Caching Integrated with Pipelined Prefetching

Alexander Gaysinsky
Caching Integrated with Pipelined Prefetching

Research thesis
submitted in partial fulfillment of
the requirements for the degree of
Master of Science in Computer Science

Alexander Gaysinsky

Submitted to the Senate of the Technion—Israel Institute of Technology
Cheshvan 5761 Haifa November 2000
The research thesis was done under the supervision of Doctor Hadas Shachnai in the Department of Computer Science.

I wish to express my deepest gratitude and appreciation to Dr. Hadas Shachnai for her excellent guidance which made this work possible.
# Contents

<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>Abstract</td>
<td>1</td>
</tr>
<tr>
<td>List of Symbols</td>
<td>2</td>
</tr>
<tr>
<td>1 Introduction</td>
<td>4</td>
</tr>
<tr>
<td>1.1 Problem Statement</td>
<td>4</td>
</tr>
<tr>
<td>1.2 Applications</td>
<td>7</td>
</tr>
<tr>
<td>1.2.1 Pipelined Main Memory</td>
<td>7</td>
</tr>
<tr>
<td>1.2.2 Program Execution on Fast Processors</td>
<td>8</td>
</tr>
<tr>
<td>1.2.3 Switch-based High-Speed LANs</td>
<td>9</td>
</tr>
<tr>
<td>1.3 Related Work</td>
<td>10</td>
</tr>
<tr>
<td>1.4 Main Results</td>
<td>11</td>
</tr>
<tr>
<td>1.5 Organization of this Work</td>
<td>11</td>
</tr>
<tr>
<td>2 Preliminaries</td>
<td>12</td>
</tr>
<tr>
<td>2.1 Definitions and Notation</td>
<td>12</td>
</tr>
<tr>
<td>2.2 Known Algorithms</td>
<td>14</td>
</tr>
<tr>
<td>2.3 Variants of the Algorithms EE and DEE</td>
<td>16</td>
</tr>
<tr>
<td>2.4 Variants of the CLPP</td>
<td>16</td>
</tr>
<tr>
<td>3 The Offline CLPP Problem</td>
<td>18</td>
</tr>
<tr>
<td>3.1 No Harm Rules</td>
<td>18</td>
</tr>
<tr>
<td>3.2 Optimality of Aggressive</td>
<td>20</td>
</tr>
<tr>
<td>4 The Online CLPP Problem</td>
<td>23</td>
</tr>
<tr>
<td>4.1 Complete Graphs</td>
<td>23</td>
</tr>
<tr>
<td>4.2 Directed Acyclic Access Graphs</td>
<td>24</td>
</tr>
<tr>
<td>4.3 Branch Trees</td>
<td>26</td>
</tr>
<tr>
<td>5 The Markovian CLPP on Branch Trees</td>
<td>28</td>
</tr>
<tr>
<td>5.1 Performance of DEE on Branch Trees</td>
<td>28</td>
</tr>
<tr>
<td>5.2 Homogeneous Branch Trees</td>
<td>29</td>
</tr>
<tr>
<td>Section</td>
<td>Page</td>
</tr>
<tr>
<td>----------------------------------------------</td>
<td>------</td>
</tr>
<tr>
<td>6 Branch Trees with Bounded Number of Leaves</td>
<td>34</td>
</tr>
<tr>
<td>6.1 A Dynamic Programming Algorithm</td>
<td>34</td>
</tr>
<tr>
<td>6.2 The DEE(L) Algorithm</td>
<td>35</td>
</tr>
<tr>
<td>7 Summary</td>
<td>39</td>
</tr>
<tr>
<td>7.1 The Contribution of this Work</td>
<td>39</td>
</tr>
<tr>
<td>7.2 Open Problems</td>
<td>39</td>
</tr>
<tr>
<td>A Bounds for the $q$-distance Fibonacci Numbers</td>
<td>41</td>
</tr>
<tr>
<td>Bibliography</td>
<td>44</td>
</tr>
</tbody>
</table>
List of Figures

1.1 An example of caching for $k = 3$ and $d = 2$ .......................... 5
1.2 An example of caching with pipelined prefetching, for $k = 3$ and $d = 2$ 5
1.3 A reference sequence on a branch tree ................................. 6
1.4 An example of the execution of online algorithm, $k = 3$ and $d = 2$ ... 6

2.1 An example of EE execution .............................................. 14
2.2 An example of DEE execution .......................... ................. 15

3.1 The cache contents of $A$, $A'$ after step $(i-1)$ in the proof of Theorem 3.1, $(i)$ ......................................................... 19

6.1 Case (i) in the proof of Theorem 6.3 ................................. 37
6.2 Case (ii) in the proof of Theorem 6.3 ................................. 37
Abstract

Caching and prefetching have been studied extensively in the past decades; however, the interaction between the two was not well understood until the work of Cao et al. (1995), who proposed to integrate caching with prefetching. They introduced the following execution model. Suppose that a program makes a sequence of $m$ accesses to data blocks. The cache can hold $k$ blocks, where $k < m$. An access to a block in the cache incurs one time unit, and fetching a missing block incurs $d$ time units. At any time while serving a request to read/write a block, the system can prefetch some block to the cache for future usage. The Caching with Prefetching (CP) problem is to determine the sequence of block evictions/prefetches, so as to minimize the time required for accessing all blocks.

In this work we consider a generalized version of the CP problem, in which requests for blocks exhibit locality of reference, i.e., any sequence of block references is a walk on an access graph $G$; in addition, we allow pipelined prefetching of data blocks, assuming that several block fetches can be held in parallel. As before, our measure is the overall execution time of a given reference sequence. We call this problem Caching with Locality and Pipelined Prefetching (CLPP). Our study is motivated from program execution on the Pentium, and other fast processors, and the pipelined operation of modern main memory controllers (such as RDRAM).

For the offline case, we show that an algorithm introduced by Cao et al. (1995) is optimal for our problem. In the online case we focus on the class of directed acyclic access graphs (DAG), and on the subclass of branch trees, which often arise in our applications. When $G$ is a DAG, we give an algorithm that is optimal within factor $\min(1 + \frac{k}{d}, 2)$ in the set of online deterministic algorithms. This algorithm is within factor $1 + o(1)$ from any optimal algorithm on branch trees.

Finally, we consider the problem of caching with prefetching under the assumption that the sequence of accessed blocks is generated by a Markov chain, and the access graph is a branch tree. This includes the study of the CLPP(L), in which the set of blocks stored in the cache at any time is a tree with at most $L$ leaves, for some $1 \leq L \leq k$. We give algorithms that achieve a factor of 2 to the optimal online.
List of Symbols

$G = (V, E)$ ........ a graph $G$ with a set of vertices $V$ and edges $E$
$(u, v)$ ........... an edge from node $u$ to node $v$
$p_a(v)$ ........... accumulated probability of the node $v$
$Path(G)$ ........... the set of directed paths in graph $G$
DAG ............... directed acyclic graph
A_PS ............... adaptive path selection
CP ................. caching with prefetching
CLPP ............... caching with locality and pipelined prefetching
S_CLPP ............ single phase caching with locality and pipelined prefetching
CLPP(L) ............ caching with locality and pipelined prefetching, with bounded number of leaves
S_CLPP(L) ........... single phase caching with locality and pipelined prefetching, and bounded number of leaves
AGG ............... the aggressive algorithm
EE ................. the eager execution algorithm
DEE ............... the disjoint eager execution algorithm
DEE(L) ............. disjoint eager execution with bounded number of leaves
DP ................. the dynamic programming algorithm
LRU ............... the least recently used algorithm
FIFO ............... the first-in-first-out algorithm
OPT ............... the optimal offline algorithm
$\sigma$ ............. a sequence of block references
$Pr(\sigma)$ .......... the probability of a reference sequence $\sigma$
$|\sigma|$ ............ the length of a reference sequence $\sigma$
$C(\sigma)$ .......... the execution cost of a sequence $\sigma$
$A(\sigma)$ .......... the cost incurred by an algorithm $A$ on a reference sequence $\sigma$
$OPT(\sigma)$ ........ the cost incurred by an optimal offline algorithm on a reference sequence $\sigma$
\( c_{A,k,d}(G) \) ........ the competitive ratio of an algorithm \( A \) on a graph \( G \), when the cache size is \( k \) and the delivery time is \( d \)

\( c_{k,d}(G) \) ........ the competitive ratio of an optimal online algorithm on a graph \( G \), for fixed \( k,d \)

\( c_k(G) \) ........... the competitive ratio of an optimal online algorithm on a graph \( G \), for the arbitrary \( d \) and a fixed \( k \)

\( c_{k,d}^D(G) \) ........ the competitive ratio of an optimal deterministic online algorithm on a graph \( G \), for fixed \( k,d \)

\( c_k^D(G) \) ........... the competitive ratio of an optimal deterministic online algorithm on a graph \( G \), for the arbitrary \( d \) and fixed \( k \)

\( \| x \| \) ............. the module of a complex number \( x \)

\( O \) ................... upper bound

\( \Omega \) ................... lower bound

\( T \) ................... a graph \( G = (V,E) \) which is a tree

\( T' \) ................... a sub-tree of a tree \( T \)

\( \text{height}(T) \) ...... the maximal distance from the root of \( T \) to a leaf

\( l(T) \) ................... the set of leaves of a tree \( T \)

\( \text{child}(T) \) ........ the set of direct children of the leaves of a tree \( T \)

\( g(n) \) ............... \( n \)th element in the \( g \)-distance Fibonacci sequence

\( f(n) \) ............... the number of nodes in a homogeneous tree \( T \) with accumulated probability \( p_n = p^a \)
Chapter 1

Introduction

1.1 Problem Statement

Caching and prefetching were intensively studied for several decades as two separate problems (see, e.g., Borodin et al. [2], Irani et al. [7], Krishnan and Vitter [12]). Cao et al. proposed in [4] to integrate caching with prefetching, and introduced the following execution model. Suppose that we have a fast memory of size $k$, to which we need to fetch file blocks (or program pages) from secondary storage, whose size is $n$, $n > k \geq 1$. The delivery time of a block from secondary storage is $d$ time units. The access time to a block in the cache is one time unit. At any time while servicing the requests to read/write blocks, the system can prefetch some block to the cache, for future usage. The Caching with Prefetching (CP) problem is to determine the sequence of block evictions/prefetches, so as to minimize the time required for accessing all blocks.

In this work we consider a generalized version of the CP problem, in which requests for blocks exhibit locality of reference, i.e., any sequence of block references is a walk on an access graph $G$; in addition, we allow pipelined prefetching of data blocks, assuming that several block fetches can be held in parallel. As before, our measure is the overall execution time of a given reference sequence.

Formally, we have a set of $n$ blocks $b_1, b_2, \ldots, b_n$ held in secondary storage. The access graph for $b_1, b_2, \ldots, b_n$, $G = (V, E)$, is a graph in which each node corresponds to a block in the set. The sequence of block references has to obey the locality constraints imposed by the edges of $G$: following a request to a block (node) $u$, the next request has to be to $u$ or to a node $v$, such that $(u, v) \in E$.

The usage of pipelined prefetching implies that at any time $t$, we can start prefetching a single block to some entry in the cache. The cost of servicing the $i$th request, $r_i$, is $1 + d(r_i)$, where $0 \leq d(r_i) \leq d$ is the remaining delivery time of $r_i$. The total cost for the reference sequence $\sigma = \{r_1, \ldots, r_t\}$ is given by $C(\sigma) = |\sigma| + \sum_{i=1}^{t} d(r_i)$.

The problem of Caching with Locality and Pipelined Prefetching (CLPP) can
be stated as follows:

Given a cache of size \( k \geq 1 \), a delivery time \( d \geq 1 \), and a reference sequence \( \sigma = \{r_1, \ldots, r_t\} \), find a policy for prefetching and caching that minimizes \( C(\sigma) \).

![Cache Example](image1.png)

Figure 1.1: An example of caching for \( k = 3 \) and \( d = 2 \)

![Cache Example](image2.png)

Figure 1.2: An example of caching with pipelined prefetching, for \( k = 3 \) and \( d = 2 \)

**Example 1.1** Consider a program whose block accesses are given by the sequence “DEBCA”. The cache can hold three blocks, and initially \( A, B \) and \( C \) are in the cache; fetching a block takes two time units. Figure 1.1 shows the execution of the optimal caching-by-demand algorithm (see, e.g., in [2]), that takes 11 time units\(^1\).

Figure 1.2 shows an execution of the Aggressive (AGG) algorithm (see in Section 2.2), that combines caching with pipelined prefetching; thus, the reference sequence is completed within 7 time units.

\(^1\)Note, that if a block \( b_i \) is replaced in order to bring the block \( b_j \), then \( b_i \) becomes unavailable for access when the fetch is initiated; \( b_j \) can be accessed when the fetch terminates, i.e., after \( d \) time units.
This suggests that pipelined prefetching can be helpful. The challenge of a
good algorithm is to achieve a maximum overlap between accesses to blocks in
the cache and fetches.

We focus on the class of directed acyclic access graphs (DAG), and in particular
on the subclass of branch trees (see in Section 2.1), which often arise in our
applications.

We study the CLPP problem both in the offline case, where the whole refer-
ence sequence, $\sigma$, is known in advance, and the online case, where $\sigma$ is revealed
along the execution of the algorithm. In the next example we show the operation
of an online algorithm for the CLPP (in which the access graph is a branch tree).

![Reference sequence on a branch tree](image)

**Figure 1.3:** A reference sequence on a branch tree

![Execution example](image)

**Figure 1.4:** An example of the execution of online algorithm, $k = 3$ and $d = 2$

**Example 1.2** Consider the access graph in Figure 1.3; suppose that the unknown
sequence of block accesses is “ABEF”. The cache can hold three blocks; fetching a
block takes two time units. Initially the cache is empty. Figure 1.4 shows the on-
line operation of the algorithm Eager Execution (EE). This algorithm maintains
a subtree of the original tree of the program in the cache. The circled vertices in
Figure 1.3 show the subtree fetched by EE.
In each step (one time unit) EE initiates a fetch of a single block to the cache. The contents of the cache are shown in Figure 1.4 for each step. The stage indicates the remaining delivery time for each block \( b_i \), i.e., \( d(b_i) \). A block that is in stage 0 is accessed and evicted from the cache. Thus, for example, in step 2 \( d(B) = 2 \), while \( d(A) = 1 \).

Note that EE fetches to the cache both \( B \) and \( C \). In step 3 \( A \) is accessed and the next requested block, \( B \), becomes known. Therefore, in step 4, the fetch of block \( C \) is aborted, and EE starts to fetch the block \( D \). At this step \( B \) is accessed and the next requested block, \( E \), becomes known. Thus, in step 5 the fetch of \( D \) is aborted and EE initiates the fetch of \( E \); in step 6 EE starts to fetch \( F \) and so on.

We also consider the problem of caching with prefetching under the assumption that the sequence of blocks accessed is generated by a Markov chain, and the access graph is a branch tree. This includes the study of the CLPP(L), in which the set of blocks stored in the cache at any time is a tree, with at most \( L \) leaves, for some \( 1 \leq L \leq k \). The Markovian model provides a theoretical abstraction for locality of references in programs: each reference sequence is associated with probability, which depends on the probabilities of the individual requests along the sequence, and the “transition probabilities”, much as in real programs. In fact, certain simple properties of real programs (e.g., the fact that data-dependent loop is typically executed many times before exiting) can be modeled well.

## 1.2 Applications

### 1.2.1 Pipelined Main Memory

Since the 1970’s, the frequency of the PC processor doubles every two years, according to Moore’s law (see, e.g., Hennesy and Patterson [5]). This rapid rate has come both from advances in the technology used to build computers, and from innovation in computer design. At the same time, memory speedup is far from such improvement trends. The traditional approach to overcome the mismatch between memory and CPU performance is to use a hierarchical memory system, consisting of several cache levels. Caching helps to reduce the average access time to memory due to locality of reference, but they cannot solve the bandwidth problems occurring during cache-fill operations.

The pipeline technique is already used in the existing memory system, \( SDRAM \). Unfortunately, parallel accesses to memory addresses can be done only in a small address window. Recent modifications of the \( SDRAM \) system, proposed by Mathew et al. [14], help to increase parallelism. The paper proposes a new memory controller, \( PVA \), that reduces the effect of the long \( SDRAM \) latency, i.e., the time that elapses until the CPU receives from memory the requested data. \( PVA \) uses two approaches, namely, overlapping address accesses and reordering
of the command sequence for improving locality of accesses, so addresses fall in the memory window, where accesses can be done in parallel. Thus, system bandwidth increases significantly.

The new memory system, \textit{RDRAM}, is designed by Rambus [17] to address application needs for additional bandwidth. Except for the small gap of 1 to 4 clock cycles required for switching between Read and Write modes, all commands are fully parallelized. With appropriate communication policy, RDRAM performs about 95\% of the commands in parallel.

An essential difference between RDRAM and the SDRAM is in the extended and close to entire ability of the RDRAM to accept new data requests, while supplying previously referenced data. The number of requests that can be handled in parallel is called the \textit{throughput}. Suppose, that the CPU cache size is \( k \) and the RDRAM throughput is \( d \), then the problem of finding an optimal memory-CPU protocol yields an instance of the CLPP.

1.2.2 Program Execution on Fast Processors

Modern processors use pipelining for speeding up the execution of programs. The pipeline is composed of a set of segments, each corresponding to a step in a single instruction; an instruction that is fetched into the pipe is processed in stages and completes its execution once its last step was terminated successfully. The throughput of the processor is the rate at which instructions complete their execution successfully and leave the pipe. The speedup factor is the ratio between the throughput of a pipeline processor and the throughput of a processor that does not use pipelining. Ideally, this ratio equals to the length of the pipeline. In practice, the speedup achieved by using a pipeline is much smaller.

Indeed, when the program starts executing a branch, the next instruction that will be executed is known only when that branch is resolved. However, delaying the pipeline until the branch execution is completed is undesirable. Therefore, mechanisms for \textit{speculative execution} of code are used to forecast the instruction path that will be taken by the program after each branch.

Given the set of possible execution paths of the program, our objective is to choose after each execution step the next instruction, that will be fetched into the pipe, such that the overall execution time of the program is minimized. More formally, we represent the set of the execution paths of the program as a binary tree, that we call the \textit{branch tree} of the program: each internal node represents an instance of a conditional branch; an edge represents a basic block in the program, i.e., a sequence of non-branch instructions between two successive branches. The leaves of the tree represent termination instructions of the program.

Using a branch predictor we associate with each node a \textit{prediction probability}: this is the probability that the predictor makes the correct guess on this node. The processor has to construct the execution path of the program. This is done by maintaining in the pipe a sub-tree of instructions, whose root \( r \) is the branch
that is currently processed; when this branch is resolved, the next node \( r' \) in the program path becomes the root of the sub-tree held in the pipe, and any node that is unreachable from \( r' \) is flushed from the pipe. This is known as the Adaptive Path Selection (A_PS) problem.

The A_PS problem is a special case of the Markovian CLPP, in which the access graph is a branch tree, the cache size (representing the pipe length) is some \( k > 1 \) and \( d = k - 1 \). We discuss this special case of the CLPP in Chapter 5.

The usage of speculative execution of code typically results in the simultaneous (tentative) execution of multiple paths in the program. This requires the processor design to be more complicated. A main factor in the added complexity is the number of prefetched paths, which can be held in parallel in the pipe. The number of paths is given by the number of leaves of the sub-tree of instructions fetched into the pipe. We discuss the resulting variant of the CLPP problem in Chapter 6.

1.2.3 Switch-based High-Speed LANs

Switch-based networks are becoming popular, to meet the increasing demand for higher network performance. Some aspects of such networks were studied by Yang and Ni [22].

Traditional shared-medium Local Area Networks (LANs) do not provide satisfactory throughput and latency for some communication intensive applications, especially those that require the transmission of multimedia data. Switch-based networks can give higher bandwidth and therefore make such applications possible. Various switching hubs were developed for Ethernet, Fast Ethernet and ATM.

For example, in ATM networks, the data unit is a fixed size cell of 53 bytes; the function of the network is to switch packets from input ports to output ports. ATM can also pipeline data: the switch fabric is based on many small switch elements. Packets that cannot be delivered immediately due to channel contention inside the switch can be buffered at the input FIFO.

Suppose, that an application running on a client’s machine requests data from the host. Once the connection is established between the client and the host, packages including requests are delivered from the client, and the requested data is then transmitted from the host. Suppose, that the path between the client and the host consists of \( S \) switches. Clearly, at most \( 2S \) requests can be processed simultaneously. The problem of finding an optimal algorithm for handling the client’s requests can be described as an instance of the CLPP, in which \( k \) is the size of the local cache at the client, and \( d = 2S \).
1.3 Related Work

The CP problem was introduced by Cao et al. [4]: the paper studied algorithms for the offline version of the problem. An algorithm called Aggressive (AGG) was shown to yield a \( \min(1 + \frac{d}{k}, 2) \)-approximation to the optimal.

Karlin and Kimbrel studied in [9] a generalized version of the CP, where we have \( r \) storage units (e.g., disks), and we can prefetch \( r \) blocks in parallel. The paper gives performance bounds for several offline algorithms in this setting. The Aggressive algorithm is shown to achieve a ratio of \((1 + \frac{r}{k})\) to the optimal.

The paging (or caching) problem has been widely studied (comprehensive surveys appear e.g., in [2], [6]). The traditional Sleator-Tarjan competitive analysis of paging [18] makes no restriction on the reference sequence generated by the adversary\(^2\).

In this model, it was shown that every deterministic online algorithm has a competitive ratio at least \( k \), where \( k \) is the main store (or cache) size, and that several algorithms used in practice, including Least Recently Used (LRU) and First-In-First-Out (FIFO) are \( k \)-competitive, i.e., optimal by this measure. Yet, in practice LRU is almost always superior than FIFO, and in fact, its competitive ratio is typically a small constant. One reason for this discrepancy between the theoretical analysis and empirical results is that real programs exhibit locality of reference, i.e., the set of possible reference sequences has to be consistent with an underlying access graph reflecting this locality.

Borodin et al. [2] and Irani et al. [7] propose to use access graphs in studying paging algorithms. The access graph model enabled to distinguish between the competitive ratios of LRU and FIFO on specific access graphs (e.g., trees). Borodin et al. [2] present an online algorithm (called FAR), that is strongly competitive on any access graph; Irani et al. [7] study online algorithms that perform well on directed access graphs.

Karlin et al. introduced in [8] the Markov paging problem, in which the access graph model is combined with the generation of reference sequences by a Markov chain. Specifically, the transition from a reference to a page \( u \) to the reference to a page \( v \) (both represented as nodes in the access graph of the program) is done with some fixed probability. The paper presents an algorithm whose fault rate is at most a constant factor from the optimal, for any Markov chain.

There has been some earlier work on the \( \lambda \text{PS} \) problem, in the context of pipeline processors: The Eager Execution (EE) algorithm (see in Section 2.2) was shown to perform well in practice (see, e.g., [1] and [20]), however no theoretical performance bound were given for this algorithm. Uht and Sindagi introduced in [20] the Disjoint Eager Execution algorithm and gave empirical study of its performance.

Raghavan et al. showed in [16] that by using Markov decision theory, it

\(^2\)Throughout this work, we sometimes call an optimal offline algorithm ‘the adversary’.
is possible to give an optimal algorithm for the $A_{PS}$ problem. However, this algorithm is impractical in requiring amount of computation that is exponential in $k$, the length of the pipe. Also it was shown in [16] that for the special case of a homogeneous branch tree, where the transition parameter $p$ is close to 1, $DEE$ is optimal to within a constant factor for the $A_{PS}$ problem. In the present work we strengthen this result, and show, that $DEE$ is within factor $1 + o(1)$ from the optimal on homogeneous branch trees, for any $\frac{1}{2} \leq p \leq 1$.

1.4 Main Results

We present the following results for the CLPP problem:

- We show that AGG is optimal for the offline CLPP, on any access graph and for any $d, k \geq 1$.

- For the online CLPP we show that
  
  - If $G$ is a complete graph of size at least $(2k + 1)$, then any marking algorithm is within factor $1 + 2/k$ from the optimal in the set of deterministic on-line algorithms on $G$.
  
  - If $G$ is a DAG then EE is optimal to within factor $\min(1 + \frac{k}{d}, 2)$ in set of deterministic online algorithms, for any $k, d \geq 1$. In the special case where $G$ is a branch tree, EE is within factor $1 + o(1)$ from any optimal online algorithm for the CLPP.

- For the Markovian CLPP on branch trees we show that DEE achieves a ratio of $(1 + o(1))$ to the optimal on homogeneous trees, and the ratio 2 for a general Markov chain.

- Finally, for the bounded Markovian CLPP, we give an algorithm that achieves a ratio 2 to the optimal on branch trees.

1.5 Organization of this Work

In Chapter 2 we give definitions and some technical results that will be used in later chapters. In Chapter 3 we study the offline CLPP, and show the optimality of the AGG algorithm. In Chapter 4 we discuss the online CLPP: Section 4.1 discusses complete graphs; Sections 4.2 and 4.3 refer to DAGs and branch trees.

Chapter 5 deals with the Markovian CLPP on branch trees, and in Chapter 6 we study the CLPP on branch trees with bounded number of leaves. Finally, in Chapter 7 we summarize the contribution of this work and discuss some problems for future study.
Chapter 2

Preliminaries

2.1 Definitions and Notation

We now define the performance measures that will be used in our study of algorithms for the CLPP problem. Let $G$ denote the access graph; $k$ is the size of the cache; $d$ is the delivery time of a block, and $\text{OPT}$ is an optimal offline algorithm for the CLPP problem. In other words, $\text{OPT}$ is an algorithm that achieves the minimal possible execution time for any sequence of block accesses; $\text{OPT}$ knows the whole sequence in advance.

Recall, that any reference sequence $\sigma$ is a path in $G$.\footnote{When $G$ is a directed graph, $\sigma$ forms a directed path in $G$. We assume that $r_i \neq r_{i+1}$, for all $1 \leq i < |\sigma|$. Clearly, this makes the problem no easier.} We denote by $\text{Path}(G)$ the set of paths in $G$. Let $A(\sigma)$, $\text{OPT}(\sigma)$ denote the cost incurred by A and $\text{OPT}$ respectively for the execution of $\sigma$. We use competitive analysis (see, e.g., in [3]) to establish performance bounds for online algorithms for the CLPP.

**Definition 2.1** The competitive ratio of an online algorithm $A$ on a graph $G$, for fixed $k$ and $d$, is given by

$$c_{A,k,d}(G) = \sup_{\sigma \in \text{Path}(G)} \frac{A(\sigma)}{\text{OPT}(\sigma)}$$  \hspace{1cm} (2.1)

In the Markovian model, each path $\sigma$ in $G$ is associated with a probability. The probability of a path $\sigma = r_1, r_2, \ldots, r_t$ is given by

$$Pr(\sigma) = \prod_{i=1}^{t-1} p_{r_i,r_{i+1}}$$  \hspace{1cm} (2.2)

where $p_{r_i,r_{i+1}}$ is the transition probability from $r_i$ to $r_{i+1}$. 

12
Definition 2.2 The expected performance ratio of an algorithm \( A \) in the Markovian model on a graph \( G \) for fixed \( k \) and \( d \), is

\[
e_{A,k,d}(G) = \sum_{\sigma \in \text{Path}(G)} \Pr(\sigma) \frac{A(\sigma)}{\text{OPT}(\sigma)}
\]  

(2.3)

where \( \Pr(\sigma) \) is given in (2.2).

The following notations will be used for presenting the results in the next chapters.

Definition 2.3 \( c_{k,d}(G) \) (\( c_{k,d}^{D}(G) \)) is the competitive ratio of an optimal (deterministic) online algorithm for the CLPP, on an access graph \( G \), with fixed \( k \) and \( d \). Thus,

\[
c_{k,d}(G) = \inf_{A \text{ is online} \ c_{A,k,d}(G)}
\]  

(2.4)

and

\[
c_{k,d}^{D}(G) = \inf_{A \text{ is deterministic online} \ c_{A,k,d}(G)}
\]  

(2.5)

where \( c_{A,k,d} \) is given in (2.1).

Definition 2.4 \( c_{k}(G) \) (\( c_{k}^{D}(G) \)) is the competitive ratio of an optimal (deterministic) online algorithm on \( G \), for a cache of size \( k \) and arbitrary \( d \). Thus,

\[
c_{k}(G) = \sup_{d \in \mathbb{N}} c_{k,d}(G)
\]  

(2.6)

and

\[
c_{k}^{D}(G) = \sup_{d \in \mathbb{N}} c_{k,d}^{D}(G)
\]  

(2.7)

In the Markovian model we replace in Definitions 2.3 and 2.4 competitive ratio with expected performance ratio, and use \( e_{A,k,d}(G) \) as given in (2.3).

Finally, an access graph \( G \) is called a branch tree, if \( G \) is a directed binary tree \( T \), in which any internal node has degree 2.

In the Markovian model, there is a transition probability \( p_{u,v} \) from any node \( u \) to its son \( v \). Let \( v, w \) be the left and right son of \( u \); then, \( p_{w,v} = 1 - p_{v,w} \). Assume, w.l.o.g. that \( p_{u,v} \geq p_{u,w} \), then we call \( p_{u,v} \) the local probability of \( u \). Denote by \( \{v_0 = r, v_1, \ldots, v_n = v\} \) the path from the root of \( T \), \( r \), to some node \( v \), then the accumulated probability of \( v \) is given by

\[
p_{u}(v) = \prod_{i=0}^{n-1} p_{v_i,v_{i+1}}.
\]  

(2.8)
2.2 Known Algorithms

We now describe three of the algorithms that will be studied in this work. These algorithms were introduced in earlier papers, in the studies of special cases of the CLPP problem, namely, the CP and the A_PS problems.

- **Aggressive (AGG)** is an offline algorithm proposed by Cao et al. in [4] for the CP problem: AGG always prefetches the next block not in the cache at the earliest possible moment, replacing the block whose next request is farthest in the future. The earliest possible moment is the time in which prefetching becomes profitable, i.e., there is a block in the cache whose next reference is after the first reference to the block that will be fetched. An example of the execution of AGG is given in Figure 1.2.

- The **Eager Execution (EE)** algorithm was proposed for the A_PS problem by Wang and Uht in [21]. We study the performance of EE for the CLPP problem on DAGs. EE always fetches a missing block (or “hole” in $G$) that is closest to the currently accessed block, and discards a block that is unreachable from the current position.

**Example 2.1** An example for the execution of EE is given in Figure 2.1. In this example EE operates on a binary tree with a cache of size $k = 7$, and $d = 6$. Initially the cache is empty. EE chooses a complete sub-tree as follows: First, EE fetches the root of the tree, $N_1$; then it selects the two direct sons $N_2$, $N_3$. Finally, EE initiates the fetches of $N_4$, $N_5$, $N_6$ and $N_7$.

---

2The thick edges show the selected sub-tree.
- **Disjoint Eager Execution (DEE)** is an online algorithm for the $\Lambda_{\text{PS}}$ problem that was studied by Uht and Sindagi in [20]. We study the performance of DEE for the CLPP on Markovian branch trees. DEE fetches the “hole” that is most likely to appear in $\sigma$ from the current position; it discards a block that cannot be accessed in the future.

**Example 2.2** An example of the execution of DEE is given in Figure 2.2. In this example, DEE runs on homogeneous branch tree, with local probability $p = 0.7$ for each node in the tree$^3$. The cache size is $k = 7$ and $d = 6$. DEE starts the selection from the root $N_1$, and continues as follows: each step, DEE selects a node $v$ that has the maximal accumulated probability $p_a(v)$. The accumulated probabilities of the nodes are

\[
\begin{align*}
    - & p_a(N_1) = 1 \\
    - & p_a(N_2) = 0.7 \\
    - & p_a(N_4) = 0.49 \\
    - & p_a(N_8) = 0.343 \\
    - & p_a(N_3) = 0.3
\end{align*}
\]

$^3$The thick edges show the selected sub-tree, and the numbers near vertices show the order of selection.
\[- p_a(N16) = 0.24 \]
\[- p_a(N5) = 0.21 \]

Hence, the nodes are selected in the above order.

### 2.3 Variants of the Algorithms EE and DEE

We now describe some variants of the EE and DEE used for deriving the results in the next chapters. All algorithms operate in phases. Note that DEE' and EE' operate sometimes as offline algorithms, since they use “future information” supplied by the adversary.

- **EE’** starts fetching to the cache, at the beginning of each phase, a sub-graph of $k$ nodes; at time $d$ the adversary reveals to EE’ the last ‘correct’ node in this sub-graph.
- **Lazy-EE** fetches to the cache a sub-graph of $k$ nodes; then, Lazy-EE stalls for $d$ time units.
- **DEE’** selects in each phase the sub-tree of size $k$ selected by DEE. At the end of each phase (i.e. after $k$ steps), the adversary reveals to DEE’ the last ‘correct’ node in its sub-tree.
- **Lazy-DEE** operates as follows: in each phase, it selects the sub-tree selected by DEE in the first $k$ steps, and then stalls for $k$ steps.

### 2.4 Variants of the CLPP

In the following we define the *Single Phase CLPP* that we use in our proofs. Generally, in this problem we examine the performance of an algorithm $A$, assuming it operates on $\sigma$ for $k$ time units. Initially the cache is empty. A can start fetching a subtree of blocks, rooted at $r_1$; the size of this subtree is at most $k$.

Let $G$ be a DAG, with a single source $r_1$ and $k \geq 1$ an integer. Suppose that $\sigma = \{r_1, \ldots, r_k\}$ is a reference sequence of length $k$. Recall that $\sigma$ is a path in $G$. For a given sub-graph $G' \subseteq G$, let $\sigma \cap G'$ denote the maximal set of vertices in $G'$ that form a prefix of $\sigma$. Let $A$ be an online deterministic algorithm. Initially, $A$ knows only that $r_1 \in \sigma$. In the *Single Phase CLPP (S CLPP)* problem, $A$ needs to select a sub-graph $G_A \subseteq G$ of $k$ vertices, such that $\sigma \cap G_A$ is maximal.

For a selected sub-graph, $G_A$, we denote by

$$B(G_A) = \min_{\sigma \in \text{Path}(G), \, |\sigma| = k} |\sigma \cap G_A|$$

(2.9)
the *benefit* of $A$ from the selection of $G_A$. Thus, we seek an algorithm $A$, whose benefit is maximal.

We also define a natural extension of $S_{-CLPP}$ to the *Markovian* case. Let $T$ be a branch tree. In the *Markovian Single Phase CLPP* (*Markovian $S_{-CLPP}$*) we need to choose a subtree $T_A$ of $T$ of size $k$, such that the expected number of “correct” vertices, i.e., the vertices in the set $\sigma \cap T_A$, is maximal. Formally, let

$$B(T_A) = \sum_{\sigma \in \text{Path}(T)} \Pr(\sigma) |\sigma \cap T_A|$$

be the expected *benefit* of an online algorithm $A$. In the *Markovian $S_{-CLPP}$*, we seek an algorithm whose expected benefit is maximal.
Chapter 3

The Offline CLPP Problem

In the offline case, we are given the reference sequence, and our goal is to achieve maximal overlap between prefetching and references to blocks in the cache, so as to minimize the overall execution time of the sequence.

We first show that a set of rules formulated by Cao et al. in [4], to characterize the behavior of optimal algorithms for the CP problem, applies also for the offline CLPP problem. We refer to these rules below as "no harm" rules.

3.1 No Harm Rules

Theorem 3.1 There exists an optimal algorithm \( A \), which satisfies the following rules:

1. A fetches the next block in the reference sequence that is missing in the cache.

2. A evicts the block whose next reference is furthest in the future.

3. A never replaces a block \( B \) by a block \( C \), if \( B \) will be referenced before \( C \).

Proof: We use induction to show, that any algorithm \( A \) can be transformed to an algorithm \( A' \) which satisfies the three "no harm" rules in the first \( i \) references, \( i \geq 1 \), without incurring extra cost.

Base: Assume that the cache is initially empty. Any optimal algorithm starts by fetching the first block in the reference string, and the three rules are satisfied.

Induction step: We consider separately each of the three rules.

(i) Rule 1: Suppose that \( A \) follows rule 1 in the first \( (i - 1) \) references. Indeed, if \( A \) does not initiate a fetch in step \( i \), then rule 1 continues to hold. Suppose then that \( A \) fetches some block, \( r_n \), and discards the block \( r_m \), thus violating rule 1.
Specifically, assume that the reference sequence is
\[ \sigma_1 = (r_1, \ldots, r_i, r_{i+1}, \ldots, r_l, \ldots, r_n, \ldots, r_m, \ldots, r_s, \ldots) \]
and the first missing block in the cache is \( r_i \neq r_n \) (The cache contents are given in Figure 3.1). We define an algorithm \( A' \), which follows rule 1 in step \( i \), without increasing the overall execution time, i.e., in step \( i \) \( A' \) fetches \( r_i \). Thus, after step \( i \) the cache contents of \( A \) and \( A' \) differ in one block. Now, \( A' \) will take the same actions as \( A \), until one of the following occurs.

a. if \( A \) fetches \( r_i \), then \( A' \) fetches \( r_n \).

b. if \( A \) discards \( r_n \), \( A' \) discards \( r_i \).

Note, that at least \( a. \) has to occur before the reference to \( r_n \), since \( A \) has to fetch \( r_i \) to the cache. After \( a. \) or \( b. \) occurs, we get that the cache contents of \( A \) and \( A' \) are identical. Thus, \( A' \) with not incur extra miss on \( r_n \).

The proof is similar for the case where rule 2 or rule 3 was violated.

(ii) **Rule 2:** Assume that \( A \) satisfies rules 1 and 2 in the first \((i - 1) \) steps. As before, it is sufficient to consider the case where the contents of \( A' \)'s cache change in step \( i \), i.e., \( A \) evicts a block. Assume again that the reference sequence is \( \sigma_1 \): \( A \) violates rule 2 by discarding \( r_m \) rather than \( r_s \), that is referenced later. Then \( A' \) discards \( r_s \) and acts like \( A \), until one of the following occurs:

a. \( A \) discards \( r_s \) and fetches some \( r_f \neq r_m \), then \( A' \) fetches \( r_f \) and discards \( r_m \);
b. A fetches \( r_m \) and discards \( r_d \neq r_s \), then \( A' \) fetches \( r_s \), and discards \( r_d \).

We note, that at least \( b \). has to occur before the next reference to \( r_s \), therefore \( A' \) will not pay with extra miss on that block.

(iii) **Rule 3:** Assume now that \( A \) satisfies rules 1, 2, 3 in the first \((i-1)\) steps. Suppose that the reference sequence is

\[
\sigma_2 = (r_1, \ldots, r_i, r_{i+1}, \ldots, r_m, \ldots, r_n, \ldots),
\]

and the first missing block is \( r_n \). In step \( i \) \( A \) initiated a fetch of \( r_n \) and discards \( r_m \), that is referenced before \( r_n \); \( A' \) does no fetch in step \( i \); then, \( A' \) operates like \( A \) until either

a. \( A \) fetches \( r_m \) and discards some block \( r_d \neq r_n \): in this case \( A' \) discards \( r_d \) and fetches \( r_n \), or

b. \( A \) discards \( r_n \) and fetches \( r_f \neq r_m \), then \( A' \) discards \( r_m \) and fetches \( r_f \).

As in (i) and (ii), we get that \( A' \) has the same cache contents like \( A \) after one of the above occurs. This completes the proof.

\[\square\]

In the remainder of this section we consider only optimal algorithms that follow the “no harm” rules. Clearly, once an algorithm \( A \) decides to fetch a block, these rules define uniquely the block that should be fetched, and the block that will be evicted. Thus, the decision to be made by any optimal offline algorithm for the CLPP is when to start the next fetch.

### 3.2 Optimality of Aggressive

Recall, that the algorithm \( \text{AGG} \) follows the “no harm” rules, and fetches each block at the earliest opportunity. We now show, that this algorithm is the best possible for the CLPP.

**Theorem 3.2** \( \text{AGG} \) is an optimal offline algorithm for the CLPP problem.

**Proof:** We show by induction, that for all \( i \geq 1 \), any optimal offline algorithm \( A \) which satisfies the “no harm” rules, can be modified to act like \( \text{AGG} \) in the first \( i \) steps, without harming \( A' \)’s optimality.

**Base:** Assume that the cache is initially empty, then both \( A \) and \( \text{AGG} \) fetch \( r_1 \).

**Induction Step:** Assume that \( A \) acts like \( \text{AGG} \) in the first \((i-1)\) steps. We distinguish between three cases in the reference \( r_i \):

20
(i) A initiates a fetch of some block \( r_i \), then since A satisfies the “no harm” rules, and AGG fetches in the first opportunity, clearly, AGG acts in step \( i \) like A (\( r_i \) is missing in AGG’s cache, and can be fetched).

(ii) If AGG does no fetch, then since A satisfies the “no harm” rules, A cannot start a fetch as well, and the induction hypothesis holds after step \( i \).

(iii) Finally, assume that AGG initiates a fetch, and A does not. Suppose that AGG fetches \( r_n \) and discards \( r_m \). We define the algorithm A’, which operates like AGG in step \( i \), and then proceeds like A, until one of the following occurs:

- a. If A fetches \( r_n \) and discards \( r_d \neq r_m \), then A’ fetches \( r_m \) and discards \( r_d \).
- b. A discards \( r_m \) and fetches \( r_f \neq r_n \), then A’ discards \( r_n \) and fetches \( r_f \).

Note, that at least \( a \) will occur before the next reference to \( r_n \) (which precedes the next reference to \( r_m \)). This completes the proof. 

\[ \square \]

The greediness of AGG plays an important role in the case where \( d < k \), as shown in the next result.

**Corollary 3.1** If \( d < k \) then in any reference sequence, AGG incurs a single miss: in the first reference.

**Proof:** We show that for any \( i \geq 1 \), when AGG accesses \( r_i \) in the cache, each of the blocks \( r_{i+1}, \ldots, r_{i+d} \) is either in the cache or being fetched. The proof is by induction on \( i \).

**Base:** \( i = 1 \). Assuming that the cache is initially empty, then AGG starts at time \( t = 1 \) to fetch \( r_1 \), and stalls in the next \( d - 1 \) time units, in which it initiates the fetches of \( r_2, \ldots, r_d \). Thus, AGG accesses \( r_1 \) in the cache at time \( t = d + 1 \), and the claim holds.

**Induction Step:** Assume that the claim holds till \( r_i \), and we show for \( r_{i+1} \). We handle two cases:

(i) If at the time \( r_i \) is accessed, \( r_{i+d} \) is in the cache or being fetched, then AGG will not interrupt this fetch or discard \( r_{i+d} \) from the cache, by the “no harm” rules.

(ii) If \( r_{i+d} \) is not being fetched, and it is not in the cache, then \( r_{i+d} \) is the first missing block in the cache (By the induction hypothesis, all the preceding blocks are either in the cache or being fetched). By the “no harm” rules, AGG will not discard any of the blocks \( r_i, \ldots, r_{i+d-1} \) for fetching \( r_{i+d} \). In addition, \( d < k \), hence there exists in the cache at least one block \( r_x \), whose next reference is after \( r_{i+d} \); AGG will discard the block \( r_x \) and initiate a fetch of \( r_{i+d} \). We get that the claim holds also after the \( i \)th reference.
Chapter 4

The Online CLPP Problem

4.1 Complete Graphs

We first show, that for a complete access graph, the lower bound obtained for deterministic algorithms for the classic paging problem, remains the same for the CLPP; however, we require that $G$ is sufficiently large, namely, the number of vertices is at least $(2k + 1)$.

**Theorem 4.1** For any $k \geq 1$, and $G$ of size at least $(2k + 1)$,

\[ e_k^D(G) \geq k - 1 \]  

(4.1)

**Proof:** Consider the case where $d = k - 1$. Recall, that when $d < k$, any optimal offline algorithm stalls for $d$ time units in the first reference, and any other reference incurs the cost 1. At any time, the on-line algorithm $A$ can hold $k$ blocks in the cache, while fetching $k$ other blocks. Since $|V| \geq 2k + 1$, there exists a block $r_f$, that is not in the cache and is not being fetched. We can construct a sequence, in which the next reference is to the current missing block, $r_f$. Thus, $A$ stalls in each reference for $d = k - 1$ time units. This completes the proof. \qed

In the following we show that the lower bound derived in Theorem 4.1 cannot be substantially improved: the ratio in (4.1) can be achieved, to within an additive factor 2 by a caching-by-demand algorithm. Consider the set of marking algorithms proposed for the classical caching (paging) problem (see, e.g., [2], [19], [25] and comprehensive surveys in [3], [15]). A marking algorithm proceeds in phases. At the beginning of a phase all the blocks in the cache are unmarked. Whenever a block is requested, it is marked. On a fault, the marking algorithm evicts an unmarked block from the cache and fetches the requested one. A phase ends on the first ‘miss’ in which all the blocks in the cache are marked (equivalently, a phase ends when $k$ distinct blocks have been requested during this phase). At this point all the blocks become unmarked and a new phase begins.
Theorem 4.2 For any access graph $G$ and $k$, $d \geq 1$, if $A$ is a marking algorithm, then
\[ c_{A,k,d}(G) \leq k + 1 \tag{4.2} \]

Proof: Let $A$ be a deterministic marking algorithm. Recall, that marking algorithms work in phases. We denote by $n_j$ the number of references in phase $j$, $j \geq 1$.

We start the proof with calculation of the cost incurred by $\text{OPT}$ for the execution of phase $j$ of $A$: each phase includes accesses to $k + 1$ distinct blocks; thus, any algorithm (including $\text{OPT}$) has to fetch at least one block from secondary memory to the cache. Also, if a phase consists of $n_j$ accesses, any algorithm has to spend $n_j$ steps on the execution of $n_j$ accesses. Therefore, $\text{OPT}$ needs at least $\max(n_j, d)$ steps to complete the execution of phase $j$ of $A$.

Now we calculate the cost incurred by $A$ in phase $j$. $A$ fetches blocks only on a fault and it can fetch at most $k$ blocks within phase $j$. Therefore, the cost incurred by $A$ for phase $j$ is at most $kd + n_j$. This yields the desired ratio:

\[ c_{A,k,d}(G) \leq \frac{n_j + kd}{\max(n_j, d)} = \frac{n_j}{\max(n_j, d)} + \frac{kd}{\max(n_j, d)} \tag{4.3} \]
\[ \leq 1 + k \]

From the Theorems 4.1 and 4.2 we conclude that marking algorithms are close to the optimal in the set of deterministic algorithms on complete graphs, as summarized in our next result.

Corollary 4.1 For any $k \geq 1$, if $G$ is a complete graph of size at least $(2k + 1)$, then any marking algorithm, $A$, is within factor $1 + \frac{2}{k}$ from the optimal in the set of deterministic online algorithms on $G$.

4.2 Directed Acyclic Access Graphs

Let $G$ be a DAG, with a single source $r_1$ and $k \geq 1$ an integer. Suppose that $\sigma = \{r_1, \ldots, r_k\}$ is a reference sequence of length $k$. Recall that $\sigma$ is a path in $G$. For a given sub-graph $G' \subseteq G$, let $\sigma \cap G'$ denote the maximal set of vertices in $G'$ that form a prefix of $\sigma$. Let $A$ be an online deterministic algorithm. Initially, $A$ knows only that $r_1 \in \sigma$. In the Single Phase CLPP ($S_{\text{CLPP}}$) problem, $A$ needs to select a sub-graph $G_A \subseteq G$ of $k$ vertices, such that $\sigma \cap G_A$ is maximal.
For a selected sub-graph \( G_A \), we denote by

\[
B(G_A) = \min_{\sigma \in \text{Path}(G)} \frac{|\sigma \cap G_A|}{|\sigma| = k}
\]

the benefit of \( A \) from the selection of \( G_A \). Thus, we seek an algorithm \( A \), whose benefit is maximal.

The next result shows that \( EE \) is optimal in the set of online deterministic algorithms for the \( S_{\text{CLPP}} \).

**Lemma 4.3** Given a DAG \( G \), and \( k \geq 1 \), for any deterministic online algorithm \( A \)

\[
B(G_{EE}) \geq B(G_A).
\]

**Proof:** Note, that \( A \) selects the sub-graph \( G_A \), before it knows any of the vertices \( r_2, \ldots, r_k \). By the definition of \( B(G_A) \), in the worst case \( \sigma \cap G_A \) is the shortest path from \( r_1 \) to some vertex \( v \in G \), such that \( v \notin G_A \) (we call such a vertex a “hole”). \( EE \) always selects the hole that is closest to \( r_1 \). Hence, the length of the shortest path from \( r_1 \) to a vertex not in \( G_{EE} \) is maximal. \( \square \)

**Theorem 4.4** If \( G \) is a DAG then for any \( k, d \geq 1 \)

\[
e_{EE,k,d}(G) \leq \min(1 + k/d, 2) \cdot c_{k,d}(G).
\]

**Proof:** Consider two algorithms, \( EE' \) and Lazy-EE, that operate in phases. Both \( EE' \) and Lazy-EE start each phase with no fetches in progress; also both algorithms start each phase, \( i \), by fetching to the cache a sub-graph of \( k \) vertices: the first vertex fetched in this sub-graph is the source of the DAG at the beginning of phase \( i \), denoted by \( r_i \). \( EE' \) and Lazy-EE select the same sub-graph, the one that \( EE \) selects, and for both algorithms, the \( i \)-th phase ends when the last correct vertex in the sub-graph becomes known.

The main difference between \( EE' \) and Lazy-EE is in following:

- For any \( i \geq 1 \), \( d \) time units after phase \( i \) started, the adversary reveals to \( EE' \) the last correct vertex in the sub-graph fetched to the cache in phase \( i \); this vertex becomes \( r_{i+1} \).

- at any phase \( i \geq 1 \), Lazy-EE stalls for \( d \) time units once it has initiated the fetches of \( k \) vertices.

Note that by Lemma 4.3, \( EE' \) outperforms any online algorithm, since it fetches in each phase the same sub-graph as \( EE \), but waits only \( d \) time units for knowing \( r_{i+1} \). The length of each phase of \( EE' \) is \( d \) time units. Also, \( EE \) performs as good as Lazy-EE, since it never stalls. Finally, if \( k > d \) we can take
\( k = d \), since we can replace any online algorithm by its version that uses only \( d \) blocks in the cache. Hence,

\[
c_{EE,k,d}(G) \leq \min\left(\frac{k + d}{d}, 2\right)
\]  \hspace{1cm} (4.7)

which gives the statement of the theorem. \qed

### 4.3 Branch Trees

We now show that if \( G \) is a branch tree \( T \) then the result in Theorem 4.4 can be tightened, in the case where \( d = k - 1 \). This case is of particular interest in the application of the CLPP to pipeline execution of programs on fast processors (see Section 1.2).

**Theorem 4.5** If \( T \) is a branch tree then

\[
c_{EE,k,k-1}(T) \leq (1 + o(1))c_{k,k-1}(T).
\]  \hspace{1cm} (4.8)

We use in the proof the next lemma.

**Lemma 4.6** If \( T \) is a branch tree then \( c_{k,k-1}(T) \geq k/\lg k \).

**Proof:** We prove a lower bound on the expected performance ratio of any deterministic algorithm, on problem instances chosen from a specific probability distribution. The theorem will then follow, from Yao’s method [24].

Assume that \( d = k - 1 \). Suppose that \( T \) is rooted at \( r_1 \). The adversary generates the reference sequence \( \sigma \) as follows: at node \( i \), the adversary proceeds to the left child with probability \( 1/2 \).

Let \( v \) be a node in \( T \), and denote by \( h(v) \) the depth of \( v \) in \( T \). (The depth of \( r_1 \) is 0). The accumulated probability of \( v \), \( p_a(v) \) is the probability that the adversary selects \( v \) for \( \sigma \). Obviously, in our case this probability depends on \( h(v) \) and is equal to

\[
p_a(v) = \frac{1}{2^{h(v)}}
\]  \hspace{1cm} (4.9)

Now, we allow the online algorithm, \( A \), to start fetching the first \( k \) blocks at time \( t = 0 \) (rather than one block per time unit); then, \( A \) waits for \( k \) steps and starts fetching another set of blocks at time \( k \).

The goal of \( A \) is to maximize the expected number of hits (the ‘benefit’). Consider an algorithm \( A \), that proceeds as follows. First, Asorts the nodes in \( T \) in decreasing order by the probabilities that they belong to \( \sigma \), and then fetches the first \( k - 1 \) nodes in the list. In our case, \( A \) takes a balanced subtree of \( k \) nodes, rooted in \( r_1 \).
We can now calculate the expected benefit of any online algorithm. Let $v_j$ be the $j$th node fetched to the cache. The expected number of hits in the first $k$ time units is
\[
\sum_{j=1}^{k} p_a(v_j) = \sum_{j=1}^{k} \left( \frac{1}{2} \right)^{h(v_j)}
\]  
(4.10)

We can see that the above algorithm, $A$, maximizes this value, since it selects $k$ nodes with the highest probabilities. Hence, the expected benefit of any online algorithm is bounded by the height of a balanced tree of $k$ nodes, that is $\lg k$.

Any optimal offline algorithm will incur a miss on $r_1$, while any other reference costs 1 time unit. Hence, its total cost is $|\sigma| + d$, while $A$’s expected cost is at least $|\sigma|k/\lg k$.

Proof of Theorem 4.5: Consider a variant of the Lazy-EE, which operates in phases: in phase $j$ it fetches into the cache a complete binary branch tree of size $n = k/\lg k$; then, it initiates no fetches in the next $k$ time units.

This algorithm incurs on the average the cost $k + n$ for the execution of a reference sequence of length $\lg n$. In addition, EE performs at least as well as this algorithm, on any reference sequence. Thus,
\[
c_{EE,k,k-1} \leq \frac{n + k}{\lg n} = \frac{k(1 + 1/\lg k)}{\lg k - \lg \lg k} = \frac{k}{\lg k} (1 + o(1))
\]  
(4.11)

Using Lemma 4.6 we get the statement of the theorem. □
Chapter 5

The Markovian CLPP on Branch Trees

DEE is a natural greedy algorithm for the CLPP in the Markovian model. In this section we analyze the performance of DEE on branch trees, and show that it is optimal to within a constant factor in the set of online algorithms on these graphs. As shown below, this constant is reduced to \((1 + o(1))\) for a special class of Markov chains (that we call homogeneous), namely, in each state we face the same set of choices, with a fixed transition probability for each choice (see in Section 5.2). As in Section 4.3, we assume in our discussion of the Markovian CLPP on branch trees that \(d = k - 1\).

5.1 Performance of DEE on Branch Trees

Let \(T\) be a branch tree, and \(k \geq 1\) an integer. Suppose that \(\sigma\) is a reference sequence of length \(k\). Recall, that \(\sigma\) is a path in \(T\). For a given subtree \(T_A \subseteq T\) chosen by an online algorithm \(A\), we denote by \(\sigma \cap T_A\) the set of vertices in \(T_A\), that appear in \(\sigma\). In the Markovian Single Phase CLPP (Markovian S-CLPP) we need to choose a subtree \(T_A\) of \(T\) of size \(k\), such that the expected number of “correct” vertices, i.e., the vertices in the set \(\sigma \cap T_A\), is maximal.

Formally, let

\[
B(T_A) = \sum_{\sigma \in \text{Path}(T)} \Pr(\sigma) |\sigma \cap T_A|
\]

be the expected benefit of an online algorithm \(A\). We seek an algorithm whose expected benefit is maximal.

The next result shows that DEE is optimal in the set of online algorithms for the Markovian S-CLPP.

Lemma 5.1 (Raghavan et al. [16]) Given a branch tree \(T\), and \(k \geq 1\), for any online algorithm \(A\)

\[
B(T_{DEE}) \geq B(T_A).
\]

(5.2)
The optimality of DEE for the Markovian S_CLPP will be used for obtaining a performance bound for DEE, when applied to the CLPP problem.

**Theorem 5.2** DEE is optimal to within factor 2 in the set of online algorithms on branch trees.

**Proof:** The proof technique is similar to the proof of Theorem 4.4: we define the DEE' and Lazy-DEE algorithms, that operate in phases. DEE' selects in each phase the sub-tree of size $k$, $T_{DEE}$, selected by DEE in solving the Markovian S_CLPP problem. At the end of each phase (i.e. after $k$ steps), the adversary reveals to DEE' the last ‘correct’ node in its sub-tree. By Lemma 5.1, DEE’ outperforms any on-line algorithm.

Lazy-DEE operates as follows: in each phase, it selects the sub-tree $T_{DEE}$ in the first $k$ steps, and then stalls for $k$ steps. Thus, Lazy-DEE starts a new phase every $2k$ steps, while DEE’ starts every $k$ steps. This yields the desired ratio of 2. \(\square\)

### 5.2 Homogeneous Branch Trees

We call a Markovian access graph $G$ a homogeneous branch tree $T$, if $T$ is a complete binary branch tree, and for some $0 < p < 1$, the transition probability from any node $v_j$ to its left child is $p$, and $1 - p$ to its right child. We assume that $p \geq 1/2$ \(^1\). Thus, in such access graph, the reference sequence, $\sigma$, is constructed by proceeding in each step from the current node either to the left child (with the fixed probability $p$) or to the right child (with probability $1 - p$).

In the following we derive an asymptotic expression for the expected benefit of DEE, when solving the Markovian S_CLPP on a homogeneous branch tree, with any parameter $1/2 \leq p < 1$. Our computations are based on a Fibonacci-type analysis, which suits well the homogeneous case. We use in the analysis the next technical lemma.

**Lemma 5.3** For any $\frac{1}{2} \leq p < 1$ such that $p^n = 1 - p$ and $q \in N$, let

\[
\alpha = (1 - p)q + p, \\
\delta \in \{0, 1\}
\]

then the height of the sub-tree chosen by DEE for the Markovian S_CLPP is given by

\[
\text{height}(T_{DEE}) = \log_{1/p}((1 - p)\alpha k + p) + \delta + o(1) \tag{5.4}
\]

**Proof:** Let $q$ be defined as in (5.3), and denote by $f(n)$ the number of vertices in $T$ with accumulated probability $p^n$. Then, $f(n)$ can be computed recursively as follows:

\(^1\)Otherwise we can take $p' = 1 - p$, and switch the right with the left child in each node
(i) For $1 \leq n < q$, since $p^n > 1 - p$, there is a single node with accumulated probability $p^n$. This node is reached by proceeding from the root of $T$ on a path of length $n$, such that in each node we choose the left child.

(ii) For $n \geq q$, we get a node with accumulated probability $p^n$, either by taking the left child of a node with accumulated probability $p^{n-1}$, or by taking the right child of a node, whose accumulated probability is $p^{n-q}$.

Hence, we get that

$$f(n) = \begin{cases} 
0 & n < 0 \\
1 & \text{if } 0 \leq n < q \\
f(n-1) + f(n-q) & \text{otherwise}
\end{cases}$$

Note, that $f(n)$ can be written in terms of the $q$-distance Fibonacci numbers. Specifically,

$$f(n) = g(n + q - 1) \quad (5.5)$$

Hence, we get from (A.14) that

$$f(n) = b_q r_q^{n+q-1}(1 + o(1)), \quad (5.6)$$

where $r_q$ is the single root of the equation (A.1) in the interval $[1, 2]$. Using (5.3), it is easy to verify that $r_q = 1/p$ satisfies (A.1). From (A.16) and (5.3) we get that

$$b_q = \frac{1}{p^n-1} \left( \frac{q-1}{p^n-2} \right) = \frac{1-p}{p\alpha} \quad (5.7)$$

Hence, from (5.6)

$$f(n) = \frac{1-p}{p\alpha} \frac{1}{p^{n+q-1}}(1 + o(1)) = \frac{1 + o(1)}{\alpha p^n}. \quad (5.8)$$

Let $h$ be the maximal integer satisfying

$$\sum_{n=0}^{h} f(n) \leq k. \quad (5.9)$$

Hence, we get that

$$k \geq \frac{1 + o(1)}{\alpha} \sum_{n=0}^{h} \frac{1}{p^n} = \frac{1 + o(1)}{\alpha} \frac{1 - (1/p)^{h+1}}{1 - 1/p} = \frac{1 + o(1)}{\alpha (1 - p)} \left( \frac{1}{p^n} - p \right). \quad (5.10)$$
Note, that any vertex with accumulated probability $p^h$ is in $T_{DEE}$ and there exists a vertex with accumulated probability $p^{h+1}$ which is not in $T_{DEE}$. Thus,

$$k \leq \sum_{n=0}^{h+1} f(n) = \frac{1 + o(1)}{\alpha(1 - p)} \left( \frac{1}{p^{h+1}} - p \right).$$

(5.11)

Combining (5.10) and (5.11) we get that

$$(1 + o(1)) \frac{1}{p^h} \leq k(1 - p) \alpha + p(1 + o(1)) \leq (1 + o(1)) \frac{1}{p^{h+1}}$$

(5.12)

and we find the value of $h$ by taking a logarithm (to the base $1/p$) from both sides of (5.12). We note, that $\log_p (1 + o(1)) = o(1)$, and

$$h = \log_p(k(1 - p) \alpha + p(1 + o(1))) = \log_p(k(1 - p) \alpha + p) + o(1).$$

(5.13)

From the above discussion $h \geq \text{height}(T_{DEE}) - 1$. This yields the statement of the lemma.

Using Lemma 5.3 we can get an asymptotic expression for the expected benefit of DEE.

**Theorem 5.4** For any $\frac{1}{2} \leq p < 1$, the expected benefit of DEE in solving the Markovian $S_{CLPP}$ problem is

$$B(T_{DEE}) = \frac{1 + \log_p((1 - p) \alpha k + p)}{\alpha} (1 + o(1))$$

(5.14)

We use in the proof the next lemmas. The first lemma is an alternative formulation of (5.1).

**Lemma 5.5** ([23]) For any $k \geq 1$, the expected benefit of any algorithm $A$ in solving the Markovian $S_{CLPP}$ on a branch tree $T$ is given by:

$$B(T_A) = \sum_{v \in T_A} p_a(v).$$

(5.15)

**Lemma 5.6** For any $k \geq 1$, $B(T_{DEE})$ is a continuous function of $p$.

**Proof:** Denote by $T_k$ a subtree of size $k$ rooted in $r$, the root of $T$, and let $f_v(p) = p_a(v)$, for any $v \in T$. Using Lemmas 5.1 and 5.5 we can write

$$B(T_{DEE}) = \max_{T_k \subseteq T} B(T_k) = \max_{T_k \subseteq T} \sum_{v \in T_k} f_v(p) \equiv F(p)$$

(5.16)

Note that for any $v \in T$, $f_v(p) = p^s(1 - p)^t$, for some $0 \leq s, t \leq k$. Thus, $f_v(p)$ is a continuous function of $p$. Also, recall that if $f, g$ are continuous functions, then $f + g$ and $\max(f, g)$ are continuous too. We conclude that $F(p)$ is a continuous function of $p$. 

31
Proof of Theorem 5.4: First, we prove the theorem for \( p \) satisfying \( p^q = 1 - p \), where \( q \in N \). Using (5.9) and (5.11) we can write

\[
\sum_{n=0}^{h} p^n f(n) \leq B(T_{DEE}) \leq \sum_{n=0}^{h+1} p^n f(n)
\]

(5.17)

From (5.8) we get that

\[
\sum_{n=0}^{h+1} p^n f(n) = \sum_{n=0}^{h+1} \frac{1 + o(1)}{\alpha} = \frac{(h + 2)(1 + o(1))}{\alpha},
\]

(5.18)

and

\[
\sum_{n=0}^{h} p^n f(n) = \frac{(h + 1)(1 + o(1))}{\alpha}
\]

(5.19)

By (5.4), \( h \) tends to infinity, when \( k \) increases (also, note that \( h = \Omega(\log k) \)). Hence, we can assume that \((h + 2)(1 + o(1)) = (h + 1)(1 + o(1))\). We can write

\[
B(T_{DEE}) = \frac{(h + 1)(1 + o(1))}{\alpha}
\]

(5.20)

Substituting \( h \) with the value given in (5.13) we get the statement of the theorem.

Now, we need to show the statement of the theorem for \( p \) satisfying \( p^q = 1 - p \), where \( q \notin N \). By Lemma 5.6 \( B(T_{DEE}) \) is a continuous function of \( p \). Note, that the function \( q = \log_p(1 - p) \) is continuous in \( p \) and increasing to infinity, therefore (5.14) holds in an infinite set of points \( p_1, p_2, \ldots \in [\frac{1}{2}, 1] \); the limit of this sequence is \( p = 1 \). Thus,

\[
\lim_{p \to 1} B(T_{DEE}) = \frac{1 + \log_p((1 - p)\alpha k + p)}{\alpha}(1 + o(1)) = 0
\]

(5.21)

This yields the statement of the theorem.

\[ \square \]

Theorem 5.7 DEE is within factor \( 1 + o(1) \) from the optimal in the set of algorithms for the Markovian CLPP on homogeneous branch trees, for any \( 1/2 \leq p \leq 1 \).

Proof: Let \( B^* = B(T_{DEE}) \) be defined as in (5.14). We first show, that any online algorithm for the Markovian CLPP on homogeneous branch trees has a competitive ratio at least \( k/B^* \). Suppose that we let an online algorithm \( A \) run for \( k \) steps, in which \( A \) initiates \( k \) fetches of blocks; then, the adversary reveals to \( A \) the last “correct” node in \( T_A \). Obviously, this gives an advantage to \( A \) (since, by the pipeline property, after \( k \) steps \( A \) accesses only the first block in the cache). The optimal strategy here is to run DEE’ for \( k \) steps, and then to

32
start a new phase, taking the last “correct” node to be the new root of the access graph $T$.

Now, consider the Lazy-DEE, that fetches in each phase $\frac{k}{B^*}$ nodes, and then stalls for $k$ steps. Hence, we can partition the execution of $\sigma$ to phases, each of length $k + \frac{k}{B^*}$. The adversary accesses $k + \frac{k}{B^*}$ blocks in a phase, while the expected benefit of Lazy-DEE in a phase is

$$
\frac{1 + o(1)}{\alpha} \left[ 1 + \log_{\frac{1}{p}}((1-p)\alpha \frac{k}{B^*} + p) \right] \geq \frac{1 + o(1)}{\alpha} \left[ 1 + \log_{\frac{1}{p}}\left(\frac{(1-p)\alpha k + p}{B^*}\right)\right]
$$

$$
= \frac{1 + o(1)}{\alpha} \left[ 1 + \log_{\frac{1}{p}}((1-p)\alpha k + p) - \log_{\frac{1}{p}} B^* \right]
$$

$$
= B^* - \frac{1 + o(1)}{\alpha} \log_{\frac{1}{p}} B^*
$$

The last inequality holds since $B^* \geq 1$. Finally, we divide the expected benefit of Lazy-DEE in a single phase by the benefit of the adversary to get the expected performance ratio:

$$
c_{Lazy-DEE,k,k-1}(T) < \frac{k + \frac{k}{B^*}}{B^* - \log_{\frac{1}{p}} B^*}
$$

$$
= \frac{k}{B^*} \left(1 + \frac{1}{B^* - \log_{\frac{1}{p}} B^*}\right)
$$

$$
= \frac{k}{B^*} (1 + o(1)).
$$

As shown above, $k/B^*$ is a lower bound on the expected performance ratio of any online algorithm. This completes the proof. \qed
Chapter 6

Branch Trees with Bounded Number of Leaves

Recall, that the DEE algorithm was proposed for speculative execution of code. As shown in Chapter 5, DEE performs well on branch trees. Yet, this algorithm is hard to implement. This is due to the large number of execution paths of a program that DEE may simultaneously choose.

This motivates our study of a variant of the Markovian CLPP, in which we add the following constraint. At any time, the sub-tree of blocks held in the cache by the online algorithm can have at most \( L \) leaves. We call this variant: the \( CLPP \) \textit{with Bounded Number of Leaves} \((CLPP(L))\).

6.1 A Dynamic Programming Algorithm

Consider the following generalized version of the Markovian \( S_{-}CLPP \) problem on a branch tree \( T \). The online algorithm \( A \) has to choose a sub-tree \( T_A \) of \( T \) of size \( k \), such that

(i) \( T_A \) has at most \( L \) leaves, for some \( L \geq 1 \).

(ii) The expected number of “correct” vertices is maximized.

We call this problem \textit{Markovian S\textunderscore CLPP}\((L)\). Indeed, in the special case where \( L \geq k \), we get the Markovian \( S\textunderscore CLPP \) problem. Denote by \( |T| \) the size of the branch tree \( T \).

**Theorem 6.1** The Markovian \( S\textunderscore CLPP(L) \) problem can be optimally solved in \( O(L^2k^2|T|) \) steps.

**Proof:** W.l.o.g. we assume that the height of \( T \) is bounded by \( k \) \footnote{Otherwise we can solve the problem on a sub-graph of \( T \).}. We note, that any sub-tree of an optimal sub-tree \( T_k^* \) is also optimal. Hence, we can use
a bottom-up technique for solving the Markovian $S\_CLPP(L)$ problem. We now describe a dynamic programming (DP) algorithm for finding $T_k^*$. We calculate for each $v \in T$ an $L \times k$ matrix $w_v$; $w_v[m, n]$ is the weight (or the expected benefit) of an optimal sub-tree of $T$ rooted at $v$, which has $n$ vertices and $m$ leaves.

We initialize the matrix $w_v$, for any $v \in V$, as follows:

$$w_v[m, n] = \begin{cases} 
1 & \text{if } m = n = 1 \\
0 & \text{if } m = n = 0 \\
-\infty & \text{otherwise}
\end{cases}$$

The entries of $w_v$ are computed recursively: suppose that the vertex $v$ has a left child $x$ and a right child $y$, with the respective transition probabilities $p$ and $1 - p$, then we write

$$w_v[m, n] = 1 + \max_{0 \leq m_0 \leq m} \max_{0 \leq n_0 \leq n-1} (pw_x[m_0, n_0] + (1 - p)w_y[m - m_0, n - n_0 - 1]) \quad (6.1)$$

When calculating the maximum, we can also save the indices $m_0, n_0$, for which the maximum was obtained. Now, if $r$ is the root of the tree $T$, to get an optimal sub-tree it is sufficient to find $\max_{1 \leq m \leq L} w_r[m, k]$. We can construct the tree $T_k^*$ top-down, using the indices saved for each local maximum.

We now calculate the complexity of the DP algorithm. Note, that for each vertex $v \in V$ we compute $L \cdot k$ entries in the matrix $w_v$; each entry requires $O(L \cdot k)$ steps. Hence, the overall running time of DP is $O(L^2 k^2 |T|)$. \hfill \Box

**Corollary 6.1** There is a 2 optimal on-line algorithm for the $CLPP(L)$ problem.

**Proof:** We use the optimality of the DP algorithm for the Markovian $S\_CLPP(L)$. As in the proof of Theorem 4.4, we define the more powerful algorithm DP', and the algorithm Lazy-DP. Comparing the performance ratios of these two algorithms, we obtain a factor of 2. \hfill \Box

### 6.2 The DEE(L) Algorithm

In the case where $T$ is a homogeneous branch tree, we show that for solving the Markovian $S\_CLPP(L)$ problem the DP algorithm can be replaced with a variant of the DEE algorithm, whose complexity is $O(k \log k)$.

The algorithm DEE(L) operates as follows. Let $T'_i$ be the sub-tree obtained after $i$ vertices were selected. $T'_i = \{r\}$, where $r$ is a root of $T$. Let $l(T_i')$ be the set of leaves of $T_i'$. Denote by $\text{child}(T'_i)$ the set of vertices which are the children of the leaves of $T'_i$, i.e.,

$$\text{child}(T'_i) = \{v \in V | (u, v) \in E \text{ and } u \in l(T'_i)\}. \quad (6.2)$$
In step \((i+1)\) \textsc{DEE}(L) chooses the vertex \(v\) from the neighborhood of current sub-tree \(v \in \text{child}(T'_i)\), such that number of leaves would not exceed \(L\) and the accumulated probability of \(v\) is maximal. Or in other words, \(v \in \text{child}(T'_i)\), such that \(|l(T'_i \cup \{v\})| \leq L\) and \(p_a(v)\) is maximal.

\textbf{Lemma 6.2} The running time of \textsc{DEE}(L) is \(O(k \lg k)\) steps.

\textbf{Proof:} We partition the execution of \textsc{DEE}(L) into two stages. In the first stage, the number of leaves in the selected sub-tree is smaller than \(L\). In this stage, \textsc{DEE}(L) will maintain a heap including the vertices in \(\text{child}(T'_i)\) along with their accumulated probabilities. In step \(i\), \(1 \leq i \leq k\) one vertex will be deleted from the heap, and two new vertices (the children of the vertex selected in this step) will be inserted. Since we need to choose \(k\) vertices, the heap size in each step will be at most \(2k\). Hence, the complexity of insert/delete operation is \(O(\lg k)\), and the overall running time of the first stage is \(O(k \lg k)\).

In the second stage, the selected sub-tree has exactly \(L\) leaves (Note, that this stage may not be reached). Hence, we can select among the vertices in \(\text{child}(T'_i)\) only vertices which are \textit{left} sons of the vertices in \(l(T'_i)\), and \textsc{DEE}(L) selects in this set the vertex with maximal accumulated probability. The size of the heap is now bounded by \(L \leq k\), and the overall running time is again \(O(k \lg k)\). \(\square\)

\textbf{Theorem 6.3} \textsc{DEE}(L) is optimal for the Markovian \textsc{S-CLPP}(L) problem on homogeneous trees.

\textbf{Proof:} We use induction to show that in step \(i\), \(1 \leq i \leq k\) the sub-tree \(T'_i\) selected by \textsc{DEE}(L) is contained in an optimal sub-tree of size \(k\), \(T^*_k\).

\textbf{Basis:} \(i = 1\), then \(T'_1 = \{r\} \subseteq T^*_k\) (since the optimal sub-tree contains the root of \(T\)).

\textbf{Induction Step:} Assume that \(T^*_{i-1} \subseteq T^*_k\) for some optimal sub-tree \(T^*_k\). Let \(v\) be the vertex selected by \textsc{DEE}(L) in step \(i\), and let \(U = T^*_k - T^*_{i-1}\) be the set of vertices in \(T^*_k\) which are not in \(T^*_{i-1}\). Suppose that \(v \notin T^*_k\), then we show how \(T^*_k\) can be modified to include \(v\) without harming its optimality.

(i) There is a vertex \(w \in U\) that is not a single child (see in Figure 6.1 (a)), then we cut the edge connecting \(w\) to its father in \(T\); and replace \(v\) with the subtree rooted by \(w\) (Figure 6.1 (b)). First, we note that this change does not affect the local probabilities in the sub-tree of \(w\), due to the homogeneity of the tree \(T\). Also, clearly, \(p_a(v) \geq p_a(w)\), since \(v\) has at least the maximal accumulated probability in the set \(\text{child}(T^*_{i-1}) \cap U\). Hence, the new accumulated probability of any vertex \(x\) in the sub-tree of \(w\) is given by

\[ p'_a(x) = \frac{p_a(v)}{p_a(w)} \cdot p_a(x) \geq p_a(x). \] (6.3)
We conclude, that the weight of the modified tree is not less than the weight of $T_k^*$. Finally, we note, that our transformation has not increased the number of leaves in $T_k^*$.

(ii) Each vertex in $U$ is a single child, i.e., $U$ is set of paths, where each path is connected to a leaf in $T_{i-1}^*$ (Figure 6.2 (a)).

a. If $v$ has a sibling $w$ in $U$, then we can replace $v$ with the path rooted in $w$. Clearly, the resulting tree is still optimal,

b. Suppose, that $v$ does not have a sibling in $U$. Recall, that the tree $T_i'$ has at most $L$ leaves. Paths in $U$ are connected to the vertices in $l(T_i')$. Hence, there exists a path rooted in a vertex $w \in U$; we can replace $v$ with this path (Figure 6.2 (b)). If $|l(T_i')| = |l(T_{i-1}')| + 1$, then replacing $v$ by a path may increase by one the size of the set of leaves of $T_k^*$; however, since $U$ is a set of paths, we have that

$$|l(T_k^*)| = |l(T_{i-1}')|.$$

Hence, the resulting tree has at most $L$ leaves.
This completes the proof.
Chapter 7

Summary

7.1 The Contribution of this Work

In this work we studied the problem of caching integrated with pipelined prefetching. The CLPP problem has natural applications in memory controller policies, and in program execution on pipelined processors. We examined the CLPP problem in the offline setting, as well as the online and the Markovian case. We showed that EE is optimal to within factor 2 in the set of deterministic online algorithms on DAGs, and that AGG is an optimal offline algorithm. In the Markovian model, DEE was shown to be nearly optimal on branch trees.

7.2 Open Problems

Several interesting problems remain open:

- Can randomized online algorithms yield better competitive ratios on DAGs?
- How efficiently can we select an optimal sub-graph in the Markovian model, on a DAG?
- Can EE (or DEE) be adapted to yield efficient online algorithm for other classes of graphs (e.g., SPGs in the paper of Irani et al. [7] or graphs which contain cycles).
- DEE was shown to be optimal to within factor 2 on branch trees, with an arbitrary Markov chain. The derivation of this bound relies on our technique, of solving first the single phase problem. The experimental study of Raghavan et al. [16] shows, that in practice this bound is in fact close to 1. Can other technique be applied to tighten this bound?
- Finally, in defining the cost of an access request, we do not distinguish write accesses from read accesses. Indeed, this suits well the nature of reference
sequences in program execution on fast processors (Section 1.2.2), which consist of reads only. However, in other applications, such as pipelined main memory, accesses include reads and writes. In practice, writes are different from reads, e.g., full-block writes do not require bringing the block into the cache. Treating differently read and write operation would make the analysis more accurate.
Appendix A

Bounds for the $q$-distance Fibonacci Numbers

Definition A.1 For any integer $q \geq 2$, the $n$-th number in the $q$-distance Fibonacci sequence is given by

$$g(n) = \begin{cases} 
0 & \text{if } n < q - 1 \\
1 & \text{if } n = q - 1 \\
g(n - 1) + g(n - q) & \text{otherwise}
\end{cases}$$

Note, that in the special case where $q = 2$, we get the well known Fibonacci numbers (see, e.g., Knuth [11]).

Lemma A.1 For any $n \geq 1$ and a given $q \geq 2$, $g(n) = O\left(\frac{r^{n-q}}{q}\right)$, where $1 < r = r(q) \leq 2$.

Proof: For finding an expression for $g(n)$, $n \geq q$, we need to solve the equation generated from the recursion formula, i.e.,

$$x^n = x^{n-1} + x^{n-q}. \quad (A.1)$$

In other words, we need to find the roots of the polynomial $p(x) = x^q - x^{q-1} - 1$. First, we find that

$$p'(x) = qx^{q-1} - (q - 1)x^{q-2}. \quad (A.2)$$

Note, that $p'(x)$ has only two roots: $x_0 = 0$, $x_1 = \frac{q-1}{q}$, which are not roots of $p(x)$. Hence, we have that $gcd(p(x), p'(x)) = 1$. By a well known theorem from Algebra (see, e.g., Lang [13]), $p(x)$ has no multiple roots. Hence, the general form of the above Fibonacci sequence is

$$g(n) = \sum_{i=1}^{q} b_i r_i^n, \quad (A.3)$$
where \( r_1, r_2, \ldots, r_q \) are all the roots of the polynomial \( p(x) \).

We also note, that for \( 1 \leq x \leq 2 \) \( p(x) \) is monotonically non-decreasing, since \( p'(x) \geq 0 \) in this interval. As

\[
p(1) = -1 \tag{A.4}
\]

and

\[
p(2) = 2^q - 2^{q-1} - 1 = 2^{q-1} - 1 \geq 0, \tag{A.5}
\]

we get that \( p(x) \) has a single root \( r_q \in R^+ \) in the interval \([1, 2]\).

Denote by \( \| x \| \) the module of a complex number \( x \). Now, we claim that \( \| r_i \| < r_q \) for all \( i < q \). The claim trivially holds for any \( r_i \) satisfying \( \| r_i \| \leq 1 \), thus we may assume that \( \| r_i \| > 1 \). If \( r_i \) is a root of \( p(x) \), then

\[
0 = p(r_i) = \| r_i^q - r_i^{q-1} - 1 \| \geq \| r_i \|^q - \| r_i \|^{q-1} - 1 = p(\| r_i \|) \tag{A.6}
\]

Hence, we get that \( p(\| r_i \|) \leq 0 \), and from the fact that \( p(x) \) is non-decreasing for \( x \geq 1 \), we conclude that \( \| r_i \| \leq r_q \).

We now show, that the last inequality is strong, i.e., \( \| r_i \| < r_q \), for any \( 1 \leq i \leq q - 1 \). Assume by contradiction, that \( \| r_i \| = r_q \), i.e., \( r_i = r_q e^{i\phi} \). Then,

\[
\| r_i \|^q = r_q^q, \quad \| r_i \|^{q-1} = r_q^{q-1} \tag{A.7}
\]

and

\[
\| r_i \|^q = \| r_i \|^{q-1} + 1 \tag{A.8}
\]

and since \( r_i \) is a root of \( p(x) \), we get that

\[
\| r_i \|^{q-1} + 1 = \| r_i \|^{q-1} + 1 \tag{A.9}
\]

The last equation means that \( 1 \) and \( r_i^{q-1} \) have the same argument (see, e.g., Lang [13]). Therefore, \( r_i^{q-1} \in R^+ \), and

\[
\phi(q - 1) = 2\pi n \text{ for some } n \in Z. \tag{A.10}
\]

However, since \( r_i^q = r_i^{q-1} + 1 \), this also implies that \( r_i^q \in R^+ \), or

\[
\phi q = 2\pi m \text{ for some } m \in Z. \tag{A.11}
\]

Equations (A.10) and (A.11) are satisfied when \( \phi = 2\pi l \), for some \( l \in Z \), which means that \( r_i = r_q \), in contradiction to the fact that \( p(x) \) has no multiple roots. We conclude that for all \( 1 \leq i \leq q - 1 \) \( \| r_i \| < r_q \).

Using equation (A.3) we can write

\[
g(n) = \sum_{i=1}^{q} b_i r_i^n = b_q r_q^n (1 + \sum_{i=1}^{q-1} \frac{b_i}{b_q}(\frac{r_i}{r_q})^n) \tag{A.12}
\]
and since $\| \frac{x_i}{r_q} \| < 1$, the sum in the RHS of (A.12) exponentially tends to zero, i.e.,

$$\lim_{n \to \infty} \sum_{i=1}^{q-1} \frac{b_i}{b_q} \left( \frac{r_i}{r_q} \right)^n = 0$$  \hspace{1cm} (A.13)

Hence, we get that

$$g(n) = b_q r_q^n (1 + o(1)).$$  \hspace{1cm} (A.14)

Now, we only need to calculate $b_q$. This can be done by solving a linear system for the first $q$ elements of the sequence $g(n)$.

$$
\begin{pmatrix}
1 & 1 & \ldots & 1 \\
1 & r_1 & \ldots & r_q \\
\vdots \\
r_1^{q-1} & r_2^{q-1} & \ldots & r_q^{q-1}
\end{pmatrix}
\begin{pmatrix}
b_1 \\
b_2 \\
\vdots \\
b_q
\end{pmatrix}
= 
\begin{pmatrix}
0 \\
0 \\
\vdots \\
1
\end{pmatrix}
$$

The determinant of the above matrix is known as Vandermonde determinant (see, e.g., Lang [13]). The general solution of such a system is

$$b_i = \prod_{\alpha=1, \alpha \neq i}^{q} (r_i - r_\alpha)^{-1}$$

Our polynomial is

$$p(x) = \prod_{i=1}^{q} (x - r_i)$$  \hspace{1cm} (A.15)

and

$$p'(r_i) = \prod_{\alpha=1, \alpha \neq i}^{q} (r_i - r_\alpha),$$

therefore we get that $b_i = \frac{1}{p'(r_i)}$. Now we can calculate all the coefficients and in particular, we have

$$b_q = \frac{1}{qr_q^{q-1} - (q - 1)r_q^{q-2}}$$  \hspace{1cm} (A.16)

Substituting into (A.14) we get that

$$g(n) = \frac{r_q^n}{q r_q^q \left( \frac{1}{r_q^q} - \frac{q-1}{q} \frac{1}{r_q^2} \right)}$$  \hspace{1cm} (A.17)

This yields the statement of the lemma. \hfill \Box


Bibliography


