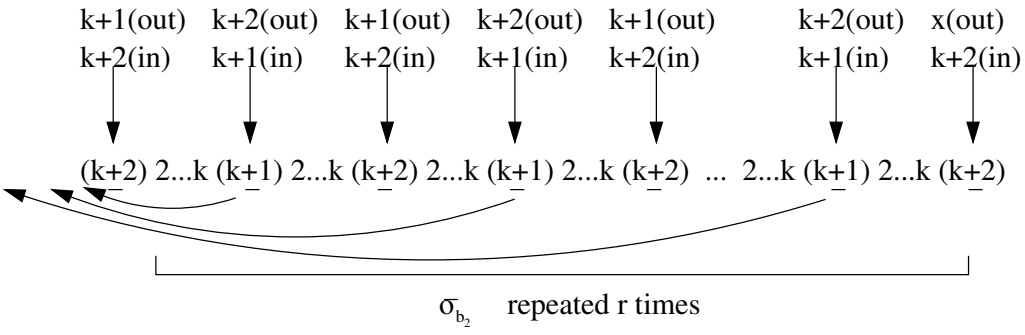

Figure 4.6: *LFD* serves σ_2. Faults are indicated by dashes. Arrows indicate the moves that *OPT* performs. Initial *LFD* cache configuration: $\{2, \ldots k, k+1\}$. Final *LFD* cache configuration: $\{2, \ldots k, k{+}1, k{+}2\}\backslash\{x\}$ where x is one of 2, ... k, k+1.

Let pages 2, 3, ..., k, and $(k{+}1)$ be in *LFD*'s cache before it starts serving σ_2 (Figure 4.6). *LFD* faults on request $\sigma_2(1)$. In each of the r copies of σ_{b_2} it faults once on request for page $k+1$ and once on request for page $k+2$ (see Figure 4.6 for more details). Therefore $cost(LFD(NR(\sigma_2))) = 2r + 1$.

OPT performs the following r moves on σ_2: for each copy relocate request for page $k{+}1$ to the first position. The new sequence of requests is denoted σ_2'. By a similar argument we get that *LFD* only faults once when serving σ_2' (Figure 4.7). Hence $cost(LFD(OPT(\sigma_2))) = 1$.

Let σ_{b_3} be the request sequence $2\ \ldots\ k\ (k+2)\ 2\ \ldots\ k\ 1$ and let $\sigma_3 = 1\ (\sigma_{b_3})^r$. Let pages

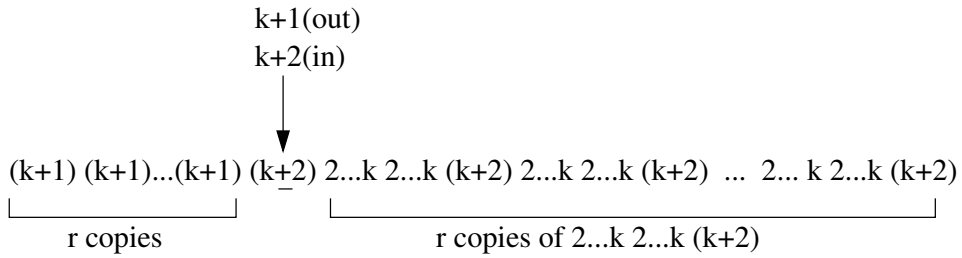

Figure 4.7: *LFD* serves σ_2. Faults are indicated by dashes. Initial *LFD* cache configuration: $\{2 \ldots k, k+1\}$. Final *LFD* cache configuration: $\{2, \ldots k, k+2\}$.

2, 3, ..., k, and $(k+2)$ be in *LFD*'s cache before it starts serving σ_3. By an argument similar to those for σ_1 and σ_2, $cost(LFD(NR(\sigma_3))) = 2r + 1$ (Figure 4.8).

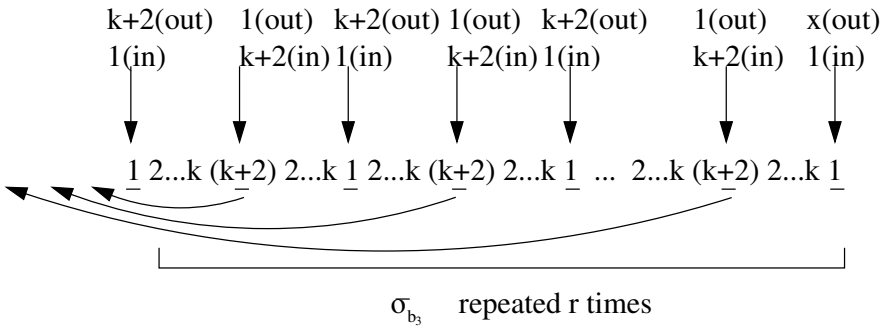

Figure 4.8: *LFD* serves σ_3. Faults are indicated by dashes. Arrows indicate the moves that *OPT* performs. Initial *LFD* cache configuration: $\{2, \ldots k, k+2\}$. Final *LFD* cache configuration: $\{1, 2, \ldots k, k+2\} \setminus \{x\}$ where x is one of 2, 3, ... k, k+2.

OPT performs the following r moves on σ_3: for each copy relocate request for page $k+2$ to the first position. The resulting request sequence is denoted by σ_3'. By an argument similar to those for σ_1 and σ_2, $cost(LFD(OPT(\sigma_3))) = 1$ (Figure 4.9).

Note that if *LFD* serves σ_2 immediately after serving σ_1, then the last evicted page in σ_1 is page 1 (Figures 4.4 and 4.6). Therefore the cache configuration at end of σ_1 is

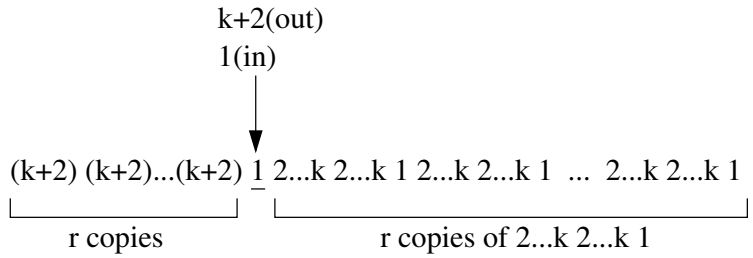

Figure 4.9: *LFD* serves σ_3'. Faults are indicated by dashes. Initial *LFD* cache configuration: $\{2, \ldots k, k+2\}$. Final *LFD* cache configuration: $\{1, 2, \ldots k\}$.

$\{2, 3, \ldots, k, k+1\}$, the same as the initial cache configuration required for σ_2. Similarly if *LFD* serves σ_3 just after it finishes serving σ_2, then the final eviction in σ_2 is that of page $k+1$ ($x = k+1$) (Figures 4.6 and 4.8). Hence the configuration at the end of σ_2 is $\{2, 3, \ldots, k, k+2\}$ and is identical to the configuration required at the beginning of σ_3. By a similar argument if *LFD* serves σ_1 again after serving σ_3, its cache configuration at the end of σ_3 matches the initial configuration for σ_1 (Figures 4.4 and 4.8).

Let $\sigma_0 = 1\ 2\ \ldots\ k(\sigma_1\sigma_2\sigma_3)^m$ ($1\ 2\ \ldots\ k$ followed by m copies of $\sigma_1\sigma_2\sigma_3$). Note that σ_0 is arbitrarily long ($m \to \infty$). The first k requests in σ_0 ensure that *LFD*'s initial cache configuration is $1\ 2\ \ldots\ k$ as required by σ_1 and by σ_1'. Using the previous observations and the *LFD* analysis on each individual subsequence we conclude that *LFD* faults $2r+1$ times on each of σ_1, σ_2, and σ_3. Therefore $cost(LFD(NR(\sigma_0))) = k + 3(2r+1)m$.

By Figures 4.4, 4.6, and 4.8 we observe that the final cache configuration for σ_1' is the same as the initial cache configuration for σ_2'. Similarly the final cache configurations for σ_2' and σ_3' are identical to the initial cache configurations for σ_3' and σ_1' respectively. By the analysis of *LFD* when serving each subsequence we obtain that *LFD* faults exactly once on

each of σ'_1, σ'_2, and σ'_3. Hence $cost(LFD(OPT(\sigma_0))) = k + 3m$.

$$\lim_{n\to\infty} \frac{cost(LFD(NR(\sigma_0)))}{cost(LFD(OPT(\sigma_0)))} = \lim_{m\to\infty} \frac{k + 3(2r+1)m}{k + 3m} = 2r + 1$$

The result follows. ■

Theorems 4.6 and 4.7 yield that the reordering scheme (NR, LFD) is a tight $(2r + 1)$-approximation.

Theorem 4.8 *In the* UNIDIRECTIONAL *model, the reordering scheme* (NR, LFD) *has an approximation factor of* $(2r + 1)$.

4.8.3 Optimal Ordering for a Special Case

The goal of this section is to develop an optimal ordering for the special case when both the cache size, k, and the reordering threshold, r, are equal to 1. The reader should then focus on this case, although specific note of the values of r and k will be made in each theorem or lemma.

First we prove some properties of an optimal reordering scheme. We describe a dynamic programming algorithm that computes the *LFD* cost of the optimal ordering. We show how to compute the optimal ordering as well.

Recall the definition of a run (see Definition 1.5). We make the following observation about the cost of processing a fixed request sequence using the eviction policy *LFD*.

Observation 4.2 *Let σ be a request sequence. If the cache size is equal to* 1*, then the cost of processing σ using eviction policy* LFD *is equal to the number of runs in σ.*

The proof follows immediately by noticing that, when *LFD* serves σ it only faults on the first request of each run in σ. Before we present the following definition, recall that all moves are implicitly forward moves (Lemma 4.1).

Let σ be a request sequence. We say that $m = (j, i)$ is a move of

- **type ①** if m relocates request $\sigma(j) = x$ to a position adjacent to a preceding copy of x
- **type ②** if m relocates request $\sigma(j) = x$ to fall between two adjacent requests, one for y and the other for z, where $y \neq z$, $y \neq x$, $z \neq x$ (if $i = 1$, then $\sigma(j) = x$ is moved in front of $z = \sigma(1)$ and $z \neq x$)
- **type ③** if m relocates request $\sigma(j) = x$ between requests $\sigma(i-1) = y$ and $\sigma(i) = y$, where $y \neq x$.

For an illustration, see Figure 4.10.

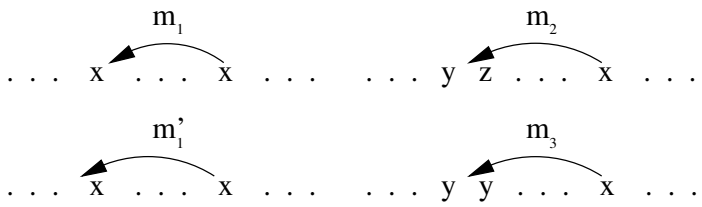

Figure 4.10: Type ① moves: m_1 and m_1'. Type ② move: m_2. Type ③ move: m_3

Lemma 4.3 *Let the cache size k be equal to 1 and the reordering threshold r be an arbitrary positive integer. Let σ be a request sequence and j be an index in σ ($1 \leq j \leq n$). Let $m_1 = (j, i_1)$, $m_2 = (j, i_2)$, and $m_3 = (j, i_3)$ be moves of type ①, ②, and ③, respectively.*

For a fixed index j, all moves of a given type have the same value and the following equality holds for the values of each individual move

$$v(m_1) = v(m_2) + 1 = v(m_3) + 2$$

Proof:

Let $m = (j, i)$ be a move that relocates $\sigma(j) = x$ to position i. We consider three cases: $\sigma(j-1) = \sigma(j+1) \neq x$ (case 1), at least one of $\sigma(j-1)$ and $\sigma(j+1)$ is equal to x (case 2), and $\sigma(j-1)$, $\sigma(j+1)$, and x are all distinct (case 3).

Consider case 1 where $\sigma(j-1) = \sigma(j+1) \neq x$. Note that $\sigma(j) = x$ is a run of length 1. One of the effects of move m is to merge the two runs that contain $\sigma(j-1)$ and $\sigma(j+1)$, respectively (i).

If m is of type ①, then $\sigma(j) = x$ is joined to a previous run of x. By (i) and Observation 4.2, $v(m) = 2$. If m is of type ②, no other runs are merged or broken. By (i) and Observation 4.2, $v(m) = 1$. If m is of type ③, then $\sigma(j) = x$ breaks the run containing $\sigma(i-1)$ and $\sigma(i)$. By (i) and Observation 4.2, $v(m) = 0$. This implies the equalities stated in the lemma hold for case 1.

In case 2, at least one of $\sigma(j-1)$ and $\sigma(j+1)$ is equal to x. By a similar analysis, if m is of type ①, $v(m) = 0$. If m is of type ②, $v(m) = -1$. If m is of type ③, $v(m) = -2$. The

equalities hold.

Consider case 3 where $\sigma(j-1)$, $\sigma(j+1)$, and x are all distinct. By a similar analysis, if m is of type ①, $v(m) = 1$. If m is of type ②, $v(m) = 0$. If m is of type ③, $v(m) = -1$. The equalities hold. ■

We now define two cases of type ① moves called **A** and **B**, and one case of type ② moves called **C**. Let σ be a request sequence and $\sigma(j) = x$ be a run of length 1 in σ ($\sigma(j-1) \neq x$ and $\sigma(j+1) \neq x$). We say that $m = (j, i)$ is a move of

- **type A** if $\sigma(j-1) \neq \sigma(j+1)$ and $\sigma(j) = x$ is moved to a position adjacent to a preceding copy of x ($\sigma(i) = x$ or $\sigma(i-1) = x$ or both)
- **type B** if $\sigma(j-1) = \sigma(j+1)$ and $\sigma(j) = x$ is moved to a position adjacent to a preceding copy of x
- **type C** if $\sigma(j-1) = \sigma(j+1)$ and $\sigma(j) = x$ is moved to fall between two adjacent requests, one for y and the other for z, where y and z are distinct from x and from one another (if $i = 1$, then $\sigma(j) = x$ is moved in front of $z = \sigma(1)$ and $z \neq x$)

We show that any positive move must be either of type A, B, or C.

Lemma 4.4 *Let the cache size k be equal to 1 and the reordering threshold r be an arbitrary positive integer. All positive moves that are applied to σ are moves of type A, B or C. Furthermore, moves of type A, B, and C have value 1, 2, and 1, respectively.*

Proof:

Recall the proof of Lemma 4.3. If m is of type $\boxed{\text{A}}$, then m is of type ① in case 3 of that proof. Hence $v(m) = 1$. If m is of type $\boxed{\text{B}}$, then m is of type ① in case 1 of that proof. Therefore $v(m) = 2$. If m is of type $\boxed{\text{C}}$, then m is of type ② in case 1 of that proof, so $v(m) = 1$.

Notice that these 3 cases are the only ones among the 9 combinations presented in the proof of Lemma 4.3 such that $v(m) > 0$. ■

We describe some specific types of moves (**I**, **II**, and **III**) that will be of direct use in developing an optimal ordering. Let σ be a request sequence. Consider all moves of type $\boxed{\text{A}}$ that relocate request $\sigma(j)$. The move among them, of smallest range, is a move of type I. Similarly, a move $m = (j, i)$ of type II (respectively III) is the move whose range is smallest among all moves of type $\boxed{\text{B}}$ (respectively $\boxed{\text{C}}$) that relocate request $\sigma(j)$.

Definition 4.15 *Let σ be a request sequence. Let $\sigma(j) = x$ be a run of length 1 ($\sigma(j-1) \neq x$ and $\sigma(j+1) \neq x$). Let $m = (j, i)$ be a move that is applied to σ. We say m is of*

- **type I** *if $\sigma(j-1) \neq \sigma(j+1)$ and $\sigma(j) = x$ is moved to next to the closest preceding x*
- **type II** *if $\sigma(j-1) = \sigma(j+1)$ and $\sigma(j) = x$ is moved next to the closest preceding x*
- **type III** *if $\sigma(j-1) = \sigma(j+1)$ and $\sigma(i : j-1)$ is a run and $\sigma(i-1) \neq \sigma(j) = x$ (if $i = 1$, then $\sigma(j)$ is moved in front of the run $\sigma(1 : j-1)$)*

See Figure 4.11 for an illustration of type I, II, and III moves.

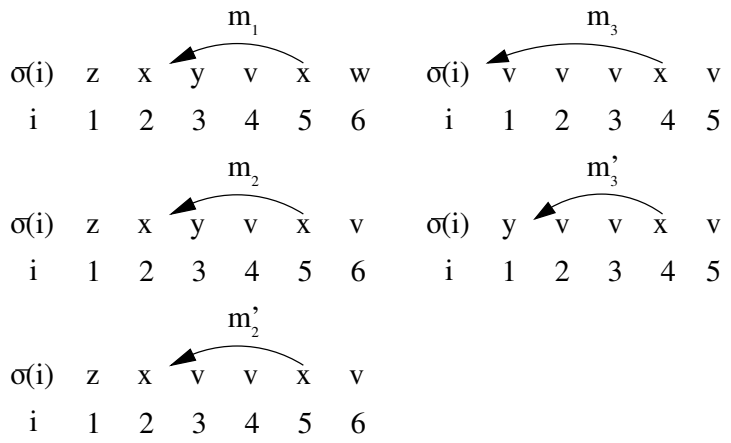

Figure 4.11: Type I move: $m_1 = (5, 3)$. Type II moves: $m_2 = (5, 3)$, $m_2' = (5, 3)$. Type III moves: $m_3 = (4, 1)$, $m_3' = (4, 2)$

We now introduce the concept and notation for a **value of a move relative to a sequence of moves** which will be used in the proof of Theorem 4.9. Recall that σ_M denotes the ordering of σ after the sequence of moves M has been performed; hence $\sigma_{M,m}$ is the ordering when move m is subsequently performed and $\sigma_{M,N}$ is the ordering that results by applying the sequence of moves M, then the sequence of moves N to σ.

Definition 4.16 *Let m be a move and M be a sequence of moves such that m is not in M. By $v_M(m)$ we denote the* **value of m relative to M**, *that is the value of move m after all moves in M have been performed.*

Formally, $v_M(m) = cost(LFD(\sigma_M)) - cost(LFD(\sigma_{M,m}))$.

Let M be the sequence of moves performed by reordering algorithm $\mathcal{E}_1$.

Theorem 4.9 *Let r and k be equal to 1 and σ be a request sequence. For any reordering scheme $(\mathcal{E}_1, LFD)$, there exists a reordering scheme $(\mathcal{E}_2, LFD)$ such that the request reorder-*

ing algorithm $\mathcal{E}_2$ uses only moves of type I, II, or III and the cost of LFD is smaller on $\mathcal{E}_2(\sigma)$ than on $\mathcal{E}_1(\sigma)$:

$$cost(LFD(\mathcal{E}_2(\sigma))) \leq cost(LFD(\mathcal{E}_1(\sigma))).$$

Proof:

Let M be the sequence of moves performed by reordering algorithm $\mathcal{E}_1$. By Theorem 4.1 ($r = 1$) we can assume that no moves in M overlap one another. By the definition of overlapping moves (Definition 4.11), all ranges of moves in M are pairwise disjoint. Therefore, no matter in what order the moves in M are performed, the final ordering is the same. Hence for the rest of this proof, M will denote the set of moves that $\mathcal{E}_1$ performs.

We show how to modify set M into M' such that M' contains only moves of type I, II, and III and $cost(LFD(\sigma_{M'})) \leq cost(LFD(\sigma_M))$. If reordering algorithm $\mathcal{E}_2$ performs the moves in M', the result follows.

The modification process is as follows: let $m = (j, i)$ be the first move in left to right order (by from-coordinate) in M such that m is not of type I, II, or III. If there is no such move, we are done ($M' = M$).

If there exists such move m, either m is of type $\boxed{\text{A}}$ (case 1), $\boxed{\text{B}}$ (case 2) or $\boxed{\text{C}}$ (case 3), or it is not of any of these types (case 4).

Focus on case 1 where m is of type $\boxed{\text{A}}$. Since m is not of type I, there exists a move $m' = (j, i')$ ($i' > i$) such that m' is of type I (this implies m' is also of type $\boxed{\text{A}}$). Assume that all moves in $M \setminus \{m\}$ have been performed. Recall the notation introduced in Definition 4.16.

Apply Lemma 4.4 to moves m and m' and obtain that $v_{M\setminus\{m\}}(m') = v_{M\setminus\{m\}}(m) = 1$. $M' = M \setminus \{m\} \cup \{m'\}$ contains one less undesirable move and $cost(LFD(\sigma_{M'})) = cost(LFD(\sigma_M))$. Since σ_M satisfies the reordering constraints and the range of m' is a subset of the range of m, $\sigma_{M'}$ also satisfies the reordering constraints.

Case 2, where m is of type B, is very similar to case 1.

In case 3, m is of type C. Let h be the smallest index less than j such that $\sigma(h : j-1) = v^{j-h}$ where $v = \sigma(j-1)$. Since m is of type C, $y = \sigma(i-1) \neq z = \sigma(i)$. Since y and z cannot be both equal to v, $h \geq i$. Since m is not of type III, $h \neq i$. Hence $h > i$.

Move $m' = (j, h)$ is either of type II if $\sigma(h-1) = \sigma(j)$ or of type III if $\sigma(h-1) \neq \sigma(j)$ (implicitly m' is also of type B or C, respectively).

Apply Lemma 4.4 to moves m and m' and obtain that $v_{M\setminus\{m\}}(m) = 1$ and $v_{M\setminus\{m\}}(m') \in \{1, 2\}$. $M' = M \setminus \{m\} \cup \{m'\}$ contains one less undesirable move and $cost(LFD(\sigma_{M'})) \leq cost(LFD(\sigma_M))$. Since σ_M satisfies the reordering constraints and the range of m' is a subset of the range of m, $\sigma_{M'}$ also satisfies the reordering constraints.

Focus on case 4 where m is not of type A, nor B, nor C. Apply Lemma 4.4 to move m and obtain $v_{M\setminus\{m\}}(m) \leq 0$. $M' = M \setminus \{m\}$ contains one less undesirable move and $cost(LFD(\sigma_{M'})) \leq cost(LFD(\sigma_M))$. Obviously $\sigma_{M'}$ still satisfies the reordering constraints.

By repeating the transformation process, we get a set M' that uses only moves of type I, II or III such that $cost(LFD(\sigma_{M'})) \leq cost(LFD(\sigma_M))$. Let reordering algorithm $\mathcal{E}_2$ perform the moves in M'. The result follows. ■

Corollary 4.2 *Let the cache size k and reordering threshold r both be equal to* 1*. There exists an optimal reordering scheme using LFD such that the moves it performs are of type I, II or III.*

For this special case (both r and k are equal to 1), we give a dynamic programming algorithm (*DP*) to compute the optimal cost, which is the *LFD* cost of the optimal ordering. Let σ be a request sequence. Let $C(j)$ denote the *LFD* cost of the optimal ordering of the prefix $\sigma(1:j)$. Let $C_1[j]$ denote the *LFD* cost of the optimal ordering of $\sigma(1:j)$ if $\sigma(j)$ is **not** moved. Let $C_2[j]$ be the *LFD* cost of the optimal ordering of $\sigma(1:j)$ if $\sigma(j)$ is moved next to its closest copy (we call this a **copy** move). Let $C_3[j]$ be the *LFD* cost of the optimal ordering of $\sigma(1:j)$ if $\sigma(j)$ is moved immediately before the run containing $\sigma(j-1)$ (we call this an **evacuation** move). Observe that it is possible that a move is both a copy move and and an evacuation move. For instance $m = (4, 3)$ applied to $\sigma = x\ y\ x\ y$ is such a move.

The *DP* algorithm computes $C_1[j]$, $C_2[j]$ and $C_3[j]$, $\forall j \in \{1, 2, \ldots, n\}$ and then computes $C(n)$ by formula 4.5 and returns it. Note that $C(n)$ is the *LFD* cost of the optimal ordering. After the proof of Theorem 4.10 we explain how to obtain the optimal ordering.

$$C(j) = \min\{C_1[j], C_2[j], C_3[j]\}, \quad \forall j \in \{1, 2, \ldots, n\} \tag{4.5}$$

First we give a formula for $C_1[j]$.

$$C_1[j] = \begin{cases} 1 & \text{if } j = 1 \quad (1.1) \\ 1 & \text{if } j = 2 \ \&\ \sigma(1) = \sigma(2) \quad (1.2) \\ 2 & \text{if } j = 2 \ \&\ \sigma(1) \neq \sigma(2) \quad (1.3) \\ C_1[j-1] & \text{if } j \geq 3 \ \&\ \sigma(j-1) = \sigma(j) \quad (1.4) \\ \min\{C_1[j-1]+1, C_2[j-1]+1\} & \text{if } j \geq 3 \ \&\ \sigma(j-1) \neq \sigma(j) \quad (1.5) \\ & \ \&\ \sigma(j-2) \neq \sigma(j) \\ \min\{C_1[j-1]+1, C_2[j-1], C_3[j-1]\} & \text{if } j \geq 3 \ \&\ \sigma(j-1) \neq \sigma(j) \quad (1.6) \\ & \ \&\ \sigma(j-2) = \sigma(j) \end{cases}$$

See Figure 4.12 for an illustration of lines (1.5) and (1.6) in the above formula. Note that z may be equal to v, but the essential is that $z \neq x$.

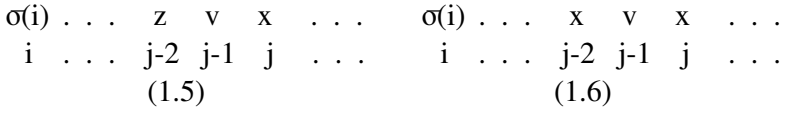

Figure 4.12: Cases (1.5) and (1.6) in the $C_1[j]$ formula

We now focus on the formula for $C_2[j]$. Let g be the index of the closest preceding copy of $\sigma(j)$. If such an index does not exist, then $g = 0$. Observe that $g = j - 1$ implies $\sigma(j-1) = \sigma(j)$.

Let $c_{g,j}$ denote the cost incurred by *LFD* when processing request sequence $\sigma' = \sigma(g : j)$.

$$C_2[j] = \begin{cases} +\infty & \text{if } j = 1 \qquad (2.1) \\ +\infty & \text{if } j \geq 2 \ \&\ g = 0 \qquad (2.2) \\ C_1[j] & \text{if } j \geq 2 \ \&\ g = i - 1 \qquad (2.3) \\ C_1[g] + c_{g+1,j-1} & \text{if } j \geq 2 \ \&\ 1 \leq g \leq j-2 \qquad (2.4) \\ & \ \&\ \sigma(g+1) = \sigma(g+2) \\ \min\{C_1[g] + c_{g+1,j-1}, C_2[g+1] + c_{g+2,j-1}\} & \text{if } j \geq 2 \ \&\ 1 \leq g \leq j-2 \qquad (2.5) \\ & \ \&\ \sigma(g+1) \neq \sigma(g+2) \end{cases}$$

See Figure 4.13 for an illustration of cases (2.4) and (2.5) in the above formula.

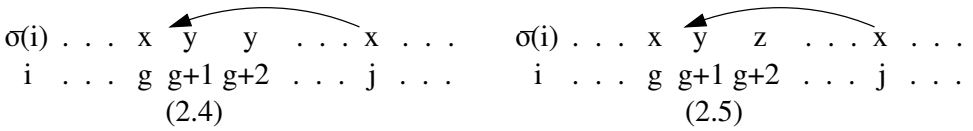

Figure 4.13: Cases (2.4) and (2.5) in the $C_2[j]$ formula

We now present the formula for $C_3[j]$. Let h be the starting index of the run containing $\sigma(j-1)$. If $j = 1$ then by convention $h = 0$. Note that if $j \geq 2$ then $h \geq 1$. The $C_3[j]$ formula is presented as a four-part formula.

If $j = 1$ then $C_3[j] = +\infty$. (3.1)

If $j \geq 2$ and $h = 1$ then

$$C_3[j] = \begin{cases} 1 & \text{if } \sigma(j-1) = \sigma(j) \qquad (3.2) \\ 2 & \text{if } \sigma(j-1) \neq \sigma(j) \qquad (3.3) \end{cases}$$

If $j \geq 2$ and $h = 2$ then

$$C_3[j] = \begin{cases} 2 & \text{if } \sigma(j-1) = \sigma(j) \quad (3.4) \\ 2 & \text{if } \sigma(j-1) \neq \sigma(j) \ \& \ \sigma(1) = \sigma(j) \quad (3.5) \\ 3 & \text{if } \sigma(j-1) \neq \sigma(j) \ \& \ \sigma(1) \neq \sigma(j) \quad (3.6) \end{cases}$$

If $j \geq 2$ and $h \geq 3$ then

$$C_3[j] = \begin{cases} C_1[h] & \text{if } \sigma(j-1) = \sigma(j) \quad (3.7) \\ C_2[j] & \text{if } \sigma(j-1) \neq \sigma(j) \quad (3.8) \\ & \ \& \ \sigma(h-1) = \sigma(j) \\ \min\{C_1[h-1]+2, C_2[h-1]+2\} & \text{if } \sigma(j-1) \neq \sigma(j) \quad (3.9) \\ & \ \& \ \sigma(h-1) \neq \sigma(j) \\ & \ \& \ \sigma(h-2) \neq \sigma(j) \\ \min\{C_1[h-1]+2, C_2[h-1]+1, C_3[h-1]+1\} & \text{if } \sigma(j-1) \neq \sigma(j) \quad (3.10) \\ & \ \& \ \sigma(h-1) \neq \sigma(j) \\ & \ \& \ \sigma(h-2) = \sigma(j) \end{cases}$$

See Figure 4.14 for an illustration of cases (3.8), (3.9) and (3.10) in $C_3[j]$ formula. Note that z may be equal to y or v, but the essential is that $z \neq x$.

Observe that due to the above dependencies, the cost values must be computed in increasing order of j and at each step the computation order must be: first compute $C_1[j]$,

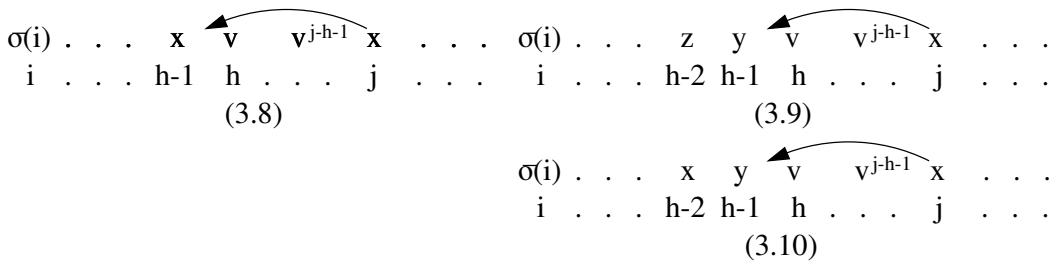

Figure 4.14: Cases (3.8), (3.9), and (3.10) in the $C_3[j]$ formula

then $C_2[j]$, and then $C_3[j]$.

Theorem 4.10 *The DP algorithm computes the LFD cost of an optimal ordering for the case when both the cache size k and reordering threshold r are equal to* 1.

Proof:

Let σ be the initial request sequence. The result follows immediately once we prove that $C(j)$ is the *LFD* cost of the optimal ordering of request sequence $\sigma(1:j)$. If $\sigma(j)$ is not moved, then the *LFD* cost of the optimal ordering of request sequence $\sigma(1:j)$ is equal to $C_1[j]$ (i).

If $\sigma(j)$ is moved (denote the move by $m = (j, i)$), by Corollary 4.2 we can assume that m is of type I, II, or III. Recall the definition of copy moves and evacuation moves that was presented after Corollary 4.2 and the definition of the moves of type I, II, and III (Definition 4.15). Observe that if m is of type I, then it is a copy move. If m is of type II, then it can be either a copy move or an evacuation move (recall there exist moves that are both copy moves and evacuation moves, i.e.: move m_2' of type II in Figure 4.11). If m is of type III, then it is an evacuation move. By definition of $C_2[j]$ and $C_3[j]$ it follows that, if $\sigma(j)$ is moved, then the *LFD* cost of the optimal ordering of request sequence $\sigma(1:j)$ is

equal to the minimum of $C_2[j]$ and $C_3[j]$ (ii). By (i) and (ii) above, formula 4.5 is correct.

We prove the correctness of the formulas for $C_1[j]$, $C_2[j]$, and $C_3[j]$, line by line. The proof is by induction on j. We assume the correctness of the formulas for $C_1[i]$, $C_2[i]$, and $C_3[i]$, $\forall i \in \{1, 2, \ldots, j-1\}$ and prove the correctness of the formulas for $C_1[j]$, $C_2[j]$, and $C_3[j]$.

Focus first on the $C_1[j]$ formula. The base cases are **lines (1.1) to (1.3)**. By inspection, all these formulas are correct. We demonstrate the correctness of the remaining lines (**(1.4) to (1.6)**) in the $C_1[j]$ formula.

Consider line (1.4) where $j \geq 3$, if $\sigma(j-1) = \sigma(j)$. By definition of $C_1[j]$, $\sigma(j)$ is not moved. Relocating $\sigma(j-1)$ forward is not a type I, II, or III move (iii). Let a be the *LFD* cost of the optimal ordering of $\sigma(1:j)$, when $\sigma(j)$ is not moved. Let b be the *LFD* cost of the optimal ordering of $\sigma(1:j)$, when both $\sigma(j-1)$ and $\sigma(j)$ are not moved. By (iii) and Corollary 4.2 we obtain that $a = b$. Since $\sigma(j-1) = \sigma(j)$, b is equal to the *LFD* cost of the optimal ordering of $\sigma(1:j-1)$, when $\sigma(j-1)$ is not moved. By the induction hypothesis, we obtain that $b = C_1[j-1]$. Since $C_1[j] = C_1[j-1]$, we get that $C_1[j] = a$, so line (1.4) is correct.

We now analyze lines (1.5) and (1.6) in the $C_1[j]$ formula (see Figure 4.12). Consider line (1.5) where $\sigma(j-1) \neq \sigma(j)$ and $\sigma(j-2) \neq \sigma(j)$. Since $\sigma(j)$ is not moved, a move involving $\sigma(j-1)$ cannot be of type II, nor III (iv). Let a be the *LFD* cost of the optimal ordering of $\sigma(1:j)$, when $\sigma(j)$ is not moved. Let b be the *LFD* cost of the optimal ordering of $\sigma(1:j)$, when both $\sigma(j-1)$ and $\sigma(j)$ are not moved. Let c be the *LFD* cost

of the optimal ordering of $\sigma(1:j)$, when $\sigma(j)$ is not moved and $\sigma(j-1)$ is moved next to its previous copy. Let d be the *LFD* cost of the optimal ordering of $\sigma(1:j-1)$, when $\sigma(j-1)$ is not moved. Let e be the *LFD* cost of the optimal ordering of $\sigma(1:j-1)$, when $\sigma(j-1)$ is moved next to its previous copy. By (iv) and Corollary 4.2, $a = \min\{b, c\}$. Since $\sigma(j-1) \neq \sigma(j)$, we get $b = d+1$. Since $\sigma(j-2) \neq \sigma(j)$, we obtain $c = e+1$. By the induction hypothesis, $d = C_1[j-1]$ and $e = C_2[j-1]$. Since $C_1[j] = \min\{C_1[j-1]+1, C_2[j-1]+1\}$, we get $C_1[j] = \min\{b, c\} = a$, which proves the correctness of line (1.5).

Now consider line (1.6) in the $C_1[j]$ formula, where $\sigma(j-1) \neq \sigma(j)$ and $\sigma(j-2) = \sigma(j)$. Since $\sigma(j)$ is not moved, the move involving $\sigma(j-1)$ cannot be of type I (v). Let a be the *LFD* cost of the optimal ordering of $\sigma(1:j)$, when $\sigma(j)$ is not moved. Let b_1 be the *LFD* cost of the optimal ordering of $\sigma(1:j)$, when both $\sigma(j-1)$ and $\sigma(j)$ are not moved. Let c_1 be the *LFD* cost of the optimal ordering of $\sigma(1:j)$, when $\sigma(j)$ is not moved and $\sigma(j-1)$ is moved next to its previous copy. Let d_1 be the *LFD* cost of the optimal ordering of $\sigma(1:j)$, when $\sigma(j)$ is not moved and $\sigma(j-1)$ is moved immediately before the run containing $\sigma(j-2)$. Let b_2 be the *LFD* cost of the optimal ordering of $\sigma(1:j-1)$, when $\sigma(j-1)$ is not moved. Let c_2 be the *LFD* cost of the optimal ordering of $\sigma(1:j-1)$, when $\sigma(j-1)$ is moved next to its previous copy. Let d_2 be the *LFD* cost of the optimal ordering of $\sigma(1:j-1)$, when $\sigma(j-1)$ is moved immediately before the run containing $\sigma(j-2)$.

By (v) and Corollary 4.2, $a = \min\{b_1, c_1, d_1\}$. Since $\sigma(j-1) \neq \sigma(j)$, we get $b_1 = b_2 + 1$. Since $\sigma(j-2) = \sigma(j)$, we obtain $c_1 = c_2$ and $d_1 = d_2$. By the induction hypothesis, $b_2 = C_1[j-1]$, $c_2 = C_2[j-1]$ and $d_2 = C_3[j-1]$. Since $C_1[j] = \min\{C_1[j-1]+1, C_2[j-1], C_3[j-1]\}$,

it follows that $C_1[j] = \min\{b_1, c_1, d_1\} = a$. Line (1.6) is correct.

Focus now on the correctness of the $C_2[j]$ formula. By inspection, the base case formula (**line (2.1)**) is correct. We prove the correctness of the remaining lines (**(2.2) to (2.5)**) in the $C_2[j]$ formula.

By definition of $C_2[j]$, $\sigma(j)$ is moved next to its closest preceding copy which is in some position g. If $g = 0$, there is no previous copy next to which $\sigma(j)$ can be moved, so $C_2[j]$ is set to $+\infty$ (**line (2.2)**). If $g = j - 1$, then $\sigma(j)$ is already next to its copy ($\sigma(j-1) = \sigma(j)$). The optimal cost is equal to $C_1[j]$ (**line (2.3)**).

Now consider lines (2.4) and (2.5) where $1 \leq g \leq j - 2$ (see Figure 4.13). Note that the closest preceding copy of $\sigma(j)$ is in position g in the initial request sequence. Since $r = 1$, no request can be bumped back more than 1 position. Therefore an optimal reordering scheme could not have first moved $\sigma(g)$ and afterwards moved $\sigma(j)$ next to the new position of $\sigma(g)$. Hence we can assume request $\sigma(g)$ remains in position g. Notice that an optimal reordering scheme might move request $\sigma(g+1)$ before moving $\sigma(j)$. There are two cases.

In the first case (line (2.4)), $\sigma(g+1) = \sigma(g+2)$, so a move involving $\sigma(g+1)$ cannot be of type I, nor II, nor III. By Corollary 4.2 there exists an optimal reordering scheme that would not perform such a move. By the induction hypothesis, $C_1[g]$ is the optimal cost for $\sigma(1:g)$ when $\sigma(g)$ is not moved. Since $\sigma(g) \neq \sigma(g+1)$ and $\sigma(j)$ is a hit for *LFD* after it is moved, the optimal cost for $\sigma(1:j)$ is $C_1[g]$ plus the cost incurred by *LFD* in processing request sequence $\sigma' = \sigma(g+1:j-1)$ (line (2.4)).

In the second case (line (2.5)), $\sigma(g+1) \neq \sigma(g+2)$; we have to consider two costs:

the optimal cost if $\sigma(g+1)$ is not moved (same as the cost in line (2.4)) and the optimal cost if $\sigma(g+1)$ is moved. Note that if $\sigma(g+1)$ is moved, then $\sigma(g)$ gets bumped back one position to index $g+1$, so $\sigma(j)$ is moved into position $g+2$ instead of $g+1$. Since $\sigma(j)$ is moved next to its closest previous copy, $\sigma(g+2:j-1)$ does not contain copies of $\sigma(j)$, hence $\sigma(g) = \sigma(j) \neq \sigma(g+2)$. A move involving $\sigma(g+1)$ cannot be of type II or III, so by Corollary 4.2, we need only consider the case that it is a move of type I. By the induction hypothesis, $C_2[g+1]$ is the optimal cost for $\sigma(1:g+1)$, when $\sigma(g+1)$ is moved next to its closest preceding copy. If $\sigma(g+1)$ is moved next to its closest preceding copy, the optimal cost for $\sigma(1:j)$ is $C_2[g+1]$ plus the cost of processing request sequence $\sigma' = \sigma(g+2:j-1)$ using *LFD* (vi). Therefore the optimal cost in line (2.5) is the minimum of costs in line (2.4) and (vi).

Finally we demonstrate the **correctness of the formula for $C_3[j]$**. This is the optimal cost for $\sigma(1:j)$ when $\sigma(j)$ is moved immediately before the run containing request $\sigma(j-1)$. Recall h is the starting index of the run containing $\sigma(j-1)$. By inspection, the base cases' formulas for $C_3[j]$ (**lines (3.1) to (3.6)**) are correct. We now show the correctness of the remaining four cases (**lines (3.7) to (3.10)**) (see Figure 4.14).

Consider line (3.7) where $\sigma(j-1) = \sigma(j)$. $\sigma(j)$ is moved at the beginning of the run that it is part of. The optimal cost for this case is the same as the optimal cost on $\sigma(1:h)$, when $\sigma(h)$ is not moved. By the induction hypothesis, $C_1[h]$ is the optimal cost on $\sigma(1:h)$, when $\sigma(h)$ is not moved. This proves the correctness of line (3.7).

Focus on line (3.8) where $\sigma(j-1) \neq \sigma(j)$ and $\sigma(h-1) = \sigma(j)$ (line (3.8)). $\sigma(j)$ is

moved next to its closest previous copy. This move is both a copy move and an evacuation move. We have already shown that $C_2[j]$ is the optimal cost for $\sigma(1:j)$ if σ_j is moved to its closest preceding copy (evacuation move). Hence the optimal cost for $\sigma(1:j)$, when $\sigma(j)$ is moved immediately before the run containing $\sigma(j-1)$ is equal to $C_2[j]$. Since $C_3[j] = C_2[j]$, line (3.8) is correct.

The remaining two cases **(lines (3.9) and (3.10))** have the following in common: $\sigma(j-1) \neq \sigma(j)$ and $\sigma(h-1) \neq \sigma(j)$. $\sigma(j)$ is moved next to $\sigma(h-1)$ which is not its copy. If $\sigma(h-1)$ is not moved, the optimal cost is $C_1[h-1]$ to which we must add 2 (vii). This is true since after $\sigma(j)$ is moved, $\sigma(h:j)$ has exactly two runs and $\sigma(h-1) \neq \sigma(j)$. We have also used the induction hypothesis that $C_1[h-1]$ is the optimal cost for $\sigma(1:h-1)$, when $\sigma(h-1)$ is not moved. We now focus on the differences between lines (3.9) and (3.10).

In line (3.9), $\sigma(h-2) \neq \sigma(j)$ (see Figure 4.14). Hence $\sigma(h-2:j) = z\ y\ v^{j-h}\ x$ $(z \neq x)$ before $\sigma(j) = x$ is moved and $\sigma'(h-2:j) = z\ y\ x\ v^{j-h}$ after $\sigma(j) = x$ is moved. z may be equal to y or v, but the essential is that $z \neq x$. $\sigma'(h-1) = y$ is preceded by z and followed by x. Since $z \neq x$, a move involving $y = \sigma'(h-1)$ cannot be of type II nor III. By Corollary 4.2, this move can be only of type I. If this move is performed, the optimal cost is $C_2[h-1]$ plus 2 ($z \neq x$ and $\sigma'(h:j)$ is a run of x followed by a run of v) (viii). We have used the induction hypothesis that $C_2[h-1]$ is the optimal cost on $\sigma(1:h-1)$, when $\sigma(h-1)$ is moved next to its closest preceding copy. The optimal cost in this case is the minimum of the expressions (vii) and (viii) (line (3.9)).

In line (3.10), $\sigma(h-2) = \sigma(j)$ (see Figure 4.14). Hence $\sigma(h-2:j) = x\ y\ v^{j-h}\ x$ before

$\sigma(j) = x$ is moved and $\sigma'(h-2:j) = x\ y\ x\ v^{j-h}$ after $\sigma(j) = x$ is moved. $\sigma'(h-1) = y$ is preceded by x and followed by x. A move m involving $y = \sigma'(h-1)$ cannot be of type I. By Corollary 4.2 m must be of type II or III. If m is performed, *LFD* does not fault on $\sigma'(h) = x$. Since $\sigma'(h+1:j)$ is a run of v so the cost of serving it using *LFD* is 1. If m is of type II, then the optimal cost is $C_2[h-1]+1$ (ix). If m is of type III, then the optimal cost is $C_3[h-1]+1$ (x). We have used the induction hypothesis twice: first, that $C_2[h-1]$ is the optimal cost on $\sigma(1:h-1)$, when $\sigma(h-1)$ is moved next to its closest preceding copy and second, that $C_3[h-1]$ is the optimal cost on $\sigma(1:h-1)$, when $\sigma(h-1)$ is moved just before the run containing $\sigma(h-2)$. The optimal cost in this last case is the minimum of the expressions (vii), (ix), and (x) (line (3.10)). This concludes the analysis, *DP* returns the cost of the optimal ordering. ■

We now explain how to obtain the optimal ordering. Let $S_1[j]$ be some sequence of moves such that the ordering $\sigma_f(1:j)$, obtained by applying them, has *LFD* cost equal to $C_1[j]$ ($\forall j \in \{1,2,\ldots,n)$. Similarly we define $S_2[j]$ (respectively $S_3[j]$) to be a sequence of moves such that the ordering $\sigma_f(1:j)$, obtained by applying them, has *LFD* cost equal to $C_2[j]$ (respectively $C_3[j]$). The formulas for $S_1[j]$, $S_2[j]$, and $S_3[j]$ can easily be derived from the formulas for $C_1[j]$, $C_2[j]$ and $C_3[j]$ while using the exact same cases and line notation.

Let ϕ denote the sequence of 0 moves (empty sequence). By inspection, $S_1[j] = \phi$ in lines (1.1) to (1.3) (for a description of these cases, see formula for $C_1[j]$), $S_2[j] = \phi$ in lines (2.1) and (2.2) (see formula for $C_2[j]$) and $S_3[j] = \phi$ in line (3.1) (see formula for $C_3[j]$). Also by

inspection, $S_3[j] = (j, h)$ in lines (3.2) to (3.6) (see formula for $C_3[j]$).

Cases that involve a single term in the expression of the optimal cost are straight forward. I.e.: $S_1[j] = S_1[j-1]$ in line (1.4) (see formula for $C_1[j]$), $S_2[j] = S_1[j]$ in line (2.3) (see formula for $C_2[j]$), $S_2[j] = S_1[g], (j, g+1)$ in line (2.4) (see formula for $C_2[j]$), $S_3[j] = S_1[h], (j, h)$ in line (3.7) (see formula for $C_3[j]$), and $S_3[j] = S_2[j]$ in line (3.8) (see formula for $C_3[j]$).

For the cases that involve the "minimum" of 2 or 3 terms, we select the sequence of moves that corresponds to the term with the minimum value. I.e.: formula for $C_1[j]$ in line (1.6) is $C_1[j] = \min\{C_1[j-1]+1, C_2[j-1], C_3[j-1]\}$. If $C_1[j-1]+1$ is the minimum, then $S_1[j] = S_1[j-1]$. If $C_2[j-1]$ is the minimum, then $S_1[j] = S_2[j-1]$. If $C_3[j-1]$ is the minimum, then $S_1[j] = S_3[j-1]$. All other cases that involve a minimum are similar to this one and use the same rule of choosing the sequence of moves that corresponds to the term with the minimum value.

The following result is about the time and space requirements of computing the optimal cost using the *DP* algorithm.

Theorem 4.11 *The* DP *algorithm can be implemented to run in* $\Theta(n)$ *time and use* $\Theta(n)$ *space.*

Proof:

The time and space consuming calculations are following: computing the g and h indices in $C_2[j]$ and $C_3[j]$ formulas, and $c_{g,j-1}$ and $c_{g+1,j-1}$ in $C_2[j]$.

To speed up the calculations we pre-compute the values of g and h. More precisely we

have two arrays G and H of size n such that $G[j]$ is the g index for $C_2[j]$ and $H[j]$ is the h index for $C_3[j]$.

Recall that $g = G[j]$ is the maximum index, no greater than j, such that $\sigma(g) = \sigma(j)$. If g doesn't exist, then $g = 0$. In words $G[j]$ is the index of the previous copy of $\sigma(j) = x$ that is closest to position j. Array G is computed in $\Theta(n)$ time and space as follows: for each distinct page x requested in σ have a linked list L_x that stores the indices (in increasing order) of requests for x in σ. A main level linked list stores the pointers to each of these L_x lists. After the lists are created go through them one by one, say current one is L_x with indices $j_1 \leq j_2 \ldots \leq j_m$. Then $G[j_1] = 0$ and $G[j_e] = j_{e-1}$, $\forall e \in \{2, 3, \ldots, m\}$.

Recall $h = H[j]$ is the minimum index less than j such that $\sigma(h : j-1) = v^{j-h}$ where $v = \sigma_{j-1}$. In words h is the start index of the run that contains request $\sigma(j-1)$. Array H is computed in linear time and space as well.

$$H[j] = \begin{cases} 0 & \text{if } j = 1 \\ 1 & \text{if } j = 2 \\ j-1 & \text{if } j \geq 3 \ \& \ \sigma(j-2) \neq \sigma(j-1) \\ H[j-1] & \text{if } j \geq 3 \ \& \ \sigma(j-2) = \sigma(j-1) \end{cases}$$

Observe that array C_1 can be computed in $\Theta(n)$ time using $\Theta(n)$ space. The same holds for array C_3 if array H is pre-computed. To compute $C_2[j]$ the most time-consuming operation is to compute $c_{g,j-1}$ and $c_{g+1,j-1}$. Computing g takes $O(1)$ time given that array G is pre-computed.

Recall $g \leq j$. Since $k = 1$ it is not difficult to show that the following formula holds.

$$c_{g,j} = \begin{cases} c_{1,j} - c_{1,g} & \text{if } \sigma(g) \neq \sigma(g+1) \\ c_{1,j} - c_{1,g} + 1 & \text{if } \sigma(g) = \sigma(g+1) \end{cases}$$

Therefore if we pre-compute the $c_{1,j}$ values, $\forall j \in \{1, 2, \ldots, n\}$ then we can compute $c_{g,j}$ in $O(1)$ time. Notice that since k is 1 we can keep track of the *LFD* cost at each position j while computing the *LFD* cost for the whole request sequence σ. The $c_{1,j}$ values can be computed in $\Theta(n)$ time and storing them requires $\Theta(n)$ space. Since $c_{g,j}$ is computed in $O(1)$ time, it follows that we can compute the $C_2[j]$ array in $\Theta(n)$ time. The result follows. ■

4.8.4 Online Reordering Schemes

Recall the concepts of an online reordering algorithm and an online reordering scheme (Definition 4.4 and 4.5). By definition, online reordering algorithms cannot observe any requests past the current one, can move forward only the current request, and make irrevocable decisions.

A lower bound result for online reordering schemes is presented next.

Theorem 4.12 *In the* UNIDIRECTIONAL *model, for k equal to 1 and arbitrary r, there is a lower bound of* 1.5 *on the competitive ratio of* **any** *online reordering scheme.*

Proof:

Let $(\mathcal{R}, LFD)$ be an arbitrary online reordering scheme. We show how an adversary can

construct σ_0 such that

$$cost(LFD(\mathcal{R}(\sigma_0))) \geq 1.5 \cdot cost(LFD(OPT(\sigma_0)))$$

where (OPT, LFD) is an optimal reordering scheme.

The adversary selects a request sequence σ_0 such that $\sigma_0(1:2) = x_1\ x_2$. At step 2, $\mathcal{R}$ has two choices: either move $\sigma_0(2) = x_2$ or do nothing. If $\mathcal{R}$ moves $\sigma_0(2)$, then the adversary sets $n = r+2$ and $\sigma_0(3:n) = x_2^r$. Note that the optimal ordering of the original request sequence $\sigma_0 = x_1\ x_2\ x_2^r$ is σ_0 itself. Hence $cost(LFD(OPT(\sigma_0))) = 2$. However $\mathcal{R}$'s reordered request sequence so far (after the second step) is $\sigma' = x_2\ x_1\ x_2^r$. Since the request for x_1 cannot be bumped back more that $r-1$ positions, $\mathcal{R}$ cannot move **all** requests in the third run (x_2^r) past the request for x_1. Therefore $\mathcal{R}$'s final ordering is $x_2^{n_1}\ x_1\ x_2^{n_2}$ where $n_1, n_2 \geq 1$ and $n_1 + n_2 + 1 = n$. Hence $cost(LFD(\mathcal{R}(\sigma_0))) = 3$. Since $cost(LFD(OPT(\sigma_0))) = 2$, the result follows.

If $\mathcal{R}$ does not move $\sigma_0(2) = x_2$, the adversary sets $n = r+3$ and $\sigma_0(3:n) = x_1^{r+1}$. Given $\sigma_0 = x_1\ x_2\ x_1^{r+1}$, observe that the optimal ordering is $x_2\ x_1\ x_1^{r+1}$ and $cost(LFD(OPT(\sigma_0))) = 2$. When $\mathcal{R}$ is at step 3, its request sequence is the same as the initial one: $x_1\ x_2\ x_1^{r+1}$. Since the request for x_2 cannot be bumped back more that r positions, $\mathcal{R}$ cannot move all requests in the x_1^{r+1} run past the request for x_2. Hence $\mathcal{R}$'s final ordering is $x_1^{n_1}\ x_2\ x_1^{n_2}$ where $n_1, n_2 \geq 1$ and $n_1 + n_2 + 1 = n$. We get that $cost(LFD(\mathcal{R}(\sigma_0))) = 3$. Since $cost(LFD(OPT(\sigma_0))) = 2$ the result follows.

Observe the result holds for arbitrarily long request sequences as well. The adversary

can construct a request sequence σ such that σ consists of m copies of σ_0 and each copy is made of requests to completely distinct pages. I.e.: the first copy has requests to pages x_1 and x_2, the second has requests to pages x_3 and x_4, etc. By a similar analysis it follows that $cost(LFD(\mathcal{R}(\sigma_0))) \geq 3m$ and $cost(LFD(OPT(\sigma_0))) = 2m$. Hence

$$\frac{cost(LFD(\mathcal{R}(\sigma_0)))}{cost(LFD(OPT(\sigma_0)))} \geq 1.5$$

which proves the result. ■

We now focus on a particular online reordering scheme which we call $(Greedy, LFD)$. Given the current j-th request reordering algorithm *Greedy* moves $\sigma(j)$ to the closest position i $(i < j)$ such that this move diminishes the *LFD* cost of processing current prefix $\sigma(1:j)$.

We now present a definition about the value of a move relative to a given prefix of a request sequence.

Definition 4.17 *Let $1 \leq g \leq h \leq n$. Let m be a move that is applied to request sequence σ_g. Let σ_{g+1} be the resulting request sequence.*

The **value** *of move m* **relative to the prefix** $\sigma_g(1:h)$ *is*

$$v_h(m) = cost(LFD(\sigma_g(1:h))) - cost(LFD(\sigma_{g+1}(1:h)))$$

Given the notation above, *Greedy* performs m if $v_j(m) > 0$. Moreover i $(i < j)$ is the maximum index such that $v_j(m') > 0$ where move $m' = (j, i)$. In words *Greedy* breaks ties

between moves by the size of their range, so the chosen move has the smallest range.

We now present a result regarding the competitiveness of reordering scheme $(Greedy, LFD)$.

Theorem 4.13 *In the* UNIDIRECTIONAL *model for $k = 1$ and $r \geq 2$, there is a lower bound of $\frac{2r+1}{3}$ on the competitive ratio of reordering scheme $(Greedy, LFD)$.*

Proof:

We give counterexamples that imply the lower bound for each of two possible cases: the reordering threshold, r, is either even or odd.

Consider first the case where r is even, of value $r = 2b$. Let $\sigma_0 = x_1\ (x_2\ x_1^{r+1}\ x_3\ x_1^{r+1})^b$ and (OPT, LFD) be an optimal reordering scheme. Since the requests in σ_0 are to 3 distinct pages, the optimal cost must be at least 3. To get an optimal ordering move all requests for x_2 at the front of the request sequence, then move all requests for x_3 to the front. We obtain is $x_2^b\ x_3^b\ x_1\ (x_1^{r+1}\ x_1^{r+1})^b$. Observe that this ordering satisfies the constraints of the UNIDIRECTIONAL model. Since k is equal to 1, Observation 4.2 implies that this ordering has *LFD* cost equal to 3 ($b > 0$ since $r \geq 2$). Hence $cost(LFD(OPT(\sigma_0))) = 3$ (i).

Recall the definition of *Greedy* above. When serving σ_0, *Greedy* proceeds as follows: at step j, it observes $\sigma_0(1 : j)$ and has to decide if it should move $\sigma_0(j)$ or not ($\forall j \in \{1, 2, \ldots, n\}$). The first change occurs at its third step, when it observes $\sigma_0(1 : 3) = x_1\ x_2\ x_1$. Move $m = (3, 2)$ has relative value $v_3(m) = 1$, so *Greedy* performs it. At the fourth step, it observes $sigma(1 : 4) = x_1\ x_1\ x_2\ x_1$. Move $m = (4, 3)$ has relative value $v_4(m) = 1$ so *Greedy* performs it as well. Repeating this argument we obtain that *Greedy* performs moves $(5, 4)$, $(6, 5)$, $\ldots$, $(r + 2, r + 1)$ before it gets to step $r + 3$. At step $r + 3$, the ordering is

$\sigma' = x_1^{r+1}\ x_2\ x_1\ x_3\ x_1^{r+1}\ (x_2\ x_1^{r+1}\ x_3\ x_1^{r+1})^{b-1}$. Since request $\sigma'(r+2) = x_2$ has been pushed back r positions, *Greedy* cannot move request $\sigma'(r+3) = x_1$. At step $r+4$, *Greedy* does nothing since moving $\sigma'(r+4) = x_3$ does not improve the *LFD* cost of the current prefix. At step $r+5$, *Greedy* observes $\sigma'(1:r+5) = x_1^{r+1}\ x_2\ x_1\ x_3\ x_1$. Move $m = (r+5, r+4)$ has relative value $v_{r+5}(m) = 1$ so *Greedy* performs it. Repeating this similar argument, we obtain the following ordering for step $2r+5$: $\sigma'' = x_1^{r+1}\ x_2\ x_1^{r+1}\ x_3\ x_1\ (x_2\ x_1^{r+1}\ x_3\ x_1^{r+1})^{b-1}$. Since $\sigma''(2r+4) = x_3$ has been bumped back r positions, $\sigma''(2r+5) = x_1$ cannot be moved.

Repeating this argument for each subsequence $x_1\ x_2\ x_1^{r+1}\ x_3\ x_1^{r+1}$, we get that the final ordering produced by *Greedy* is $\sigma_f = (x_1^{r+1}\ x_2\ x_1^{r+1}\ x_3)^b\ x_1$. Since k is equal to 1, apply Observation 4.2 to get that $cost(LFD(\sigma_f)) = 4b+1 = 2r+1$. Hence $cost(LFD(Greedy(\sigma_0))) = 2r+1$ (ii). By definition of the competitive ratio and (i) and (ii) above, it follows that the competitive ratio of reordering scheme $(Greedy, LFD)$ is at least $\frac{2r+1}{3}$.

Consider now the case where r is odd, of value $r = 2b+1$. The proof is similar to the one in the first case where r is even ($r = 2b$). The difference is that when r is odd, an extra copy of sequence $x_2\ x_1^{r+1}$ needs to be appended to the end of the request sequence used in the first case. Let $\sigma_0 = x_1\ (x_2\ x_1^{r+1}\ x_3\ x_1^{r+1})^b\ x_2\ x_1^{r+1}$. The final ordering obtained by *Greedy* is $(x_1^{r+1}\ x_2\ x_1^{r+1}\ x_3)^b\ x_1^{r+1}\ x_2\ x_1$ and its *LFD* cost is $4b+3 = 2r+1$. An optimal ordering is $x_2^{b+1}\ x_3^b\ x_1\ (x_1^{r+1}\ x_1^{r+1})^b\ x_1^{r+1}$. Since $b > 0$ ($r \geq 2$), the *LFD* cost on the optimal ordering is 3. The competitive ratio of reordering scheme $(Greedy, LFD)$ is at least $\frac{2r+1}{3}$. ■

4.8.5 Conservative Online Reordering Schemes

In this section we focus on a subset of the class of online reordering schemes that we call **conservative online reordering schemes**. We now introduce notation for the value of a move m when a request sequence is appended at the end of the current prefix sequence (before m is performed).

Definition 4.18 *Let a request sequence σ_s of length h be appended at the end of request sequence $\sigma_g(1:j)$. Let the resulting request sequence be denoted by σ. Let σ' be the request sequence obtained by applying move $m = (j,i)$ to σ. The* **value of m relative to time t and suffix sequence σ_s** *is the difference of* LFD *cost on prefix sequences $\sigma(1:t)$ and $\sigma'(1:t)$:*

$$v_{t,s}(m) = cost(LFD(\sigma(1:t))) - cost(LFD(\sigma'(1:t)))$$

The **current** *and* **full** *values of m relative to suffix σ_s are the values of m relative to σ_s and to times j and $j+h$, respectively ($v_{j,s}(m)$ and $v_{j+h,s}(m)$).*

The concept of a conservative online reordering scheme is introduced next.

Definition 4.19 *A* **conservative online reordering scheme** *is an online reordering scheme where the reordering algorithm performs a move m if for every request sequence σ_s, the full value of m relative to suffix sequence σ_s is non-negative.*

Recall the definition of an online reordering scheme (Definition 4.4). Since any conservative online reordering scheme is implicitly an online reordering scheme, it can move only the

current request $\sigma_0(j)$ at the j-th step (σ_0 denotes the initial request sequence). Using the notation introduced in Definition 4.18 we obtain that for any move $m = (j, i)$ performed by a conservative online reordering scheme and any possible suffix σ_s starting at index $j + 1$, $v_{j+h,s}(m) \geq 0$ where h is the length of σ_s. The full value of any move m performed by a conservative online reordering scheme is non-negative (no matter what the suffix is).

Decompose the sequence of moves performed by a conservative online reordering scheme such that $M = M_1,\{m\}, M_2$. We focus on the special case when r is equal to 1. Since the reordering algorithm is online, the moves in M are in increasing order by from-coordinate (Definition 4.1). By Theorem 4.1 we can assume no moves in M overlap. Therefore all moves in M_2 are performed on suffix σ_s. Let σ_{s,M_2} be the request sequence obtained by applying moves in M_2 to suffix σ_s. By the definition of conservative online reordering schemes, the value of m relative to σ_{s,M_2} is non-negative. Since this holds for any move m in M, the cost incurred by any conservative online reordering scheme is at most the cost incurred by reordering scheme $(NoReordering, LFD)$ when r is 1. By using this observation and applying Theorem 4.6 we obtain the following upper bound result.

Corollary 4.3 *In the* UNIDIRECTIONAL *model with r equal to* 1*, no conservative online reordering scheme has a competitive ratio larger than* 3*.*

4.8.6 Online Reordering Schemes with Look-Ahead

Recall the concept of an online reordering scheme (see Definitions 4.5 and 4.4). We now present the concept of online reordering schemes with look-ahead.

Definition 4.20 *An online reordering scheme has* **look-ahead** *of size l if, given the current request in position j, the reordering scheme can observe all requests in $\sigma(j : j + l)$.*

Note that the look-ahead l does not include the current request $\sigma(j)$. Observe that in previous sections, online reordering schemes had look-ahead of 0. Also note that offline reordering schemes (Definition 4.5) have look-ahead $n - 1$, where n is the length of the request sequence.

Theorem 4.14 *An online reordering scheme requires look-ahead of size at least $n - 3$ to compute the optimal cost for a request sequence of length n, assuming k and r both equal to* 1.

Proof:

We show how the adversary can construct σ_0 such that a look-ahead of size at least $n-3$ is required to compute an optimal ordering. Note that k and r are equal to 1. Let $\sigma_0 = x_1$ x_2 x_1 $x_3^{c_3}$ $x_4^{c_4}$ $x_5^{c_5}$ $\ldots$ $x_{s-1}^{c_{s-1}}$ $x_s^{c_s}$ x. x is either x_1, x_2, or x_{s+1} (the adversary picks its value) and $c_i \geq 1$, $\forall i \in \{3, 4, \ldots, s\}$ are arbitrary positive integers such that $|\sigma_0| = 4 + \sum_{i=3}^{s} c_i = n$.

Let $(\mathcal{R}, LFD)$ be online reordering scheme that computes the optimal cost. The proof is by contradiction. Assume that the reordering algorithm R has a look-ahead of l such that $l \leq n - 4$. Therefore it cannot observe the last request $\sigma_0(n) = x$ when it serves the first three requests: $\sigma_0(1)$, $\sigma_0(2)$, and $\sigma_0(3)$.

Given the current request $\sigma_0(j)$, an online reordering algorithm has only two choices to make at this step: it can move $\sigma_0(j)$ or do nothing. Note this is the only time when $\mathcal{R}$ can

move $\sigma_0(j)$. Let $(\mathcal{R}, LFD)$ serve σ_0. The first choice for $\mathcal{R}$ occurs when the current request is $\sigma_0(2)$. $\mathcal{R}$ has two choices: to move $\sigma_0(2)$ or to do nothing.

If it moves $\sigma_0(2) = x_2$, then the request sequence becomes $\sigma_1 = x_2\ x_1\ x_1\ x_3^{c_3}\ x_4^{c_4}\ x_5^{c_5}$ $\ldots\ x_{s-1}^{c_{s-1}}\ x_s^{c_s}\ x$. The adversary sets $x = x_2$. Note that the reordering threshold r is equal to 1, and $\sigma_1(2) = x_1$ has already been bumped back 1 position. Hence at the n-th step, $\mathcal{R}$ cannot move $\sigma_1(n) = x_2$ next to its copy in position 1 ($\sigma_1(1) = x_2$). Therefore the best ordering it can achieve is σ_1. It follows that $cost(LFD(\mathcal{R}(\sigma_0))) \geq cost(LFD(\sigma_1)) = 2 + s - 2 + 1 = s + 1$. Let (OPT, LFD) be an optimal reordering scheme. Notice that OPT has to perform moves $m_1 = (3, 2)$ and $m_2 = (n, 4)$ or in words, move $\sigma_0(3) = x_1$ next to its previous copy ($\sigma_0(1) = x_1$) and then move $\sigma_0(n) = x_2$ next to its previous copy which is in position 3 after m_1 is performed. The optimal ordering is $\sigma_f = x_1\ x_1\ x_2\ x_2\ x_3^{c_3}\ x_4^{c_4}\ x_5^{c_5}$ $\ldots\ x_{s-1}^{c_{s-1}}\ x_s^{c_s}$. Hence $cost(LFD(OPT(\sigma_0))) = cost(LFD(\sigma_f))) = 2 + s - 2 = s$. Note that $cost(LFD(\mathcal{R}(\sigma_0))) \geq s + 1 = cost(LFD(OPT(\sigma_0))) + 1$. It follows that $(\mathcal{R}, LFD)$ is not an optimal reordering scheme.

If $\mathcal{R}$ does not move $\sigma_0(2)$, then no changes are made. The current request is $\sigma_0(3)$. $\mathcal{R}$ has two choices again: either move $\sigma_0(3) = x_1$ next to its previous copy ($\sigma_0(1) = x_1$) or do nothing.

If $\mathcal{R}$ moves $\sigma_0(3) = x_1$, the new ordering is $\sigma_1 = x_1\ x_1\ x_2\ x_3^{c_3}\ x_4^{c_4}\ x_5^{c_5}\ \ldots\ x_{s-1}^{c_{s-1}}\ x_s^{c_s}$ x. The adversary sets $x = x_1$. Because the reordering threshold r is equal to 1 and $\sigma_1(3) = x_2$ has already been bumped back 1 position, $\mathcal{R}$ cannot move $\sigma_1(n) = x_1$ next to its previous copy in position 2 ($\sigma_1(2) = x_1$). Therefore σ_1 is the best ordering $\mathcal{R}$ can

achieve. Hence $cost(LFD(\mathcal{R}(\sigma_0))) \geq cost(LFD(\sigma_1)) = 2 + s - 2 + 1 = s + 1$. Note that an optimal ordering is obtained by applying by moves $m_1 = (2, 1)$ and $m_2 = (n, 4)$ to σ_0. Let (OPT, LFD) be the optimal reordering scheme that performs these moves. The final request sequence obtained by (OPT, LFD) is $\sigma_f = x_2\ x_1\ x_1\ x_1\ x_3^{c_3}\ x_4^{c_4}\ x_5^{c_5}\ \ldots\ x_{s-1}^{c_{s-1}}\ x_s^{c_s}$. Hence $cost(LFD(OPT(\sigma_0))) = cost(LFD(\sigma_f)) = 2 + s - 2 = s$. Observe that $cost(LFD(\mathcal{R}(\sigma_0))) \geq s + 1 = cost(LFD(OPT(\sigma_0))) + 1$. It follows that the reordering scheme $(\mathcal{R}, LFD)$ is not optimal.

If $\mathcal{R}$ does not move $\sigma_0(3) = x_1$, then the adversary sets x to be x_{s+1}. In the best case, $\mathcal{R}$ does not performs any moves on the remaining suffix of the request sequence. The best ordering that $\mathcal{R}$ can achieve is the initial request sequence, $\sigma_0 = x_1\ x_2\ x_1\ x_3^{c_3}\ x_4^{c_4}\ x_5^{c_5}\ \ldots\ x_{s-1}^{c_{s-1}}\ x_s^{c_s}\ x_{s+1}$. Hence $cost(LFD(\mathcal{R}(\sigma_0))) \geq cost(LFD(\sigma_0)) = 3 + s - 2 + 1 = s + 2$. Note that $\sigma_f = x_1\ x_1\ x_2\ x_3^{c_3}\ x_4^{c_4}\ x_5^{c_5}\ \ldots\ x_{s-1}^{c_{s-1}}\ x_s^{c_s}\ x_{s+1}$ is an optimal ordering. Let (OPT, LFD) be an optimal reordering scheme that produces σ_f. It follows that $cost(LFD(OPT(\sigma_0))) = cost(LFD(\sigma_f)) = 2 + s - 2 + 1 = s + 1$. Observe that $cost(LFD(\mathcal{R}(\sigma_0))) \geq s + 2 = cost(LFD(OPT(\sigma_0))) + 1$. Therefore the reordering scheme $(\mathcal{R}, LFD)$ is not optimal.

We have shown that in all cases the reordering scheme $(\mathcal{R}, LFD)$ is not optimal, hence an optimal online reordering scheme does require look-ahead of size at least $n - 3$. ■

Chapter 5

Conclusions

In this thesis we analyzed three variations of the caching problem: web caching under the Torng framework (Chapter 2), the relative competitive ratio (Chapter 3), and caching with request reordering (Chapter 4). Our main results concern caching with request reordering, which we divided into two sections, Sections 4.7 and 4.8, one for each reordering model, Bidirectional and Unidirectional respectively. In the Bidirectional model we focused on offline caching algorithms, whereas in the Unidirectional model we studied both offline and online reordering schemes.

5.1 Summary of Results

In Chapter 2 we proved new results for the Bit model combined with the Torng framework: the competitive ratio of any marking eviction policy is smaller than $\min\left\{\frac{k}{s_{min}}, 1 + s_{max}p\right\}$. If we restrict any marking eviction policy M to serve only (a, k)-referenced request sequences,

then its competitive ratio is less than $\min\left\{\frac{k}{s_{min}}, 1+\frac{s_{max}p}{a}\right\}$. If a is larger than the fault cost of the average-sized page in σ, then the competitive ratio of M is smaller than by 2.

Regarding the relative competitiveness ratio (Chapter 3), we proved several results that combine the basic or Torng frameworks with the CLASSICAL model. For the basic framework combined with CLASSICAL model, we showed that there exists an access graph G such the competitive ratio of eviction policy EDO on G relative to LRU is infinite (Theorem 3.2). For details on all other results, see Theorem 3.1 and Corollaries 3.3 and 3.1.

The third and most important variation is caching with request reordering (Chapter 4). We extended a result by Albers [3] to show that offline algorithm $BMIN$ is 3-competitive in the BIDIRECTIONAL model with non-optional caching (Definition 1.2). For the UNIDIRECTIONAL model we proved the following main results. For the case when both r and k are equal to 1, we presented a fast dynamic programming algorithm which computes the optimal ordering and its LFD cost. We also showed that $(NoReordering, LFD)$ is a tight $(2r+1)$-approximation reordering scheme. We proved that for k equal to 1 there exists a lower bound of 1.5 on the competitiveness of any online reordering scheme. For online reordering scheme $(Greedy, LFD)$ we showed a lower bound of $\frac{2r+1}{3}$ assuming k is equal to 1. We also demonstrated an upper bound of 3 on the competitiveness of any conservative online reordering scheme when k is equal to 1. We also showed that any online reordering scheme requires look-ahead of size at least $n-3$ to compute the optimal cost for a request sequence of length n assuming both r and k are equal to 1.

5.2 Open Problems

Several problems for caching with request reordering are still open. One research area could be the BIDIRECTIONAL model: What is the exact competitive ratio of the *BMIN* algorithm? Is there an optimal algorithm for the BIDIRECTIONAL model that runs in polynomial time? Are there online reordering schemes that are constant-competitive in the BIDIRECTIONAL model?

We do not know the time complexity status of the optimal offline reordering scheme for the UNIDIRECTIONAL model. Extending the offline optimal reordering scheme (DPR, LFD) in Section 4.8.3 (even to $k = 1$ and $r = 2$) is a challenging task. We conjecture that the problem is NP-hard for arbitrary values of k and r.

Some problems regarding online reordering schemes in the UNIDIRECTIONAL model are open as well: what is the competitive ratio of $(Greedy, LFD)$? Can we determine a lower bound, expressed in terms of k and r, on the competitive ratio of any online reordering scheme?

Another interesting research area is the analysis of online reordering schemes with look-ahead l. How is the competitive ratio affected by doubling or tripling the size of the look-ahead? Can we get a constant competitive ratio if the size l of the look-ahead is $O(r)$?

Appendix A

Appendix

A.1 Miscellaneous Results

This section contains a few stand-alone results that are potentially useful for future research in caching with request reordering. Note that none of the other results presented in this thesis depend on or use any of the results presented in this section.

Next we present a useful observation about the running time and space of processing a request sequence using eviction policy *LFD*. Note that no detailed analysis of the run time and space required by *LFD* exists in the literature.

Observation A.1 *Let σ be a request sequence of length n. LFD can be implemented such that processing σ takes $O(n \log k)$ or $O(n \log \log n)$ time and $O(n)$ space.*

Proof:

For the special case $k = 1$ notice that *LFD* has runtime linear in n and requires constant

space. For $k \geq 2$ we create an integer array N of size n such that $N[i]$ is the index of the next request for x after position i where $x = \sigma(i)$. If there are no more requests for x after position i then $N[i] = +\infty$.

I.e.: for $\sigma = x_1\ x_2\ x_1\ x_3\ x_4\ x_3\ x_1\ x_2\ x_3$, $N = 3\ 8\ 7\ 6\ +\infty\ 9\ +\infty\ +\infty\ +\infty$.

Observe the N array can be computed in $\Theta(n)$ time and space using the following technique: for each distinct page x requested in σ we create a linked list L_x that stores the indices (in increasing order) of requests for x in σ. A main level linked list stores the pointers to each L_x list. Observe all the lists can be created with a single pass of σ. Once they are created, we traverse them one at a time. If the current one is L_x with indices $i_1 \leq i_2 \ldots \leq i_m$ then $N[i_h] = i_{h+1}$, $\forall h \in \{1, 2, \ldots, m-1\}$ and $N[i_m] = +\infty$.

We also keep a balanced binary search tree T of size k that stores only info about pages that are in cache. The elements at each node are pair of type (n_x, x) where n_x is the index of the next request for x. The indices n_x serve as keys in the tree.

Whenever a page $y = \sigma(i)$ is requested, *LFD* checks if it is in cache by searching for the i. This works since if y is in cache, then $n_y = i$. If i is not found in T then y is not in cache. This test takes $O(\log k)$ time. If y is in cache, then the key n_y of y needs to be updated to $N[i]$. The tree rebalancing can be performed in $O(\log k)$.

If y is not in cache, then it must be inserted in T. Before performing the insertion *LFD* checks if its cache is full. If this is the case, it must evict the page in cache whose next request is furthest in the future. This page is the one whose key is largest in T. Searching T for this key and deleting it requires $O(\log k)$ time. Finally LFD must insert y into T which

takes time $O(\log k)$.

All of the operations take $O(\log k)$ time for each of the n steps. The total time is $O(n \log k)$ and total space is $O(n)$.

An alternative implementation would be to use a van Emde Boas priority queue [9, 26] for T since the values of the keys are from the set $\{1, 2, \ldots n\}$. All of the above operations would take $O(\log \log n)$ per step, hence the total runtime would be $O(n \log \log n)$. ■

Now recall the chunk definition (Definition 1.6). Recall that we analyze caching with reordering under the CLASSICAL cost model. Since all pages have size equal to 1, the number of distinct pages in a chunk is always k, except for possibly the last chunk in the request sequence.

Let σ be a request sequence. Recall the partitioning of σ into chunks that was presented after Definition 1.6. Note that in the next result, the order of chunks is the one implied by this partition.

Observation A.2 *In the* UNIDIRECTIONAL *model, an optimal reordering scheme does not always maximize chunks in the order they occur in the request sequence.*

Proof:

Recall that by σ^m we mean m consecutive copies of request sequence σ. Consider the following example where k is 1 and $r > 2$.

Let $\sigma_0 = x_1\ (x_2\ x_3)^{r+1}\ x_1^r$. If chunks are to be maximized in the order in which they appear in σ_0, the last r requests for page x_1 must be moved to the front of σ_0. Since the

requests for pages x_2 and x_3 would be bumped back r positions, it would not be possible to perform any other moves. Therefore a reordering scheme, that maximizes chunks in order they appear in σ_0, produces a final request sequence $\sigma_f = x_1^{r+1}\ (x_2\ x_3)^{r+1}$. Since the cache size, k, is 1, it follows that $cost(LFD(\sigma_f)) = 2r + 3$.

A much better ordering (in fact an optimal one) can be obtained by moving the last r requests for page x_2 next to $\sigma_0(2) = x_2$. The final request sequence is $\sigma'_f = x_1\ x_2^{r+1}\ x_3^{r+1}\ x_1^r$ and its cost satisfies $cost(LFD(\sigma'_f)) = 4$. The result follows. ■

The following results describe properties about the behaviour of *LFD* on a chunk. The motivation was this: we wanted to bound the number of *LFD* faults on a chunk, so that we can obtain good bounds on the cost of *LFD* on a fixed request sequence and, implicitly, good bounds on the approximation factor or competitive ratio of a reordering scheme. Note that Corollary A.1 gives an upper bound on the cost incurred by *LFD* on a chunk. However we were not able to get a tight bound on the number of chunks in a request sequence, therefore we could not obtain good bounds on the cost of *LFD* on a fixed request sequence. This approach to proving bounds on the approximation factor or competitive ratio of a reordering scheme was not successful. We now present our results related to this approach in hope that new insights might arise from them.

We introduce the following notation: given a chunk B in some request sequence σ, let B_p denote the set of pages requested in B. Since we are dealing with the Classical model each chunk contains at most k distinct pages, so $|B_p| \leq k$. For each page x in B_p, let $f(x, B)$

denote the number of times that x is requested in B.

Lemma A.1 *Let k be arbitrary ($k \geq 1$). Let B be a chunk in request sequence σ and* LFD *serve σ. For any page x in B_p, LFD does not fault on any of the last $f(x, B) - 1$ requests to x in B.*

Proof:

We use proof by contradiction. Assume the i-th request ($i \geq 2$) for x in B (denote it $\overline{x}$) is a fault for *LFD*. The time this request is serviced is $\sigma^{-1}(\overline{x})$.

Since there is at least one previous request $\overline{x'}$ for page x in B, it must be that x was evicted after $\overline{x'}$ was served but before $\overline{x}$ was served. Let the request that caused a fault and this eviction of x, be request $\overline{y}$. x was evicted at time $\sigma^{-1}(\overline{y})$.

By the definition of *LFD*, at time $\sigma^{-1}(\overline{y})$, x was the page in cache whose next request was furthest in the future. Therefore all other $k - 1$ pages present in cache at time $\sigma^{-1}(\overline{y})$ were requested again before the next request for x. Since the next request for x is in B, all these $k - 1$ next requests must occur in B as well. Since request $\overline{y}$ was a fault, it must be distinct from all k pages in cache at time $\sigma^{-1}(\overline{y})$. Hence there exist at least $k + 1$ distinct pages in B. This contradicts the chunk3 definition, so the assumption is false. ■

The following upper bound on the cost of *LFD* in processing a chunk follows immediately from Lemma A.1, by noting that the number of distinct pages requested in any chunk is at most k.

Corollary A.1 *Let k be arbitrary ($k \geq 1$), σ be a request sequence and B be a chunk in σ. Let LFD serve σ. The cost incurred by LFD in processing B is at most k.*

A.2 Notation

Observe that some symbols (i, u, p, ...) have multiple uses but the correct meaning is obvious from the context.

$H_i = \sum_{j=1}^{i} \frac{1}{j} = i$-th harmonic number

$\phi = \frac{1+\sqrt{5}}{2}$ = golden ratio; also the empty set

k = size (in bytes) of the cache

k = maximum number of pages in cache (CLASSICAL cost model only)

σ = a request sequence

U = universe of requests (all pages that can be requested in σ)

u = size of the universe of requests

$n = |\sigma|$ = length of σ

i, j, g, h = indices of requests in σ

σ_0 = initial request sequence

σ_f = final request sequence (after a sequence of moves have been applied to σ_0)

σ_{g+1} = the request sequence obtained by applying move m to σ_g

$\sigma(i) = i$-th request in σ

$\sigma(i:j)$ = subsequence of σ containing $\sigma(i)$, $\sigma(i+1)$ ... up to $\sigma(j)$ ($i \leq j$)

$\sigma_g^{-1}(overlinex)$ = position of request $\overline{x}$ in σ_g

$C_{ALG,i}$ = contents of algorithm ALG's cache just before $\sigma(i)$ is served

$m = (j, i)$ = single move m that relocates $\sigma_g(j)$ to position i ($i \leq j$) (Definition A.15)

m = a number of repetitions (copies) of a request sequence

M = a sequence of moves

$\sigma^m = \sigma$ repeated m times

R_m = range of move m (Definition A.15)

$v(m)$ = value of move m (Definition A.17)

σ_M = request sequence obtained by applying all moves in M to σ

$v_{t,s}(m)$ = value of m relative to time t and suffix sequence σ_s (Definition A.20)

$v_h(m)$ = value of m relative to prefix $\sigma_g(1 : h)$ (Definition A.19)

$s(x)$ = size (in bytes) of page x

s_{min} = minimum size (in bytes) of a page in U

s_{max} = maximum size (in bytes) of a page in U

$c_f(x)$ = cost of fault on page x

s_σ = sum (in bytes) of sizes of all requests in σ

B = a chunk (Definition A.4)

$B(\sigma, k)$ = number of chunks in σ

$L(\sigma, k)$ = average chunk3 size in σ (Definition A.5)

$L_r(\sigma)$ = average request size in σ (Definition A.5)

a = parameter for (a, k)-referenced request sequences; they satisfy $L(\sigma, k) \geq ak$

$\mathcal{E}$ = cache eviction policy

$cost(\mathcal{E}(\sigma))$ = cost of processing σ using eviction policy $\mathcal{E}$

$c_{\mathcal{E}}$ = competitive ratio of eviction policy $\mathcal{E}$

$cost(\mathcal{E}(\sigma(g:h)))$ = cost of processing $\sigma(g:h)$ using eviction policy $\mathcal{E}$ and starting with an empty cache at g-th position

$c_{g,j} = cost(LFD(\sigma(g:j)))$ = cost incurred by LFD on $\sigma(g:j)$ starting with an empty cache at position g

$(\mathcal{R}, \mathcal{E})$ = a caching scheme composed of reordering algorithm $\mathcal{R}$ and eviction policy $\mathcal{E}$

$\mathcal{R}(\sigma)$ = ordering obtained by applying reordering algorithm $\mathcal{R}$ to σ

$cost(\mathcal{E}(\mathcal{R}(\sigma)))$ = cost of processing request sequence $\mathcal{R}(\sigma)$ using $\mathcal{E}$ as an eviction policy.

$c(\mathcal{R}, LFD)$ = competitive ratio of online caching scheme $(\mathcal{R}, LFD)$

$c(G, A, B)$ = competitive ratio of algorithm A on access graph G relative to algorithm B (Definition A.10)

l = size of look-ahead (Definition A.22)

p, q, x, y, z, v, w, u = requested pages

p = fault penalty for 1 byte

LFD = Belady's *Longest Forward Distance* algorithm

LRU = *Least Recently Used* algorithm

$FIFO$ = *First In First Out* algorithm

MRU = *Most Recently Used* algorithm

$LIFO$ = *Last In First Out* algorithm

LFU = *Least Frequently Used* algorithm

EDO = Evict Diametrically Opposed algorithm

NR = No Reordering algorithm

A.3 Definitions

Definition A.1 *In* **demand caching** *a page can be brought into cache if and only if it is the currently faulting page. We call such an operation a* **demand admission**. *An eviction policy using this model is called a* **demand eviction policy**.

In **non-demand caching** *any page can be brought into cache at any time and the additional processing cost is equal to the fault cost of the page that was brought into cache. We call such an operation a* **non-demand admission**.

Definition A.2 *In* **optional caching** *a faulting page may or may not be brought into cache. In* **non-optional caching** *every faulting page must be brought into cache.*

Definition A.3 *Let σ be a request sequence. We call subsequence $\sigma(i:j)$ a* **run** *if and only if it is a maximal subsequence of requests for a particular page; more precisely, for some page x*

(i) $\sigma(i:j) = x^{j-i+1}$ (i.e.: $j-i+1$ requests for x), and

(ii) $\sigma(i-1) \neq \sigma(i)$ and $\sigma(j) \neq \sigma(j+1)$

Definition A.4 *Let σ be a request sequence. The* **size** *of subsequence $\sigma(i:j)$ is the sum (in bytes) of sizes of all* **distinct** *pages requested in B. Given index $i > 0$, let $j \geq i$ be the*

maximum index such that the size of $\sigma(i:j)$ *is less than or equal to* k*. We say that* $\sigma(i:j)$ *is the* **chunk** *starting at index* i*.*

Definition A.5 *Let* σ *be a request sequence,* n *be its length and* s_σ *be the sum (in bytes) of sizes of all requests in* σ*. Let* $B(\sigma, k)$ *denote the number of chunks in* σ*. The* **average chunk size** *in* σ *is* $L(\sigma, k) = \frac{s_\sigma}{B(\sigma,k)}$ *and the* **average request size** *in* σ *is* $L_r(\sigma) = \frac{s_\sigma}{n}$*.*

Definition A.6 *Let* σ *be a request sequence that satisfies* $L(\sigma, k) \geq ak$*. We call* σ *an* (a, k)**-referenced request sequence**.

Definition A.7 ([6]) (Caching Models)

CLASSICAL - *Uniform page size and uniform cost of fault).*

WEIGHTED - *Uniform page size and variable cost of fault.*

FAULT - *Variable page size and uniform cost of fault. [19]*

GENERALIZED - *Variable page size and variable cost of fault.*

BIT - *Variable page size and cost of fault equals p times the page size (p is a constant). [19]*

DUAL-SIZE - *Two page sizes and variable cost of fault. [24]*

REAL - *Variable page size and cost of a fault is* $w_1 \cdot$ pageSize $+w_2$

(w_1 *and* w_2 *are positive constants). [24]*

Definition A.8 *The* **basic framework** *is one where hit costs are equal to* 0 *and fault costs are equal to* 1*. The* **Torng framework** *is one such that hit costs are equal to* 1*, and fault cost for a page* x *is equal to* $c_f(x) = p \cdot s(x) + 1$ *where* $s(x)$ *is the size in bytes of page* x *and* p *is the fault penalty for* 1 *byte.*

Definition A.9 *An* **access graph** *is a graph that models a particular set of possible request sequences. Each page in the universe of requests has a corresponding vertex in the access graph* G*. Given* 2 *vertices* v_p *and* v_q*,* $(v_p \rightarrow v_q)$ *is a (directed) edge if* q *is contained in the set of pages that can be immediately requested after a request for page* p*. An undirected edge* (v_p, v_q) *signifies both* p *and* q *can be requested immediately one after the other.*

Definition A.10 *Let* G *be an access graph and* S_G *be the set of all request sequences that are consistent with* G*. Let* $\mathcal{E}_1$ *and* $\mathcal{E}_2$ *be two eviction policies.* $\mathcal{E}_1$ *is* **c-competitive on G relative to** $\mathcal{E}_2$ *if*

$$\exists \text{ constant } b \text{ such that } \forall \sigma \in S_G, cost(\mathcal{E}_1(\sigma)) \leq c \cdot cost(\mathcal{E}_2(\sigma)) + b$$

The **competitive ratio of $\mathcal{E}_1$ on G relative to $\mathcal{E}_2$***, denoted by* $c(G, \mathcal{E}_1, \mathcal{E}_2)$ *is the smallest value of* c *such that* $\mathcal{E}_1$ *is* c*-competitive on* G *relative to* $\mathcal{E}_2$*.*

Definition A.11 *Let* σ *be a request sequence and* $(\mathcal{R}, \mathcal{E})$ *be a reordering scheme. Let* $\mathcal{R}(\sigma)$ *be the request sequence obtained by applying* $\mathcal{R}$ *to* σ*. The* **cost of reordering scheme** $(\mathcal{R}, \mathcal{E})$*, denoted by* $\boldsymbol{cost(\mathcal{E}(\mathcal{R}(\sigma)))}$*, is the cost of processing request sequence* $\mathcal{R}(\sigma)$ *using* $\mathcal{E}$ *as an eviction policy.*

Definition A.12 *Let σ be a request sequence, $\mathcal{R}$ be a deterministic offline reordering algorithm, and OPT be an optimal offline reordering algorithm. We say that the offline reordering scheme $(\mathcal{R}, LFD)$ is an* **α-approximation** *if*

$$cost(LFD(\mathcal{R}(\sigma))) \leq \alpha \cdot cost(LFD(OPT(\sigma))) + b, \forall \sigma \text{ and some constant } b$$

The **approximation factor** *of $(\mathcal{R}, LFD)$, denoted by $\alpha(\mathcal{R}, LFD)$ is the smallest value of α such that $(\mathcal{R}, LFD)$ is an α-approximation.*

Definition A.13 *Let σ be a request sequence, $\mathcal{R}$ be a deterministic online reordering algorithm, and OPT be an optimal offline reordering algorithm. We say that the online reordering scheme $(\mathcal{R}, LFD)$ is* **c-competitive** *if*

$$cost(LFD(\mathcal{R}(\sigma))) \leq c \cdot cost(LFD(OPT(\sigma))) + b, \forall \sigma \text{ and some constant } b$$

The **competitive ratio** *of $(\mathcal{R}, LFD)$, denoted by $c(\mathcal{R}, LFD)$ is the smallest value of c such that $(\mathcal{R}, LFD)$ is c-competitive.*

Definition A.14 *Let $r \geq 1$ be a fixed parameter. Let σ_0 be the initial request sequence and σ_f be the final request sequence that a reordering algorithm produces. The constraints that any reordering algorithm must satisfy are:*

- FEDER *model ([14]): request $\overline{y}$ can be served before request $\overline{x}$ if $\sigma_0^{-1}(\overline{y}) - \sigma_0^{-1}(\overline{x}) < r$, $\forall$ requests $\overline{x}$ and $\overline{y}$ in σ_0*

- BIDIRECTIONAL *model:* $|\sigma_f^{-1}(\overline{x}) - \sigma_0^{-1}(\overline{x})| \leq r$, $\forall$ *request* $\overline{x}$ *in* σ_0
- UNIDIRECTIONAL *model:* $\sigma_f^{-1}(\overline{x}) - \sigma_0^{-1}(\overline{x}) \leq r$, $\forall$ *request* $\overline{x}$ *in* σ_0

In all three models the parameter r *is called the* **reordering threshold**.

Definition A.15 *Let* σ *be any request sequence. A* **move** $m = (j, i)$ *applied to* σ *is an operation that relocates request* $\sigma(j)$ *to position* i. *Let the resulting request sequence be denoted by* σ'. *Move* m *is a* **forward move** *if* $\sigma(j)$ *is moved closer to the front of* σ $(j > i)$. *Move* m *is a* **backward move** *if* $\sigma(j)$ *is moved closer to the back of* σ $(j < i)$. *Its* **from-coordinate** *and* **to-coordinate** *are* j *and* i *respectively. If* m *is a forward move, the* **range** *of* m *is* $R_m = [\![i, j]\!]$ *and* m *shifts every request in* $\sigma(i : j-1)$ *backward by one position. I.e.:* $\sigma'(1 : i-1) = \sigma(1 : i-1)$, $\sigma'(i) = \sigma(j)$, $\sigma'(i+1 : j) = \sigma(i : j-1)$, *and* $\sigma'(j+1 : n) = \sigma(j+1 : n)$. *If* m *is a backward move, the* **range** *of* m *is* $R_m = [\![j, i]\!]$ *and* m *shifts every request in* $\sigma(j+1 : i)$ *forward by one position. I.e.:* $\sigma'(1 : j-1) = \sigma(1 : j-1)$, $\sigma'(j : i-1) = \sigma(j+1 : i)$, $\sigma'(i) = \sigma(j)$, *and* $\sigma'(i+1 : n) = \sigma(i+1 : n)$.

Definition A.16 *Let* σ *be a request sequence. Let* m_1 *and* m_2 *be two moves that are applied to* σ *and* $R_1 = [\![i_1, j_1]\!]$ *and* $R_2 = [\![i_2, j_2]\!]$ *be their ranges. If* $i_1 \leq i_2 \leq j_1$ *or* $i_2 \leq i_1 \leq j_2$, *then* m_1 *and* m_2 *are* **overlapping** *moves.*

Definition A.17 *Let* σ_{g+1} *be the request sequence obtained by applying move* m *to request sequence* σ_g. *The* **value of a move** m *is*

$$v(m) = cost(LFD(\sigma_g)) - cost(LFD(\sigma_{g+1}))$$

Definition A.18 *Let m be a move. If $v(m) > 0$ then m is a* **positive move**. *If $v(m) < 0$ then m is a* **negative move**. *If $v(m) = 0$ then m is a* **neutral move**.

Definition A.19 *Let $1 \leq g \leq h \leq n$. Let m be a move that is applied to request sequence σ_g. Let σ_{g+1} be the resulting request sequence.*

The **value** *of move m* **relative to the prefix** *$\sigma_g(1:h)$ is*

$$v_h(m) = cost(LFD(\sigma_g(1:h))) - cost(LFD(\sigma_{g+1}(1:h)))$$

Definition A.20 *Let a request sequence σ_s of length h be appended at the end of request sequence $\sigma_g(1:j)$. Let the resulting request sequence be denoted by σ. Let σ' be the request sequence obtained by applying move $m = (j, i)$ to σ. The* **value of m relative to time t and suffix sequence σ_s** *is the difference of* LFD *cost on prefix sequences $\sigma(1:t)$ and $\sigma'(1:t)$:*

$$v_{t,s}(m) = cost(LFD(\sigma(1:t))) - cost(LFD(\sigma'(1:t)))$$

The **current** *and* **full** *values of m relative to suffix σ_s are the values of m relative to σ_s and to times j and $j + h$, respectively ($v_{j,s}(m)$ and $v_{j+h,s}(m)$).*

Definition A.21 *An* **offline reordering algorithm** *is a reordering algorithm that can observe all requests in the request sequence, move of any of them, and revoke any of its decisions. An* **online reordering algorithm** *is a reordering algorithm which cannot observe any requests past the current request, can move forward only the current request, and makes irrevocable decisions.*

Definition A.22 *An online reordering scheme has* **look-ahead** *of size l if, given the current request in position j, the reordering scheme can observe all requests in $\sigma(j : j + l)$.*

Bibliography

[1] D. Achlioptas, M. Chrobak, and J. Noga. Competitive analysis of randomized paging algorithms. *In Proceedings of the 4th European Symposium on Algorithms, LNCS 1136*, pages 419–430, 1996.

[2] D. Achlioptas, M. Chrobak, and J. Noga. Competitive analysis of randomized paging algorithms. *Theoretical Computer Science*, 234:203–218, 2000.

[3] S. Albers. New results on web caching with request reordering. In *Proceedings of the 16th Annual ACM Symposium on Parallel Algorithms*, pages 84–92, 2004.

[4] S. Albers, S. Arora, and S. Khanna. Page replacement for general caching problems. In *Proceedings of the 10th annual ACM-SIAM Symposium on Discrete algorithms*, 1999.

[5] S. Angelopoulos, R. Dorrigiv, A. López-Ortiz, and J. I. Munro. Private communication. 2004.

[6] K. Au, N. Dumir, C. Gaspar, B. Genc, and S. M. Liong. Generalized caching. *Project report for course CS860, School of Computer Science, University of Waterloo, Waterloo, ON*, 2004.

[7] O. Bahat and A. Makowski. Optimal replacement policies for non-uniform cache objects with optional eviction. In *Proceedings of the IEEE Infocom 2003 Conference*, 2003.

[8] L. A. Belady. A study of replacement algorithms for virtual storage computers. *IBM Systems Journal*, 5:78–101, 1966.

[9] P. Van Emde Boas. Preserving order in a forest in less than logarithmic time and linear space. *Information Processing Letters*, 6:80–82, 1977.

[10] A. Borodin and R. El-Yaniv. *Online Computation and Competitive Analysis*. Cambridge University Press, 1998.

[11] A. Borodin, S. Irani, P. Raghavan, and B. Schieber. Competitive paging with locality of reference. *Journal of Computer and System Sciences*, 50(2):244–258, 1995.

[12] P. Cao and S. Irani. Cost-aware WWW proxy caching algorithms. In *Proceedings of the 1997 Usenix Symposium on Internet Technologies and Systems (USITS-97)*, Monterey, CA, 1997.

[13] M. Chrobak and J. Noga. LRU is better than FIFO. In *Proceedings of the 9th Annual ACM-SIAM Symposium on Discrete Algorithms*, pages 78–81. Society for Industrial and Applied Mathematics, 1998.

[14] T. Feder, R. Motwani, R. Panigrahy, S. Seiden, R. van Stee, and A. Zhu. Combining request scheduling with web caching. *Theoretical Computer Science*, 324(2-3):201–218, 2004.

[15] A. Fiat and A. R. Karlin. Randomized and multipointer paging with locality of reference. In *Proceedings of the 27th Annual ACM Symposium on Theory of Computing (STOC)*, pages 626–634, 1995.

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

[17] J. L. Hennessy and D. A. Patterson. *Computer Architecture: A Quantitative Approach.* Morgan Kaufmann, 2nd edition, 1996.

[18] S. Hosseini-Khayat. On optimal replacement of nonuniform cache objects. *IEEE Transactions on Computers*, 49(8):769–778, 2000.

[19] S. Irani. Page replacement with multi-size pages and applications to web caching. *Algorithmica*, 33(3):384–409, 2002.

[20] S. Irani. Randomized weighted caching with two page weights. *Algorithmica*, 32(4):624–640, 2002.

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

[22] A. R. Karlin, M. S. Manasse, L. Rudolph, and D. D. Sleator. Competitive snoopy caching. *Algorithmica*, 3(1):70–119, 1988.

[23] E. Koutsoupias and C. H. Papadimitriou. Beyond competitive analysis. *SIAM Journal on Computing*, 30(1):300–317, 2000.

[24] A. López-Ortiz. Private communication. 2004.

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

[26] R. Kaas P. Van Emde Boas and E. Zijlstra. Design and implementation of an efficient priority queue. *Mathematical Systems Theory*, 10:99–127, 1977.

[27] P. Raghavan and M. Snir. Memory versus randomization in online algorithms. *In Proceedings of the 16th ICALP, LNCS 372*, pages 687–703, 1989.

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

[29] E. Torng. A unified analysis of paging and caching. *Algorithmica*, 20:175–200, 1998.

[30] D. Wells. *The Penguin Dictionary of Curious and Interesting Numbers*. Middlesex, England: Penguin Books, 1986.

[31] N. E. Young. On-line caching as cache size varies. In *Proceedings of the 2nd Annual ACM-SIAM symposium on Discrete Algorithms*, pages 241–250, 1991.

[32] N. E. Young. On-line paging against adversarially biased random inputs. *Journal of Algorithms*, 37(1):218–235, 2000.

[33] N. E. Young. Competitive paging and dual guided online weighted caching and matching algorithms. *Ph.D. thesis. Department of Computer Science, Princeton University, Princeton, NJ*, October 1991.