The Thesis committee for Anil Kumar Katti
certifies that this is the approved version of the following thesis:

Competitive Cache Replacement Strategies for a Shared Cache

APPROVED BY

SUPERVISING COMMITTEE:

Vijaya Ramachandran, Supervisor

Greg Plaxton

Competitive Cache Replacement Strategies for a Shared Cache

by

Anil Kumar Katti, B.E.

THESIS

Presented to the Faculty of the Graduate School of

The University of Texas at Austin

in Partial Fulfillment

of the Requirements

for the Degree of

MASTER OF SCIENCE IN COMPUTER SCIENCE

THE UNIVERSITY OF TEXAS AT AUSTIN

May 2011

Acknowledgments

I acknowledge the role my advisor Vijaya Ramachandran and sincerely thank her for the constant support and encouragement. She introduced me to the world of research in theory, which I throughly enjoyed over the last couple of years. I can now appreciate the beauty in subtle algorithms and theory behind them. Along with research, Dr. Ramachandran also helped me understand the importance of perseverance in life. Thank you, Professor!

I would also like to thank my parents for their blessings and continued support. I acknowledge the role played by my wife Divya. Even though the last 2 years were difficult, her attitude never reflected it. I thank her for the support. I am also grateful to all my friends from whom I derive constant inspiration. Special thanks to Chao Raun who was my project partner for two semesters. Technical discussions I have had with her were extremely helpful in creating this thesis.

I wholeheartedly thank Greg Plaxton for his help with the Thesis review and the Algorithms course. Both of these helped me immensely in giving this thesis a good shape. I also acknowledge and appreciate the role played by all professors in the Department of Computer Science at The University of Texas at Austin. Special thanks to all professors with whom I took courses — Vijaya Ramachandran, Peter Stone, Dana Ballard, Greg Plaxton, Maggie Myers, Michael Walfish, Jayadev Misra and Al Bovik.

Thank you all for this wonderful journey!

Anil Kumar Katti

The University of Texas at Austin
May 2011

Competitive Cache Replacement Strategies for a Shared Cache

Anil Kumar Katti, M.S.C.S
The University of Texas at Austin, 2011

Supervisor: Vijaya Ramachandran

We consider cache replacement algorithms at a shared cache in a multicore system which receives an arbitrary interleaving of requests from processes that have full knowledge about their individual request sequences. We establish tight bounds on the competitive ratio of deterministic and randomized cache replacement strategies when processes share memory blocks. Our main result for this case is a deterministic algorithm called GLOBAL-MAXIMA which is optimum up to a constant factor when processes share memory blocks. Our framework is a generalization of the application controlled caching framework in which processes access disjoint sets of memory blocks. We also present a deterministic algorithm called RR-PROC-MARK which exactly matches the lower bound on the competitive ratio of deterministic cache replacement algorithms when processes access disjoint sets of memory blocks. We extend our results to multiple levels of caches and prove that an exclusive cache is better than both inclusive and non-inclusive caches; this validates the experimental findings in the literature. Our results could be applied to shared caches in multicore systems in which processes work together on multithreaded computations like Gaussian elimination paradigm, fast Fourier transform, matrix multiplication, etc. In these computations, processes have full knowledge about their individual request sequences and can share memory blocks.

Table of Contents

List of Tables

Chapter 1

Introduction

This thesis studies cache replacement algorithms for a shared cache in the multicore setting. Cache replacement algorithms decide which block in the cache needs to be replaced by the newly requested block in order to minimize the number of cache misses incurred on a request sequence. The decision taken by a cache replacement algorithm has to be online (i.e., the eviction decision has to be based only on the requests seen in the past and the current request) in order for the algorithm to be implementable in practical systems. The caching problem is derived from the classical paging problem [22] and hence most of the results on the paging problem directly hold for the caching problem. These problems have a rich historical background and practical importance. A number of models and frameworks have been proposed in the literature to develop and analyze cache replacement algorithms. We give a quick overview of the previous models before describing our model, results and approach.

The competitive analysis framework proposed by Sleator and Tarjan [22] for the paging problem has been used extensively to measure and compare the performance of different cache replacement algorithms. The competitive ratio of a cache replacement algorithm ALG is defined as the ratio of number of cache misses incurred by ALG to the number of cache misses incurred by an optimal offline cache replacement algorithm (OPT) on a worst case request sequence. Belady [4] proposed a simple offline greedy strategy which evicts the block that is requested farthest in the future request sequence (FITF) from the cache in case of a cache miss. It was also shown in [4] that FITF is optimal in the classical caching framework.

Borodin et al. [6] introduced the notion of access graphs in order to model locality of reference. An access graph is a graph (undirected or directed) that governs the pattern of block requests in the request sequence. The nodes

in the access graph represent memory blocks and the edges in the access graph control the pattern of requests in the following fashion: The request that follows a request to node x could either be another request to x or to some neighbor of x in the access graph. A natural algorithm called FAR was proposed for the access graph framework in [6]. It was shown that FAR is optimal (up to a constant factor) in the access graph framework by Irani, Karlin and Philips [16].

Two different approaches to the caching in the parallel setting were considered by Fiat and Karlin [12] and Cao et al. [8]. Fiat and Karlin [12] introduced the multi-pointer access graph framework in which an undirected access graph with multiple pointers pointing to the nodes of the graph is assumed to govern the request sequence reaching the cache. The multiple pointers are used to model multiple processors accessing data in parallel. Cao et al. [8] introduced the application controlled caching framework in which processes (or applications) having varying degrees of knowledge about their future request sequences share a cache. In both of these frameworks, the interleaving of requests from the individual request sequences as seen at the shared cache is assumed to be fixed, and is assumed to be adversarial for the competitive ratio.

Cao et al. [8] presented experimental results in the application controlled caching framework to support the deterministic algorithm that they proposed. Barve et al. [3] formalized the application controlled caching framework where processes access disjoint sets of memory blocks and each process has varying degrees of knowledge about its future request sequence. The interleaving of requests from the request sequences is assumed to be fixed and adversarial as in [8, 12]. In [3], lower bounds were established on the competitive ratio of deterministic and randomized online cache replacement algorithms at a shared cache when each process has full knowledge about its future request sequence. It was shown in [3] that the algorithm proposed in [8] is optimal up to a constant factor in the application controlled caching framework when each process has full knowledge about its future request sequence. Note that the case when processes have no knowledge about their future request sequences reduces to the classical (or sequential) caching and hence a simple deterministic cache replacement algorithm like LRU is optimal in this scenario. Barve et

al. also proposed a randomized algorithm in the application controlled caching framework and proved that its competitive ratio was optimal up to a constant factor when each process has full knowledge about its future request sequence. The scenario in which the processes share memory blocks was left as an open problem. We consider this open problem in our thesis.

We consider the application controlled caching framework where processes share memory blocks and each process has full knowledge about its future request sequence. The interleaving of requests from these request sequences is still assumed to be fixed and adversarial. We call this generalization of the application controlled caching framework the *shared memory framework*. We establish tight bounds (up to a constant factor) on the competitive ratio of deterministic and randomized algorithms in this framework. Our main result in this case is a deterministic algorithm called GLOBAL-MAXIMA which is optimum up to a constant factor. Note that even though each process has full knowledge about its individual future request sequence, the cache replacement algorithm at the shared cache does not have any control of or knowledge about the interleaving of requests from these request sequences. On the other hand, offline algorithms have complete control of and knowledge about the interleaving.

We call the application controlled caching framework where processes access disjoint sets of memory blocks and have full knowledge about their individual request sequence the *disjoint memory framework*. In this framework, we propose a natural algorithm called RR-PROC-MARK which matches the lower bound established in [3].

Modern multicore systems have multiple levels of caches to get better caching performance. We model multicore systems with multiple levels of caches using the hierarchical caching framework. We consider LRU and RR-PROC-MARK and analyze their competitive ratio at inclusive, exclusive and non-inclusive L_2 caches. We prove that an exclusive cache is better than inclusive and non-inclusive caches.

1.1 An overview of the thesis

In this section we describe the model that we consider in our research, state previously known results and discuss our results, motivation and approach.

1.1.1 Disjoint and shared memory frameworks

Our main focus is on the analysis of cache replacement algorithms at a shared cache in the disjoint and shared memory frameworks. We formally describe these frameworks as follows:

Features common to both disjoint and shared memory frameworks: The following features are common to both the disjoint memory framework and the shared memory framework. We list features specific to each of these two frameworks after listing the common features.

1. Processes access a shared cache and the cache replacement algorithms are designed for the shared cache.

2. Each process has full knowledge about its individual request sequence. By request sequence we mean the exact order in which memory blocks are requested by the process in the future.

3. An interleaving of requests from these request sequences is seen at the shared cache. We analyze the performance of cache replacement algorithms developed for the shared cache with respect to a worst case interleaving.

4. The online algorithms do not have any knowledge about the interleaving of requests from these request sequences reaching the shared cache.

5. The optimal offline algorithm on the other hand, has complete control of and knowledge about the interleaving of these request sequences.

6. The interleaving of requests from these request sequences is assumed to be fixed. i.e., the same interleaving reaches the shared cache for

all of the cache replacement algorithms, including the optimal offline cache replacement algorithm — irrespective of their cache replacement decisions.

Feature specific to the disjoint memory framework: In the disjoint memory framework, the processes request disjoint sets of memory blocks. Thus, every block in the memory can be accessed by at most one process.

Feature specific to the shared memory framework: In the shared memory framework, the processes share memory blocks. Thus, any block in the memory can be accessed by any subset of processes.

1.1.1.1 Connecting these models to the real world

Relating to the real world: In a number of parallel shared memory applications, processes or cores work together to solve a given problem and hence share memory blocks. In most of these computations, each process also has full knowledge about its future request sequence of its current task. For instance, the computation of Gaussian elimination paradigm as discussed by Chowdhury and Ramachandran [9] has this type of behavior. Even the computation of matrix multiplication and fast Fourier transform have this type of behavior. Chowdhury et al. [10] discuss these types of computations in more detail.

Another scenario in which the shared memory model relates well to the real world is when modern multicore processor systems are equipped with hardware and software prefetchers [7, 20]. These units predict the future request sequence in an online fashion for most of the computations.

Both of these examples motivate us to consider the shared memory framework where processes share memory blocks and each process has full knowledge about its request sequence.

Deviation from the real world: In modern multicore systems, requests from all of the cores reach the shared cache in parallel. When a cache miss is incurred, only the core that requested the block which was a cache miss

gets delayed. All other cores can continue requesting memory blocks. Thus, the interleaving of request sequences depends on the cache misses incurred by the cache replacement algorithm at the shared cache. We refer to this type of interleaving as free interleaving. Hassidim [15] and Lopez-Ortiz and Salinger [18] considered free interleaving while analyzing cache replacement algorithms at the shared cache.

On the other hand, modern operating systems interrupt processes for a number of reasons (like, serving system calls, priority scheduling, etc.) and hence the interleaving does not necessarily depend on just the delays due to cache misses. This was stated as the motivation behind fixed and adversarial interleaving considered in [8]. We consider this type of interleaving in our thesis.

1.1.2 Notation used in this thesis

1. We let p processes share a cache of size k. We let P_i denote the ith process.

2. Observe that k is at least as large as p. This is because each process should have at least one block in the cache for its use. Usually, k is orders of magnitude larger than p.

3. We use σ_i to represent the request sequence of the ith process. Recall that, the request sequence is the exact order in which memory blocks are requested by the process in the future.

4. An arbitrary interleaving of $\sigma_1, \sigma_2, \ldots, \sigma_p$ is seen at the shared cache. We develop cache replacement algorithms for the shared cache and analyze their performance with respect to a worst case interleaving (represented by σ). Recall that the same interleaving is seen by both online and offline algorithms. The difference is that offline algorithm knows the exact sequence of requests in σ but online algorithm doesn't.

5. If a memory block q can be accessed by a process P_i, we say that q *belongs to* P_i and P_i *owns* q: these terms make sense only in the disjoint memory framework. In the shared memory framework, any block can be accessed by a subset of processes.

1.1.3 Results

The first part of Table 1.1 and 1.2 describes the previously known results for deterministic and randomized algorithms respectively.

The following are some important contributions of our research:

1. **The shared memory framework:**
   (a) Lower bounds on the competitive ratio of deterministic and randomized algorithms in the shared memory framework. [Section 4.3]
   (b) A deterministic algorithm called GLOBAL-MAXIMA which is optimal (up to a constant factor ≤ 5) in the shared memory framework. [Section 4.4]
2. **The disjoint memory framework:**
   (a) A deterministic algorithm called RR-PROC-MARK which closes the gap between the upper and lower bounds on the competitive ratio of deterministic algorithms in the disjoint memory framework. [Section 3.3]
   (b) Analysis of RR-PROC-MARK in a more realistic cost model called the full access cost model. [Section 3.3.3]
3. **The hierarchical caching framework:** Analysis of cache replacement algorithms at inclusive, non-inclusive and exclusive L_2 caches in the hierarchical caching framework. [Chapter 5]

Results on deterministic algorithms: Our results on deterministic algorithms are tabulated in the second part of Table 1.1.

We classify cache replacement algorithms in the shared memory framework into 2 types — *local* and *global* algorithms. Local algorithms make eviction decisions based on the local knowledge available at a particular process and on the other hand, global algorithms make eviction decisions based on the global knowledge available from all of the p processes.

In the shared memory framework, we establish a lower bound of $\frac{p}{2}\log\frac{4(k+1)}{3p}$ on the competitive ratio of deterministic local and global algorithms. We give a natural deterministic marking algorithm called GLOBAL-MAXIMA and establish an upper bound of $2(p\ln(\frac{ek}{p})+1)$ on its competitive ratio. Further, we show that, if the cache replacement strategy used by deterministic local algorithms is known, the lower bound on the competitive ratio can be pushed to k. Note that an upper bound of k can be obtained by using a simple deterministic algorithm like LRU at the shared cache by completely ignoring knowledge about future request sequences.

In the disjoint memory framework, we introduce a family of algorithms called the process marking algorithms that is motivated from the randomized algorithm presented in [3]. We give a natural deterministic process marking algorithm called RR-PROC-MARK which matches the lower bound of $p+1$ on the competitive ratio of deterministic algorithms established by [3]. We analyze the performance of RR-PROC-MARK in a more realistic cost model called the *full access cost model*.

Results on randomized algorithms: Our result on randomized algorithms is tabulated in the second part of Table 1.2.

In the shared memory framework, we establish a lower bound of $\frac{1}{2}\log(k+1)$ on the competitive ratio of randomized algorithms. Note that an upper bound of H_k can be obtained by using a randomized algorithm like PARTITION (due to McGeoch and Sleator [19]) at the shared cache by completely ignoring knowledge about future request sequences. We conclude that randomization is not of much help in the shared memory framework.

Results in the hierarchical caching framework: We analyze the competitive ratio of cache replacement algorithms in a hierarchical caching framework. Our results in the hierarchical caching framework are presented in Table 1.3. We introduce the concept of effective size of the L_2 cache in order to analyze online cache replacement algorithms at inclusive, exclusive and non-inclusive L_2 caches as seen in modern multicore systems like Intel-Nehalem and AMD-Shanghai. We consider cache replacement algorithms in the sequential

and parallel disjoint memory models and establish upper and lower bounds on their competitive ratio.

1.1.4 Our approach

Our algorithms and analysis build on the methodology developed for multi-pointer access graph [12] and application controlled caching [3] frameworks.

Our approach for the shared memory framework: We present a deterministic cache replacement algorithm called GLOBAL-MAXIMA in the shared memory framework. GLOBAL-MAXIMA proceeds in phases. Upon a cache miss, GLOBAL-MAXIMA evicts a block from the cache that has the maximum global distance among all of the blocks that were not requested in the current phase. The global distance of a memory block x is the minimum over all of the local distances of x. Further, the local distance of x with respect to a particular process is the number of unmarked blocks in cache that occur before the first request to x in the request sequence of that process.

GLOBAL-MAXIMA can be considered as a parallel adaptation of FAR which was proposed in [6] for the single-pointer access graph framework. FAR proceeds in phases. Upon a cache miss, FAR evicts a block from the cache which is farthest from the set of already requested blocks in the access graph. We describe access graphs in more detail in the next chapter.

We use the concept of holes (introduced in [3]) for analysis. A hole essentially represents a memory block that is missing from the cache. Upon a request to a block that is not in the cache, an existing block is evicted and hence a hole is created in its place. Request to this missing block causes another cache miss. This process continues till the end of the phase. We bound the total cost of GLOBAL-MAXIMA by bounding the maximum number of cache misses attributed to each hole during a given phase of the algorithm.

In order to establish lower bounds on the deterministic and randomized algorithms in the shared memory framework, we fix the request sequence for each of the p processes and construct an adversarial interleaving. For the randomized case we use the von Neumann minimax principle as described by

Yao [27] to give a probability distribution on the interleaving. This principle states that the expected cost of any deterministic algorithm on the probabilistic interleaving gives a lower bound on the cost of the randomized algorithms on the interleaved sequence.

Our approach for the disjoint memory framework: We present a deterministic cache replacement algorithm called RR-PROC-MARK in the disjoint memory framework. It improves the competitive ratio from $2(p+1)$ to $p+1$ (lower bound presented in [3]). This algorithm is motivated from the following observation about the algorithm proposed in [8]: upon a request to a clean block, the algorithm proposed in [8] evicts a block from the cache resulting in the creation of a hole. In the worst case, all of these holes could be created in one of the p processes. They all could move from one process to another in a group, leading to a maximum number of cache misses for every such movement. This led to a factor of 2 in the upper bound on the competitive ratio. In contrast, RR-PROC-MARK carefully distributes these holes among all of the processes by using a simple scheme (round robin) for selecting processes which are then asked to make evictions. We show that this scheme is effective in reducing the constant factor in the upper bound.

Our approach for the hierarchical caching framework: Modern multicore systems have a hierarchy of caches in order to obtain a better caching performance. We present the hierarchical caching framework to model such systems. We consider two levels of shared caches and compare the inclusive, exclusive and non-inclusive properties in both sequential and parallel disjoint memory frameworks.

For the sequential caching framework, we consider LRU and obtain an upper bound on the competitive ratio when the L_2 cache is inclusive, exclusive and non-inclusive of the L_1 cache. For the parallel disjoint memory framework, we consider RR-PROC-MARK and obtain an upper bound on the competitive ratio first in the (h,k)-paging context and then at the L_2 cache.

Table 1.1: Competitive ratio of deterministic cache replacement algorithms. Algorithms in the classical caching framework assume no knowledge about future request sequence and the algorithms in the parallel caching frameworks assume that each process has full knowledge about its individual request sequence.

Deterministic cache replacement algorithms		
Model	**Lower Bound**	**Upper Bound**
Known results		
Classical-caching (sequential)	k [22]	k [22] LRU, FIFO
Disjoint memory framework (parallel)	$p+1$ [3]	$2(p+1)$ [8] and [3] LRU-PROC-MARK
Shared memory framework (parallel)	Open	Open
Our results		
Disjoint memory framework (parallel)	$p+1$ [3]	$p+1$ [Thm 3.3.9] RR-PROC-MARK
Shared memory framework (parallel)	$\frac{p}{2}\log\frac{4(k+1)}{3p}$ [Thm 4.3.2]	$2(p\ln(\frac{ek}{p})+1)$ [Thm 4.4.1] GLOBAL-MAXIMA
Upper bound on RR-PROC-MARK in full access cost model		
$1+\frac{2(H_p+O(1))(b-1)}{b+1}$ [Thm 3.3.10]		

Table 1.2: Competitive ratio of randomized cache replacement algorithms. Algorithms in the classical caching framework assume no knowledge about future request sequence and the algorithms in the parallel caching frameworks assume that each process has full knowledge about its individual request sequence.

Randomized cache replacement algorithms		
Model	**Lower Bound**	**Upper Bound**
Known results		
Classical-caching (sequential)	H_k [13]	H_k [19] PARTITION
Disjoint memory framework (parallel)	H_{p-1} [3]	$2H_{p-1} + 2$ [3] RAND-PROC-MARK
Shared memory framework (parallel)	Open	Open
Our result		
Shared memory framework (parallel)	$\frac{1}{2}\log(k+1)$ [Thm 4.3.4]	H_k [19] PARTITION

Table 1.3: Competitive ratio of cache replacement strategies in the (h, k)-paging framework. Algorithms in the classical caching framework assume no knowledge about future request sequence and the algorithms in the parallel caching frameworks assume that each process has full knowledge about its individual request sequence.

(h,k)-**paging framework**		
Model	**Lower Bound**	**Upper Bound**
Known results		
Sequential (Deterministic)	$\frac{k}{k-h+1}$ [22]	$\frac{k}{k-h+1}$ [22] LRU, FIFO
Sequential (Randomized) when $h = k$	H_k [13]	H_k [19] PARTITION
Sequential (Randomized) when $h < k$ and $\frac{k}{k-h} > e$ for a constant e	$\ln \frac{k}{k-h} - \ln\ln \frac{k}{k-h} - \frac{2}{k-h}$ [28]	$2(\ln \frac{k}{k-h} - \ln\ln \frac{k}{k-h} + \frac{1}{2})$ [28] RAND-MARK
Our result		
Parallel disjoint memory (h,k)-paging (Deterministic) when $h = k$ [Thm 5.3.1]	$p + 1$	$p + 1$ RR-PROC-MARK
Parallel disjoint memory (h,k)-paging (Deterministic) when $h \leq c \cdot k$: for a constant $c < 1$ [Thm 5.3.1]	O(1)	O(1) RR-PROC-MARK
Parallel disjoint memory (h,k)-paging (Randomized)	Open	$\frac{2k}{k-h+k/(H_{p-1}+1)}$ [Thm 5.3.4] RAND-PROC-MARK

Table 1.4: Effective size of the L_2 cache for online marking algorithms. These values of k and h can be directly applied for the algorithms presented in the previous two tables in order to obtain the competitive ratio at the shared L_2 cache.

Model	**Effective cache size (k)**
Inclusive cache	k_2
Non-inclusive cache	k_2
Exclusive cache	$k_1 + k_2$

Note: Proofs can be found in [Thm 5.2.2]. The effective size of the L_2 cache for the optimal offline algorithm is $h = h_1 + h_2$.

Chapter 2

The Caching Problem

The caching problem is similar to the *classical paging problem* which was introduced in the virtual memory context. In the paging problem, the RAM (which is modelled as the main memory) is the fast memory and the disk drive is the slow memory whereas in the caching problem, the cache is the fast memory and the main memory is in fact considered the slow memory (in comparison to the cache). Caches help bridge the gap between the speed of the processor and the main memory. Caches are typically placed much closer to the processor than the main memory and hence they are both faster and smaller (due to chip real estate constraints). Because they are much smaller in size when compared to the main memory, the number of memory blocks that can be stored in them is limited. This leads to the problem of caching.

In general a cache lies in between a processor or a set of processors and the main memory. When the processor wants to access a memory block, it sends a block request to the cache. If the cache already contains the requested block, the block request is served immediately. If the cache does not have the requested block, it sends a block request to the main memory. The block is fetched from the main memory into the cache before the processor can use it. A *cache miss* occurs when the processor requests a memory block which does not exist in the cache and a *cache hit* occurs when the processor requests a memory block which exists in the cache. A cache miss is more expensive than a cache hit because the requested block has to be read from the main memory and the main memory is usually orders of magnitude slower than the cache.

Upon a cache miss, an existing block should be *evicted* from the cache in order to accommodate the newly requested block. The caching problem is to determine the block that should be evicted from the cache on every cache miss. The goal of a *cache replacement algorithm* is to solve the cache problem

while incurring a minimum number of cache misses on any given sequence of block requests. A cache replacement algorithm could be either *online* or *offline*. Online cache replacement algorithms make eviction decisions based on just the current block request and block requests seen till that moment. Offline cache replacement algorithms, on the other hand, make eviction decisions based on the entire sequence of block requests, including the future block requests. Any cache replacement algorithm used in practice has to be online in order to be implementable.

In this chapter, we consider cache replacement algorithms in different scenarios — the classical caching context, caching with access graph, caching in the parallel context and caching at multiple levels of caches. We describe the model proposed in the literature for all these scenarios and discuss the performance of cache replacement algorithms proposed in the past. In each of these models, the performance of the cache replacement algorithms is measured using an analytical tool called *competitive analysis*. We first describe competitive analysis, before presenting a survey on the models and cache replacement algorithms proposed in the literature.

Competitive analysis

Competitive analysis is an analytical tool developed by Sleator and Tarjan [22] to measure and compare the performance of cache replacement algorithms. Let ALG be an online cache replacement algorithms and let σ represent an arbitrary request sequence (sequence of block requests). We use $\text{A}(\sigma)$ to represent the number of cache misses incurred by a cache replacement algorithm A on block requests in σ. The competitive ratio of ALG is defined as the ratio of number of cache misses incurred by ALG to the number of cache misses incurred by an optimal offline cache replacement algorithm (OPT) on the worst case request sequence. More formally, the competitive ratio of ALG is said to be c if there exists a constant c' such that:

$$\forall \sigma : \text{ALG}(\sigma) \leq c\text{OPT}(\sigma) + c'$$

Belady [4] proposed a simple offline greedy algorithm called FITF which evicts a block from the cache that is requested farthest in the future request

sequence upon a cache miss. It was also proved in [4] that FITF is optimal in the classical caching framework. FITF is also referred to as LFD in the literature. LFD stands for longest forward distance indicating that the block which is evicted by LFD has the longest forward distance in the future request sequence.

2.1 A survey of cache replacement algorithms

In this section, we present the models and cache replacement algorithms developed in the past. In each of these models, we describe the known algorithms and present known upper and lower bounds on their competitive ratio.

2.1.1 Classical (or sequential) caching framework

The classical caching framework consists of a processor, cache and a main memory. A number of deterministic and randomized cache replacement algorithms have been proposed in the classical caching framework. A few of these algorithms are used in practical systems. We present a few algorithm that have been well studied in the literature:

1. **Deterministic algorithms:**
   (a) LRU: Upon a cache miss, the least recently used block is evicted from the cache.
   (b) FIFO: Upon a cache miss, the block which was first brought into the cache is evicted from the cache.
2. **Randomized algorithms:**
   (a) RAND [21]: Upon a cache miss, a random block is evicted from the cache.
   (b) MARK-RAND [13]: Upon a cache miss, a random block that was not requested in the current phase is evicted from the cache. The algorithm keeps track of the blocks that were not requested during the current phase by using a *mark* bit.

It was shown in [22] that the competitive ratio of both LRU and FIFO is at most k — size of the cache and that both LRU and FIFO are optimal in the classical caching framework. The optimality was established by proving a lower bound of k on the competitive ratio of deterministic algorithms. The competitive ratio of RAND was shown to be at most k in an expected sense by Raghavan and Snir [21]. The attractive feature of RAND is its simplicity. It attains the same competitive ratio as an optimal deterministic cache replacement algorithm (ex. LRU) with minimal time and space complexity. Fiat et al. [13] proved that RAND was far from optimal in the classical caching framework by establishing a lower bound of H_k on the competitive ratio of randomized cache replacement algorithms. Fiat et al. also presented MARK-RAND and established an upper bound of $2H_k$ on its competitive ratio. Later, the H_k lower bound was matched by an algorithm called PARTITION proposed by McGeoch and Sleator [19].

Marking algorithms: Based on the definition of MARK-RAND, a family of algorithms called marking algorithms was introduced by Borodin et al. in [6]. A marking algorithm proceeds in marking phases. A phase starts with all the blocks in the cache unmarked. A block gets marked when a request to it is served. Upon a cache miss, a marking algorithm evicts an unmarked block from the cache. A marking algorithm could be either *explicit* or *implicit*.

Explicit and implicit marking algorithms: An explicit marking algorithm follows the above mentioned marking scheme explicitly. On the other hand, an implicit marking algorithm does not follow the marking scheme. But if we split the request sequence into multiple phases, it can be observed that an implicit marking algorithm never evicts a block that was requested during the current phase.

Observe that MARK-RAND is an explicit marking algorithm and LRU is an implicit marking algorithm. FIFO on the other hand is a non-marking algorithm. Since on a particular sequence, FIFO could evict a block that was requested during the current phase.

Trong [26] generalized the results from [22] to deterministic marking algorithms. It was shown in [26] that any deterministic marking algorithm is k-competitive in the classical caching framework.

(h, k)-paging framework: The (h, k)-paging framework was introduced in [22] as a variation on the classical caching framework. In this framework, the online algorithm is given more resources (i.e. a bigger cache) than the offline algorithm. The online algorithm is given a cache of size k and the offline algorithm is given a cache of size h ($k \geq h$). Sleator and Tarjan [22] proved that LRU and FIFO are optimal in this framework by established an upper bound of $\frac{k}{k-h+1}$ on their competitive ratio and a lower bound of $\frac{k}{k-h+1}$ on the competitive ratio of deterministic algorithms.

2.1.2 Access graph framework

It has been observed that in practice, LRU has a competitive ratio much less than k [29]. It is also seen that LRU performs much better than FIFO in practice even though both LRU and FIFO have the same competitive ratio. This inconsistency in the competitive ratio has been attributed to *locality of reference* by Borodin et al. [6]. Locality of reference refers to the notion that after a request to a block, say x, a block that is both spatially and temporally closer to x has a higher probability of getting requested in practical computations. This notion was not captured in the classical competitive analysis presented in [22].

Different models were proposed in the literature to capture the concept of locality of reference. In [6], locality of reference has been modeled in the form of an access graph. Koutsoupias and Papadimitriou [17] modeled locality of reference in form of a diffused adversary model. The access graph framework has been quite influential and our research derives motivation from this framework. We describe the access graph framework in detail in the following part of this subsection.

An access graph is a graph G that governs the pattern of block requests in the request sequence. The nodes in the access graph represent the memory blocks and the edges control the pattern of requests in the following fashion:

The request that follows a request to node x could either be another request to x or to some neighbor of x in G. The competitive ratio of an online algorithm ALG on an access graph G is defined as the ratio of number of cache misses incurred by ALG to the number of cache misses incurred by an optimal offline algorithm (OPT) on a worst case request sequence which is a valid walk on G. More formally, the competitive ratio of ALG is said to be $c(G)$ on a graph G if there exists a constant c' such that:

$$\forall \sigma \in walks(G) : \text{ALG}(\sigma) \leq c(G)\text{OPT}(\sigma) + c'$$

We use $walks(G)$ to represent the set of valid walks on the access graph G. It was shown in [6] that LRU is optimal when the access graph is an undirected tree and that the competitive ratio of LRU is at most a constant times the competitive ratio of FIFO on any access graph.

Borodin et al. [6] established a lower bound of $\lceil \log(k+1) \rceil$ on the competitive ratio of deterministic and randomized algorithms on a $k+1$-node cycle. In [6], both LRU and FIFO were shown to be far from optimal on a $k+1$-node cycle by establishing an upper bound of k on their competitive ratio. A deterministic marking algorithm called FAR was presented in [6] for the access graph framework. Later, it was proved that FAR is optimal up to a constant factor for any graph by Irani et al. [16]. Motivated by the analysis of FAR, a randomized algorithm was presented and analyzed by Fiat and Karlin in [12].

Multi-pointer access graph framework Fiat and Karlin [12] considered the multi-pointer access graph framework consisting of multiple pointers pointing at nodes in the access graph. The multi-pointer access graph framework was proposed to address caching in the parallel context. Multiple pointers model multiple processes sharing a cache and requesting memory blocks in parallel. An arbitrary interleaving of requests from these processes reaches the shared cache. An algorithm called MPALG was proposed along with the the multi-pointer access graph framework in [12]. It was also shown in [12] that MPALG was optimal up to a constant factor in this framework.

2.1.3 Application controlled caching framework

The notion of giving control to applications to make system level decisions was extremely influential in the operating systems community during the nineties. Applications (or processes) usually have much better knowledge about the resources (for instance: memory blocks) they might need in future than the operating system. The application controlled caching framework exploits this feature. In this framework, each process has varying degrees of knowledge about its future request sequence and the operating system relies on that knowledge to make better cache replacement decisions. The application controlled caching framework was proposed by Cao et al. in [8]. A deterministic algorithm was proposed along with the model which picks a process that owns the least recently used (LRU) page and asks it to make an eviction. The picked process makes an eviction based on its knowledge about its future request sequence.

Barve et al. [3] considered the application controlled caching framework where processes access disjoint sets of memory block and each process has varying degrees of knowledge about its future request sequence. In [3], lower bounds were established on the competitive ratio of deterministic and randomized algorithms and a randomized algorithm was proposed which was optimal (up to a constant factor) in this framework.

We model these two scenarios — where processes access disjoint sets of memory blocks and where processes share memory blocks as two different models: the *disjoint memory framework* and the *shared memory framework*. In both of these frameworks, we assume that each process has full knowledge about its future request sequence. Recall that the case when processes have no knowledge about their future request sequence reduces to the classical caching problem.

The disjoint memory framework When the application controlled caching framework was first introduced [8], it was assumed that processes access disjoint sets of memory blocks. We model this assumption as the disjoint memory framework. When processes access disjoint sets of memory blocks, a block evicted by a process cannot be requested by any other process. Algorithms in this framework exploit this feature to obtain better competitive ratio. We

present a natural deterministic algorithm which matches the lower bound established in [3]. We consider this framework in detail in Chapter 3.

The shared memory framework We model the scenario in which processes share memory blocks as the shared memory framework. Any memory block can be requested by any subset of processes. This assumption makes the adversary more powerful. We develop a natural deterministic marking cache replacement algorithm which is optimal up to a constant factor and establish lower bounds on the performance of deterministic and randomized algorithms. We consider this framework in detail in Chapter 4.

2.1.4 Hierarchical caching framework

We present the hierarchical caching framework to model multicore processor systems like Intel-Nehalem and AMD-Shanghai [1, 11]. A multicore system typically consists of multiple cores and a hierarchy of caches in order to obtain better caching performance. The cores in a multicore system are analogous to processes in the application controlled caching framework. On the other hand, only a single level shared cache was considered in the application controlled caching framework whereas multiple levels of caches are considered in the hierarchical caching framework. These caches could be either private to each core or shared among all the cores. For instance, the L_1 and L_2 caches are private in both Intel-Nehalem and AMD-Shanghai processors and the L_3 cache is shared. We consider all levels of caches to be shared among all cores in our thesis.

The cache hierarchy consists of at least two levels — L_1 and L_2 caches. With two levels of caches, the L_2 cache could be either inclusive, exclusive or non-inclusive of the L_1 cache. We are interested in the analysis of the performance of cache replacement algorithms at the shared L_2 cache in each of these cases. We consider this framework in detail in Chapter 5.

Chapter 3

Disjoint memory framework

In this chapter we present our results on cache replacement algorithms in the *disjoint memory framework*. Section 3.1 describes and motivates the disjoint memory framework. In Section 3.2, we present a survey on known algorithms and results in this model. In Section 3.3, we present a natural deterministic algorithm called RR-PROC-MARK which is optimal in the disjoint memory framework.

3.1 Disjoint memory framework description

The disjoint memory framework was proposed and formalized in [8] and [3] respectively. We described the model in Chapter 1. As a review, we describe the model below:

1. We let p processes share a cache of size k. We let P_i denote the ith process.

2. Each process has full knowledge about its request sequence. We use σ_i to represent the request sequence of the ith process. Recall that this request sequence is the exact order in which memory blocks are requested by the process P_i in the future.

3. An arbitrary interleaving of $\sigma_1, \sigma_2, \ldots, \sigma_p$ is seen at the shared cache. The analysis of algorithms at the shared cache is with respect to a worst case interleaving (represented by σ).

4. The online algorithm at the shared cache does not have any knowledge about the interleaving of request sequences from the p processes.

5. The optimal offline algorithm (OPT) on the other hand, has complete control of and knowledge about the interleaving.

6. The interleaving is assumed to be fixed, i.e., the same interleaving is assumed to reach the shared cache for all of the cache replacement algorithms, including the optimal offline algorithm — irrespective of the cache replacement decisions taken at the shared cache.

7. The processes request disjoint sets of memory blocks. Thus, every block in the memory can be accessed by at most one of the p processes. If block q can be accessed by process P_i, we say that q *belongs to* P_i and P_i *owns* q.

Research on cache replacement algorithms for the application controlled caching framework was initiated in [8] and [3]. In the application controlled caching framework, each process is assumed to have varying degrees of knowledge about its future request sequence — full knowledge or no knowledge. The cache replacement algorithms developed in the application controlled caching framework were analyzed in both of these cases. Note that the application controlled caching framework reduces to the sequential caching framework when processes have no knowledge about their individual request sequences. A simple algorithm like LRU at the shared cache is optimum in this framework.

This motivates us to consider the case when each process has full knowledge about its future request sequence and model it as the disjoint memory framework. We continue our search for more efficient cache replacement algorithms when each process has full knowledge about its future request sequence. Results from the application controlled caching framework for the case when each process has full knowledge about its future request sequence clearly hold for the disjoint memory framework. In the next section, we discuss all such results from the application controlled caching framework.

3.2 Previous results in this model

In this section, we present a survey on the cache replacement algorithms proposed in the application controlled caching framework. In Subsection 3.2.1,

we present a family of algorithms called *process marking algorithms* motivated by the process marking scheme used in the randomized algorithm proposed in [3]. We present a few lemmas for all of the process marking algorithms in Subsection 3.2.2.

A deterministic cache replacement algorithm which picks a process that owns the least recently used (LRU) block and asks it to make an eviction was proposed in [8]. We observe that the proposed algorithm is an implicit process marking algorithm and since the algorithm uses LRU to pick a process, we call it LRU-PROC-MARK from now on. This algorithm was not given any specific name in [8].

The process picked by LRU-PROC-MARK makes an eviction based on its knowledge about its future request sequence. If the process picked by the algorithm has full knowledge about its future request sequence, it makes a *good eviction*. We define the term good eviction later. If it does not have knowledge about its future request sequence, a global LRU block is evicted.

Following the algorithm in [8], Barve et al. [3] analyzed LRU-PROC-MARK for the case when each process has full knowledge about it future request sequence and established an upper bound of $2(p+1)$ on its competitive ratio. A few important bounds on the competitive ratio of deterministic and randomized cache replacement algorithms were also established in [3]. A lower bound of $p+1$ was established on the competitive ratio of deterministic cache replacement algorithms in the application controlled caching framework for the case when each process has full knowledge about its future request sequence.

A lower bound of H_{p-1} was established on the competitive ratio of randomized cache replacement algorithms in the application controlled caching framework for the case when each process has full knowledge about its future request sequence. A simple randomized algorithm was also proposed in [3], which uses a simple randomized algorithm to pick a process, which is then asked to make an eviction. We observe that the randomized algorithm proposed in [3] is a process marking algorithm similar to LRU-PROC-MARK. We call this algorithm RAND-PROC-MARK from now on.

The process picked by RAND-PROC-MARK makes an eviction based on its knowledge about its future request sequence. If the process picked by the

algorithm has full knowledge about its future request sequence, it makes a good eviction.

We close the gap between the upper and lower bounds on the competitive ratio of deterministic cache replacement algorithms in the disjoint memory framework by introducing a simple scheme for picking processes. We propose an algorithm called RR-PROC-MARK which, instead of picking a process that owns the least recently used (LRU) block, uses the round robin (RR) scheme to pick a process, which is then asked to make an eviction. The process picked by the algorithm uses MARK-FITF cache replacement algorithm to make an eviction. We define MARK-FITF later. We establish an upper bound of $p + 1$ on the competitive ratio of our algorithm in the disjoint memory framework.

Two level cache replacement strategy: In order to exploit the knowledge each process has about its future request sequence, algorithms proposed in the application controlled caching framework take decisions at two levels:

- **process-selection**: A process (also referred to as a *victim process* in [3]) is picked and asked to make an eviction.
- **block-eviction**: The process picked by the algorithm evicts one of the blocks that belong to it from the cache based on its knowledge about its future request sequence.

LRU-PROC-MARK uses LRU for process-selection and the process picked by the algorithm makes a good eviction decision when it has full knowledge about its future request sequence. RAND-PROC-MARK uses a simple randomized algorithm for process-selection and the process picked by the algorithm makes a good eviction decision when it has full knowledge about its future request sequence. RR-PROC-MARK uses a round robin strategy for process-selection and MARK-FITF for block-eviction.

Marking algorithms: Before describing good evictions, we present a quick review of the marking algorithms. Marking algorithms are a family of cache replacement algorithms which proceed in marking phases. A phase starts with

all of the blocks unmarked. A block gets marked when a request to it is served. Upon a cache miss, one of the unmarked blocks is evicted from the cache. The marking scheme employed in these algorithms ensures that a block requested during a particular phase is not evicted from the cache till the end of that phase. This property gives marking algorithms an optimum competitive ratio of k in the sequential caching framework. The definition of good eviction is implicitly based on this marking scheme.

Good set and good eviction: The term *good eviction* was introduced in [3], along with the term *good set*. A good set for process P_i is a set of blocks consisting of the unmarked block in the cache which is requested farthest in the future request sequence of P_i, say u_i, and all of the marked blocks in the cache that belong to P_i which are requested after u_i. A process is said to have evicted a good block, or made a good eviction if it evicts a block from its good set. Note that even if the online algorithm is not a marking algorithm, good set and good eviction terms can still be defined by considering an implicit marking scheme during analysis. Also note that the good set of a process could contain both marked and unmarked blocks.

3.2.1 Process marking algorithms

Similar to the marking algorithms in the sequential caching framework, we consider a family of cache replacement algorithms for the disjoint memory framework which we call the process marking algorithms. In process marking algorithms, processes are marked in addition to the memory blocks. The notion of marking processes was first introduced for RAND-PROC-MARK in [3]. Our contribution is to formalize this notion into a general framework of algorithms for the disjoint memory framework. We establish a few new lemmas for processes marking algorithm and generalize the lemmas that were established in [3] specifically for RAND-PROC-MARK to all of the process marking algorithms.

Definition: Process marking algorithms are a family of cache replacement algorithms which proceed in marking phases. A phase starts with all blocks

and processes unmarked. A block gets marked when a request to it is served. A process gets marked when all of the blocks that belong to it in the cache are marked. Upon a cache miss, one of the unmarked processes is picked and asked to make an eviction. The process picked by the algorithm evicts one of the unmarked blocks that belongs to it from the cache.

Explicit and implicit process marking algorithms: Similar to the marking algorithms, process marking algorithms are either explicit or implicit. An explicit process marking algorithm maintains an explicit *mark* bit for every process and block. It follows the marking scheme described in the definition of the process marking algorithms explicitly. On the other hand, an implicit process marking algorithm always pick an unmarked process and evict an unmarked block from the cache if the process marking scheme was followed. However, an implicit process marking algorithm chooses not to maintain the *mark* bit and follow the process marking scheme explicitly.

Process *only* marking algorithms: In contrast to the process marking algorithms, process *only* marking algorithms evict both marked and unmarked blocks from the cache. These algorithms still use the process marking scheme defined in the previous paragraph and pick unmarked processes which are asked to make evictions. The algorithms proposed in [8] and [3] for the application controlled caching framework were process only marking algorithms. Recall that processes in these algorithms make good evictions (which could include eviction of marked blocks).

In the sequential caching framework, RAND-MARK [13] and FAR [6] are examples of explicit marking algorithms and LRU is an implicit marking algorithm. In the disjoint memory framework, RAND-PROC-MARK [3] is an example of an explicit process only marking algorithm and LRU-PROC-MARK [8] is an implicit process only marking algorithm. RR-PROC-MARK (presented in the next section) is an example of an explicit process marking algorithm. We summarize below the three algorithms discussed in the disjoint memory framework till now.

1. RAND-PROC-MARK [3]: Upon a cache miss, a random *unmarked* process (chosen with uniform probability) is picked and asked to make an eviction. The process picked by the algorithm evicts a good block from the cache if it has full knowledge about its future request sequence. This is an explicit process only marking algorithm.

2. LRU-PROC-MARK [8]: Upon a cache miss, a process that owns the least recently used (LRU) block in the cache is picked and asked to make an eviction. The process picked by the algorithm evicts a good block from the cache if it has full knowledge about its future request sequence. This is an implicit process only marking algorithm.

3. RR-PROC-MARK [This thesis]: Upon a cache miss, an *unmarked* process is picked using the round robin scheme and asked to make an eviction. i.e., the first unmarked process which has process id greater than that of the most recently process picked by the algorithm is picked and asked to make an eviction. The process picked by the algorithm uses MARK-FITF to evict an *unmarked* block from the cache. This is an explicit process marking algorithm.

MARK-FITF **as a block-eviction policy:** A simple offline cache replacement algorithm called FITF was proposed for the sequential caching framework in [4]. Upon a cache miss, FITF evicts a block from the cache that is requested farthest in the future request sequence. It was also shown that FITF was optimal in [4]. A marking version of FITF is MARK-FITF. Instead of evicting a block which is requested farthest in the future request sequence, MARK-FITF evicts an unmarked block which is requested farthest in the future request sequence. Note that when a process P_i uses MARK-FITF to make an eviction, a good eviction is ensured. This is because, MARK-FITF always evicts u_i, the unmarked block in the cache which is requested farthest in the future request sequence of P_i. Recall that u_i always belongs to the good set of process P_i.

3.2.2 Lemmas for process marking algorithms

Before presenting lemmas for process marking algorithms, we review a few terms from [3].

1. **Clean block:** A block q is said to be a clean block with respect to a particular phase of a process marking algorithm if q does *not exist* in the cache at the start of this phase.

2. **Non-clean block:** A block q is said to be a non-clean block with respect to a particular phase of a process marking algorithm if q *exists* in the cache at the start of this phase.

3. **Hole:** When a non-clean block q is evicted from the cache in order to serve the request to a clean block, we say that a hole h is *created* at q. A hole basically suggests that the block q is missing from the cache.

4. **Hole association:** Since the hole h is created due to the eviction of q, we also say that h is *associated* with q. If q belongs to process P_i, we say that h is *associated* with P_i.

5. **Hole movement:** Let q be requested again by P_i at some point during the phase. Since q is not in the cache, another unmarked block q' is evicted in order to serve the request to q. At that point, we say that the hole h now *moves* from q to q'. It gets associated with q' from now on. If q' belongs to another process $P_{i'}$, we also say that h *moved* from P_i to $P_{i'}$.

6. **Relating cache misses and holes:** Every cache miss results either in creation or in movement of a hole. In case of a process marking algorithm, a hole is always associated with an unmarked block (recall that, always an unmarked block is evicted from the cache). Since all clean blocks are marked when they are brought in, holes are always associated with non-clean blocks. Further, holes are always created at unmarked processes and when a hole moves, it always moves into an unmarked process. This is because, always an unmarked process is picked and asked to make an eviction.

The first two lemmas in this section generalize lemmas presented in [3] for RAND-PROC-MARK to all of the process marking algorithm.

Lemma 3.2.1. Let u_i be an unmarked block in the cache belonging to process P_i that is requested farthest in P_i's future request sequence at some point during a phase. If u_i is requested during the same phase, process P_i is marked by the time the request for block u_i is served.

Proof. The proof of this lemma is directly based on the proof of Lemma 6.2 in [3]. Before the request to u_i is served, requests to all other unmarked blocks in cache belonging to P_i are served and hence they are all marked. Even u_i is marked when a request to it is served. All of the blocks in cache that belong to P_i are marked and hence by the definition of a process marking algorithm, P_i is marked by the time the request to u_i is served.

□

Lemma 3.2.2. If the processes picked by a process marking algorithm always make good evictions, the following statement holds: by the time the request to a block q associated with a hole is served during the phase, the process P_i that owns q is marked.

Proof. The proof of this lemma is again directly based on the proof of Lemma 6.4 in [3]. Since q is associated with a hole, q was evicted by P_i at some point during the phase. The process P_i would have chosen either u_i, the unmarked block that was requested farthest in P_i's future request sequence at that point or a block that is requested after u_i in its request sequence for eviction.

Hence, either $q = u_i$ or the request to q is after the request to u_i in P_i's request sequence. In both of these cases, the request to u_i is served by the time the request to q is served. Lemma 3.2.1 proves that P_i is marked by the time the request to u_i is served. Hence, P_i is marked by the time the request to its block associated with a hole is served.

□

We prove 3 additional results for process marking algorithms in the remaining part of this section.

Lemma 3.2.3. Consider a marking phase of a process marking algorithm with l clean block requests. The cost of an optimal offline algorithm (OPT) for this phase is at least $l/2$ in an amortized sense.

Proof. The key observation is that every process marking algorithm is also a marking algorithm. This is because the block marking scheme used in the process marking algorithms is exactly similar to the scheme used in marking algorithms for the sequential caching framework. An amortized lower bound of $l/2$ was established on the cost of OPT for a marking phase with l clean block requests in [13]. The same lower bound holds in the case of process marking algorithms.

□

Lemma 3.2.4. Once a process P_i gets marked during the phase of a process marking algorithm, it remains marked till the end of the phase.

Proof. The marking scheme used by a process marking algorithm ensures that once a block gets marked, it remains marked till the end of the phase. A process P_i gets marked when all of the blocks belonging to P_i in the cache are marked.

If P_i does not request any new block after getting marked, it remains marked till the end of the phase since blocks belonging to P_i in the cache remain marked.

If P_i requests a new block, say r, after getting marked, r gets marked when it is served. The process P_i remains marked in this case as well since blocks belonging to P_i in the cache are marked.

□

Lemma 3.2.5. The number of holes associated with every unmarked process is non-decreasing and the number of holes associated with every marked process is non-increasing in case of a process marking algorithm.

Proof. First, we shall prove that the number of holes associated with an unmarked process is non-decreasing. Consider an unmarked process P_i. The number of holes associated with P_i can decrease only when one or more of these holes get filled. By filled, we mean, P_i requests the blocks associated with one or more of these holes. Lemma 3.2.2 proves that P_i is marked by the time its request to a block associated with a hole is served. This implies that the process P_i gets marked when the number of associated holes decreases and once marked, P_i remains marked till the end of the phase (Lemma 3.2.4). Hence, the number of holes associated with P_i is non-decreasing till it remains unmarked.

Next, we shall prove that the number of holes associated with a marked process is non-increasing. Consider a marked process P_i. The number of holes associated with P_i can increase only when P_i evicts one of its blocks from the cache. The process P_i can make an eviction only when it is picked by the process marking algorithm to make an eviction. Recall that a process marking algorithm always picks an unmarked process and asks it to make an eviction. This implies that the process P_i is never picked and asked to make an eviction. Hence, the number of holes associated with P_i is non-increasing after it gets marked.

□

3.3 Deterministic process marking algorithm : RR-PROC-MARK

In this section, we present a natural deterministic algorithm for the disjoint memory framework called RR-PROC-MARK. It improves the competitive ratio from $2(p+1)$ to $p+1$. This algorithm is motivated by the following observation about LRU-PROC-MARK: upon a request to a clean block, LRU-PROC-MARK evicts a non-clean block from the cache resulting in the creation of a hole. Let l be the number of such holes created during a particular phase of LRU-PROC-MARK. In the worst case, all of these l holes are created in one of the p processes. They all move from one process to another in a group, leading to l cache misses on every such movement. This leads to a total of $lp + l$ cache misses per phase (the extra $+l$ term is to account for the number

of cache misses at the time of creation of these holes). Lemma 3.2.3 proves that the cost of OPT is at least $l/2$ per phase in an amortized sense. Hence, the competitive ratio of LRU-PROC-MARK is at most $2(p+1)$.

In contrast, RR-PROC-MARK carefully distributes these holes among all of the processes by using a simple scheme (round robin) for selecting processes. We show that this scheme is effective in reducing the constant factor in the upper bound. Two additional useful features of RR-PROC-MARK are that it is a fair algorithm (every process is asked to make almost equal number of evictions) and the algorithm is computationally very efficient.

3.3.1 Description of RR-PROC-MARK

RR-PROC-MARK is an explicit process marking algorithm. It proceeds in marking phases. A phase starts with all of the blocks and processes unmarked. A block gets marked when a request to it is served. A process gets marked when all its blocks get marked. Upon a cache miss, an unmarked process is picked using the round robin scheme and asked to make an eviction. The process picked by the algorithm evicts an unmarked block from the cache that belongs to it and is requested farthest in its future request sequence (MARK-FITF).

Participating process: A process P_i is said to be a participating process during a particular phase of RR-PROC-MARK if it owns one or more blocks in the cache at the start of the phase. RR-PROC-MARK picks only the participating processes and asks them make to evictions during any given phase. This is ensured by marking all of the non-participating processes at the start of the phase (note that a marked process is never picked by our algorithm). Also, note that, once a process gets marked, it remains marked till the end of the phase (Lemma 3.2.4).

RR-PROC-MARK(P_i, r):

Input: Requested cache block r and the process that owns r, P_i.

Output: Eviction decision.

- **if** r is in the cache: Mark r if it is not marked; mark process P_i if all its blocks are marked. No eviction is needed in this case.
- **if** r is *not* in the cache:
  1. If blocks in the cache are marked (end of phase):
     (a) Start a new phase by unmarking all of the blocks and processes.
     (b) Mark all non-participating processes at this point.
     (c) Reset *victim_process* to 0.
  2. *process-selection:* use ROUND-ROBIN to pick the next unmarked process.
     (a) $victim\_process = victim\_process + 1$
     (b) **while** $P_{victim\_process}$ is marked:
     $$victim\_process = (victim\_process \mod p) + 1$$
     (c) $P_{victim\_process}$ is picked and is asked it to make an eviction.
  3. *block-eviction:* use MARK-FITF to evict a block q from the cache such that:
     (a) q is unmarked
     (b) q belongs to $P_{victim\_process}$
     (c) q appears farthest in $P_{victim\_process}$'s future request sequence.
  4. Bring r in place of q and mark r; mark process P_i if all its blocks are marked.
  5. Mark $P_{victim\_process}$ if it does not own any unmarked block in cache at this point.

3.3.2 Competitive ratio of RR-PROC-MARK

In this section we establish an upper bound of $\max(10, p + 1)$ on the competitive ratio of RR-PROC-MARK. Our approach involves establishing a bound on the cost of RR-PROC-MARK and OPT on every process marking phase. Using these bounds, we prove an upper bound on the competitive ratio of RR-PROC-MARK.

Consider an arbitrary phase of RR-PROC-MARK with l clean block requests and p' participating processes. We prove the following key results:

- Lemma 3.3.5 of Section 3.3.2.1 proves that the cost of RR-PROC-MARK is at most $\frac{l}{2}(p'+1)$ for this phase, when $l < p'$.
- Lemma 3.3.8 of Section 3.3.2.2 proves that the cost of RR-PROC-MARK is at most $l \cdot (H_{p'} + 2)$ for this phase, when $l \geq p'$.

Using the above two results and the bound on $cost(\text{OPT})$, we prove that the competitive ratio of RR-PROC-MARK is at most $\max(10, p+1)$ in Lemma 3.3.9 of Section 3.3.2.3.

Road map: The basic idea behind our approach to bound the cost of RR-PROC-MARK is to split the interleaved request sequence in the current phase into a number of stages. We establish bounds on the cost of RR-PROC-MARK for each of these stages. We define stages differently for the case when $l < p'$ and $l \geq p'$. Recall that a cache miss is either due to a request to a clean block or due to a request to a non-clean block which is associated with a hole. We bound the number of cache misses due to requests to clean and non-clean blocks separately. The number of cache misses due to requests to clean blocks is bounded by the number of clean block requests (l for the current phase). On the other hand, the number of cache misses due to requests to non-clean blocks which are associated with holes depends on how our algorithm distributes these holes.

3.3.2.1 Upper bound on the cost of RR-PROC-MARK when $l < p'$

Recall that l is the number of clean block requests and p' is the number of participating processes during the current phase. We shall prove that the cost of RR-PROC-MARK is at most $\frac{l}{2}(p'+1)$ for this phase, when $l < p'$.

Stage: We split the request sequence in the current phase into a number of stages as follows:

Stage 1 starts with the first request during the current phase and ends just before the request that results in an unmarked process getting associated with 2 holes for the first time. Stage 2 starts with the request that results in an unmarked process getting associated with 2 holes for the first time and ends just before the request that results in an unmarked process getting associated with 3 holes for the first time and so on.

More generally, stage j starts with the request that results in an unmarked process getting associated with j holes for the first time and ends just before the request that results in an unmarked process getting associated with $j+1$ holes for the first time.

By Lemma 3.2.5 we know that the number of holes associated with an unmarked process is non-decreasing. This implies that after the first request in stage j is served, exactly one unmarked process has j associated holes and all of the other unmarked processes have less than j associated holes.

Since there are at most l holes present at any point, we have *at most l stages* during the current phase.

Notations used for the case when $l < p'$

1. Let u_j be the number of unmarked processes just before the start of stage j.
2. Let l_j be the number of clean block requests during stage j.
3. Let λ_j be the total number of clean block requests from the start of stage 1 through the end of stage j.
4. Let m_j be the number of cache misses during stage j.
5. Let *cost*(RR-PROC-MARK) represent the total number of cache misses incurred by RR-PROC-MARK during the current phase.

Observations:

1. For $1 \leq j \leq l : \lambda_j = \sum_{i=1}^{j} l_i$ and $\lambda_j \leq l$: follows directly from the definitions of λ_j, l_i and l.

2. $u_1 = p'$: the number of unmarked processes just before the start of stage 1 is exactly equal to the number of participating processes (note that all non-participating processes are marked at the start of the phase).

3. $cost(\text{RR-PROC-MARK}) = \sum_{j=1}^{l} m_j$: follows directly from the definitions of m_j and $cost(\text{RR-PROC-MARK})$.

4. Lemma 3.3.1 proves that the number of holes associated with every unmarked process is exactly $j - 1$ just before the start of stage j.

5. Lemma 3.3.3 proves that the number of unmarked processes is at most $\frac{\lambda_{j-1}}{j-1}$ just before the start of stage j. For $1 \leq j \leq l : u_j \leq \frac{\lambda_{j-1}}{j-1}$.

6. Lemma 3.3.4 proves that the number of cache misses during stage j is bounded by the number of unmarked processes at the start of the stage. For $1 \leq j \leq l : m_j \leq u_j$.

Lemma 3.3.5 proves the key result for this section. It proves that the cost of RR-PROC-MARK is at most $\frac{l}{2}(p' + 1)$ for the current phase when $l < p'$.

Lemma 3.3.1. The number of holes associated with every unmarked process is exactly $j - 1$ just before the start of stage j.

Proof. From the definition of stage j, note that the number of holes associated with every unmarked process just before the start of stage j is strictly less than j. We shall prove that the number of holes associated with every unmarked process just before the start of stage j is exactly $j - 1$. We prove our claim by induction.

Base case: $j = 1$. Just before the start of stage 1, there are no holes in the system. Hence, every process hence has exactly 0 associated holes.

Inductive assumption: Assume that the statement holds just before the start of stage $j-1$. Restating the statement for stage $j-1$:

Just before the start of stage $j-1$, every unmarked process has exactly $j-2$ associated holes.

Inductive step: We shall prove that the statement holds just before the start of stage j.

From the inductive assumption, every unmarked process has exactly $j-2$ associated holes just before the start of stage $j-1$. We shall now prove that every unmarked process is picked exactly once and asked to make an eviction during stage $j-1$. This will increase the number of holes associated with every unmarked process by exactly one and hence by the end of stage $j-1$ (or equivalently, just before the start of stage j) every unmarked process will have exactly $j-1$ associated holes.

We can trivially prove that none of the unmarked processes are picked more than once and asked to make evictions during stage $j-1$. If some unmarked process, say P_i was picked more than once and asked to make evictions during stage $j-1$, the number of holes associated with P_i will exceed $j-1$ and hence stage j has already begun.

We now prove that every unmarked process is picked at least once and asked to make an eviction during stage $j-1$. A process P_i is picked and asked to make an eviction in Line 2c, when the following condition holds:

$$victim\_process = i \wedge P_i = unmarked$$

In order to prove that every unmarked process is picked and asked to make an eviction at least once, we have to prove the following:

$\forall 1 \leq i \leq p' : victim\_process = i$ at some point during the $j-1^{st}$ stage.

Let stage j start with the process P_x. i.e., let P_x be the unmarked process which was picked and asked to make an eviction to serve the the first request in stage j.

The process P_x has exactly $j-1$ associated holes just before the start of stage j and exactly $j-2$ associated holes just before the start of stage $j-1$. Hence, P_x was picked and asked to make an eviction at some point during stage $j-1$. At this point, $victim\_process = x$. Notice that $victim\_process$ is only changed in Line 2a of our algorithm during a given phase. The only way it changes is by the following operation:

$$victim\_process = victim\_process \mod p+1$$

The first cache miss after the point when P_x was picked and asked to make an eviction during stage $j-1$ changes $victim\_process$ to $x \mod p+1 \neq x$. From that point onwards, $victim\_process$ takes on all values in the set $\{1, 2, \ldots, p\} - \{x\}$ before it is equal to x (at the start of stage j). $\forall 1 \leq i \leq p' : victim\_process = i$ at some point during stage $j-1$. Hence proved.

□

The following is a corollary for Lemma 3.3.1. This corollary continues to hold for the case when $l \geq p'$ because of the following reasons:

1. The definition of the stage for the case when $l < p'$ does not depend on the $l < p'$ inequality.

2. The proof of Lemma 3.3.1 is just based on the definition of a stage and does not use $l < p'$ inequality in its proof.

Corollary 3.3.2. The difference between the maximum number of holes associated with an unmarked process and the minimum number of holes associated with an unmarked process is at most 1 at any point during the phase.

Proof. This corollary follows from Lemma 3.3.1. Consider some point in time during the current phase of our algorithm. Let this point be in stage j. Lemma 3.3.1 proves that the number of holes associated with every unmarked process just before the start of stage j is exactly $j-1$ and the number of holes associated with every unmarked process just before the start of the $j+1$st stage is exactly j. Further, the number of holes associated with an unmarked process is non-decreasing.

Hence, at any point during stage j, the maximum number of holes associated with an unmarked process is at most j and the minimum number of holes associated with an unmarked process is at least $j-1$. Hence the difference is at most 1.

□

Lemma 3.3.3. Recall that λ_{j-1} is the total number of clean block requests from the start of stage 1 through the end of stage $j-1$. The total number of unmarked processes just before the start of stage j (u_j) is at most $\frac{\lambda_{j-1}}{j-1}$.

Proof. Recall that the number of holes in the system at some point during the phase is exactly equal to the number of clean block requests till that point. Hence, the total number of holes in the system just before the start of stage j is at most λ_{j-1}. Further, Lemma 3.3.1 proves that the number of holes associated with every unmarked process just before the start of stage j is exactly $j-1$.

Every unmarked process has exactly $j-1$ associated holes and the remaining holes are associated with marked processes. Let x be the number of holes associated with marked processes. The number of holes associated with any process is non-negative. Hence, $x \geq 0$.

$$\begin{aligned} \lambda_{j-1} &= u_j \cdot (j-1) + x \\ &\geq u_j \cdot (j-1) \\ u_j &\leq \frac{\lambda_{j-1}}{j-1} \end{aligned}$$

Hence, the total number of unmarked processes just before the start of stage j (u_j) is at most $\frac{\lambda_{j-1}}{j-1}$.

□

Lemma 3.3.4. The number of cache misses during stage j (m_j) is no more than the number of unmarked processes just before the start of stage j (u_j).

Proof. We shall prove this lemma for an arbitrary stage (stage j).

The main observation for this lemma is as follows: every cache miss is served by exactly one unmarked process (which is picked by our algorithm). The process picked by the algorithm evicts one of its unmarked blocks leading to an increase in the number of associated holes by exactly 1. Hence, every cache miss increases the number of holes associated with exactly one unmarked process by exactly one. In order to bound the number of cache misses during stage j, we bound the increase in number of holes associated with all of the unmarked processes during this stage.

From Lemma 3.3.1 we have that every unmarked process has exactly $j-1$ associated holes just before the start of stage j and every unmarked process has exactly j associated holes just before the start of stage $j+1$. Hence,

1. The increase in number of holes associated with an unmarked process is at most 1 during stage j.
2. Also, we have at most u_j unmarked processes during stage j. Because once a process gets marked it remains marked till the end of the phase.

The number of cache misses during stage j is at most $\sum_{i=1}^{u_j} 1 = u_j$. Hence proved.

□

Lemma 3.3.5. Consider a phase of RR-PROC-MARK with l clean block requests and p' participating processes. The cost of RR-PROC-MARK is at most $\frac{l}{2}(p'+1)$ for this phase when $l < p'$.

Proof. From Observation 3, we have,

$$cost(\text{RR-PROC-MARK}) = \sum_{j=1}^{l} m_j \tag{3.1}$$

and from Lemma 3.3.4, for all j such that $1 \leq j \leq l$, we have,

$$m_j \leq u_j \tag{3.2}$$

From the Equations 3.1 and 3.2, we have,

$$cost(\text{RR-PROC-MARK}) \leq \sum_{j=1}^{l} u_j \tag{3.3}$$

From Observation 2 and Lemma 3.3.3, we have,

$$u_1 \leq p' \tag{3.4}$$

$$\text{for all } j \text{ such that } 2 \leq j \leq l : u_j \leq \frac{\lambda_{j-1}}{j-1} \tag{3.5}$$

From Equations 3.3, 3.4 and 3.5, we have,

$$\begin{aligned} cost(\text{RR-PROC-MARK}) &\leq p' + \sum_{j=2}^{l} \frac{\lambda_{j-1}}{j-1} \\ &\leq p' + \sum_{j=1}^{l-1} \frac{\lambda_j}{j} \end{aligned}$$

Note that for all i such that $1 \leq i \leq p' : \lambda_i \leq l$. Hence,

$$\begin{aligned} cost(\text{RR-PROC-MARK}) &\leq p' + l \cdot \sum_{j=1}^{l-1} \frac{1}{j} \\ &\leq p' + l \cdot H_{l-1} \end{aligned}$$

Given $l < p'$, we shall prove that $p' + l \cdot H_{l-1}$ is at most $\frac{l}{2}(p'+1)$. Rearranging $p' + l \cdot H_{l-1} \leq \frac{l}{2}(p'+1)$, we need to prove:

$$\frac{l \cdot (2H_{l-1} - 1)}{l-2} \leq p'$$

Since $l < p'$, it suffices to prove:

$$\begin{aligned} \frac{l \cdot (2H_{l-1} - 1)}{l-2} &\leq l \\ 2H_{l-1} - 1 &\leq l - 2 \end{aligned}$$

This holds for all l such that $l \geq 6$. An upper bound of $\frac{l}{2}(p'+1)$ can be established on *cost*(RR-PROC-MARK) for $l < 6$ by considering case-by-case analysis.

□

We aim to establish a tighter bound on the competitive ratio of RR-PROC-MARK when $l \geq p'$. We hence proceed to the next subsection in which we establish an upper bound of $2(H_{p'}+2)$ on the competitive ratio of RR-PROC-MARK.

3.3.2.2 Upper bound on the cost of RR-PROC-MARK when $l \geq p'$

Recall that l is the number of clean block requests and p' is the number of participating processes during the current phase. We shall prove that the cost of RR-PROC-MARK is at most $l \cdot (H_{p'}+2)$ for this phase, when $l \geq p'$.

Stage: We split the request sequence in the current phase into a number of stages as follows:

The stage 0 starts with the first request during the current phase and ends just before the request that results in the first participating process getting marked. The stage 1 starts with the request that results in the first participating process getting marked and ends just before the request that results in the second participating process getting marked and so on.

More generally, stage j starts with the request that results in the jth participating process getting marked and ends just before the request that results in the $j+1$st participating process getting marked.

Since there are exactly p' participating processes, we have *exactly* $p'+1$ *stages* (including stage 0) during the current phase.

Without loss of generality, we assume that $\{P_1, P_2, \ldots, P_{p'}\}$ is the set of participating processes, and that P_j gets marked due to the first request in stage j for $1 \leq j \leq p'$.

Notations used for the case when $l \geq p'$

1. Let u_j be the number of unmarked processes just before the start of stage j.
2. Let l_j be the number of clean block requests during stage j.
3. Let λ_j be the total number of clean block requests from the start of stage 0 through the end of stage j.
4. Let m_j be the number of cache misses during stage j.
5. Let *cost*(RR-PROC-MARK) represent the total number of cache misses incurred by RR-PROC-MARK during the current phase.
6. (new for this case) Let n_i be the total number of requests to non-clean blocks which are not in the cache by the participating process P_i during the current phase.
7. (new for this case) Let N be the total number of requests to non-clean blocks which are not in the cache during the *entire phase*. Note that $N \geq \sum_{i=1}^{p'} n_i$. We shall soon prove that N is in fact equal to $\sum_{i=1}^{p'} n_i$.

Observations:

1. For $0 \leq j \leq p' : \lambda_j = \sum_{i=0}^{j} l_i$ and $\lambda_j \leq l$: follows directly from the definitions of λ_j, l_i and l.
2. $u_0 = p'$: the number of unmarked processes just before the start of stage 0 is exactly equal to the number of participating processes (note that all of the non-participating processes are marked at the start of the phase).
3. For $1 \leq j \leq p' : u_j = p' - j + 1$: exactly one unmarked process gets marked in every stage (except stage 0). Hence, the number of unmarked processes decreases by 1 in each stage (except stage 0).
4. *cost*(RR-PROC-MARK) $= \sum_{j=0}^{p'} m_j$: follows directly from the definitions of m_j and *cost*(RR-PROC-MARK).

5. Recall that non-clean blocks are those that exist in the cache at the start of the phase and non-participating processes do not own any blocks in cache at the start of the phase. Hence, only the participating processes can request non-clean blocks.

6. Lemma 3.3.6 proves that the total number of requests to non-clean blocks which are not in the cache during the entire phase is exactly equal to the total number of requests to non-clean blocks which are not in the cache by all of the p' participating process. $N = \sum_{i=1}^{p'} n_i$.

7. Recall that requesting a hole means requesting the block associated with the hole. Also, recall that holes are associated only with non-clean blocks in our algorithms.

8. Every hole h can only be requested by the participating process that owns the non-clean block associated h: Let q be the non-clean block associated with h. In the disjoint memory framework, q belongs to at most one process. Since q is a non-clean block, it belongs to exactly one participating process, say P_i. Hence, only P_i can request h which is associated with q.

9. Every participating process P_i can only request associated holes: From the previous observation, we know that P_i can request a hole associated with it. We need to show that P_i cannot request the hole h which is not associated with P_i. Since h is not associated with P_i, it should be associated with a non-clean block, say q belonging to some other participating process. Since q does not belong to P_i, it cannot request h which is associated with q.

10. Lemma 3.3.7 proves that the number of holes associated with P_i just before it gets marked is at most $\frac{\lambda_{i-1}}{u_i} + 1$.

Lemma 3.3.8 proves the key result for this section. It proves that the cost of RR-PROC-MARK is at most $l \cdot (H_{p'} + 2)$ for the current phase when $l \geq p'$.

Lemma 3.3.6. The total number of requests to non-clean blocks which are not in the cache during the entire phase is exactly equal to the total number of requests to non-clean blocks which are not in the cache by all of the p' participating process.

Proof. Since only the participating processes can request non-clean blocks (by Observation 5), the number of requests to non-clean blocks which are not in the cache during the entire phase is exactly equal to the total number of requests to non-clean blocks which are are not the cache by all of the p' participating processes.

□

Lemma 3.3.7. The number of holes associated with P_i just before it gets marked is at most $\frac{\lambda_{i-1}}{u_i} + 1$.

Proof. Recall that P_i gets marked due to the first request in stage i. Hence, it is sufficient to bound the maximum number of holes associated with an unmarked process just before the start of stage i to bound the maximum number of holes associated with P_i just before it gets marked.

Recall that the number of holes in the system at some point during the phase is exactly equal to the number of clean block requests till that point. Hence, the total number of holes in the system just before the start of stage i is at most λ_{i-1}. We have a total of u_i unmarked processes just before the start of stage i. Out of u_i unmarked processes, at least one has less than or equal to $\frac{\lambda_{i-1}}{u_i}$ associated holes. Hence the minimum number of holes associated with an unmarked process is at most $\frac{\lambda_{i-1}}{u_i}$.

Corollary 3.3.2 proves that the difference between the maximum number of holes associated with an unmarked process and the minimum number of holes associated with an unmarked process is at most 1 at any point during the current phase. Recall that Corollary 3.3.2 can be used for the case when $l \geq p'$ because it is based only on the definition of the stage in the previous case and not on the $l < p'$ inequality as such.

Hence the maximum number of holes associated with an unmarked process just before the start of stage i is at most $\frac{\lambda_{i-1}}{u_i} + 1$. In other words, the maximum number of holes associated with P_i just before it gets marked at most $\frac{\lambda_{i-1}}{u_i} + 1$.

□

Lemma 3.3.8. Consider a phase of RR-PROC-MARK with l clean block requests and p' participating processes. The cost of RR-PROC-MARK is at most $l \cdot (H_{p'} + 2)$ for this phase when $l \geq p'$.

Proof. The main observation for this lemma is the following: every cache miss is either due to a request to a clean block or due to a request to a non-clean block which is not in the cache. The number clean block requests during stage j is exactly l_j. The number of requests to non-clean blocks which are not in the cache during the *entire phase* is exactly N. Further, from Lemma 3.3.6, we have,

$$N = \sum_{i=1}^{p'} n_i$$

Hence,

$$\begin{aligned} \textit{cost}(\text{RR-PROC-MARK}) &= \sum_{j=0}^{p'} l_j + N \\ &= \sum_{j=0}^{p'} l_j + \sum_{i=1}^{p'} n_i \end{aligned}$$

A request to a non-clean block which is not in the cache is the same as a request to a hole associated with that non-clean block. In order to bound n_i, we will need to bound the number of requests to holes by each of the participating processes.

From Observations 8 and 9, P_i can only request holes that are associated with it. The process P_i gets marked as soon the first hole associated with it is requested. From that point onwards, the number of holes associated with P_i

is non-increasing and non-negative. Hence, the number of requests to holes by P_i is bounded by the number of associated holes just before it gets marked.

From Lemma 3.3.7, the number of holes associated with P_i just before it gets marked is at most $\frac{\lambda_{i-1}}{u_i} + 1$. Hence,

$$cost(\text{RR-PROC-MARK}) \leq \sum_{j=0}^{p'} l_j + \sum_{i=1}^{p'} (\frac{\lambda_{i-1}}{u_i} + 1)$$

Note that, $\sum_{j=0}^{p'} l_j = l$ and for all i such that $1 \leq i \leq p' : \lambda_i \leq l$. Further, by Observation 3, for all i such that $1 \leq i \leq p'$, we have,

$$u_i = p' - i + 1$$

Hence,

$$\begin{aligned} cost(\text{RR-PROC-MARK}) &\leq l + \sum_{i=1}^{p'} (\frac{l}{p' - i + 1} + 1) \\ &\leq l + p' + l \cdot \sum_{i=1}^{p'} \frac{1}{p' - i + 1} \\ &\leq l + p' + l \cdot H_{p'} \\ &\leq l \cdot (1 + p'/l + H_{p'}) \end{aligned}$$

Given $l \geq p'$, the cost of RR-PROC-MARK is at most $l \cdot (H_{p'} + 2)$.

□

3.3.2.3 Upper bound on the competitive ratio

Theorem 3.3.9. The competitive ratio of RR-PROC-MARK in the disjoint memory framework is at most $\max(10, p + 1)$.

Proof. Consider an arbitrary phase of RR-PROC-MARK with l clean block requests and p' participating processes. We have the following two key results:

- Lemma 3.3.5 in Section 3.3.2.1 proves that the cost of RR-PROC-MARK is at most $\frac{l}{2}(p'+1)$ for this phase, when $l < p'$.
- Lemma 3.3.8 in Section 3.3.2.2 proves that the cost of RR-PROC-MARK is at most $l \cdot (H_{p'} + 2)$ for this phase, when $l \geq p'$.

Since the number of participating processes during any given phase is at most p. The cost of RR-PROC-MARK is at most $\max(l \cdot (H_p + 2), \frac{l}{2} \cdot (p+1))$ for this phase.

On the other hand, OPT incurs at least $l/2$ cache misses per phase in an amortized sense. We can conclude that the competitive ratio of RR-PROC-MARK is at most $\max(2(H_p + 2), p + 1)$.

Note: $2(H_p + 2)$ is less than $p + 1$ when $p \geq 9$. Hence, the competitive ratio is at most $\max(10, p + 1)$.

□

3.3.3 RR-PROC-MARK in full access cost model

The *full access cost model* is proposed in [5] as a more realistic cost model for the sequential caching framework. In contrast to the classical competitive analysis in which we measure the ratio of the total number of cache misses incurred by the online algorithm to the total number of cache misses incurred by the offline algorithm, in the full access cost model, we measure the ratio of the total *access cost* incurred by the online algorithm to the total access cost incurred by the offline algorithm to serve the entire request sequence.

Access cost: Access cost of a block request is the time taken to serve the request. We let t_{hit} and t_{miss} represent the time taken to serve a request that results in a cache hit and a cache miss, respectively. Let n be the total number of requests in the given request sequence σ and m be the total number of cache misses incurred by ALG on this sequence. The access cost of ALG on

σ represented by $accesscost(\text{ALG})$ is given by:

$$\begin{aligned} accesscost(\text{ALG}) &= (n-m) \cdot t_{hit} + m \cdot t_{miss} \\ &= n \cdot t_{hit} + m \cdot (t_{miss} - t_{hit}) \end{aligned}$$

In order to keep the equations simple, we let $t_{miss} = b$ and $t_{hit} = 1$. The access cost of ALG is reduced to:

$$accesscost(\text{ALG}) = n + m \cdot (b-1)$$

It was shown in [5] that the competitive ratio of any marking algorithm in the full access cost model is at most $1 + \frac{(k-1)b}{k+b}$ in sequential caching framework. We consider the competitive ratio of RR-PROC-MARK in the full access cost model in the disjoint memory framework. Recall that each process has full knowledge about its individual request sequence in the disjoint memory framework.

We show that the competitive ratio of RR-PROC-MARK in the full access cost model is at most $1 + \frac{2(H_p+O(1))(b-1)}{b+1}$. Further, under the very reasonable assumption that $k \geq pH_p$ (note that k is usually orders of magnitude larger than p), we prove that the competitive ratio is at most $1 + \frac{2(H_p+O(1))(b-1)}{2(H_p+O(1))+b-1}$.

The main intuition behind the better performance of RR-PROC-MARK is as follows. We observed in Subsection 3.3.2.2 that RR-PROC-MARK performs extremely well (with an upper bound of $2(H_p + 2)$ on the competitive ratio) when the number of clean block requests in every phase is more than the number of participating processes. On the other hand, when the number of clean block requests is less, the total number of cache misses incurred by RR-PROC-MARK is naturally less. Hence, RR-PROC-MARK incurs relatively lesser number of cache misses for the entire range of l. This motivates us to consider its competitive ratio in the full access cost model which instead of counting just the number of cache misses, computes the fraction of cache misses in the entire request sequence.

Theorem 3.3.10. The competitive ratio of RR-PROC-MARK in the full access cost model is at most $1 + \frac{2(H_p+O(1))(b-1)}{b+1}$. Further, when $k \geq pH_p$, the competitive ratio is at most $1 + \frac{2(H_p+O(1))(b-1)}{2(H_p+O(1))+b-1}$.

Proof. Consider an arbitrary phase of RR-PROC-MARK with l clean block requests and p' participating processes. From Lemmas 3.3.5 and 3.3.8, we have the following for the number of cache misses incurred by RR-PROC-MARK:

$$cost(\text{RR-PROC-MARK}) \leq \begin{cases} p' + lH_{l-1} & \text{if } l < p' \\ l + p' + lH_{p'} & \text{if } l \geq p' \end{cases}$$

Let n be the total number of requests during the current phase. Recall that, $n \geq k$. We have the following for the full access cost of RR-PROC-MARK:

$$accesscost(\text{RR-PROC-MARK}) \leq \begin{cases} n + (p' + lH_{l-1})(b-1) & \text{if } l < p' \\ n + (l + p' + lH_{p'})(b-1) & \text{if } l \geq p' \end{cases}$$

On the other hand, OPT incurs at least $l/2$ cache misses during this phase in an amortized sense. The cost of OPT in the full access cost model is at least $n + (l/2)(b-1)$ per phase in an amortized sense. In order to keep equations compact, we let $s = b - 1$. The competitive ratio of RR-PROC-MARK in the full access cost model is:

$$\frac{accesscost(\text{RR-PROC-MARK})}{accesscost(\text{OPT})} \leq \begin{cases} \frac{n+(p'+lH_{l-1})s}{n+(l/2)s} & \text{if } l < p' \\ \frac{n+(l+p'+lH_{p'})s}{n+(l/2)s} & \text{if } l \geq p' \end{cases}$$

Rewriting the above equations:

$$\frac{accesscost(\text{RR-PROC-MARK})}{accesscost(\text{OPT})} \leq \begin{cases} 1 + \frac{(p'+l(H_{l-1}-1/2))s}{n+(l/2)s} & \text{if } l < p' \\ 1 + \frac{(H_{p'}+p'/l+1/2)ls}{n+(l/2)s} & \text{if } l \geq p' \end{cases}$$

Note that the total number of requests (n) in the considered phase of RR-PROC-MARK is at least k (since the phase ends only after all of the k blocks in the cache are marked).

$$\frac{accesscost(\text{RR-PROC-MARK})}{accesscost(\text{OPT})} \leq \begin{cases} 1 + \frac{(p'+l(H_{l-1}-1/2))s}{k+(ls/2)} & \text{if } l < p' \\ 1 + \frac{(H_{p'}+3/2)ls}{k+(ls/2)} & \text{if } l \geq p' \end{cases}$$

We now consider each of these equations individually.

Case 1: $l < p'$

$$\frac{accesscost(\text{RR-PROC-MARK})}{accesscost(\text{OPT})} \leq 1 + \frac{(p' + l(H_{l-1} - 1/2))s}{k + (l/2)s}$$

We can observe that the expression is monotonically increasing with l and achieves maximum value at the maximum value of l. Maximum value of l in this case is p'. We have,

$$\begin{aligned}\frac{accesscost(\text{RR-PROC-MARK})}{accesscost(\text{OPT})} &\leq 1 + \frac{(p' + p'(H_{p'-1} - 1/2))s}{k + (p'/2)s} \\ &\leq 1 + \frac{(H_{p'} + 1/2)s}{k/p' + s/2}\end{aligned}$$

Recall that $k \geq p \geq p'$. The cache should at least as large as the number of processes in order to accommodate at least one block requested by each of the p processes. Since $k \geq p'$, we have $k/p' \geq 1$. Hence,

$$\begin{aligned}\frac{accesscost(\text{RR-PROC-MARK})}{accesscost(\text{OPT})} &\leq 1 + \frac{(H_{p'} + 1/2)s}{1 + s/2} \\ &\leq 1 + \frac{2(H_{p'} + 1/2)s}{2 + s}\end{aligned}$$

Further, k is usually orders of magnitude larger than p. Under a very reasonable assumption that $k \geq pH_p \geq p'H_{p'}$, we have:

$$\begin{aligned}\frac{accesscost(\text{RR-PROC-MARK})}{accesscost(\text{OPT})} &\leq 1 + \frac{(H_{p'} + 1/2)s}{H_{p'} + s/2} \\ &\leq 1 + \frac{2(H_{p'} + 1/2)s}{2H_{p'} + s}\end{aligned}$$

Case 2: $l \geq p'$

$$\begin{aligned}\frac{accesscost(\text{RR-PROC-MARK})}{accesscost(\text{OPT})} &\leq 1 + \frac{(H_{p'} + 3/2)ls}{k + (ls/2)} \\ &\leq 1 + \frac{(H_{p'} + 3/2)s}{k/l + s/2}\end{aligned}$$

We can observe that the expression is monotonically increasing and achieves its maximum at the maximum value of l. Note that the maximum number of cache misses incurred by RR-PROC-MARK in any marking phase is k. Recall that the number of cache misses incurred by RR-PROC-MARK is at most $l \cdot (H_{p'} + 2)$ for this phase when $l \geq p'$. This quantity cannot exceed k. Hence,

$$\begin{aligned} l \cdot (H_{p'} + 2) &\leq k \\ \frac{k}{l} &\geq (H_{p'} + 2) \end{aligned}$$

This implies that the competitive ratio of RR-PROC-MARK in the full access cost model when $l \geq p'$ is at most:

$$\frac{accesscost(\text{RR-PROC-MARK})}{accesscost(\text{OPT})} \leq 1 + \frac{2(H_{p'} + 3/2)s}{2(H_{p'} + 2) + s}$$

We conclude by stating our results for full access model as follows: When $k \geq p$:

$$\frac{accesscost(\text{RR-PROC-MARK})}{accesscost(\text{OPT})} \leq \begin{cases} 1 + \frac{2(H_p+1/2)s}{2+s} & \text{if } l < p' \\ 1 + \frac{2(H_p+3/2)s}{2(H_p+2)+s} & \text{if } l \geq p' \end{cases}$$

When $k \geq pH_p$:

$$\frac{accesscost(\text{RR-PROC-MARK})}{accesscost(\text{OPT})} \leq \begin{cases} 1 + \frac{2(H_p+1/2)s}{2H_p+s} & \text{if } l < p' \\ 1 + \frac{2(H_p+3/2)s}{2(H_p+2)+s} & \text{if } l \geq p' \end{cases}$$

Representing constants by $O(1)$ and substituting $s = b - 1$, the competitive ratio of RR-PROC-MARK in full access cost model is at most $1 + \frac{2(H_p+O(1))(b-1)}{b+1}$ when $k \geq p$ and $1 + \frac{2(H_p+O(1))(b-1)}{2(H_p+O(1))+b-1}$ when $k \geq pH_p$.

□

Chapter 4

Shared memory framework

In this chapter we present our results on cache replacement algorithms in the *shared memory framework*. We classify cache replacement algorithms in the shared memory framework into two families — *local* and *global* algorithms. We establish lower bounds on the competitive ratio of deterministic local and global cache replacement algorithms in the shared memory framework. The main contribution of this chapter in terms of algorithm is a deterministic global marking algorithm called GLOBAL-MAXIMA. We prove that GLOBAL-MAXIMA is optimal (up to a constant factor ≤ 5) in the shared memory framework. Further, we establish a lower bound on the competitive ratio of randomized cache replacement algorithms in the shared memory framework. We show that a randomized cache replacement algorithm called PARTITION [19] proposed in the classical caching framework is optimal up to a constant factor in the shared memory framework.

Section 4.1 describes and motivates the shared memory framework. In Section 4.2, we describe local and global algorithms. In Section 4.3, we present lower bounds on the competitive ratio of deterministic (local and global) and randomized algorithms in this model. In Section 4.4, we present a natural deterministic marking algorithm called GLOBAL-MAXIMA which is optimal (up to a constant factor ≤ 5) in the shared memory framework.

4.1 Shared memory framework description

The shared memory framework is a generalization of the disjoint memory framework which was introduced in [8] and discussed in Chapter 3. In contrast to the disjoint memory framework, processes in the shared memory framework can share memory blocks, i.e., every block in the memory can be

accessed by any subset of processes.

We present a formal description of the shared memory framework below. Points 1 through 6 are identical to the corresponding points in the disjoint memory framework description. The last point is specific to the shared memory framework:

1. We let p processes share a cache of size k. We let P_i denote the ith process.

2. Each process has full knowledge about its request sequence. We use σ_i to represent the request sequence of the ith process. Recall that this request sequence is the exact order in which memory blocks are requested by the process P_i in the future.

3. An arbitrary interleaving of $\sigma_1, \sigma_2, \ldots, \sigma_p$ is seen at the shared cache. The analysis of algorithms at the shared cache is with respect to a worst case interleaving (represented by σ).

4. The online algorithm at the shared cache does not have any knowledge about the interleaving of request sequences from the p processes.

5. The optimal offline algorithm (OPT) on the other hand, has complete control of and knowledge about the interleaving.

6. The interleaving is assumed to be fixed, i.e., the same interleaving is assumed to reach the shared cache for all the cache replacement algorithms, including the optimal offline algorithm — irrespective of the cache replacement decisions taken at the shared cache.

7. The processes request non-disjoint sets of memory blocks. Thus, every block in the memory can be accessed by any subset of p processes.

Motivation for the shared memory framework: Processes share memory blocks in a number of parallel shared memory algorithms when they work on different parts of the computations together. In several of such applications, processes also have perfect knowledge about the sequence of requests they plan

to request in the future since they work on a well-defined computation like matrix multiplication or Gaussian elimination paradigm [9, 10]. Observe that in these computations, the interleaving of requests from different processes reaching the shared cache still remains adversarial since the interleaving depends on factors like the difference in the clock period, interrupts from the operating systems, etc. This motivates us to consider the shared memory framework which is a generalization of the application controlled caching framework.

4.2 Local and global algorithms

We classify algorithms in the shared memory framework into two families — local and global algorithms. Recall that each process in the shared memory framework has full knowledge about its future request sequence.

- **Local algorithms:** Local algorithms are a family of cache replacement algorithms which take eviction decisions based on the local knowledge of the request sequence available at one process.
- **Global algorithms:** Global algorithms are a family of cache replacement algorithms which aggregate knowledge about the request sequences from all the p processes and uses this global information to take an eviction decision.

We establish a lower bound of k on the competitive ratio of the deterministic local algorithms when the adversary knows the eviction strategy used by the algorithm in Subsection 4.3.2. Recall that a simple deterministic no-knowledge algorithm like LRU achieves a competitive ratio of k [22] at the shared cache by completely ignoring the knowledge about the future request sequences. This motivates us to search for better cache replacement algorithms in the shared memory framework which use the available information regarding the request sequences of all the p processes. We will show that deterministic global algorithms have a lower bound of $\frac{p}{2}\log\frac{4(k+1)}{3p}$ on their competitive ratio in Subsection 4.3.3. We establish a lower bound of $\frac{1}{2}\log(k+1)$ on the competitive ratio of randomized algorithms in Subsection 4.3.4.

4.3 Lower bound on the competitive ratio

In this section we establish lower bounds on the competitive ratio of deterministic (local and global) and randomized algorithms in the shared memory framework. In Subsection 4.3.1, we present the individual request sequences which we use to establish lower bounds on the competitive ratio of both deterministic (local and global) and randomized algorithms. The main idea is to fix the individual request sequences for all three types of algorithms and control the interleaving in an adversarial manner.

4.3.1 Individual request sequences

Our approach for establishing the lower bounds involves fixing the individual request sequences for all the p processes and building an adversarial interleaving. In fact, we use the exact same request sequence for all the lower bound proofs that we present in this section.

We group the p processes into $p/2$ pairs (each pair containing 2 processes). We assume that p is even in order to keep the analysis simple. The ith pair consists of process P_{2i-1} and P_{2i}. Both the processes in every pair request the exact same set of memory blocks but in different order.

The lower bound request sequence consists of requests to $k+1$ distinct memory blocks. These blocks are evenly distributed among $p/2$ pairs of processes. Hence, every pair requests $2(k+1)/p$ distinct memory blocks. Processes in the ith pair request the following distinct memory blocks:

$$(2i-2)\frac{k+1}{p}+1, (2i-2)\frac{k+1}{p}+2, \ldots, 2i\frac{k+1}{p}$$

Process P_{2i-1} requests these blocks in the order mentioned above and process P_{2i} requests these blocks in the reverse order (Table 4.1).

By defining an interleaving of requests from the individual request sequences in Table 4.1, we establish a lower bound on the competitive ratio of marking algorithms (deterministic local, global and randomized algorithms). We use the notion of *repeated requests to already requested blocks* (described in the next paragraph) to generalize the established lower bound to non-marking

Table 4.1: Individual lower bound request sequences

1^{st} Pair		2^{nd} Pair		$\cdots$	ith Pair	
P_1	P_2	P_3	P_4		P_{2i-1}	P_{2i}
1	$2\frac{k+1}{p}$	$2\frac{k+1}{p}+1$	$4\frac{k+1}{p}$		$(2i-2)\frac{k+1}{p}+1$	$2i\frac{k+1}{p}$
2	$2\frac{k+1}{p}-1$	$2\frac{k+1}{p}+2$	$4\frac{k+1}{p}-1$		$(2i-2)\frac{k+1}{p}+2$	$2i\frac{k+1}{p}-1$
.	.	.	.		.	.
.	.	.	.		.	.
$2\frac{k+1}{p}-1$	2	$4\frac{k+1}{p}-1$	$2\frac{k+1}{p}+2$		$2i\frac{k+1}{p}-1$	$(2i-2)\frac{k+1}{p}+1$
$2\frac{k+1}{p}$	1	$4\frac{k+1}{p}$	$2\frac{k+1}{p}+1$		$2i\frac{k+1}{p}$	$(2i-2)\frac{k+1}{p}$

algorithms. The idea is to prove that on a small tweak of the above defined request sequence, a non-marking algorithm always incurs at least as many cache as an optimal marking algorithm. This allows a lower bound established on marking algorithms to extend for all the cache replacement algorithms.

Repeated requests to already requested blocks: The notion of repeatedly requesting already requested blocks was introduced and used in the lower bound proofs in [6]. The intention is make an algorithm pay by extra cache misses if it is not a marking algorithm by repeatedly requesting the blocks that have already been requested during the phase.

As a quick review we present the main difference between marking and non-marking algorithms. Recall that marking algorithms proceed in marking phases. At the start of a phase, all the blocks in the cache are unmarked. A block gets marked when a request to it is served. Upon a cache miss, marking algorithms evicts an *unmarked* block. When all the blocks in the cache are marked, they all are unmarked and a new phase starts. We call this a *marking phase*.

In contrast to the marking algorithms, non-marking algorithms do not follow the marking scheme and hence evict any block (even marked) from the cache upon a cache miss. Intuitively, marking algorithms perform better than

non-marking algorithms in the shared memory framework because marking algorithms retain blocks that were requested during a phase till the end of the phase. If an algorithm evicts a block that was recently brought in, the adversary can easily request the evicted block and make the algorithm incur extra cache miss without it incurring any cache miss.

We modify our lower bound sequences presented in Table 4.1 to accommodate this flexibility for the adversary by repeatedly requesting already requested blocks. The request sequence of process P_i is given by:

$$\sigma_i = s_1, s_2, \ldots, s_j$$

The new request sequence σ_i' is given by:

$$\sigma_i' = s_1, t_1, s_2, t_2, \ldots, s_j, t_j$$

Where $t_j = s_1, s_2, \ldots, s_j$. For instance, consider the blocks requested by P_1:

$$\sigma_1 = 1, 2, 3, \ldots, 2\frac{k+1}{p} - 1, 2\frac{k+1}{p}$$

We modify this sequence to the set of blocks that were already requested by P_1 before requesting a new block. The changed sequence looks as follows:

$$1^*, 1, 2^*, 1, 2, 3^* \ldots, 1, 2, 3, \ldots, 2\frac{k+1^*}{p}, 1, 2, \ldots, 2\frac{k+1}{p}$$

We use the notation of x^* in the above sequence to represent the first request to block x. Observe that the sequence that is described above forces any cache replacement algorithm not to evict blocks that were requested previously during the current phase. Because if at some point, an algorithm evicts a block that was requested during the current phase, the adversary can easily request that block by marking at most one block.

Thus an algorithm incurs at least as many cache misses as the optimal marking algorithm. Hence, establishing a lower bound on the competitive ratio of marking algorithms is sufficient. The same lower bound extends to other algorithms. For completeness, we give the final set of individual request sequences used in our lower bound proofs in Table 4.2.

Table 4.2: Individual lower bound request sequences with repeated requests to already requested blocks for ith pair of processes

ith Pair	
P_{2i-1}	P_{2i}
$[(2i-2)\frac{k+1}{p}+1]^*$	$[2i\frac{k+1}{p}]^*$
$(2i-2)\frac{k+1}{p}+1, [(2i-2)\frac{k+1}{p}+2]^*$	$2i\frac{k+1}{p}, [2i\frac{k+1}{p}-1]^*$
.	.
.	.
$(2i-2)\frac{k+1}{p}+1, (2i-2)\frac{k+1}{p}+2, \ldots, [2i\frac{k+1}{p}]^*$	$2i\frac{k+1}{p}, 2i\frac{k+1}{p}-1, \ldots, [(2i-2)\frac{k+1}{p}]^*$

In conclusion, the individual request sequences can be easily modified to make a non-marking algorithm incur at least as many cache misses as a marking algorithm in the shared memory framework. Hence, we just concentrate on marking algorithms and establish lower bounds on the competitive ratio of deterministic and randomized marking algorithms using the individual request sequence presented in Table 4.1.

4.3.2 Lower bound for deterministic local algorithms

Theorem 4.3.1. The competitive ratio of any deterministic local algorithm in the shared memory framework is at least k provided the adversary knows the eviction strategy used by the algorithm.

Proof. We start with a deterministic local marking algorithm ALG and establish a lower bound on the competitive ratio of ALG. Using the technique of repeated requests to already requested blocks, we generalize the lower bound to all the deterministic local algorithms.

We assume that the eviction policy used by the algorithm is known beforehand. Without loss of generality, we assume that upon a cache miss, the local algorithm evicts an unmarked block that is requested farthest in the future request sequence (MARK-FITF) of a particular process. If a different

eviction strategy is used, the adversary can suitably modify the individual request sequences. For the case when MARK-FITF is used, the individual request sequences for p processes are as specified in Table 4.1. Once the individual request sequences are fixed, the adversary just controls the interleaving of requests from these sequences. We shall describe construction of one phase in this proof. The remaining phases can be constructed in a similar fashion.

Without loss of generality, we assume that the shared caches of both ALG and OPT are initially warm and contain $1, 2, \ldots, k$ blocks. If this is not the case, the adversary can request a set of blocks before starting the lower bound sequence in order to bring both these caches to the same state.

The first phase starts with process P_p requesting $k+1$. Since $k+1$ does not exist in the cache, ALG takes an eviction decision based on the local knowledge of one of the p processes, say P_{2i-1}. Since we assume that an unmarked block that it requested farthest in its future request sequence of P_{2i-1} is evicted, ALG ends up evicting $2i\frac{k}{p}$. Note that P_{2i} can immediately request this block. The adversary makes P_{2i} request $2i\frac{k}{p}$. This request results in another cache miss. In order to serve this request ALG uses the local knowledge of another process to make an eviction and the pattern repeats. This continues till all the blocks in the cache are marked. At that point, the adversary makes all the p processes request the remaining blocks in their request sequences in order to start a new phase.

Note that ALG ends up incurring a cache miss on every request during this phase. The optimal offline algorithm OPT on the other hand incurs just 1 cache miss by evicting the block that is requested farthest in the interleaved request sequence when it incurs a cache miss on $k+1$. The last cache miss of the first phase falls into the second phase and hence, the lower bound on the competitive ratio of a deterministic local marking algorithm for this phase is k. Since every phase proceeds in a similar fashion, the lower bound on the competitive ratio of deterministic local marking algorithms is k.

Recall that previously we proved that the number of cache misses incurred by any non-marking algorithm is at least as many as an optimal marking algorithm when the request sequence is tweaked slightly. Hence the lower bound of k holds for any deterministic local algorithm when the eviction strategy used by the algorithm is known.

□

4.3.3 Lower bound for deterministic global algorithms

Theorem 4.3.2. The competitive ratio of any deterministic global algorithm in the shared memory framework is at least $\frac{p}{2}\log\frac{4(k+1)}{3p}$.

Proof. We start with a deterministic global marking algorithm ALG and establish a lower bound on the competitive ratio of ALG — similar to the proof for lower bound on the competitive ratio of deterministic local algorithms. Using the technique of repeated requests to already requested blocks, we generalize the lower bound to all the deterministic global algorithms. The individual request sequences for the p processes are again as specified in Table 4.1.

Observe that the cache of ALG has exactly k blocks and hence one out of these $k+1$ distinct memory blocks in the request sequences, say q, does not exist in the cache. The block q appears in the request sequences of two out of p processes, say P_{2i-1} and P_{2i}. The adversary picks one of these two processes such that minimum number of blocks get marked before q is requested again. Let P_{2i} be the picked process. The adversary makes P_{2i} request q. Since q was not in the cache, request to q results in a cache miss. In order to serve this request another block is deterministically chosen for eviction and the pattern repeats.

Lemma 4.3.3 which follows this theorem proves that the number of cache misses incurred by ALG in the first phase is at least $\frac{p}{2}\log\frac{4(k+1)}{3p}$. OPT on the other hand incurs just 1 cache miss by evicting the block that is requested farthest in the interleaved request sequence when it incurs a cache miss during this phase. Hence, the lower bound on the competitive ratio of deterministic global marking algorithm for this phase is $\frac{p}{2}\log\frac{4(k+1)}{3p}$. Since every phase proceeds in a similar fashion, the lower bound on the competitive ratio of deterministic global marking algorithms is $\frac{p}{2}\log\frac{4(k+1)}{3p}$.

□

Lemma 4.3.3. The total number of cache misses incurred by any deterministic global marking algorithm in the first phase of the request sequence is at least $\frac{p}{2}\log\frac{4(k+1)}{3p}$.

Proof. Consider a deterministic global marking algorithm ALG. We prove that ALG incurs at least $\log \frac{4(k+1)}{3p}$ cache misses for every pair of processes. i.e., given a pair of processes, we prove that ALG incurs at least $\log \frac{4(k+1)}{3p}$ cache misses on the blocks requested by these processes before all the blocks requested by these processes are marked. Since we have $p/2$ such pairs, the cost of ALG is at least $\frac{p}{2} \log \frac{4(k+1)}{3p}$. We consider the ith pair (P_{2i-1} and P_{2i}) and establish a bound on the number of cache misses incurred by ALG for this pair.

Let $U = \{(2i-2)\frac{k+1}{p}+1, (2i-2)\frac{k+1}{p}+2, \ldots, 2i\frac{k+1}{p}\}$ be the set of blocks in the request sequences of P_{2i-1} and P_{2i}. Let U_j be the set of all unmarked blocks in the request sequences of P_{2i-1} and P_{2i} just before the jth block in U is evicted by ALG. Further, we use u_j to represent $|U_j|$.

Note that the adversary does not request blocks in U till a block $\in U$ is evicted from the cache and hence $U_1 = U$. In order to bound the number of cache misses incurred by ALG on the blocks in U before all the blocks in U are marked, we bound u_j in terms of u_{j-1} for an arbitrary j. Consider the state of the request sequences of P_{2i-1} and P_{2i} just before the $j-1^{st}$ block in U was evicted. The number of unmarked blocks in the request sequences of P_{2i-1} and P_{2i} at this point is exactly u_{j-1}. Let q be the $j-1^{st}$ block in U that is evicted by ALG.

Let the number of unmarked blocks requested before the first request to q in the request sequence of P_{2i-1} and P_{2i} be $d_{2i-1}(q)$ and $d_{2i}(q)$, respectively. Adversary requests q such that at most $\min(d_{2i-1}(q), d_{2i}(q))$ unmarked blocks in U_{j-1} are marked before requesting q again. Further, q also gets marked as soon as the request to q is served. Hence, at most $\min(d_{2i-1}(x), d_{2i}(x)) + 1$ blocks in U_{j-1} get marked by the time the request to q is served.

After the request to q is served, the adversary does not request any other block belonging to the request sequences of P_{2i-1} and P_{2i} till the jth element in U is evicted. Hence u_j is equal to the number of elements left unmarked in the the request sequences of P_{2i-1} and P_{2i} after the request to q is served. Hence,

$$u_j = u_{j-1} - (\min(d_{2i-1}(q), d_{2i}(q)) + 1) \tag{4.1}$$

Now, we bound $\min(d_{2i-1}(q), d_{2i}(q))$ from above. Observe that the sequence of unmarked blocks in the request sequence of P_{2i-1} always remains

an exact reverse of the sequence of unmarked blocks in the request sequence of P_{2i}. Hence for all x in U_{j-1}, we have,

$$\min(d_{2i-1}(x), d_{2i}(x)) \leq \lceil u_{j-1}/2 \rceil - 1$$

Since $d_i(x) \leq d_i(q)$, we have,

$$\min(d_{2i-1}(q), d_{2i}(q)) \leq \lceil u_{j-1}/2 \rceil - 1 \tag{4.2}$$

From Equations 4.1 and 4.2, we have,

$$u_j \geq u_{j-1} - \lceil u_{j-1}/2 \rceil$$

and by rearranging the terms, we get,

$$u_j \geq \lfloor u_{j-1}/2 \rfloor \tag{4.3}$$

From the Equation 4.3 and the fact that $u_1 = 2(k+1)/p$, we can prove by induction on j (for all j such that $j \geq 2$) that:

$$u_j \geq \frac{k+1}{2^{j-2}p} - \sum_{l=0}^{j-2} \frac{1}{2^l}$$

Base case: $j = 2$: The claim clearly holds for the case when $j = 2$ because, from Equation 4.3, we have:

$$u_2 \geq \lfloor u_1/2 \rfloor$$

Substituting $\frac{k+1}{p}$ for u_1,

$$u_2 \geq \frac{k+1}{p} - 1$$

Induction step: Assume that the claim holds for $j-1$. We shall prove that it holds for j.

$$\begin{aligned} u_j &\geq \lfloor u_{j-1}/2 \rfloor \\ &\geq \lfloor \frac{\frac{k+1}{2^{j-3}p} - \sum_{l=0}^{j-3} \frac{1}{2^l}}{2} \rfloor \\ &\geq \frac{k+1}{2^{j-2}p} - \frac{1}{2} \sum_{l=0}^{j-3} \frac{1}{2^l} - 1 \\ &\geq \frac{k+1}{2^{j-2}p} - \sum_{l=0}^{j-2} \frac{1}{2^l} \end{aligned}$$

Hence proved.

The phase ends for the ith pair of processes when there is just one unmarked block left. Hence, we let u_j to go to 1:

$$\begin{aligned} 1 &\geq \frac{k+1}{2^{j-2}p} - \sum_{l=0}^{j-2} \frac{1}{2^l} \\ &\geq \frac{k+1}{2^{j-2}p} - (2 - \frac{1}{2^{j-2}}) \end{aligned}$$

Rearranging,

$$\begin{aligned} 3 \cdot 2^{j-2} &\geq \frac{k+1}{p} + 1 \\ j &\geq \log \frac{4(k+1)}{3p} \end{aligned}$$

Hence a global algorithm ALG incurs at least $\log \frac{4(k+1)}{3p}$ cache misses on blocks in U before all the blocks in U get marked. Hence, ALG incurs at least $\frac{p}{2} \log \frac{4(k+1)}{3p}$ cache misses in the entire phase.

□

4.3.4 Lower bound for randomized algorithms

Theorem 4.3.4. The competitive ratio of any randomized algorithm in the shared memory framework is at least $\frac{1}{2}\log(k+1)$.

Proof. As discussed before, we just concentrate on marking algorithms in the proof. We shall later generalize it to any randomized cache replacement algorithm.

In order to establish a lower bound on the competitive ratio of randomized marking algorithms, we use the von Neumann minimax principle as described by Yao [27]. We give a probability distribution on the interleaving of requests from the individual request sequences (Table 4.1). We then calculate

the expected number of cache misses incurred by any deterministic marking algorithm ALG. This will give us a lower bound on the cost of randomized marking algorithms in an expected sense.

Now we describe the construction of the interleaved sequence along with the probability distribution. We maintain a mark bit for every block in the cache. Initially, all the blocks that exist in the cache of ALG are unmarked. We mark a block when the adversary requests it. The following algorithm is used to generate a phase of the interleaved request sequence:

- Repeat the following steps till all the blocks in the cache are marked:
  1. *Start a new stage:* for i from 1 through $p/2$: repeat the following steps:
     (a) Pick one of the two processes in the ith pair (P_{2i-1} or P_{2i}) with equal probability (1/2).
     (b) Let u be the total number of unmarked blocks in the request sequences of the ith pair of processes. Request the first $u/2$ unmarked blocks belonging to the process picked in step 1a.
  2. Mark all the blocks that were requested in the first step.

We show that the expected cost of ALG on the probabilistic interleaving described above is at least $\log(k+1)$. Let k_j be the total number of distinct unmarked blocks in the request sequences of all the p processes at the start of the jth stage. Exactly $k_j/2$ blocks are requested and hence marked in the jth stage. The phase ends when all the blocks in the cache are marked. Hence, the total number of stages in a given phase is no less than $\log k_1$.

In every stage, each unmarked block is requested with probability 1/2. Since one of the k_j unmarked blocks is missing from the cache, the expected number of cache misses incurred by ALG in the jth stage is 1/2.

Hence, the total number of cache misses incurred by ALG in the entire phase is at least $\frac{1}{2}\log k_1$ in an expected sense. Since $k_1 = k+1$, the expected number of cache misses incurred by ALG is at least $\frac{1}{2}\log(k+1)$.

The lower bound on the cost of a randomized marking algorithm is $\frac{1}{2}\log(k+1)$ in an expected sense. Recall that OPT incurs exactly one cache miss on any deterministic interleaving consisting of $k+1$ distinct blocks.

As noted in the previous two proofs, the lower bound for randomized marking algorithms can be generalized to randomized algorithms by using the concept of repeated requests to already requested blocks. Hence the competitive ratio of any randomized algorithm is at least $\frac{1}{2}\log(k+1)$.

□

A few observations on these lower bounds:

1. The lower bound on the competitive ratio of deterministic local algorithms suggests that relying on the local knowledge about the future request sequence of a particular process does not turn out to be useful in the shared memory framework. A simple deterministic marking algorithm like LRU matches the lower bound of k by ignoring the knowledge about the future request sequences.

2. The lower bound on the competitive ratio of randomized algorithms proves that randomization does not yield impressive results in the shared memory framework. PARTITION [19] achieves an optimal competitive ratio of H_k in the sequential caching framework. One can use PARTITION at the shared cache in the shared memory framework by ignoring the knowledge about the future request sequences. Our lower bound on randomized algorithms proves that PARTITION remains optimal (up to a constant factor) in the shared memory framework.

3. The lower bound on the competitive ratio of global algorithms gives hope that more efficient algorithms may exist in the shared memory framework. Making eviction decisions based on the global knowledge about the future request sequences from all the p processes seems to be the key for better cache replacement algorithms in this framework. In the next section, we develop a deterministic global marking algorithm called GLOBAL-MAXIMA motivated by this observation.

4.4 Deterministic global algorithm : GLOBAL-MAXIMA

We present a natural deterministic global marking algorithm called GLOBAL-MAXIMA for the shared memory framework. We shall prove that GLOBAL-MAXIMA is optimal (up to a constant factor ≤ 5) in the shared memory framework. We describe the algorithm in Subsection 4.4.1 and analyze its competitive ratio in Subsection 4.4.2.

4.4.1 Description of GLOBAL-MAXIMA

GLOBAL-MAXIMA is a global algorithm. Recall that a global algorithm aggregates knowledge regarding the future request sequences from all the p processes and makes an eviction decision based on the global knowledge. GLOBAL-MAXIMA is also an explicit marking algorithm. It proceeds in marking phases. A phase starts with all the blocks unmarked. A block gets marked when a request to it is served. Upon a cache miss, a global distance function (described in the algorithm) is applied on all the unmarked blocks in the cache. An unmarked block with the maximum global distance is evicted from the cache.

GLOBAL-MAXIMA(r):

Input: Requested cache block r.

Output: Eviction decision.

- **if** r is in the cache: Mark r if it is not marked. No eviction is needed in this case.
- **if** r is *not* in the cache:
  1. If all the blocks in the cache are marked (end of phase), start a new phase by unmarking all the blocks.
  2. Let U be the set of all unmarked blocks in the cache.
  3. *Compute local distance:* For all $x \in U$ and $1 \leq i \leq p : d_i(x) =$ number of distinct blocks $\in U$ in the request sequence of process

P_i before the first request to x. If x never occurs in the request sequence of process P_i, $d_i(x) = |U| - 1$.

4. *Compute global distance:* For all $x \in U : d(x) = \min_{i=1}^{p} d_i(x)$
5. Evict an unmarked block q such that, $q = \text{argmax}_{x \in U}\, d(x)$
6. Bring r in place of q and mark r.

4.4.2 Competitive ratio of GLOBAL-MAXIMA

In this section, we prove that the competitive ratio of GLOBAL-MAXIMA in the shared memory framework is at most $2(p \ln(ek/p) + 1)$. We use the concept of *holes* to establish an upper bound on the competitive ratio of GLOBAL-MAXIMA. Here is a quick recap of the definitions and notations around holes borrowed from [3] and stated in Chapter 3. All the points except the last 3 are exactly similar to the ones stated in Chapter 3.

1. **Clean block:** A block q is said to be clean block with respect to a particular phase of a process marking algorithm, if q does *not exist* in the cache at the start of this phase.

2. **Non-clean block:** A block q is said to be non-clean block with respect to a particular phase of a process marking algorithm, if q *exists* in the cache at the start of this phase.

3. **Hole:** When a non-clean block q is evicted from the cache, in order to serve the request to a clean block, we say that a hole h is *created* at q. A hole basically suggests that the block q is missing from the cache.

4. **Hole association:** Since the hole h is created due to eviction of q, we also say that h is *associated* with q.

5. **Hole movement:** Let q be requested again by some process at some point during the phase. Since q is not in the cache, another unmarked block q' is evicted in order to serve the request to q. At that point, we say that the hole h *moves* from q to q'. It gets associated with q' from now on.

6. **Relating cache misses and holes:** Every cache miss results either in creation or in movement of a hole. In case of a marking algorithm, a hole is always associated with an unmarked block (recall that always an unmarked block is evicted from the cache). Since all the clean blocks are marked when they are brought in, holes are always associated with non-clean blocks.

Theorem 4.4.1. The competitive ratio of GLOBAL-MAXIMA in the shared memory framework is at most $2(p\ln(ek/p)+1)$.

Before proving this theorem we make the following observations regarding the local and global distance functions (d_i and d respectively) defined and used in our algorithm.

Observations:

1. $0 \leq d_i(x) \leq k-1$ for all $1 \leq i \leq p$ and $x \in U$.
2. $0 \leq d(x) \leq k-1$ for all $x \in U$.
3. For every eviction candidate q in our algorithm, $\frac{|U|}{p} - 1 \leq d(q) \leq k-1$.

The first two bounds directly follow from the definitions of local and global distance functions (Lines 3 and 4 of our algorithm). The third bound is proved by Lemma 4.4.2 which is presented after the proof for Theorem 4.4.1.

Significance of the global distance function (d): The global distance of an unmarked block q at the time q was evicted from the cache is the minimum number of blocks that get marked before q is requested again in the current phase. This is because, there are at least $d(q)$ unmarked blocks in cache that are requested before the first request to q in the future request sequences of all the p processes. Upon a cache miss, GLOBAL-MAXIMA chooses an unmarked block q with maximum value for its global distance and evicts it from the cache. By doing so, GLOBAL-MAXIMA is ensuring that maximum number of unmarked blocks get marked before q gets requested again.

Proof. This proof is for Theorem 4.4.1. Consider an arbitrary phase of GLOBAL-MAXIMA. Let l be the number of clean block requests during this phase. These l requests to clean blocks result in creation of l holes. Consider one such hole, h.

We aim to bound the number of cache misses due to repeated requests to h during the current phase. Let u_0 be the number of unmarked blocks in the cache when h is created (due to a request to a clean block). Note that, $u_0 \leq k$. Let u_1 be the number of unmarked blocks in the cache when the non-clean block associated with h is requested for the first time.

More generally, let u_j represent the number of unmarked blocks in the cache when the non-clean block associated with h is requested for the jth time.

Lemma 4.4.3 (which is presented at the end of this proof) proves that at least $\frac{u_{j-1}}{p}$ blocks are marked between two consecutive requests ($j-1^{st}$ and jth requests) to the non-clean blocks associated with h. Hence,

$$\begin{aligned} u_{j-1} - u_j &\geq \frac{u_{j-1}}{p} \\ u_j &\leq u_{j-1} \cdot (1 - 1/p) \\ &\leq u_0 \cdot (1 - 1/p)^j \end{aligned}$$

Recall that $u_0 \leq k$,

$$u_j \leq k \cdot \frac{1}{e^{\frac{j}{p}}}$$

With at most $p \ln(k/p)$ requests to h, the number of unmarked blocks in cache reduces to p. From that point on wards, at most p cache misses are incurred before all the blocks in the cache get marked (end of the current phase).

Total number of cache misses due to h is at most $p \ln(k/p) + p = p \ln(ek/p)$.

Since every cache miss (hole) is treated in a similar fashion, the total number of cache misses due to l holes is bounded by $l + lp \ln(ek/p)$ cache

misses. The extra l additive term is to account for the number of cache misses that occur due to creation of these l holes.

$$cost(\text{GLOBAL-MAXIMA}) \leq l \cdot (p \ln(ek/p) + 1)$$

Recall that OPT incurs at least $l/2$ cache misses on this phase in an amortized sense. Hence, the competitive ratio of GLOBAL-MAXIMA in the shared memory framework is at most $2(p \ln(ek/p) + 1)$.

□

To complete the proof of the theorem, we now state and prove Lemma 4.4.2 and 4.4.3.

Lemma 4.4.2. For every eviction candidate q in our algorithm,

$$\frac{|U|}{p} - 1 \leq d(q) \leq k - 1$$

Proof. Consider an eviction candidate q in our algorithm. From the definition of d, we have,

$$d(q) \leq k - 1$$

What is left to be shown is that $d(q) \geq \frac{|U|}{p} - 1$. Observe that for an eviction candidate q and a block x in U,

$$d(q) \geq d(x)$$

hence, it is sufficient if we prove that there exists an unmarked block x in the cache for which $d(x) \geq \frac{|U|}{p} - 1$. First of all, we assume that every unmarked block in the cache is requested by at least one process. If not, there exists an unmarked block, say x which is not requested by any process. Hence, for all i such that $1 \leq i \leq p$, we have,

$$d_i(x) = |U| - 1$$

and since,

$$d(x) = \min_{i=1}^{p} d_i(x)$$

we have,

$$d(x) = |U| - 1 \geq \frac{|U|}{p} - 1$$

Let V be the set of all unmarked blocks in cache which have their global distance function less than $\frac{|U|}{p} - 1$. Hence, for all x in V,

$$d(x) < \frac{|U|}{p} - 1$$

For all, x in V, there exists an i such that $1 \leq i \leq p$, for which,

$$d_i(x) \leq \frac{|U|}{p} - 2$$

Consider any 2 elements requested by P_i — x and y. Clearly, either the first occurrence of x appears before the first occurrence of y or vice versa. Hence, $d_i(x) \neq d_i(y)$. This implies that for a given i, the number of elements with $0 \leq d_i(x) \leq \frac{|U|}{p} - 2$ is at most $\frac{|U|}{p} - 1$.

Hence, the number of elements with $0 \leq d_i(x) \leq \frac{|U|}{p} - 2$ for all i is at most $p \cdot (\frac{|U|}{p} - 1) = |U| - p$.

The remaining p unmarked blocks in $U - V$ have their global distance function at least $\frac{|U|}{p} - 1$. Hence there exists an unmarked block $x \in U$ whose global distance function is at least $\frac{|U|}{p} - 1$. Hence proved.

□

Lemma 4.4.3. Consider a hole h during the current phase of our algorithm. The number of unmarked blocks that get marked between any two consecutive requests ($j-1^{st}$ and jth request) to h is at least $\frac{u_{j-1}}{p}$ where u_{j-1} is the number of unmarked blocks in cache just before the $j - 1^{st}$ request to h.

Proof. Let q be the non-clean block which is evicted to serve the $j-1^{st}$ request to the unmarked block associated with h. The number of unmarked blocks in cache when q is evicted is exactly u_{j-1}. From Lemma 4.4.2, at the time q is evicted, we have,

$$d(q) \geq \frac{u_{j-1}}{p} - 1 \tag{4.4}$$

Since q is associated with h now, the jth request to a non-clean block associated with h is essentially a request to q. At least $d(q) + 1$ unmarked blocks get marked by the time the request q is served. The extra $+1$ term is to account for q getting marked. Hence the number of unmarked blocks in the cache after the request to q is served is

$$u_j \leq u_{j-1} - d(q) - 1$$

From Equation 4.4, we have,

$$\begin{aligned} u_j &\leq u_{j-1} - \frac{u_{j-1}}{p} \\ u_{j-1} - u_j &\geq \frac{u_{j-1}}{p} \end{aligned}$$

Hence the lemma is proved.

□

Chapter 5

Hierarchical caching framework

In this chapter we present our results on the performance of cache replacement algorithms in the hierarchical caching framework. The main motivation for considering the hierarchical caching framework is to model modern multicore processor systems which have a multi-level cache hierarchy in order to obtain better caching performance. In Section 5.1, we present a model for multicore systems with two levels of caches — L_1 and L_2. We consider the L_2 cache with 3 different properties - inclusive, exclusive and non-inclusive. In Section 5.2, we consider one core with two levels of caches and establish upper and lower bounds on the competitive ratio of deterministic cache replacement algorithms at inclusive, exclusive and non-inclusive L_2 caches. Recall that algorithms do not have any knowledge about the future request sequence in the sequential caching framework. In Section 5.3, we consider p cores with two levels of caches and establish upper and lower bounds on the competitive ratio of deterministic cache replacement algorithms at inclusive, exclusive and non-inclusive L_2 caches when each core has full knowledge about its future request sequence. In Section 5.4, we present a case study of the cache architecture in Intel Nehalem and AMD-Shanghai processors and discuss the relevance of our results in modern multicore processor systems.

5.1 Multicore systems with two levels of caches

In Chapters 3 and 4, we considered a simple framework with multiple processes sharing a cache and analyzed cache replacement algorithms at the shared cache. In this chapter we consider a theoretical model for the multicore processor systems with multiple levels of caches and analyze cache replacement algorithms at these caches. Initially, we consider a multicore processor system

with just two levels of caches and later extend these results to higher levels of caches.

A multicore processor system with two levels of caches consists of p cores, first level L_1 cache and a second level L_2 cache. The L_1 cache could be either private to each core (in which case, the system consists of p L_1 caches) or shared among all the p cores as in Chapters 3 and 4. Similarly, the L_2 cache could be either private to each core or shared among all the p cores. In our analysis, we consider both L_1 and L_2 caches to be shared among all the p cores. With two levels of caches, the contents of the L_2 cache could either *inclusive*, *exclusive* or *non-inclusive* of the contents of the L_1 cache. We describe these three types of caches in the following part of this section.

5.1.1 Inclusive, exclusive and non-inclusive caches

The L_2 cache is said to be an inclusive cache if the invariant: *"the contents of the L_1 cache are a strict subset of the contents of the L_2 cache"* is maintained. The inclusion property wastes expensive cache real estate by maintaining redundant copies of the memory blocks in both L_1 and L_2 caches. Thus, the total number of cache misses incurred by the system increases when the L_2 cache is inclusive of the L_1 cache. But, it turns out that the inclusion property improve the cache efficiency by reducing *coherence traffic* at the lower level caches. Further, it also decreases the core idle time and thereby increases the overall efficiency of the system. In the following part of this subsection, we describe the cache coherence problem and the role of inclusion property in solving the problem.

Hardware background: The L_1 cache is usually placed on the same chip as the core and hence is much faster and smaller when compared to the L_2 cache (which is typically placed on a different chip). The L_1 cache controller communicates with the core through the local processor bus. The L_2 cache controller communicates with the L_1 cache controller through the system bus. The Random Access Memory, which is modeled as the main memory lies outside of these chips and it communicates with all these cache controllers through the system bus. The cache controllers take charge of the system bus in order to read/write data from/to main memory.

Cache coherency, snooping and inclusion property: The cache coherence (or cache coherency) refers to the problem of maintaining consistency of memory blocks stored in the L_1 cache. Both hardware and software based solutions have been proposed to this problem in the literature [23, 25]. One commonly used hardware based solution for the cache coherence problem in *write-back* caches is *snooping*. In this technique, the cache controllers of the L_1 caches snoop the traffic through the system bus. There are two types of snooping — *read snooping* and *write snooping*.

In read snooping, the system bus is continually monitored by the cache controllers of the L_1 caches when they read memory blocks because there is a possibility that these blocks could be changed by another core and the changed copy might not have be updated in the main memory. In write snooping, the system bus is continually monitored by the cache controllers of the L_1 caches when they write to a memory block because there is a possibility that these blocks reside in caches of some other cores. Upon detecting an inconsistency, the cache which owns the consistent copy of the block writes it back to the main memory and all other caches reload this block from the main memory before proceeding with their current operation.

Snooping costs a lot to the L_1 cache controller in terms of both time and resources. Thus, the L_1 cache controllers will not be able to dedicate sufficient time and resources for serving block requests from the cores. To address this problem, the inclusion property was introduced. The inclusion property ensures that the L_2 cache has a copy of the contents in the L_1 cache at all time. This allows the L_1 caches to just concentrate on serving block requests from the cores the L_2 cache on the other hand, will take care of the snooping protocol. This results in a dramatic improvement in the performance of the L_1 caches and also reduces the core idle time.

In conclusion, the inclusion property has a very important role to play in multicore processor systems with multiple levels of write-back caches. The inclusion property helps the lower level caches concentrate more on serving the block requests from the cores and less on maintaining cache coherency. The upper level caches on the other hand continually snoop the system bus in order to maintain cache coherency.

Exclusive and non-inclusive caches: Even though making the L_2 cache inclusive improves the performance of the L_1 cache (when considered in isolation), the overall performance of the system degrades due to the wastage of cache real estate. In a retrospective paper on inclusive caches, Baer and Wang [2] present specific arguments against inclusive caches.

This led to two new types of caches: exclusive and non-inclusive caches. The L_2 cache is said to be an exclusive cache if the invariant: *"the contents of the L_1 and L_2 caches are disjoint"* is maintained. Non-inclusive caches on the other hand, do not enforce either of these (inclusive or exclusive) invariants.

Implementation of inclusive, exclusive and non-inclusive caches: We discuss the implementation details for inclusive, exclusive and non-inclusive caches below:

Inclusive cache: An inclusive L_2 cache is forced to include the contents of the L_1 cache. This is implemented by using inclusion bits. An inclusion bit is maintained for every cache line present in the L_1 cache. A detailed flowchart for the implementation of the inclusion property due to Tipley is presented in [24]. The technique can be described as follows: upon a cache miss at the L_1 cache, the request is sent to the L_2 cache. If the requested block exists in the L_2 cache, it is read into the L_1 cache. If the requested block does not exist in the L_2 cache, the block is read from the main memory into both L_2 and L_1 caches. A block whose inclusion bit is not set is evicted from the L_2 cache in order to accommodate the newly requested block. A block is evicted from the L_1 cache in order to accommodate the newly requested block and its corresponding inclusion bit is cleared in the L_2 cache.

Exclusive cache: An exclusive L_2 cache is forced to exclude the contents of the L_1 cache. This is implemented by using a technique called *victim caching*. A secondary cache like the L_2 cache is also referred to as a victim cache since it is used to store blocks evicted from the L_1 cache anticipating their access in the recent future. The AMD-Shanghai multicore processor system implements this technique. The technique can be described as follows: upon a cache miss at the L_1 cache, the request is sent to the L_2 cache. If the requested block exists in the L_2 cache, it is read into the L_1 cache. If the requested block does not exist in the L_2 cache, the block is read from the main

memory directly into the L_1 cache. A block, say x, is evicted from the L_1 cache. Instead of evicting x completely, it is stored in the L_2 cache. A block, say y, is evicted from the L_2 cache and x occupies the position of y in the L_2 cache.

Non-inclusive cache: This is the simplest of the three types of caches. A non-inclusive cache is implemented very similar to an inclusive cache except for the inclusion bits. Upon a cache miss at the L_1 cache, the request is sent to the L_2 cache. If the requested block exists in the L_2 cache, it is read into the L_1 cache. If the requested block does not exist in the L_2 cache, the block is read from the main memory into both L_2 and L_1 caches. A block is evicted from the L_2 cache in order to accommodate the newly requested block. A block is evicted from the L_1 cache in order to accommodate the newly requested block.

Comparing inclusive, exclusive and non-inclusive properties: The above-mentioned properties influence the efficiency of L_1 caches in terms of the time taken to serve a block request from the core. But the number of cache misses incurred by a cache replacement algorithm at the L_1 cache is not influenced and hence the competitive ratio of cache replacement algorithms at the L_1 cache remains unchanged. However, the number of cache misses incurred by cache replacement algorithms at the L_2 cache is influenced by the above-mentioned properties. Hence, we consider the competitive ratio of well known algorithms at the L_2 cache to compare inclusive, exclusive and non-inclusive properties. The number of cache misses at the L_2 cache is exactly equal to the number of requests reaching the main memory.

5.2 Sequential model with two levels of caches

In this section, we compare inclusive, exclusive and non-inclusive properties by measuring the competitive ratio of LRU at the L_2 cache in each of these three cases. In order to establish an upper bound on the competitive ratio of LRU at the L_2 cache, we establish bounds on the effective size of the L_2 cache when it is inclusive, exclusive and non-inclusive of the L_1 cache. We then extend results on the competitive ratio of LRU in the sequential (h, k)-paging framework to the L_2 cache.

Review of LRU in the sequential (h,k)-paging framework: The (h,k)-paging framework was introduced in [22] in order to analyze the performance of online cache replacement algorithms with better resources (i.e., bigger cache) against optimal offline algorithms. The online cache replacement algorithm is given a cache of size k and the offline algorithm is given a cache of size h (with $k \geq h$). It was shown in [22] that the competitive ratio of LRU with respect to an optimal offline algorithm in the (h,k)-paging framework is at most $\frac{k}{k-h+1}$. It was also shown that LRU was optimal in the (h,k)-paging framework. i.e., the competitive ratio of any deterministic cache replacement algorithms in the (h,k)-paging framework was shown to be at least $\frac{k}{k-h+1}$. For completeness, we review proof for these two results in Theorem 5.2.1.

k-phase partition: This terminology was introduced in [22]. A k-phase partition of an input request sequence σ is basically a partition of σ into phases such that each phase contains requests to exactly k distinct blocks. The first phase starts with the first request and ends just before request to the $k+1^{st}$ distinct block in the request sequence. The second phase starts with request to the $k+1^{st}$ distinct block and ends just before request to the $2k+1^{st}$ distinct block in the request sequence.

Theorem 5.2.1. [22] The competitive ratio of LRU is at most $\frac{k}{k-h+1}$ and the competitive ratio of any deterministic cache replacement algorithm is at least $\frac{k}{k-h+1}$ in the (h,k)-paging framework.

Proof. A classic proof for this theorem was presented in [22]. We restate the proof for completeness. Consider the k-phase partition of an arbitrary request sequence σ. LRU incurs at most k cache misses on the current phase (with k distinct block requests). On the other hand, OPT incurs at least $k-h+1$ cache misses on k distinct block requests. Hence the competitive ratio of LRU is at most $\frac{k}{k-h+1}$.

The lower bound on the competitive ratio of a deterministic cache replacement algorithm ALG can be established by constructing an adversarial request sequence consisting of requests to $k+1$ distinct blocks. Without loss

of generality, we assume that the cache of ALG is initially warm and contains $1, 2, 3, \ldots, k$ blocks. The adversarial request sequence begins with a request to $k+1$. Since $k+1$ does not exist in the cache, it results in a cache miss and ALG deterministically evicts a block from the cache. Without loss of generality, let the evicted block be 1. The adversary makes the next request to 1. Request to 1 again incurs a cache miss and the pattern continues. Since ALG is deterministic, the sequence can be deterministically constructed. Note that ALG incurs cache miss on every single request in the request sequence. On the other hand, OPT incurs just $k - h + 1$ cache misses for every k cache misses incurred by ALG by keeping $h - 1$ blocks fixed in its cache. Hence the competitive ratio of any deterministic algorithm in the (h, k)-paging framework is at least $\frac{k}{k-h+1}$.

□

Effective cache size: In order to analyze the competitive ratio of cache replacement algorithms in the hierarchical caching framework, we introduce a new term called the *effective cache size*. The effective size of the L_2 cache for a deterministic cache replacement algorithm ALG is said to be at least k, if ALG incurs at most k cache misses at the L_2 cache on any request sequence consisting of requests to k distinct blocks. Further, the effective size of the L_2 cache for ALG is tight if there exists a request sequence consisting of requests to k distinct blocks on which ALG incurs exactly k cache misses at the L_2 cache. We establish bounds on the effective size of the L_2 cache for marking algorithms in Theorem 5.2.2. Since LRU is an implicit marking algorithm, the bounds hold for it as well. We then, use the results on the competitive ratio of LRU in the (h, k)-paging framework and effective cache size to establish bounds on the competitive ratio of LRU in the sequential model with two levels of caches.

Notation used in sequential model with two levels of caches: We first introduce notation used in the sequential model with two levels caches. We let k_1 and k_2 represent the size of the online algorithm's L_1 and L_2 cache and h_1 and h_2 represent the size of offline algorithm's L_1 and L_2 caches respectively.

Theorem 5.2.2. The effective size of the L_2 cache for any marking algorithm is k_2 when the L_2 cache is either inclusive or non-inclusive of the L_1 cache and is $k_1 + k_2$ when the L_2 cache is exclusive of the L_1 cache.

Proof. We consider request sequences consisting of requests to k_2, k_2 and k_1+k_2 distinct blocks to establish bounds on the effective size of the L_2 cache for a marking algorithm when the L_2 cache is inclusive, non-inclusive and exclusive of the L_1 cache respectively.

Inclusive cache: Consider a sequence consisting of requests to k_2 distinct blocks. Observe that a marking algorithm incurs at most k_2 cache misses on this sequence at an inclusive L_2 cache because a block that is read into the L_2 cache is not evicted till at least k_2 new blocks are read into the cache. Hence the effective size of the L_2 cache for a marking algorithm is at least k_2 when the L_2 cache is inclusive of the L_1 cache.

Non-inclusive cache: Consider a sequence consisting of requests to k_2 distinct blocks. Observe that a few of these requests could get served by the L_1 cache and hence requests to at most k_2 distinct block requests reach the L_2 cache. On the sequence that reaches the L_2 cache, a marking algorithm incurs at most k_2 cache misses. Hence the effective size of the L_2 cache for a marking is at least k_2 when the L_2 cache is non-inclusive of the L_1 cache.

Exclusive cache: Recall that the hardware forces the L_2 cache to be exclusive of the L_1 cache. It can be observed that the L_1 and L_2 caches together hold the k_1+k_2 most recently requested blocks at all the time. Hence these two together work as a bigger cache. Consider a sequence consisting of requests to k_1+k_2 distinct blocks. Observe that a marking algorithm incurs at most $k_1 + k_2$ cache misses during the current phase at the exclusive L_2 cache. Hence the effective size of the L_2 cache for a marking algorithm is at least $k_1 + k_2$ when the L_2 cache is exclusive of the L_1 cache.

One can prove that the effective size of the L_2 cache for any deterministic algorithm is in fact at most k_2, k_2 and k_1+k_2 when the L_2 cache is inclusive, non-inclusive and exclusive of the L_1 cache respectively by consider specific adversarial sequences with requests to k_2, k_2 and $k_1 + k_2$ distinct blocks.

The adversarial sequence is constructed in the exact same manner as described in Theorem 5.2.1.

□

Since LRU is an implicit marking algorithm, these results hold for LRU as well.

Optimal offline algorithm for the sequential model with two levels of caches: We now present an optimal offline algorithm for the sequential model with two levels of caches which is an extension of FITF. We call this algorithm 2-FITF. In order to analyze the performance of inclusive, exclusive and non-inclusive, we keep the optimal offline algorithm independent of the property used at the L_2 cache. 2-FITF is defined as follows:

Upon a cache miss at the L_1 cache, the request is sent to the L_2 cache. If the requested block exists in the L_2 cache, it is read into the L_1 cache. If the requested block does not exist in the L_2 cache, the block is read from the main memory directly into the L_1 cache. A block, say x, which is requested farthest in the future request sequence is evicted from the L_1 cache. The block x is stored in the L_2 cache. The L_2 cache chooses a block, say y, which is requested farthest in the future request sequence. The L_2 cache evicts y if it is requested after x in the future request sequence and evicts x if it is requested after y in the future request sequence.

Theorem 5.2.3. When the cost measure is the total number of cache misses at the L_2 cache, 2-FITF is an optimal in the sequential model with two levels of caches.

Proof. We start with a few notes on the optimal offline algorithm in the sequential caching framework — FITF. FITF has also been referred to as LFD in the literature. LFD stands for longest forward distance. Upon a cache miss, LFD evicts a block from the cache that has the longest forward distance in the future request sequence. Belady [4] established the optimality of LFD.

We compare 2-FITF with two levels of caches (of sizes h_1 and h_2 respectively) with LFD with a single level cache of size $h_1 + h_2$ and prove that

the number of block requests reaching the main memory is same for both of these algorithms. Initially we assume that the set of memory blocks in the two levels of caches for 2-FITF is the same as the set of memory blocks in the single level cache for FITF. In order to establish the optimality of 2-FITF, we show that on every request in an arbitrary request sequence σ, 2-FITF takes the exact same decision as LFD.

In order to show that 2-FITF takes the exact same decision as LFD, it is sufficient if we consider a cache miss and prove that both of these algorithms evict the same block when their cache contents match.

Consider a request that results in a cache miss and let at this point the contents of both 2-FITF's and LFD's cache match. Let z be the block which was evicted from the two level cache structure by 2-FITF. Let x be the block with the longest forward distance in the L_1 cache and y be the block with the longest forward distance in the L_2 cache at the time the cache miss occurred. From the definition of 2-FITF, the forward distance of z is equal to the maximum of forward distances of x and y. Hence z is the block with the longest forward distance in both of these caches considered together. Recall that LFD evicts a block with the maximum forward distance. Hence, LFD also evicts z.

Further, it is easy to see that 2-FITF is not better than FITF because if 2-FITF performed better than FITF, FITF could mimic 2-FITF by splitting the one level cache into two parts — of size h_1 and h_2 respectively.

□

Note that 2-FITF forces the L_2 cache to be exclusive of the L_1 cache in order to get a better effective cache size. 2-FITF basically evicts the block that is requested farthest in the future of the request sequence from the two caches considered collectively. Hence it has an effective cache size of $h_1 + h_2$.

Theorem 5.2.4. The following are tight bounds for the competitive ratio of LRU:

1. $\frac{k_2}{k_2-(h_1+h_2)+1}$ at an inclusive L_2 cache.
2. $\frac{k_2}{k_2-(h_1+h_2)+1}$ at a non-inclusive L_2 cache.

3. $\frac{k_1+k_2}{k_1+k_2-(h_1+h_2)+1}$ at an exclusive L_2 cache.

Proof. Lemma 5.2.2 proves that the effective size of the L_2 cache for any marking algorithm (including LRU) is at least k_2, k_2 and $k_1 + k_2$ when the L_2 cache inclusive, non-inclusive and exclusive of the L_1 cache respectively. This implies that LRU incurs at most k_2, k_2 and $k_1 + k_2$ cache misses at the L_2 cache when the L_2 cache is inclusive, non-inclusive and exclusive of the L_1 cache respectively.

On the other hand, OPT incurs at least $k_2-(h_1+h_2-1), k_2-(h_1+h_2-1)$ and $k_1 + k_2 - (h_1 + h_2 - 1)$ cache misses at the L_2 cache when the sequences with k_2, k_2 and $k_1 + k_2$ distinct block requests are considered. Hence the competitive ratio of LRU in each of these cases is: $\frac{k_2}{k_2-(h_1+h_2)+1}$, $\frac{k_2}{k_2-(h_1+h_2)+1}$ and $\frac{k_1+k_2}{k_1+k_2-(h_1+h_2)+1}$ respectively.

Further, Lemma 5.2.2 also proves that the effective size of the L_2 cache for any deterministic algorithm is at most k_2, k_2 and k_1+k_2 when the L_2 cache inclusive, non-inclusive and exclusive of the L_1 cache respectively. This proves the lower bound on the competitive ratio of the cache replacement algorithms. Hence the bound on the competitive ratio of LRU is tight.

□

5.3 Parallel disjoint memory model with two levels of caches

In this section, we compare inclusive, exclusive and non-inclusive properties by measuring the competitive ratio of RR-PROC-MARK at the L_2 cache in each of these three cases when the processes have full knowledge about their future request sequences. In order to establish an upper bound on the competitive ratio of RR-PROC-MARK at the L_2 cache, we establish a bound on the effective size of L_2 cache when it is inclusive, exclusive and non-inclusive of the L_1 cache. We then establish results on the competitive ratio of LRU in the parallel (h, k)-paging framework and extend it to the L_2 cache.

Relating disjoint memory and multicore caching frameworks: The disjoint memory framework is closely related to the multicore caching framework. We present some observations regarding the relationship between these two frameworks below:

1. Both disjoint memory framework and multicore caching framework model parallel machines.
2. Processes in the disjoint memory framework are analogous to cores in the multicore caching framework.
3. Similar to processes in the disjoint memory framework, cores in the multicore caching framework have full knowledge about their future request sequences in the following two scenarios:
   (a) The cores work on a well defined computation like matrix multiplication or Gaussian elimination paradigm which have known request access pattern.
   (b) The hardware and software prefetchers ([7, 20]) predict the future request access pattern in an online manner.
4. The interleaving of these request sequences, however, cannot be predicted in both of these frameworks and hence remains adversarial.

By relating these two frameworks, we can use the algorithms developed in the disjoint memory framework on multicore caches. In this section, we consider deterministic algorithms developed in the Chapter 3 and analyze their performance in the parallel model with two levels of caches when cores have full knowledge about their future request sequences.

Before considering algorithms in the parallel model with two levels of caches, we introduce the disjoint memory (h, k)-paging framework. In Chapter 3, we considered the disjoint memory (k, k)-paging framework in which both online and offline algorithms gets shared caches of the same size — k). In the disjoint memory (h, k)-paging framework, the online algorithm gets a shared cache of size k and the offline algorithm gets a shared of size h ($k \geq h$). We establish an upper bound on the competitive ratio of RR-PROC-MARK in

the disjoint memory (h, k)-paging framework and then prove a lower bound on the competitive ratio of deterministic process marking algorithms in this framework.

Theorem 5.3.1. The competitive ratio of RR-PROC-MARK in the disjoint memory (h, k)-paging framework is at most $\max(\frac{2k}{k-h+2k/(p+1)}, \frac{2k}{k-h+k/(H_p+2)})$.

Proof. Consider an arbitrary interleaving of the request sequences from all the p cores and call it σ. We split σ into a number of phases (as per the definition of phase in RR-PROC-MARK). Consider a marking phase with l clean block requests and p' participating cores for analysis. From Lemma 3.3.5 and 3.3.8, we have the following inequalities on the number of cache misses incurred by RR-PROC-MARK.

$$cost(\text{RR-PROC-MARK}) \leq \begin{cases} \frac{l}{2}(p'+1) & \text{if } l < p' \\ p' + l + lH_{p'} & \text{if } l \geq p' \end{cases}$$

Clearly, for $l < p' \leq p$,

$$p' + lH_{l-1} \leq \frac{l}{2}(p+1)$$

for $l \geq p'$,

$$\begin{aligned} p' + l + lH_{p'} &\leq l(H_p + 2) \\ cost(\text{RR-PROC-MARK}) &\leq \begin{cases} \frac{l}{2}(p+1) & \text{if } l < p' \\ l(H_p+2) & \text{if } l \geq p' \end{cases} \end{aligned}$$

On the other hand, OPT incurs at least $(k - h + l)/2$ cache miss per marking phase in an amortized sense [28]. Competitive ratio of RR-PROC-MARK is:

$$\frac{cost(\text{RR-PROC-MARK})}{cost(\text{OPT})} \leq \begin{cases} \frac{l(p+1)}{k-h+l} & \text{if } l < p' \\ \frac{2l(H_p+2)}{k-h+l} & \text{if } l \geq p' \end{cases}$$

Since the number of cache misses incurred by any online marking algorithm is at most k per marking phase, we have the following inequalities:

When $l < p'$:

$$\begin{aligned} \frac{l}{2}(p+1) &\leq k \\ l &\leq \frac{2k}{p+1} \end{aligned}$$

When $l \geq p'$:

$$\begin{aligned} l(H_p+2) &\leq k \\ l &\leq k/(H_p+2) \end{aligned}$$

Observe that the above two expressions monotonically increases with l and hence attain maxima for the maximum value of l. Hence,

$$\frac{cost(\text{RR-PROC-MARK})}{cost(\text{OPT})} \leq \begin{cases} \frac{2k}{k-h+2k/(p+1)} & \text{if } l < p' \\ \frac{2k}{k-h+k/(H_p+2)} & \text{if } l \geq p' \end{cases}$$

The upper bound on competitive ratio of RR-PROC-MARK in the disjoint memory (h,k)-paging framework is at most $\max(\frac{2k}{k-h+2k/(p+1)}, \frac{2k}{k-h+k/(H_p+2)})$.

□

Theorem 5.3.2. The competitive ratio of any deterministic process marking cache replacement algorithms in the disjoint memory (h,k)-paging framework is at least $\max(\frac{p+1}{k-h+1}, \frac{k}{k-h+k/H_p})$.

Proof. Let ALG be a deterministic process marking cache replacement algorithm in the disjoint memory (h,k)-paging framework. We establish two bounds on the competitive ratio of ALG.

1. The first bound follows directly from the lower bound established in Theorem A.2 [3] which consists of one clean block request per phase.

2. The second bound is established for a general l.

First bound: In [3], it was shown that the cost of ALG is at least $p+1$ with one clean block request per phase. On the other hand, OPT incurs exactly $k-h+1$ cache misses per marking phase with one clean block request [28] giving us a competitive ratio of:

$$\frac{cost(\text{ALG})}{cost(\text{OPT})} \geq \frac{p+1}{k-h+1}$$

Second bound: In order to establish the second bound on the competitive ratio, we construct an explicit sequence consisting of infinitely many phases. We give construction of one such phase and prove that the cost of ALG on one phases is at least lH_p and cost of OPT is at most $k-h+l$. Hence, the competitive ratio is $\frac{lH_p}{k-h+l}$. This expression attains maxima for the maximum value of l. Since ALG incurs at most k cache misses per marking phase, $lH_p \leq k$.

$$\frac{cost(\text{ALG})}{cost(\text{OPT})} \geq \frac{k}{k-h+k/H_p}$$

What is left is the construction of a phase for which $cost(\text{ALG}) \geq lH_p$ and $cost(\text{OPT}) \leq k-h+l$. W.l.o.g, we assume that the cache of the deterministic online algorithm is initially warm and each of the p cores own exactly k/p blocks in the cache. The phase that we consider consists of a number of stages as defined below:

Stage: The zeroth stage consists of p cores collectively requesting l clean blocks.

At the end of zeroth stage, at least one core will have at least l/p holes associated with it. Without loss of generality, let P_1 be one such core. The first stage consists of request to all the holes associated with P_1 and marking all the other blocks in cache that belong to P_1. Since we are assuming that ALG is a process marking algorithm, none of the blocks in cache belonging to P_1 get evicted in later stages. All these holes are now associated with the other $p-1$ cores (P_1 does not have any holes associated with it).

At the end of first stage, at least one core will have at least $l/(p-1)$ holes associated with it. Without loss of generality, let P_2 be one such core.

The second stage consists of request to all the holes associated with P_2 and marking all the other blocks in cache that belong to P_2 and so on..

In general, at the end of $i-1^{st}$ stage, at least one core will have at least $l/(p-i+1)$ holes associated with it. Without loss of generality, let P_i be one such core. i^{th} stage consists of request to all the holes associated with P_i and marking all the other blocks in cache that belong to P_i. After the i^{th} stage, P_i will not participate in the future stages.

At the end of $p-1^{st}$ stage, exactly one core, P_p, will have all the l holes associated with it. ALG could make sure that all the blocks belonging to P_p get marked before the first hole associated with P_p is requested and thus ending the marking phase. Hence, the number of stages is at least p (including the zeroth stage). Let M_i be the number of cache misses incurred by ALG in i^{th} stage. We have the following expression for the cost of ALG,

$$cost(\text{ALG}) = \sum_{i=0}^{p-1} M_i$$

Note that $M_0 = l$ and $M_i \geq l/(p-i+1)$ (for $1 \leq i \leq p-1$).

$$\begin{aligned} cost(\text{ALG}) &\geq l + \sum_{i=1}^{p-1} l/(p-i+1) \\ &\geq l \sum_{i=1}^{p} 1/i \\ &\geq lH_p \end{aligned}$$

Hence the competitive ratio of any deterministic marking algorithm in the disjoint memory (h,k)-paging framework is at least $\max(\frac{p+1}{k-h+1}, \frac{k}{k-h+k/H_p})$.

□

Corollary 5.3.3. The competitive ratio of RR-PROC-MARK is constant when the size of the optimal offline algorithm's cache is less than a constant factor of the size of the online cache replacement algorithm.

Proof. This is a corollary of Theorem 5.3.1. We shall prove that the competitive ratio of RR-PROC-MARK is constant when $h \leq c \cdot k$ for some fixed constant $c < 1$.

Theorem 5.3.1 proves that the competitive ratio of RR-PROC-MARK in the disjoint memory (h, k)-paging framework is:

$$R_u \leq \max(\frac{2k}{k - h + 2k/(p+1)}, \frac{2k}{k - h + k/(H_p + 2)})$$

When $h \leq c \cdot k$:

$$\begin{aligned} R_u &\leq \max(\frac{2(p+1)}{(p+1)(1-c)+2}, \frac{2(H_p+2)}{(H_p+2)(1-c)+1}) \\ &= \mathcal{O}(1) \end{aligned}$$

Hence, the competitive ratio of RR-PROC-MARK is a constant when the size of the optimal offline algorithm's cache is less than a constant factor of the size of the online cache replacement algorithm.

Further, Theorem 5.3.2 proves that the competitive ratio of any deterministic algorithm in the disjoint memory (h, k)-paging framework is:

$$R_l \geq \max(\frac{p+1}{k - h + 1}, \frac{k}{k - h + k/H_p})$$

When $h \leq c \cdot k$:

$$\begin{aligned} R_l &\geq \max(\frac{p+1}{k(1-c)+1}, \frac{H_p}{H_p(1-c)+1}) \\ &= \mathcal{O}(1) \end{aligned}$$

Hence, the competitive ratio of RR-PROC-MARK is optimal up to a constant factor when $h \leq c \cdot k$.

Also note that when $h = k$, the competitive ratio of RR-PROC-MARK is at most $p + 1$ and that of a deterministic process marking algorithm is at least $p + 1$.

□

Theorem 5.3.4. The competitive ratio of RAND-PROC-MARK in the disjoint memory (h, k)-paging framework is at most $\frac{2k}{k-h+k/(H_{p-1}+1)}$.

Proof. An upper bound of $2H_{p-1}+2$ was established on the competitive ratio of RAND-PROC-MARK in [3]. We aim to establish a bound on the competitive ratio of RAND-PROC-MARK in the disjoint memory (h, k)-paging framework. Consider an arbitrary interleaving of the request sequences from all the p cores and call it σ. We split σ into a number of phases (as per the definition of phase in RAND-PROC-MARK). Consider a marking phase with l clean block requests and p' participating cores for analysis. From [3], we have the following bound on the number of cache misses incurred by RAND-PROC-MARK.

$$cost(\text{RAND-PROC-MARK}) \leq l(H_{p-1}+1)$$

On the other hand, OPT incurs at least $(k-h+l)/2$ cache miss per marking phase in an amortized sense [28]. Hence, the competitive ratio of RAND-PROC-MARK is:

$$\frac{cost(\text{RAND-PROC-MARK})}{cost(\text{OPT})} \leq \frac{2l(H_{p-1}+1)}{k-h+l}$$

Since the number of cache misses incurred by any online marking algorithm is at most k per marking phase, we have the following inequalities,

$$\begin{aligned} l(H_{p-1}+1) &\leq k \\ l &\leq k/(H_{p-1}+1) \end{aligned}$$

Observe that the above expression monotonically increases with l and hence attain maxima for the maximum value of l. Hence,

$$\frac{cost(\text{RAND-PROC-MARK})}{cost(\text{OPT})} \leq \frac{2k}{k-h+k/(H_{p-1}+1)}$$

The upper bound on competitive ratio of RAND-PROC-MARK in the disjoint memory (h, k)-paging framework is at most $\frac{2k}{k-h+k/(H_{p-1}+1)}$.

□

In the remaining part of this section, we establish a bound on the effective size of the L_2 cache for deterministic process marking algorithms in the disjoint memory framework.

Notation used in the parallel model with two levels of caches: We first introduce notation used in the parallel model with two levels caches. We let k_1 and k_2 represent the size of the online algorithm's L_1 and L_2 cache and h_1 and h_2 represent the size of offline algorithm's L_1 and L_2 caches respectively.

Effective size of the L_2 cache in the parallel model with two levels of caches: Recall that all process marking algorithms are in fact marking algorithms. Hence the effective size of the L_2 cache derived for marking algorithms in the previous section holds for all the process marking algorithms as well. Hence the effective size of the L_2 cache for RR-PROC-MARK is k_2 when the L_2 cache is either inclusive or non-inclusive of the L_1 cache and is $k_1 + k_2$ when the L_2 cache is exclusive of the L_1 cache. These effective cache sizes hold for RAND-PROC-MARK when a process marking version of the algorithm is used.

5.3.1 Extending to multiple levels of caches

Our results for LRU and RR-PROC-MARK extend from two levels of caches to multiple levels of caches.

We let the ith levels of cache be represented by L_i and let r be the number of levels of shared caches. The size of the L_i cache is represented by k_i for the online algorithm and h_i for the offline algorithm.

The L_i cache is said to be an inclusive cache if the contents of the L_i cache includes the contents all the lower level caches $(L_1, L_2, .. L_{i-1})$ and each of the lower level cache is inclusive as well. Similarly, the L_i cache is said to be an exclusive cache if the contents of the L_i cache are exclusive of all the lower level caches and each of the lower level cache is exclusive as well. The L_i cache is non-inclusive if neither of these constraints are enforced.

The effective size of the L_i cache can be obtained by extended the proof for effective size of the L_2 using induction. The key observation is that a hierarchy of shared caches can be replaced by single shared cache of size equal to the effective cache size of the top most cache. Using the effective size of the L_i cache, we can obtain an upper bound on the competitive ratio of LRU and RR-PROC-MARK.

Also, note that, the optimal offline algorithm with i levels of shared caches is an extension of 2-FITF called I-FITF. The optimal offline algorithm with i levels of shared caches tries to retain the blocks which will be requested in the recent future at the lowest level caches. Using induction, one can easily establish that I-FITF is in fact optimal in this model.

In conclusion, the effective size of the top level cache (L_r) when the cache structure is inclusive or non-inclusive is at most k_r for any deterministic marking algorithm. Further, the effective size of the L_r cache when the cache structure is exclusive is at most $\sum_{i=1} rk_r$.

5.4 Case study: Cache architectures in Intel Nehalem and AMD Shanghai

An in-depth comparison of *cache coherent non uniform memory access* (ccNUMA) multiprocessor systems with AMD (Shanghai) and Intel (Nehalem-EP) quad-core x86-64 processors is presented by Hackenberg et al. [14]. An overview of the cache structure in these processors is presented below.

Intel 2x Intel Xeon X5570 (Nehalem-EP): Intel's Nehalem processors consist of 4 cores connected by point-to-point interconnects. Each of these cores have an on-chip private L_1 and L_2 caches. All these cores share a common L_3 cache. Each of the L_1 caches are 32 KB and L_2 caches are 256 KB. L_3 cache is 8 MB in size and is inclusive of both L_1 and L_2 caches. Further, the L_2 cache is neither inclusive not exclusive of the L_1 cache. Nehalem processors use MESIF protocol in order to maintain cache coherency.

AMD 2x AMD Opteron 2384 (Shanghai): AMD's Shanghai processors consist of 4 cores connected by point-to-point interconnects. Each of these cores have an on-chip private L_1 and L_2 caches. All these cores share a common L_3 cache. Each of the L_1 caches are 64 KB and L_2 caches are 512 KB. L_3 cache is 6 MB in size and is non-inclusive of both L_1 and L_2 caches. Further, the L_2 cache is exclusive of the L_1 cache. Shanghai processors use MOESI protocol to maintain cache coherency.

In Intel Pentium M processors, inclusion was not enforced in the L_2 cache since there was not enough difference in the capacities of the L_1 and L_2 caches. It can be observed that in case of Nehalem architecture, the size of L_3 cache is nearly 8 times that of all the lower level caches combined. L_3 is an off-chip cache and hence inclusion does not consume a lot of useful on-chip cache memory. Advantages of enforcing non-inclusion over inclusion in multicore processors with hierarchical caches are discussed by Baer and Wang [2] and Zahran et al. [30].

Discussion: Our results can be mainly seen as a theoretical backing for the exclusive caches. We prove that the competitive ratio of marking algorithms like LRU is less for an exclusive L_2 cache when compared to an inclusive and non-inclusive L_2 cache. The competitive ratio of LRU was shown to be $\frac{k_1+k_2}{k_1+k_2-(h_1+h_2)+1}$ at an exclusive L_2 cache and $\frac{k_1}{k_1-(h_1+h_2)+1}$ at inclusive and non-inclusive L_2 caches. Observe that $k_2 \geq h_1 + h_2$ and hence $\frac{k_1+k_2}{k_1+k_2-(h_1+h_2)+1} \leq \frac{k_1}{k_1-(h_1+h_2)+1}$. Our results validate the experimental findings regarding the superiority of exclusive caches [2, 30].

Our results in the parallel disjoint memory (h, k)-paging framework hold directly for the shared caches in Intel Nehalem and AMD Shanghai processors. In this case we assume that each process has full knowledge about its individual request sequence. The effective cache size the L_2 cache was used to establish bounds on the competitive ratio of RR-PROC-MARK in this case.

Even though exclusive caches perform better than inclusive and non-inclusive caches in terms of competitive ratio, the cost involved in maintaining the exclusivity is usually much higher. Further, in snooping protocol based caches, the inclusion property makes lower level caches much more effective by taking care of the cache coherence at higher levels.

Table 5.1: Architecture details: Intel-Nehalem and AMD-Shanghai processors

Feature	Intel-Nehalem	AMD-Shanghai	Source
Number of cores	4	4	[14]
Cache Line size	64 Bytes	64 Bytes	[14]
L_1 cache (**private**)	32 KB	64 KB	[14]
L_2 cache (**private**)	256 KB **non-inclusive** of L_1	512 KB **exclusive** of L_1	[14]
L_3 cache (**shared**)	8 MB **inclusive** of L_1 and L_2	6 MB **non-inclusive** of L_1 and L_2	[14]
Cache coherency protocol	MESIF	MOESI	[14]
Does cache line read into L_1 get stored in L_2?	Yes. The line is first read into L_2 and it is stored there.	No. The line is directly read into L_1.	[11] & [1]
Does cache line read into L_1 get stored in L_3?	Yes. The line is first read into L_3 and it is stored there.	No. The line is directly read into L_1.	[11] & [1]
Does the evicted cache line from L_1 get stored in L_2?	No.	Yes. Shanghai uses "victim-caching". Evicted line from the L_1 cache is stored in the L_2 cache.	[11] & [1]
Does the evicted cache line from L_2 get stored in L_3?	No.	Depends. If the evicted line is being used by others cores, yes. Else, no.	[11] & [1]
What happens when a modified cache line is requested by another core?	The cache line is written back to the memory and is marked as shared.	The cache line is shared without being written back to the memory.	[11] & [1]
What happens when a modified cache line is evicted from L_1?	The cache line is written back to the memory directly.	The cache line is written back L_3 and then to the memory.	[11] & [1]

Chapter 6

Conclusion and further research

In this thesis, we analyzed cache replacement algorithms at a shared cache in the multicore setting using the classical competitive analysis. In our model, we assume that each process has full knowledge about its individual future request sequence. We also assume that the interleaving of requests from these processes that reaches the shared cache is assumed to be adversarial for the competitive analysis. In Chapter 4, we established tight bounds on the competitive ratio of deterministic and randomized cache replacement algorithms when processes share memory blocks. We also presented a deterministic global algorithm called GLOBAL-MAXIMA which is optimal up to a constant factor in this framework. The case when processes access disjoint sets of memory blocks was considered in [8] and [3]. In Chapter 3, we presented a deterministic algorithm called RR-PROC-MARK which exactly matched the lower bound on the competitive ratio of deterministic algorithm when processes access disjoint sets of memory blocks. In Chapter 5, we analyzed the principle of inclusion at a shared L_2 cache by computing the competitive ratio of well known cache replacement algorithms. We proved that an exclusive cache is better than both inclusive and non-inclusive caches; this validates the experimental findings in the literature.

The algorithm that we proposed in the disjoint memory framework (RR-PROC-MARK) is simple and computationally efficient. It is also a fair algorithm since the same number of blocks belonging to each of the participating process is evicted from the cache during every phase. On the other hand, GLOBAL-MAXIMA might be unfair on a few adversarial request sequences. On a few request sequences, GLOBAL-MAXIMA evicts blocks requested by some of the processes more often than the blocks requested by the other processes. A topic for further research is to develop a fair strategy (along the lines of

RR-PROC-MARK) which still remains optimal in the shared memory framework. Other future research problems include the following two problems in the shared memory framework: closing the gap between the lower bound of $\frac{p}{2}\log\frac{4(k+1)}{3p}$ and the upper bound of $2(p\ln\frac{ek}{p}+1)$ on the competitive ratio of deterministic algorithms and improving the computational complexity of global cache replacement algorithms in the shared memory framework. There is also a gap between the lower and upper bounds on the competitive ratio of randomized algorithms in both shared and disjoint memory frameworks which needs to be closed.

On the hierarchical caching front, one can consider extending our results to a more general hierarchy containing caches in a tree structure. With increase in number of cores in multicore systems, a natural cache hierarchy consists of private L_1 caches and shared higher level caches where the extent of sharing increases with the increase in the level of the cache.

The shared memory framework could motivate research on multi-pointer *directed* access graphs. The concept of directed access graphs was proposed in [6] to model the notion of locality of reference in a sequence of instructions reaching the instruction cache. A topic of further research is to see if we can apply the concepts developed here to the multi-pointer directed access graph framework.

Bibliography

[1] AMD. http://blogs.amd.com/developer/2008/11/13/larger-l3-cache-in-shanghai-part-i. 2009.

[2] Jean-Loup Baer and Wen-Hann Wang. Retrospective: On the inclusion properties for multi-level cache hierarchies. *International Symposium on Computer Architecture 1998*, pages 59–60, 1998.

[3] R. D. Barve, E. F. Grove, and J. S. Vitter. Application-controlled paging for a shared cache. *SIAM Journal on Computing*, 29(4), 1995.

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

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

[6] 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.

[7] D. Callahan, K. Kennedy, and A. Porterfield. Software prefetching. *Proceedings of ASPLOS IV*, pages 40–52, 1991.

[8] P. Cao, E. W Felten, and K. Lee. Application-controlled file caching policies. *Proceedings of Summer USENIX Conference, Boston, MA*, pages 171–182, 1994.

[9] R.A. Chowdhury and V. Ramachandran. The cache-oblivious gaussian elimination paradigm: Theoretical framework, parallelization and experimental evaluation. *Theory of Computing Systems*, 47(1):878 – 919, 2010.

[10] R.A. Chowdhury, F. Silvestri, B. Blakeley, and V. Ramachandran. Oblivious algorithms for multicores and network of processors. *Proceedings of IEEE IPDPS*, 2010.

[11] Intel Corporation. White paper intel xeon processor 3500 and 5500 series intel microarchitecture. 2009.

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

[13] A. Fiat, R. Karp, M. Luby, L. McGeoch, D. Sleator, and N. Young. On competitive algorithms for paging problems. *Journal of Algorithms*, 12:685–699, 1991.

[14] D. Hackenberg, D Molka, and W. E. Nagel. Comparing cache architectures and coherency protocols on x86-64 multicore smp systems. *Proceedings of the 42nd Annual IEEE/ACM International Symposium on Microarchitecture*, pages 413–422, 2009.

[15] A. Hassidim. Cache replacement policies for multicore processors. *Innovations in Computer Science (ICS)*, 2010.

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

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

[18] A. Lopez-Ortiz and A. Salinger. Paging for multicore (cmp) caches. *Technical Report*, 2010.

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

[20] T. C. Mowry, M. S. Lam, and A. Gupta. Design and evaluation of a compiler algorithm for prefetching. *Proceedings of ASPLOS V*, pages 62–73, 1992.

[21] P. Raghavan and M. Snir. Memory versus randomization in on-line algorithms. *Journal or Computer and System Sciences*, 7:79–83, 1992.

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

[23] I. Tartalja and V. Milutinovic. The cache coherence problem in shared memory multiprocessors: Software solutions. *IEEE Comp. Soc. Press, Los Alamitos, Calif.*, 1996.

[24] R. E. Tipley. Method and apparatus for achieving multilevel inclusion in multilevel cache hierarchies. *Patent Number: 5, 369,753*, 1994.

[25] M. Tomasevic and V. Milutinovic. The cache coherence problem in shared memory multiprocessors: Hardware solutions. *IEEE Comp. Soc. Press, Los Alamitos, Calif.*, 1993.

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

[27] A. C-C. Yao. Probabilistic computations: Towards a unified measure of complexity. *17th Annual Symposium on Foundations of Computer Science*, pages 222–227, 1977.

[28] Neal Young. Competitive paging and dual-guided on-line weighted caching and matching algorithms. *Dissertation, Princeton University*, 1991.

[29] Neal Young. Competitive paging as cache size varies. *Proceedings of Second ACM-SIAM Symposium on Discrete Algorithms*, 1991.

[30] M. Zahran, K. Albayraktaroglu, and M. Franklin. Non-inclusion property in multi-level caches revisited. *International Journal of Computers and Their Applications 2007*, June 2007.

Vita

Anil Kumar Katti, the eldest son of Ajit Kumar Katti and Damayanti Ajit Katti joined The University of Texas at Austin after obtaining his Bachelor of Engineering in Computer Science from National Institute of Technology, Surathkal, Karnataka, India. He pursued Master of Science in Computer Science in the Department of Computer Science at The University of Texas at Austin. He was born and brought up in Bangalore which was then a very pleasant and beautiful city in India!

Permanent address: #4, Srinivasanilaya, Sharadanagar
Chunchgatta Road, Bangalore 560062
Karnataka, India.

This thesis was typeset with LaTeX[†] by the author.

[†]LaTeX is a document preparation system developed by Leslie Lamport as a special version of Donald Knuth's TeX Program.