Strongly Competitive Algorithms for Caching with Pipelined Prefetching*

Alexander Gaysinsky[†] Alon Itai[‡] Hadas Shachnai [§¶]

Department of Computer Science
The Technion, Haifa 32000, Israel

Abstract

Prefetching and caching are widely used for improving the performance of file systems. Recent studies have shown that it is important to *integrate* the two. In this model we consider the following problem. Suppose that a program makes a sequence of m accesses (references) to data blocks. The cache can hold $k < m$ blocks. An access to a block in the cache incurs one time unit, and fetching a missing block incurs d time units. A fetch of a new block can be initiated while a previous fetch is in progress, thus, d block fetches can be in progress simultaneously. The locality of references to the cache is captured by the *access graph* model of [4]. The goal is to find a policy for prefetching and caching, which minimizes the overall execution time of a given reference sequence. This problem is called *caching with locality and pipelined prefetching (CLPP)*. Our study is motivated from the pipelined operation of modern memory controllers, and from program execution on fast processors.

In the offline case, we show that an algorithm of Cao et al. [6] is optimal. In the online case, we give an algorithm which is within factor of 2 from the optimal in the set of online deterministic algorithms, for *any* access graph, and $k, d \geq 1$. Better ratios are

*A preliminary version of this paper appeared in the proceedings of the *9th Annual European Symposium on Algorithms (ESA)*, August 2001.

[†]`csalex@tx.technion.ac.il`

[‡]`itai@cs.technion.ac.il`

[§]Currently on a leave at Bell Laboratories, Lucent Technologies, 600 Mountain Ave., Murray Hill, NJ 07974.

[¶]Corresponding author: `hadas@cs.technion.ac.il`. Telephone: +972-4-829-4359. Fax: +972-4-822-1128.

obtained for several important classes of access graphs, including *complete graphs* and *directed acyclic graphs (DAG)*. Finally, in some natural applications the CLPP problem can be modeled as a Markovian process on branch trees. For such applications we give algorithms whose expected performance ratios are within factor 2 from the optimal.

Index Terms: Caching, pipelined prefetching, access graphs, markovian access models, online algorithms.

1 Introduction

1.1 Problem Statement

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. [6], 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 and the cache can hold $k < m$ blocks. An access to a block in the cache incurs one time unit, and fetching a missing block incurs d time units. While accessing a block in the cache, the system can fetch a block from secondary storage, either in response to a cache miss (*caching by demand*), or before it is referenced, in anticipation of a miss (*prefetching*); at most one fetch can be in progress at any given time. The *Caching with Prefetching (CP)* problem is to determine the sequence of block evictions/prefetches, so as to minimize the overall time required for accessing all blocks.

Motivated by the operation of modern memory controllers, and from program execution on fast processors, we consider the problem of caching integrated with *pipelined* prefetching. Here, a fetch of a new block can be initiated while a previous fetch is still in progress. Thus, d block fetches can be in progress simultaneously.

The locality of reference in memory access patterns of real programs is captured by the *access graph* model of Borodin et al. [4]; thus, we assume that any sequence of block references is a walk on an access graph, G. As before, our measure is the overall execution time of a given reference sequence.

Formally, suppose that a set of n data blocks $b_1, b_2, \ldots, b_n$ is held in secondary storage. The access graph for the program that reads/writes into $b_1, b_2, \ldots, b_n$, is given by a directed graph $G = (V, E)$, where each vertex corresponds to a block in this set. Any sequence of block references has to obey the locality constraints imposed by the edges of G: following a request to a block (vertex) u, the next request has to be either to block u or to a block v, such that $(u, v) \in E$. Pipelined prefetching allows to initiate a prefetch one time unit after the previous prefetch.

Given a reference sequence $\sigma = \{r_1, \ldots, r_m\}$, r_i can be satisfied immediately at time t, incurring one time unit, if r_i is in the cache; otherwise, if a prefetch of r_i was initiated by an algorithm $\mathcal{A}$ at time $t_i \leq t$, there is a *stall* for $d_{\mathcal{A}}(r_i) = d - (t - t_i)$ time units. The total execution time of σ is the time to access the m blocks plus the stall time, i.e., $time(\mathcal{A}, \sigma) =$

$m + \sum_{i=1}^{m} d(r_i)$. 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 $\sigma = \{r_1, \ldots, r_m\}$, find an algorithm $\mathcal{A}$ for pipelined prefetching and caching, such that $time(\mathcal{A}, \sigma)$ is minimized.

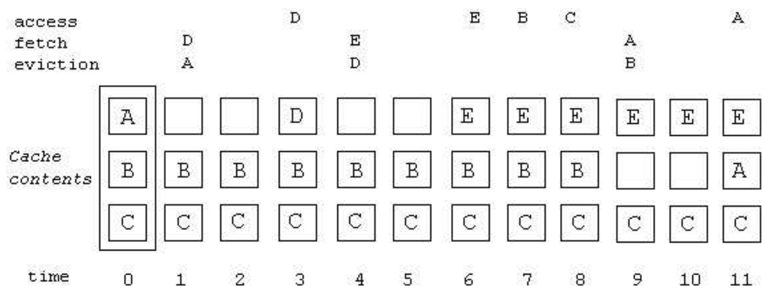

Figure 1: An example of caching ($k = 3$ and $d = 2$)

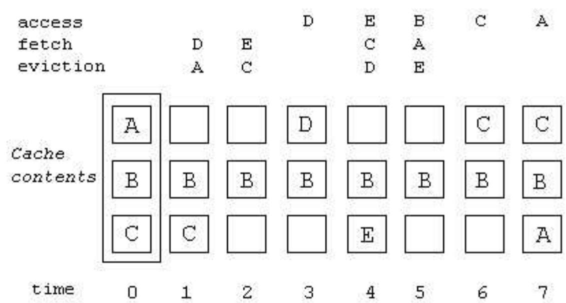

Figure 2: An example of caching with pipelined prefetching, using Algorithm AGG ($k = 3$ and $d = 2$)

Example 1.1 *Consider a program whose block accesses are given by the sequence "DEBCA" (the entire sequence is known at time* 0*). The cache can hold three blocks; fetching a block takes two time units. Initially A,B and C are in the cache. Figure 1 shows the execution of the optimal caching-by-demand algorithm [5] that incurs* 11 *time units*[1]*. Figure 2 shows an execution of the* Aggressive (AGG) *algorithm (see Section 3), which combines caching with pipelined prefetching; thus, the reference sequence is completed within* 7 *time units, which is optimal.*

[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.

The above example suggests that pipelined prefetching can be helpful. When future accesses are *fully known*, the challenge of a good algorithm is to achieve maximum overlap between accesses to blocks in the cache and fetches. Without this knowledge, achieving maximum overlap may be harmful, due to evictions of blocks that will be requested in the near future. Thus, more careful decisions need to be made, based on the structure of the underlying access graph.

1.2 Our Results

We study the CLPP problem both in the *offline* case, where the sequence of cache accesses is known in advance, and in the *online* case, where r_{i+1} is revealed to the algorithm only when r_i is accessed. In the offline case (Section 3), we show that algorithm Aggressive (AGG) introduced in [6] is optimal, for *any* access graph and any $d, k \geq 1$.

In the online case (Section 4), we give an algorithm which is within factor of 2 from the optimal in the set of online deterministic algorithms, for *any* access graph, and $k, d \geq 1$. Better ratios are obtained (in Section 5) for several important classes of access graphs, including *complete graphs* and *directed acyclic graphs (DAG)*. In particular, for complete graphs we obtain a ratio of $1 + 2/k$, for DAGs $\min(2, 1 + k/d)$, and for branch trees $1 + o(1)$.

In Section 6 we study the CLPP problem assuming a Markovian access model on branch trees, which often arise in applications. We give algorithms whose expected performance ratios are within factor 2 from the optimal for general trees, and $(1 + o(1))$ for homogeneous trees. In Section 7 we discuss the extension of our results to Markovian CLPP on branch trees with a *bounded* number of leaves. Finally, in Section 8 we summarize the contribution of this work and discuss some open problems for future study.

For deriving upper bounds on the competitive ratios of our algorithms, we develop (in Section 4) a general proof technique, that we use also for special classes of access graphs (in Section 5), and for the Markovian CLPP (in Section 6). The technique relies on comparing a *lazy* version of a given online algorithm to an optimal algorithm, which is allowed to use *parallel* (rather than *pipelined*) prefetches. The technique may be useful for tackling other problems in which *pipelined* service is granted to a set of requests in an online fashion.

1.3 Applications of the CLPP

Main memory controllers: Pipelined prefetching of data blocks is combined with caching by main memory controllers. For example, the RDRAM memory system, recently developed by Rambus [20] has the extended ability to process new data requests while accessing data in the CPU cache. The memory *throughput* is then the number of data requests that can be handled in parallel. Suppose that the CPU cache size is k and the memory throughput is d. Then the problem of finding an optimal protocol for the management of the CPU cache by the controller yields an instance of the CLPP.

Program execution on fast processors: Modern processors use pipelining for speeding up the execution of programs. Thus, in RISC architectures the execution of each instruction is partitioned into several steps: the processor starts executing the next instruction as soon as the previous instruction has completed its first step. The online nature of this process comes from the presence of *branch* instructions. Indeed, when the program starts the execution of a branch, the next instruction to be executed is known only when that branch is resolved.

The problem of fetching instructions, with the objective of keeping the pipeline as full of useful instructions as possible, can be abstracted to *adaptive path selection* in a *branch tree* [19], a binary out-tree that represents the set of the execution paths of a program (each internal vertex represents an instance of a conditional branch). At any time, the processor holds in the pipe a subtree of instructions; the selection of the next instruction to be fetched during the execution of a branch is done by a predictor. Thus, the path selection in the tree is generated by a Markov chain. Adaptive path selection 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 problem in Section 6.

The usage of speculative execution of code typically results in the simultaneous (tentative) execution of multiple paths in the program. This makes the processor design more complicated. A main factor in the added complexity is the number of prefetched paths which can be held at any time in the cache: this is the number of *leaves* of the subtree of instructions fetched into the pipe. We discuss the resulting variant of the CLPP problem in Section 7.

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 in [26]. Various switching hubs were developed for Ethernet, Fast Ethernet and ATM. For example, in ATM networks, fixed size cells of 53 bytes are switched from input ports to output ports. ATM can also *pipeline* data, i.e., a new packet can be sent from a source to its destination before the previous one has arrived.

Suppose that an application running on a client's machine requests data from the host. Once the connection between the client and the host is established, packages containing 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 data requests can be described as an instance of the CLPP, in which k is the size of the local cache at the client and the delay is $d = 2S$.

1.4 Related Work

The concept of cooperative prefetching and caching was first investigated by Cao et al. [6]. The paper studies offline prefetching and caching algorithms, where fetches are *serialized*, i.e., at most one fetch can be in progress at any given time. An algorithm called Aggressive (AGG) was shown to yield a $min(1+d/k, 2)$-approximation to the optimal. Kimbrel and Karlin [14] extended this study to storage systems which consist of r units (e.g., an array of r disks); fetches are serialized on each storage unit, thus, up to r block fetches can be processed in parallel. The paper gives performance bounds for several offline algorithms in this setting. Algorithm AGG was shown to achieve a ratio of $(1 + rd/k)$ to the optimal.

Albers et al. showed in [1] that the CP problem can be optimally solved using linear programming. Recently, Albers and Witt presented in [2] an optimal combinatorial algorithm. The papers [1, 2] give also approximation algorithms for the problem of minimizing the overall *stall time* in a system that consists of r units, for any $r > 1$.

Other papers (see e.g. [21, 24]) present experimental results for cooperative prefetching and caching, in the presence of optional program-provided hints of future accesses.

Note that the classic paging problem, where the cost of an access is zero and the cost of a fault is 1, is a special case of our problem, in which $d \gg 1$.[2] There is a wide literature on

[2]Thus, when normalizing (by factor d) we get that the delivery time equals to one, while the access time,

the caching (paging) problem. (Comprehensive surveys appear in [5, 12, 18, 7].) Borodin et al. [4] introduced the *access graph* model. The paper presents an online algorithm (called *FAR*) that is strongly competitive on any access graph. Later works (e.g., [13, 8, 10]) consider extensions of the access graph model, or give experimental results for some heuristics for paging in this model [9].

Karlin et al. [15] introduced 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 page u to the reference to page v (both represented as vertices 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 Markovian CLPP, on branch trees in which $k > 1$ and $d = k - 1$. We examine here two of the algorithms proposed in these works, the algorithms Eager Execution (EE) [3, 23] and Disjoint Eager Execution (DEE) [23]. EE was shown to perform well in practice, however, no theoretical performance bounds were derived. Raghavan et al. [19] showed, 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. We improve this result, and show (in Section 6.2) that DEE is nearly optimal on branch trees, for a large set of values of $p \in [1/2, 1]$.

2 Preliminaries

The set of reference sequences of a program to data blocks is restricted, i.e. some patterns may not occur. We model these reference sequences as paths in an *access graph* G. Namely, G is a directed graph whose vertex set is the set of blocks, and every reference sequence $\sigma = (r_1, \ldots, r_{|\sigma|})$ forms a path in G. We allow consecutive accesses to a block b_j by adding a self-loop to the corresponding vertex in G.

Denote by *Paths*(G) the set of paths in G. Let OPT be an optimal offline algorithm for CLPP. We refer to an optimal offline algorithm and the source of requests together as the *adversary*, who generates the sequence and serves the access requests offline. We use competitive analysis (see e.g. [5]) to establish performance bounds for online algorithms for our problem.

$1/d$, asymptotically tends to 0.

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

$$c_{\mathcal{A}}(G,k,d) = \sup_{\sigma \in Paths(G)} \frac{time(\mathcal{A},\sigma)}{time(OPT,\sigma)} \ ,$$

where $time(\mathcal{A},\sigma)$, $time(OPT,\sigma)$ are the times required by $\mathcal{A}$ and OPT, respectively, to execute σ.

We abbreviate the formulation of our results using the following notation. Let $c(G,k,d)$ be the competitive ratio of an optimal online algorithm for CLPP on an access graph G, for fixed k and d; if d is omitted, then d can be arbitrary. We say that $\mathcal{A}$ is *strongly competitive*, if $c_{\mathcal{A}}(G,k,d) = O(c(G,k))$. The superscript *det* restricts the set of algorithms to deterministic algorithm, e.g., $c^{det}(G,k,d)$ is the competitive ratio of an optimal deterministic online algorithm on G, for a cache of size k and delay d.

In the Markovian CLPP, the probability to access a vertex v depends only on the current block u. Namely, it is equal to the transition probability $p_{u,v}$. The *accumulated probability* of path $\pi = (v_0 = r, v_1, \ldots, v_n = v)$ is given by

$$p_a(\pi) = \prod_{i=1}^{n} p_{v_{i-1},v_i}. \tag{1}$$

The *expected performance ratio* of an algorithm $\mathcal{A}$ on a graph G, for fixed k and d, is

$$\bar{c}_{\mathcal{A}}(G,k,d) = \sum_{\sigma \in Paths(G)} Pr(\sigma) \cdot \frac{time(\mathcal{A},\sigma)}{time(OPT,\sigma)} \ .$$

Thus, in Definition 2.1, the expected performance ratio replaces competitive ratio.

Finally, an access graph G is called a *branch tree*, if G is an ordered binary out-tree in which every internal vertex has a *left* child and a *right* child. In the Markovian CLPP for a branch tree, all paths start at the root r; the *local probability* of a vertex v is $p_{u,v}$— the transition probability from its parent u; and the *accumulated probability* of v is the accumulated probability of the unique path from r to v.

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. The next lemma shows that a set of rules formulated in [6], to characterize the behavior of optimal algorithms for the CP problem, applies also for the offline CLPP problem.

Lemma 3.1 *[No harm rules] There exists an optimal algorithm $\mathcal{A}$, which satisfies the following rules : (i) $\mathcal{A}$ fetches the next block in the reference sequence that is missing in the cache; (ii) $\mathcal{A}$ evicts the block whose next reference is furthest in the future. (iii) $\mathcal{A}$ never replaces a block B by a block C, if B will be referenced before C.*

Proof: We show by induction that any algorithm $\mathcal{A}$ can be transformed to an algorithm $\mathcal{A}'$ 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 sequence, and the three rules are satisfied.

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

(i) **Rule 1:** Suppose that $\mathcal{A}$ follows rule 1 in the first $(i-1)$ references. Indeed, if $\mathcal{A}$ does not initiate a fetch in step i, then rule 1 continues to hold. Suppose that $\mathcal{A}$ fetches block r_n and discards 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 $r_l \neq r_n$ (see in Figure 3).

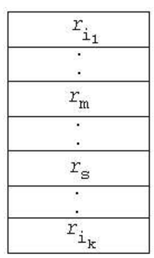

Figure 3: The cache contents of $\mathcal{A}$, $\mathcal{A}'$ after step $(i-1)$ in the proof of Lemma 3.1, $(i), (ii)$).

We define an algorithm $\mathcal{A}'$ that follows rule 1 in step i, without increasing the overall execution time, i.e. in step i $\mathcal{A}'$ fetches r_l. Thus, after step i the cache contents of $\mathcal{A}$ and $\mathcal{A}'$ differ in one block. Now, $\mathcal{A}'$ will take the same actions as $\mathcal{A}$, until one of the following occurs.

(a) $\mathcal{A}$ fetches r_l, then $\mathcal{A}'$ fetches r_n.

(b) $\mathcal{A}$ discards r_n, then $\mathcal{A}'$ discards r_l.

Note that at least (a) has to occur before the reference to r_n, since $\mathcal{A}$ has to fetch r_l to the cache. After (a) or (b) occurs, we get that the cache contents of $\mathcal{A}$ and $\mathcal{A}'$ are identical. Thus, $\mathcal{A}'$ does not incur an 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 $\mathcal{A}$ satisfies rules 1 and 2 in the first $(i-1)$ steps. As before, we need to consider only the case where the contents of $\mathcal{A}$'s cache change in step i, i.e., $\mathcal{A}$ evicts a block. Assume again that the reference sequence is σ_1: $\mathcal{A}$ violates rule 2 by discarding r_m rather than r_s, which is referenced later. Then $\mathcal{A}'$ discards r_s and acts like $\mathcal{A}$, until one of the following occurs:

(a) $\mathcal{A}$ discards r_s and fetches some $r_f \neq r_m$, then $\mathcal{A}'$ fetches r_f and discards r_m;

(b) $\mathcal{A}$ fetches r_m and discards r_d. If $r_d \neq r_s$, then $\mathcal{A}'$ fetches r_s, and discards r_d; otherwise the contents of $\mathcal{A}'$'s cache remain unchanged.

We note, that at least (b) has to occur before the next reference to r_s, therefore $\mathcal{A}'$ will not incur an extra miss on that block.

(iii) **Rule 3:** Assume now that $\mathcal{A}$ 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 $\mathcal{A}$ initiated a fetch of r_n and discards r_m, that is referenced before r_n; $\mathcal{A}'$ does no fetch in step i; then, $\mathcal{A}'$ operates like $\mathcal{A}$ until either

(a) $\mathcal{A}$ fetches r_m and discards some block $r_d \neq r_n$: then $\mathcal{A}'$ discards r_d and fetches r_n, or

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

(c) $\mathcal{A}$ discards r_n and fetches r_m, then the contents of $\mathcal{A}'$'s cache remain unchanged.

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

■

In the remainder of this section we consider only optimal algorithms that follow the "no harm" rules. Clearly, once an algorithm $\mathcal{A}$ decides to fetch a block, these rules uniquely define the block that should be fetched, and the block that will be evicted. Thus, the only decision to be made by any algorithm is *when* to start the next fetch.

Algorithm AGG, proposed by Cao et al. [6], follows the "no harm" rules; in addition, it fetches each block at the *earliest* opportunity, i.e., whenever 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 2.) As stated in our next result, this algorithm is the best possible for the CLPP.

Theorem 3.2 AGG *is an optimal offline algorithm for the CLPP problem.*

Proof: We show by induction, that for $i \geq 1$, any optimal offline algorithm $\mathcal{A}$ which satisfies the "no harm" rules, can be modified to act like AGG in the first i steps, without harming $\mathcal{A}$'s optimality.

Base: Assume that the cache is initially empty. Then, both $\mathcal{A}$ and AGG fetch r_1.

Induction Step: Assume that $\mathcal{A}$ acts like AGG in the first $(i-1)$ steps. We distinguish between two cases in the ith reference:

(i) $\mathcal{A}$ initiates a fetch of some block r_l. Then since $\mathcal{A}$ satisfies the "no harm" rules, and AGG fetches in the first opportunity, clearly, AGG acts in step i like $\mathcal{A}$ (r_l is missing in AGG's cache, and can be fetched).

(ii) $\mathcal{A}$ does not initiate a fetch, but AGG does. Specifically, suppose that AGG fetches r_n and discards r_m. We define an algorithm $\mathcal{A}'$ that operates like AGG in step i, and then proceeds like $\mathcal{A}$, until one of the following occurs:

  (a) $\mathcal{A}$ fetches r_n and discards r_d. If $r_d \neq r_m$, then $\mathcal{A}'$ fetches r_m and discards r_d; otherwise the contents of $\mathcal{A}'$'s cache remain unchanged.

  (b) If $\mathcal{A}$ discards r_m and fetches $r_f \neq r_n$, then $\mathcal{A}'$ 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). Thus, the cost of $\mathcal{A}'$ is equal to that of $\mathcal{A}$, and $\mathcal{A}'$ acts like AGG in the first $i+1$ steps.

■

The greediness of AGG plays an important role when $d < k$, as shown in the next result.

Corollary 3.3 *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 $r_{i+1}, \ldots, r_{i+d-1}$ 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 need to handle two cases:

(i) If at the time r_i is accessed, r_{i+d} is either 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 neither being fetched nor in the cache, then r_{i+d} is the first missing block (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, since $d < k$, there exists in the cache at least one block r_e, whose next reference is after r_{i+d}; AGG will discard block r_e and initiate a fetch of r_{i+d}. Hence the claim holds also after the ith reference.

■

4 The Online CLPP Problem

Let $G = (V, E)$ be an access graph. Suppose that $\sigma = (r_1, \ldots, r_\ell)$, $1 \leq \ell \leq \min\{k, d\}$, is a reference sequence to blocks. Recall that σ is a path in G and, except for the first block r_1, is unknown.

Suppose that r is the current referenced block, and $V' \subseteq V$ is the set of blocks in the cache. If s is the length of the shortest path from r to a *hole*, that is, a vertex v not in V', then provided the content of the cache is not changed, there will be no cache miss for at least s steps. We therefore define $B(r, V')$— the *benefit* of V'—as the distance from r to the closest hole.

In our attempt to construct a competitive algorithm for CLPP, we first consider algorithms that select a set of ℓ vertices to put in the cache, based solely on the initial reference r_1 and the access graph G. We call this problem the *Single Phase CLPP* (*S_CLPP*) problem. Given a reference sequence σ and the subgraph $G_{\mathcal{A}} \subseteq G$ selected by the algorithm $\mathcal{A}$, we denote by $PREF_\sigma(G_{\mathcal{A}})$ the maximal set of vertices in $G_{\mathcal{A}}$ that form a *prefix* of σ. Given that the first request in σ is r_1, an algorithm $\mathcal{A}$ is *optimal* if $\min_{\sigma \in Paths(G), |\sigma|=\ell} |PREF_\sigma(G_{\mathcal{A}})|$ is maximal. Note that maximizing the last expression is equivalent to maximizing the distance from r_1 to a hole, i.e. $B(r_1, G_{\mathcal{A}})$.

Algorithm AGG_{OL} below mimics the operation of AGG in an online fashion. Denote by $dist(v,u)$ the length of the shortest path from v to u in G.[3] Let IN_t denote the blocks that are in the cache or are being fetched at step t, and $OUT_t = V \setminus IN_t$. The following is a pseudocode description for AGG_{OL}.

Algorithm AGG $_{\text{OL}}$

for $t = 1, \ldots, \ell$ do
  Let $u = \arg\min\{dist(r_t, v) : v \in OUT_t\}$.
  Let $w = \arg\max\{dist(r_t, v) : v \in IN_t\}$.
  If $dist(r_t, u) < dist(r_t, w)$
    Evict w from the cache and initiate a fetch for u.

Lemma 4.1 *Algorithm* AGG_{OL} *is optimal in the set of deterministic online algorithms for the S_CLPP problem, for any graph G, and $k, d \geq 1$.*

Proof: Note that since $\ell \leq d$, in the S_CLLP problem we need to select a subset V' before knowing any of the vertices $r_2, \ldots, r_\ell$. Since AGG_{OL} always selects the hole that is closest to r_1, $G_{\mathcal{A}}$ maximizes the length of the shortest path from r_1 to a hole. Hence, $B(r_1, G_{\mathsf{AGG}_{\text{OL}}})$, the benefit of AGG_{OL}, is maximum. ∎

The above lemma facilitates the derivation of our main result for the online CLPP.

Theorem 4.2 *For any graph G, and $k, d \geq 1$ there exists an algorithm, $\mathcal{A}$, such that*

$$c_{\mathcal{A}}(G,k,d) \leq 2 \cdot c^{det}(G,k,d) \ .$$

Proof: We compare two algorithms, Algorithm $\mathsf{Lazy\text{-}AGG}_{\text{OL}}$ that divides reference sequence to phases and solves in each phase the S_CLPP, and Algorithm $\mathsf{Par\text{-}AGG}_{\text{OL}}$ which

[3]When u is unreachable from v $dist(v,u) = \infty$.

uses parallelism to outperform any deterministic online algorithm. The ratio between the performance of these algorithms constitutes a bound on the performance ratio for the problem.

Consider first Algorithm $\mathsf{Lazy\text{-}AGG}_{\text{OL}}$ which operates in phases. Phase i starts at some time t_i, with a stall of d time units, for fetching a missing block—r_i. Each phase is partitioned into sub-phases. Let $t_{i,j}$ be the start time of sub-phase j. The first sub-phase of phase i starts at time $t_{i,1} = t_i$. At sub-phase j, $\mathsf{Lazy\text{-}AGG}_{\text{OL}}$ invokes Algorithm AGG_{OL} to select a subset, $V_{i,j}$, of $\ell = \min(d, k)$ vertices. Some of these vertices (=blocks) are already in the cache: $\mathsf{Lazy\text{-}AGG}_{\text{OL}}$ initiates pipelined fetching of the remaining blocks. Let $r_{i,j}$ be the block that is accessed first in sub-phase j of phase i, and let $\sigma^d_{i,j}$ be the sequence of the first d block accesses in sub-phase j. We denote by

$$Good(i,j) = \sigma^d_{i,j} \cap V_{i,j}$$

the maximal set of blocks among those that are in the cache or being fetched at time $t_{i,j} + d$, that forms a prefix of $\sigma^d_{i,j}$. Let $g_j = |Good(i,j)|$. We handle separately two cases:

- If $g_{i,j} = d$, then $\mathsf{Lazy\text{-}AGG}_{\text{OL}}$ waits until d blocks were accessed in the cache; at time $t_{i,j} + 2d$ the j-th sub-phase terminates, and $\mathsf{Lazy\text{-}AGG}_{\text{OL}}$ starts sub-phase $j+1$ of phase i.

- If $g_j < d$ then at time $t_{i,j} + d + g_j$ phase i terminates and the first missing block in the cache becomes r_{i+1}.

Consider now Algorithm $\mathsf{Par\text{-}AGG}_{\text{OL}}$ that operates like $\mathsf{Lazy\text{-}AGG}_{\text{OL}}$, except that $\mathsf{Par\text{-}AGG}_{\text{OL}}$ has the advantage that in each sub-phase, j, all the prefetches are initiated in parallel and d time units after this sub-phase starts $\mathsf{Par\text{-}AGG}_{\text{OL}}$ knows the value of g_j and the first missing block in the cache. As in $\mathsf{Lazy\text{-}AGG}_{\text{OL}}$, if $g_j = d$ then $\mathsf{Par\text{-}AGG}_{\text{OL}}$ proceeds to the next sub-phase of phase i; if $g_j < d$ phase i terminates. Note that combining Lemma 4.1 with the parallel fetching property we get that $\mathsf{Par\text{-}AGG}_{\text{OL}}$ outperforms any deterministic online algorithm.

To compute $c_{\mathsf{Lazy\text{-}AGG}_{\text{OL}}}(G,k,d)/c^{det}(G,k,d)$ it suffices to compare the length of phase i of $\mathsf{Lazy\text{-}AGG}_{\text{OL}}$ and $\mathsf{Par\text{-}AGG}_{\text{OL}}$, for any $i \geq 1$. Suppose that there are $sp(i)$ sub-phases in phase i. For $\mathsf{Lazy\text{-}AGG}_{\text{OL}}$ each of the first $sp(i) - 1$ sub-phases incurs $2d$ time units, while the last sub-phase incurs $d + g$ time units, for some $1 \leq g < d$. For $\mathsf{Par\text{-}AGG}_{\text{OL}}$ each sub-phase (including the last one) incurs d time units; thus, for any sequence σ we get the ratio

$$\frac{c_{\mathsf{Lazy\text{-}AGG_{OL}}}(G,k,d)}{c^{det}(G,k,d)} \leq \frac{\mathrm{time}(\mathsf{Lazy\text{-}AGG_{OL}},\sigma)}{\mathrm{time}(\mathsf{Par\text{-}AGG_{OL}},\sigma)} \leq \frac{d+g+(sp(i)-1)2d}{d\cdot sp(i)} \leq 2\ .$$

■

5 Online CLPP on DAGs and Complete Graphs

5.1 Directed Acyclic Access Graphs

Consider now the subclass of DAGs.[4] Our next result improves the bound in Theorem 4.2, in the case where $k < d$.

Theorem 5.1 *If G is a DAG then for any cache size $k \geq 1$ and delivery time $d \geq 1$,*

$$c_{\mathsf{Lazy\text{-}AGG_{OL}}}(G,k,d) \leq \min(1+k/d,2)\cdot c^{det}(G,k,d)\ .$$

Proof: Note that in the proof of Theorem 4.2 we showed that $c_{\mathsf{Lazy\text{-}AGG_{OL}}}(G,k,d) \leq 2\cdot c^{det}(G,k,d)$.

When $k < d$ we can improve this ratio. In this case each phase consists of a single sub-phase. Indeed, since G is a DAG (i.e., no self-loops) consecutive accesses to same block cannot occur. More precisely, each block can be accessed at most once, along the execution of the program. It follows that in each phase both $\mathsf{Par\text{-}AGG_{OL}}$ and $\mathsf{Lazy\text{-}AGG_{OL}}$ have at most $k < d$ 'good' blocks in the cache (i.e., $g_1 < d$), and phase i terminates. The ratio between the length of phase i for $\mathsf{Lazy\text{-}AGG_{OL}}$ and $\mathsf{Par\text{-}AGG_{OL}}$ is then at most $(d+k)/d$. This completes the proof. ■

5.1.1 Branch Trees

The case where G is a branch tree and $d = k-1$ is of particular interest in the application of CLPP to pipeline execution of programs on fast processors (see Section 1.3). For this case we show that the bound in Theorem 5.1 can be further improved. Specifically, we show that the competitive ratio of $\mathsf{Lazy\text{-}AGG_{OL}}$ is within factor $1+o(1)$ of the optimal in the set of online (deterministic or randomized) algorithms on branch trees.

[4] Note that on this subclass of graphs $\mathsf{AGG_{OL}}$ acts exactly like algorithm EE studied in [3, 23].

Theorem 5.2 *If G is a branch tree then*

$$c_{\mathsf{Lazy\text{-}AGG_{OL}}}(G,k,k-1) \le (1+o(1))c(G,k,k-1),$$

where the $o(1)$ term refers to a function of k.

For the proof we need the following lemma.

Lemma 5.3 *If G is a branch tree then $c(G,k,k-1) \ge k/\lg k$.*

Proof: We derive 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 [28].

Assume that $d = k-1$. Suppose that the tree T is rooted at r. The adversary generates the reference sequence σ as follows. At vertex i, the adversary proceeds to the left child with probability 1/2. Let v be a vertex in T, and denote by $depth(v)$ the depth of v in T. (The depth of r_1 is 0). Recall that the accumulated probability of v, $p_a(v)$, is the probability that the adversary selects v for σ. Obviously, in our case this probability depends on $depth(v)$ and is equal to

$$p_a(v) = \frac{1}{2^{depth(v)}}$$

Now, we allow the online algorithm, $\mathcal{A}$, to start fetching the first k blocks at time $t = 0$ (rather than one block per time unit); then, $\mathcal{A}$ waits for k steps and starts fetching another set of blocks at time k. Thus, $\mathcal{A}$ solves in each phase the S_CLPP.

Recall that in the S_CLPP the goal of $\mathcal{A}$ is to maximize $PREF_\sigma(T_\mathcal{A})$, where $T_\mathcal{A} \subseteq T$ is the subtree selected by $\mathcal{A}$. Consider an algorithm $\mathcal{A}_{bal}$, that proceeds as follows. First, $\mathcal{A}_{bal}$ sorts the vertices in T in decreasing order by their accumulated probabilities, and then fetches the first k vertices in the list. In our case, $\mathcal{A}_{bal}$ takes a balanced subtree of k vertices, rooted at r.

We now calculate the expected benefit of any online algorithm. Let v_j be the jth vertex fetched to the cache. Then the expected benefit of $\mathcal{A}$ from selecting the above subtree is

$$\bar{B}(r, T_\mathcal{A}) = \sum_{j=1}^{k} p_a(v_j) = \sum_{j=1}^{k} \left(\frac{1}{2}\right)^{depth(v_j)} .$$

We note that $\mathcal{A}_{bal}$ maximizes this value, since it selects k vertices with the highest probabilities. Hence, the expected benefit of any online algorithm is bounded by the height of a balanced tree of k vertices, that is, $\lg k$.

Any optimal offline algorithm will incur a miss on the first reference, r_1 ($=r$), while any other reference costs one time unit. Hence, its total cost is $|\sigma| + d$, while $\mathcal{A}$'s expected cost is at least $|\sigma|k/\lg k$. ■

Proof of Theorem 5.2: Consider the following simple variant of $\mathsf{Lazy\text{-}AGG}_{\mathsf{OL}}$: in phase i fetch into the cache a complete binary tree of size $k' = k/\lg k$; then, stall for k time units. This algorithm incurs an average cost of $k + k'$ for accessing $\lg k'$ blocks.

$$c_{\mathsf{Lazy\text{-}AGG}_{\mathsf{OL}}}(G, k, k-1) \leq \frac{k' + k}{\lg k'} = \frac{k(1 + 1/\lg k)}{\lg k - \lg\lg k} = \frac{k}{\lg k}(1 + o(1)) \ .$$

Using Lemma 5.3 we get the statement of the theorem. ■

5.2 Complete Graphs

Suppose that G is a complete graph, i.e., G contains the edges (u, v) and (v, u) between every two vertices u and v. We first show, that on these graphs the lower bound obtained for deterministic algorithms for the classic paging problem remains valid for the CLPP.

Theorem 5.4 *If G is a complete graph, then for any cache size $k \geq 1$*

$$c^{det}(G, k) \geq k - 1 \ . \tag{2}$$

Proof: Note that it is sufficient to show that there exists some $d \geq 1$ for which $c^{det}(G, k, d) \geq k - 1$. We take $d = k - 1$. Let $|V| = k + 1$. Recall that when $d < k$, any optimal offline algorithm stalls for d time units in the first reference, and any other reference incurs cost one. Let $\mathcal{A}$ be an online deterministic algorithm. For any $t \geq 1$, if $\mathcal{A}$ accesses at time t some block, b_j, then at most $(k-1)$ other blocks can be available in the cache or in the process of being fetched. Hence, there exists a block, r_f, that is neither in the cache nor being fetched; r_f will be requested at time $t + 1$ and incur a stall of $k - 1$ time units. The same holds for any block that is being fetched by $\mathcal{A}$ to the cache. Thus, we can construct a sequence in which $\mathcal{A}$ stalls in each reference for $d = k - 1$ time units. ■

In the following we show that the lower bound derived in Theorem 5.4 cannot be substantially improved: the ratio in (2) can be achieved, to within an additive factor of 2 by a caching-by-demand algorithm.

Consider the set of *marking* algorithms proposed for the classical caching (paging) problem (see, e.g., [4]). 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 cache 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. At this point all the blocks become unmarked, and a new phase begins.

Lemma 5.5 *For any access graph G, cache size k and delivery time $d \geq 1$, if $\mathcal{A}$ is a marking algorithm, then $c_{\mathcal{A}}(G, k, d) \leq k + 1$.*

Proof: Let $\mathcal{A}$ 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 calculate the cost incurred by an optimal offline algorithm, OPT, for the execution of phase j of $\mathcal{A}$. Each phase contains accesses to $k + 1$ distinct blocks; thus, any algorithm (including 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, OPT needs at least $\max(n_j, d)$ steps to complete the execution of phase j of $\mathcal{A}$.

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

$$c_{\mathcal{A}}(G, k, d) \leq \frac{n_j + kd}{\max(n_j, d)} = \frac{n_j}{\max(n_j, d)} + \frac{kd}{\max(n_j, d)} \leq 1 + k \ .$$

∎

From Theorems 5.4 and 5.5 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.

Theorem 5.6 *On a complete graph, any marking algorithm is within factor $1 + 2/k$ from the optimal in the set of online deterministic algorithms for CLPP.*

6 The Markovian CLPP on Branch Trees

The following algorithm, known as DEE, is a natural greedy algorithm for the S_CLPP in the Markovian model. As before, suppose that the parameter is some $\ell \geq 1$.

Algorithm DEE
for $t = 1, \ldots, \ell$ do
  Fetch a missing block that from the current position
    is *most likely* to appear in σ;

Discard a block that is unreachable from the current position.

An example of the execution of DEE is given in Figure 6. In this section we analyze the performance of DEE on branch trees, and show that its expected performance ratio is at most 2. This ratio is reduced to $(1+o(1))$ for a special class of Markov chains that we call *homogeneous* (see Section 6.2).

6.1 Performance of DEE on Branch Trees

Let T be a branch tree, and $k \geq 1$ an integer. Suppose that σ is a reference sequence of length k or larger. In the *Markovian S_CLPP* we need to choose a subset of vertices $T_{\mathcal{A}}$ of T of size k, such that the *expected* size of $PREF_\sigma(T_{\mathcal{A}})$ is maximal. Formally, for a tree T rooted at r, let

$$\bar{B}(r, T_{\mathcal{A}}) = \sum_{\sigma \in Paths(T)} Pr(\sigma) |PREF_\sigma(T_{\mathcal{A}})| \tag{3}$$

be the *expected benefit* of an online algorithm $\mathcal{A}$ from $T_{\mathcal{A}}$. We seek an algorithm, $\mathcal{A}$, whose expected benefit is maximal. The next result, given in [19], shows that DEE is optimal in the set of online algorithms for the Markovian S_CLPP.

Lemma 6.1 *([19]) Given a branch tree T and $k \geq 1$, for any online algorithm $\mathcal{A}$*

$$\bar{B}(r, T_{\mathsf{DEE}}) \geq \bar{B}(r, T_{\mathcal{A}}) \ .$$

We proceed to obtain a performance bound for DEE, when applied for the CLPP.

Theorem 6.2 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 5.1. We define two algorithms Par-DEE and Lazy-DEE that operate in phases. Par-DEE selects in each phase the k blocks 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 Par-DEE the first block not in this set. By Lemma 6.1, Par-DEE outperforms any on-line algorithm.

Lazy-DEE operates as follows. In each phase, it selects the blocks selected by DEE in the first k steps, and then stalls for k steps. Thus, Lazy-DEE starts a new phase every $2k$ steps, while Par-DEE starts every k steps. This yield the desired ratio of 2. ∎

6.2 Homogeneous Branch Trees

We call a branch tree T *homogeneous*, if all the left children in T have the *same* local probability, $p \in [1/2, 1)$. In other words, the transition probabilities from any vertex u to its left child is p and to its right child is $1-p$, (otherwise we can take $p' = 1-p$, and switch the right with the left child in each vertex).

In the following we derive an explicit asymptotic expression for the expected benefit of DEE, when solving the Markovian S_CLPP on a homogeneous branch tree, with any local probability $1/2 \leq p < 1$. Our computations are based on a Fibonacci-type analysis, which suits well the homogeneous case. For any integer $q \geq 2$, the n-th number of the *q-distance Fibonacci sequence* is given by

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

This sequence can be viewed as a special case of the q-order Fibonacci sequence, given by $g(n) = \sum_{j=1}^{q} a_j g(n-j)$ (see, e.g., [16, 22]). Note that with $q = 2$, we get the well known Fibonacci numbers [16].

Lemma 6.3 *For any $n \geq 1$ and a given $q \geq 2$,*

$$g(n) = b_q x_q^n \left(1 + o(1)\right), \tag{4}$$

where

$$b_q = \frac{1}{q{x_q}^{q-1} - (q-1){x_q}^{q-2}} \tag{5}$$

and $1 < x_q \leq 2$ is the single real root of the polynomial

$$x^q - x^{q-1} - 1 \ . \tag{6}$$

We give the proof in Appendix B.

Lemma 6.4 *Let $\frac{1}{2} \leq p < 1$ satisfy that for some natural $q \in \mathbb{N}$*

$$p^q = 1 - p \tag{7}$$

and let

$$\alpha \ = \ (1-p)q + p \tag{8}$$

Then for some $\phi(p,k) \in [0,1]$ and $\delta \in \{0,1\}$, the height of the subtree chosen by DEE *for the Markovian S_CLPP with parameter k is given by*

$$height(T_{\mathsf{DEE}}) = \log_{1/p}((1-p)\alpha k + p) - \phi(p,k) + \delta + o(1). \tag{9}$$

The $o(1)$ term refers to a function of k.

Proof: Let q be defined as in (7), 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:

(i) For $1 \leq n < q$, since $p^n > 1-p$, there is a single vertex with accumulated probability p^n. This vertex is reached by proceeding from the root of T on a path of length n, such that in each vertex we choose the left child.

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

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

Then, we get from (4) that

$$f(n) = b_q {x_q}^{n+q-1}(1+o(1)) \ . \tag{10}$$

Using (7), it is easy to verify that $x_q = 1/p$ is a root of (6). From (5) and (8) we get that

$$b_q = \frac{1}{\dfrac{q}{p^{q-1}} - \dfrac{q-1}{p^{q-2}}} = \frac{1-p}{p\alpha} \ .$$

Hence, from (10)

$$f(n) = \frac{1-p}{p\alpha}\frac{1}{p^{n+q-1}}(1+o(1)) = \frac{1+o(1)}{\alpha p^n} \ . \tag{11}$$

Let h be the maximal integer satisfying

$$\sum_{n=0}^{h} f(n) \leq k \ . \tag{12}$$

From (11) we get that

$$\begin{aligned} k \geq \frac{1+o(1)}{\alpha} \sum_{n=0}^{h} \frac{1}{p^n} &= \frac{1+o(1)}{\alpha} \frac{1-(1/p)^{h+1}}{1-1/p} \\ &= \frac{1+o(1)}{\alpha(1-p)} \left(\frac{1}{p^h} - p \right). \end{aligned} \tag{13}$$

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,

$$k \leq \sum_{n=0}^{h+1} f(n) = \frac{1+o(1)}{\alpha(1-p)} \left(\frac{1}{p^{h+1}} - p \right). \tag{14}$$

Combining (13) and (14) we have

$$(1+o(1))\frac{1}{p^h} \leq k(1-p)\alpha + p(1+o(1)) \leq (1+o(1))\frac{1}{p^{h+1}} \tag{15}$$

and we find the value of h by taking a logarithm (to the base $1/p$) from both sides of (15). We note that $\log_{1/p}(1+o(1)) = o(1)$, and since h is an integer,

$$h = \log_{1/p}(k(1-p)\alpha + p) - \phi(p,k) + o(1), \quad \phi(p,k) \in [0,1]. \tag{16}$$

From the above discussion, $height(T_{\mathsf{DEE}}) \leq h+1$. This yields the statement of the lemma.

∎

Lemma 6.4 will be used for deriving an asymptotic expression for the expected benefit of DEE, as stated in our next result. Let $Q = \{p|\ p \in [1/2,1) \text{ and } p^q = 1-p,\ q \in \mathbb{N}\}$.

Theorem 6.5 *Let $\bar{B}(r, T_{\mathsf{DEE}})$ denote the expected benefit of* DEE *in solving the Markovian S_CLPP problem. Then,*

$$\bar{B}(r, T_{\mathsf{DEE}}) \geq \frac{1 + \lg_{1/p}((1-p)\alpha k + p) - \phi(p,k)}{\alpha}(1+o(1)) \tag{17}$$

in an infinite set of points S, where $Q \subset S$. The $o(1)$ term refers to a function of k.

In the proof we use two lemmas. Recall that $p_a(v)$, the accumulated probability of a vertex v, is the probability of the path from the root to v (as given in (1)). The next lemma is an alternative formulation of (3), presented by Yaniv in [27].

Lemma 6.6 *([27]) The expected benefit of any algorithm $\mathcal{A}$ in solving the Markovian S_CLPP on a branch tree T is given by*

$$\bar{B}(r, T_{\mathcal{A}}) = \sum_{v \in T_{\mathcal{A}}} p_a(v).$$

Lemma 6.6 implies that for any algorithm, $\mathcal{A}$, $\bar{B}(r, T_{\mathcal{A}})$ is monotonically increasing in p.

Corollary 6.7 *For any $k \geq 1$ $\bar{B}(r, T_{\mathsf{DEE}})$ is a monotone increasing function of p.*

Lemma 6.8 *For any $k \geq 1$, $\bar{B}(r, T_{\mathsf{DEE}})$ 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 6.1 and 6.6 we can write

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

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. Recall that if f, g are continuous functions, then $f + g$ and $\max(f, g)$ are continuous too. We conclude that $F(p)$ is continuous. ∎

Proof of Theorem 6.5: Using (12) and (14) we can write for any $p \in Q$

$$\sum_{n=0}^{h} p^n f(n) \leq \bar{B}(r, T_{\mathsf{DEE}}) \leq \sum_{n=0}^{h+1} p^n f(n) \tag{18}$$

From (11), we get that

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

Substituting h with the value given in (16), we get the statement of the theorem for all $p \in Q$. To show that (17) holds for a larger set of points S, such that $Q \subset S$, we note that from Corollary 6.7 and Lemma 6.8, for any $p \in Q$ and $\varepsilon > 0$, there exists $\delta > 0$, such that for all $p' \in (p - \delta, p]$, $F(p) - F(p') < \varepsilon$. It follows that for any $p \in Q$ and sufficiently small values of $\varepsilon > 0$, we can define an interval in which (17) holds. Denote the resulting set of points by S; then, $Q \subset S$. ∎

Theorem 6.9 DEE *is within a factor $1 + o(1)$ from the optimal for Markovian CLPP on homogeneous branch trees, for an infinitely large set of values of $p \in [1/2, 1]$.*

Proof: Let $B^* = \bar{B}(r, T_{\mathsf{DEE}})$. We first show that any algorithm for the Markovian CLPP on homogeneous branch trees has expected performance ratio at least k/B^*.

Consider Algorithm Par-DEE which operates on T for k steps: in these steps Par-DEE initiates the fetches of k blocks using the rules of DEE, starting from r, the root of T. At time k the adversary reveals to Par-DEE the *last* "correct" vertex in $T_{Par-DEE}$: Par-DEE takes the last correct vertex to be the new root of the access graph T and starts a new phase. Note that in the above operation mode Par-DEE is more powerful that any other algorithm. Indeed, by the pipeline property, after k steps Par-DEE can access only the *first* block in the cache.

Now, consider Lazy-DEE, that fetches in each phase k/B^* vertices, and then stalls for k steps. Hence, we can partition the execution of σ into phases, each of length $k' = k(1 + 1/B^*)$. The adversary accesses k' blocks in a phase, while (from Theorem 6.5), for any $p \in S$, the expected benefit of Lazy-DEE in a phase is at least

$$\begin{aligned}
\frac{1+o(1)}{\alpha}\left[1 + \lg_{\frac{1}{p}}((1-p)\alpha\frac{k}{B^*} + p) - \phi(p,k')\right] \quad &\geq \quad \frac{1+o(1)}{\alpha}\left[1 + \lg_{\frac{1}{p}}(\frac{(1-p)\alpha k + p}{B^*} - \phi(p,k'))\right] \\
&= \quad \tfrac{1+o(1)}{\alpha}[1 + \lg_{\frac{1}{p}}((1-p)\alpha k + p) - \phi(p,k') - \lg_{\frac{1}{p}} B^*] \\
&= \quad B^* - \frac{1+o(1)}{\alpha}\lg_{\frac{1}{p}} B^* \ .
\end{aligned}$$

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

$$\bar{c}_{Lazy-DEE}(T,k,k-1) < \frac{k + \frac{k}{B^*}}{B^* - \lg_{1/p} B^*} = \frac{k}{B^*}\left(1 + \frac{1 + \lg_{1/p} B^*}{B^* - \lg_{1/p} B^*}\right) = \frac{k}{B^*}(1 + o(1)) \ .$$

As shown above, k/B^* is a lower bound on the expected performance ratio of any algorithm. This completes the proof. ∎

7 Branch Trees with Bounded Number of Leaves

As mentioned above (see in Section 1.3) speculative execution of program code results in handling at the same time multiple execution paths of the program. Each of these paths needs to be allocated a *processing unit*: each processing unit chooses an execution path and prefetches the blocks on that path. When arriving at a branch instruction, the corresponding processing unit evaluates the instruction and branches accordingly. A processing unit aborts when it needs to access a block not on its path. Modern processor architectures contain several processing units capable of working in parallel. Suppose that there are L units, then L paths can be executed in parallel. The vertices of these paths form a tree with L leaves. (The leaves of this subtree may be any vertices in T.)

This motivates our study of a variant of the Markovian CLPP, in which we add the following constraint. At any time the subtree of blocks held in the cache can have at most L leaves. We call this variant: the Markovian CLPP with L leaves ($CLPP_L$).

7.1 A Dynamic Programming Algorithm

Consider the following generalization of the Markovian S_CLPP problem on a branch tree T. The algorithm $\mathcal{A}$ has to choose a subtree $T_{\mathcal{A}}$ of T of size k, such that

(i) The expected size of $PREF_\sigma(T_{\mathcal{A}})$ is maximal, and

(ii) $T_{\mathcal{A}}$ has at most L leaves, for some $L \geq 1$.

We call this problem the *Markovian S_CLPP_L*. Indeed, in the special case where $L \geq k$, we get the Markovian S_CLPP problem. Denote by $|T|$ the size of the branch tree T.

Theorem 7.1 *The Markovian S_CLPP_L problem can be optimally solved in $O(L^2k^2|T|)$ steps.*

Proof: Without loss of generality, assume that the height of T is bounded by k (otherwise we can solve the problem on a sub-graph of T). We note that any subtree of an optimal tree T_k^* is also optimal. Hence we can use a bottom-up technique for solving the $\mathrm{S\_CLPP_L}$ problem. We now describe a dynamic programming algorithm DP 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 subtree 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 vertex v's children, x and y, have respective transition probabilities p and $1-p$. Then we write

$$w_v[m,n] = 1 + \max_{0 \le m_0 \le m} \max_{0 \le n_0 \le n-1} \left(p w_x[m_0, n_0] + (1-p) w_y[m - m_0, n - n_0 - 1]\right) .$$

When calculating the maximum, we can also keep the indices m_0, n_0, for which the maximum was obtained. To get an optimal subtree, it is sufficient to find $\max_{1 \le m \le L} w_r[m,k]$, where r is the root of T. The tree T_k^* can be constructed top-down, using the indices of each local maximum.

We now calculate the complexity of Algorithm DP. 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 $O(L^2 k^2 |T|)$. ∎

Corollary 7.2 *There is a 2-optimal algorithm for the Markovian CLPP$_L$ problem.*

Proof: We use the optimality of the DP algorithm for the Markovian S_CLPP$_L$. As in the proof of Theorem 5.1, we define the more powerful Algorithm Par-DP, and Algorithm Lazy-DP. Comparing the performance ratios of these two algorithms, we obtain a factor of 2. ∎

7.2 Algorithm DEE$_L$

When T is a homogeneous branch tree, we show that for solving the Markovian S_CLPP$_L$ problem Algorithm DP can be replaced by Algorithm DEE$_L$, a variant of Algorithm DEE, whose complexity is $O(k \lg k)$. Algorithm DEE$_L$ operates as follows. Let T_i' be the subtree obtained after i vertices were selected; $T_1' = \{r\}$. Denote by $children(T_i')$ the set of children of the vertices in T_i', i.e.,

$$children(T_i') = \{v \in V | (u,v) \in E \, u \in T_i', \text{ and } v \notin T_i'\} .$$

Let $leaves(T_i')$ be the set of leaves of T_i'. In step $(i+1)$ DEE$_L$ chooses the vertex $v \in children(T_i')$, such that $|leaves(T_i' \cup \{v\})| \le L$ and the accumulated probability of v is maximal. Note that the number of leaves increases only when u is an internal vertex in T_i'.

Lemma 7.3 *The running time of* DEE$_L$ *is $O(k \lg k)$ steps.*

Proof: We partition the execution of DEE$_L$ into two stages. In the first stage, the number of leaves in the selected subtree is smaller than L. During this stage DEE$_L$ maintains a heap consisting of the vertices in $children(T_i')$ with their accumulated probabilities. In step

i, $1 \leq i \leq k$, one vertex is deleted from the heap, and two new vertices (the children of the vertex selected in this step) are inserted. Since we need to choose k vertices, $|T_i'| \leq k$ and $|children(T_i')| \leq 2k$. Hence, the heap size in each step is at most $2k$, and the complexity of an insert/delete operation is $O(\lg k)$. Thus the overall running time of the first stage is $O(k \lg k)$.

In the second stage the selected subtree has exactly L leaves. (Note that this stage need not be reached.) Since choosing a child of a non-leaf strictly increases the number of leaves, we can only choose a child of a leaf. Since the accumulated probability of a left child of a vertex is larger than that of its right sibling, we select only left children. Thus, $\mathsf{DEE_L}$ selects a left child of a leaf with maximal accumulated probability. We need therefore maintain a heap consisting only of the left children of the leaves, and its size is bounded by $L \leq k$. Hence, the overall running time is again $O(k \lg k)$. ∎

Theorem 7.4 $\mathsf{DEE_L}$ *is optimal for the Markovian S_CLPP_L problem on homogeneous trees.*

Proof: We use induction to show that in step i, $1 \leq i \leq k$ the subtree T_i' selected by $\mathsf{DEE_L}$ is contained in an optimal subtree of size k, $T^{opt(k)}$.

Basis: $i = 1$, then $T_i' = \{r\} \subseteq T^{opt(k)}$ (since the optimal subtree contains the root of T).

Induction Step: Assume that $T_{i-1}' \subseteq T^{opt(k)}$ for some optimal subtree $T^{opt(k)}$. Let v be the vertex selected by $\mathsf{DEE_L}$ in step i, and let $U = T^{opt(k)} - T_{i-1}'$ be the set of vertices in $T^{opt(k)}$ which are not in T_{i-1}'. Suppose that $v \notin T^{opt(k)}$, then we show how $T^{opt(k)}$ can be modified to include v without harming its optimality. We handle separately two cases, depending on the set U:

(i) There exists a vertex $w \in U$ that is not a single child (Figure 4 (a)). Then we modify T_{i-1}' by cutting the edge connecting w to its parent in T, and replacing the subtree rooted at w by an isomorphic subtree rooted at v. (Figure 4 (b)). First, we note that this change does not affect the local probabilities in the subtree of w, due to the homogeneity of the tree T. Also, since $T_{i-1}' \subseteq T^{opt(k)}$, either $w \in children(T_{i-1}')$ and can be selected by $\mathsf{DEE_L}$ in phase i, or w has an ancestor in $children(T_{i-1}')$, which can be selected in phase i. In both cases, the fact that v was chosen by $\mathsf{DEE_L}$ in this phase implies that $p_a(v) \geq p_a(w)$. Hence, the new accumulated probability of any vertex x in the subtree of w is given by

$$p_a'(x) = \frac{p_a(v)}{p_a(w)} \cdot p_a(x) \geq p_a(x).$$

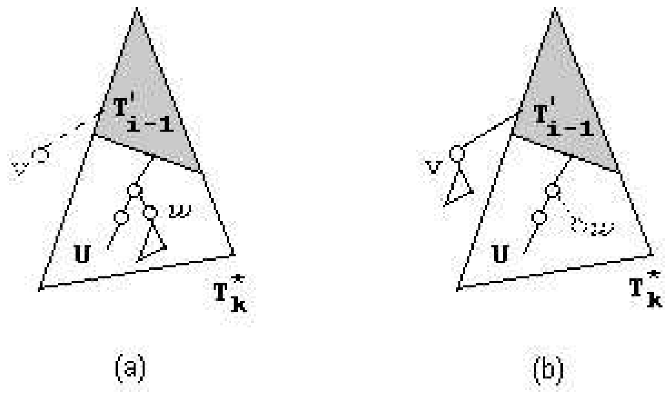

Figure 4: Case (i) in the proof of Theorem 7.4

We conclude that the weight of the modified tree is not smaller than the weight of $T^{opt(k)}$. Finally, we note that our transformation did not increase the number of leaves in $T^{opt(k)}$.

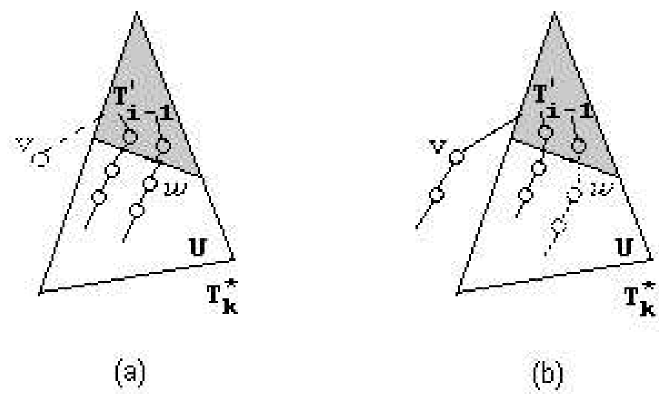

Figure 5: Case (ii) in the proof of Theorem 7.4

(ii) Each vertex in U is a single child, i.e., U is set of disjoint paths, each connected to a vertex in T'_{i-1} (Figure 5 (a)). Then, there exists a path in U rooted in a vertex $w \in children(T'_{i-1})$. Clearly, $p_a(w) \leq p_a(v)$ (otherwise, DEE_L would have selected w in iteration i). We can replace the path rooted in w in $T^{opt(k)}$ by an isomorphic path rooted in v (Figure 5 (b)).

This completes the proof. ∎

8 Discussion

We studied the problem of caching integrated with pipelined prefetching, which 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 AGG is an optimal offline algorithm, and that Lazy-AGG$_{\text{OL}}$ is optimal to within factor 2 in the set of deterministic online algorithms on *general* graphs. In the *Markovian* model, DEE was shown to be nearly optimal on branch trees.

Several interesting problems remain open:

- We have shown that randomization does not help when G is a branch tree. Can randomization help in the larger class of DAGs? in other classes of graphs?

- How efficiently can we select an optimal sub-graph in the *Markovian* model, e.g., on a DAG?

- We have shown that $c^{det}(G, k, d)$ can be achieved to within factor 2. Can we bound the value of $c^{det}(G, k, d)$ for certain classes of graphs (other than complete graphs), so we can measure our online algorithms relative to the optimal *offline*?

- 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 in [19] shows, that in practice the performance ratio of DEE is close to 1. Can other technique be applied to tighten our 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, 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 bring the analysis closer to real execution models.

Acknowledgments: The authors thank Anna Karlin, Rajeev Motwani and Prabhakar Raghavan, for stimulating discussions on this paper.

References

[1] S. Albers, N. Garg and S. Leonardi, "Minimizing Stall Time in Single and parallel Disk Systems", J. of the ACM 47(6): 969-986, 2000.

[2] S. Albers,and C. Witt, "Minimizing Stall Time in Single and Parallel Disk Systems Using Multicommodity Network Flows", in Proc. of RANDOM-APPROX 2001: 12-23.

[3] D. Bhandarkar, J. Ding, "Performance Characterization of the Pentium Pro Processor", Proc. of the 3rd International Symposium on High Performance Computer Architecture, San Antonio, 1997.

[4] A. Borodin, S. Irani, P. Raghavan and B. Schieber, "Competitive Paging with Locality of Reference", *Journal of Computer and System Science*, 1995, pp. 244-258.

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

[6] P. Cao, E. Felton, A. Karlin, K. Li, "A Study of Integrated Prefetching and Caching Strategies", in Proc. of the *ACM Joint International Conference on Measurement and Modeling of Computer Systems (SIGMETRICS/PERFORMANCE '95)*, Ottawa, May 1995.

[7] A. Fiat. and G. J. Woeginger,, "Online Algorithms, The State of the Art", Springer, 1998 (LNCS #1442).

[8] A. Fiat. and A. R. Karlin, "Randomized and Multipointer Paging with Locality of Reference", in Proc. of the *Twenty-Seventh Annual ACM Symposium on Theory of Computing*, Las Vegas, 1995, pp. 626–634.

[9] A. Fiat. and Z. Rosen, "Experimental Studies of access Graph Based Heuristics: Beating the LRU Standard?", in Proc. of the *Eighth Annual ACM-SIAM Symposium on Discrete Algorithms*, New Orleans, 1997, pp. 63–72.

[10] A. Fiat. and M. Mendel, "Truly Online Paging with Locality of Reference", in Proc. of *38th Annual Symposium on Foundations of Computer Science*, Miami Beach, 1997, 326-335.

[11] J.L. Hennesy and D.A. Patterson "Computer Architecture a Quantitative Approach", 1995.

[12] D.S. Hochbaum, *Approximation Algorithms for NP-Hard Problems*, PUS Publishing Company, 1995.

[13] S. Irani, A. R. Karlin and S. Phillips, "Strongly competitive algorithms for paging with locality of reference", *SIAM Journal Comput.*, June 1996, pp. 477-497.

[14] T. Kimbrel, A. R. Karlin, "Near-optimal parallel prefetching and caching", SIAM J. Computing 29(4): 1051-1082, 2000.

[15] A. R. Karlin, S. Phillips and P. Raghavan, "Markov paging", Proc. of the 33rd Symposium on Foundations of Computer Science, Pittsburgh, Penns., 208-216, 1992.

[16] D. E. Knuth, *The Art of Computer Programming*, Vol. 3, Addison Wesley, 1973.

[17] S. Lang, *Algebra*, *Columbia University, New York*, Ch. 5, Sec. 8. 1971.

[18] R. Motwani, P. Raghavan, *Randomized Algorithms*, Cambridge University Press, 1995.

[19] P. Raghavan, H. Shachnai, M. Yaniv, "Dynamic Schemes for Speculative Execution of Code", *MASCOTS*, 1998.

[20] Rambus Inc., "Technology Overview", August 1999.

[21] A. Tomkins, R. H. Patterson and G. Gibson. "Informed Multi-Process Prefetching and Caching", *SIGMETRICS*, 1997.

[22] R. Sedgewick and P. Flajolet, *An Introduction to the Analysis of Algorithms*, Addison-Wesley Publishing, 1996.

[23] A.K. Uht and V. Sindagi, "Disjoint Eager Execution: An optimal form of speculative execution", *MICRO-28*, 1995.

[24] G. M. Voelker, E. J. Anderson, T. Kimbrel, M. J. Feeley, J. S. Chase, A. R. Karlin and H. M. Levy, "Implementing Cooperative Prefetching and Caching in Globally-Managed Memory System". *SIGMETRICS*, 1998.

[25] S. S. H. Wang and A. K. Uht, "Ideograph/Ideogram: Framework/Architecture for Eager Evaluation", *MICRO-23*, 1990.

[26] M. Yang and L. M. Ni, "Design of Scalable and Multicast Capable Cut-Through Switches for High-Speed LANs", *Proc. of the 1998 International Conference on Parallel Processing*, pp. 324-332.

[27] M. Yaniv, "Dynamic Schemes for Speculative Execution of Code", M.Sc. Thesis, Dept. of Computer Science, The Technion, 1998.

[28] A. C.-C. Yao, "Probabilistic Computations: towards a unified measure of complexity", *Proc. of the 18th IEEE Symposium on Foundations of Computer Science*, 222-227,

A The DEE Algorithm - Example

Consider the operation of DEE on a *homogeneous* branch tree, with the local probability $p = 0.7$ for any vertex in the tree (See in Figure 6: the thick edges show the selected subtree, and the numbers by the vertices show the order of selection). Suppose that the parameter is $l = 7$. DEE starts the selection from the root, $N1$, and continues as follows: in each step, DEE selects a vertex v that has the maximal accumulated probability. The accumulated probabilities of the vertices are $p_a(N1) = 1, p_a(N2) = 0.7, p_a(N4) = 0.49, p_a(N8) = 0.343, p_a(N3) = 0.3, p_a(N16) = 0.24, p_a(N5) = 0.21$. Hence, the vertices are selected in this order.

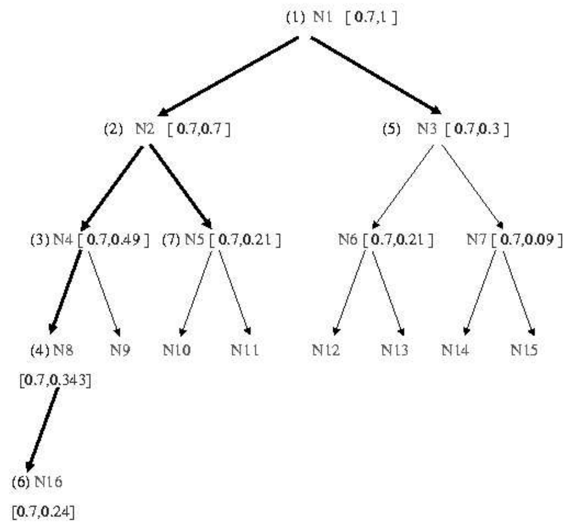

Figure 6: The execution of DEE on a branch tree for the S_CLPP with $l = 7$.

B The q-distance Fibonacci Numbers

Proof of Lemma 6.3: To find an expression for $g(n)$, $n \geq q$, we need to solve the equation generated from the recursion formula, i.e.,

$$x^n = x^{n-1} + x^{n-q}. \tag{19}$$

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

$$p'(x) = qx^{q-1} - (q-1)x^{q-2}. \tag{20}$$

Note that $p'(x)$ has only two roots: $x_0 = 0$ and $x_1 = \frac{q-1}{q}$, which are not roots of $p(x)$. It follows that $p(x)$ has no multiple roots (see, e.g., in [17]). Hence, the general form of our sequence is

$$g(n) = \sum_{\ell=1}^{q} b_\ell {x_\ell}^n, \tag{21}$$

where $x_1, x_2, \ldots, x_q$ are the roots of the polynomial $p(x)$. We also note that for $1 \leq x \leq 2$ $p(x)$ is monotone and non-decreasing, and since $p(1) = -1$, and $p(2) = 2^{q-1} - 1 \geq 0$, we get that $p(x)$ has a single root $x_q \in \mathbb{R}^+$ in the interval $[1, 2]$.

Recall that the module of a complex number $x = a + bi$ is $|x| = \sqrt{a^2 + b^2}$; then a polar representation of x is $x = re^{i\phi} = r(\cos\phi + i\sin\phi)$, where $r = |x|$ and ϕ is the argument of

x. We now show that $|x_\ell| < x_q$ for all $\ell < q$. The claim trivially holds for any x_ℓ satisfying $|x_\ell| \leq 1$, thus we may assume that $|x_\ell| > 1$. If x_ℓ is a root of $p(x)$, then

$$0 = p(x_\ell) = |{x_\ell}^q - {x_\ell}^{q-1} - 1| \;\geq\; |x_\ell|^q - |x_\ell|^{q-1} - 1 = p(|x_\ell|),$$

and since $p(x)$ is non-decreasing for $x \geq 1$, we conclude that $|x_\ell| \leq x_q$. To show that the last inequality is strong for all $1 \leq \ell \leq q-1$, assume that for some ℓ $|x_\ell| = x_q$, i.e., $x_\ell = x_q e^{i\phi}$. Then,

$$|x_\ell|^q = {x_q}^q \quad , \quad |x_\ell|^{q-1} = {x_q}^{q-1}$$

and

$$|x_\ell|^q = |x_\ell|^{q-1} + 1$$

and since x_ℓ is a root of $p(x)$, we get that

$$|{x_\ell}^{q-1} + 1| = |x_\ell|^{q-1} + 1$$

The last equation implies that 1 and ${x_\ell}^{q-1}$ have the same argument [17]. Therefore, ${x_\ell}^{q-1} \in \mathbb{R}^+$, and

$$\phi(q-1) = 2\pi n \text{ for some } n \in Z. \tag{22}$$

However, since ${x_\ell}^q = {x_\ell}^{q-1} + 1$, this also implies that ${x_\ell}^q \in \mathbb{R}^+$, or

$$\phi q = 2\pi m \text{ for some } m \in Z. \tag{23}$$

Equations (22) and (23) are satisfied when $\phi = 2\pi\ell$, for some $\ell \in Z$, which means that $x_\ell = x_q$, in contradiction to the fact that $p(x)$ has no multiple roots. We conclude that for all $1 \leq \ell \leq q-1$ $|x_\ell| < x_q$. We can now write

$$g(n) = \sum_{\ell=1}^{q} b_\ell {x_\ell}^n = b_q {x_q}^n \left(1 + \sum_{\ell=1}^{q-1} \frac{b_\ell}{b_q} \left(\frac{x_\ell}{x_q}\right)^n\right) \tag{24}$$

and since $|x_\ell / x_q| < 1$, the sum on the right hand side of (24) exponentially tends to zero, i.e.,

$$\lim_{n\to\infty} \sum_{\ell=1}^{q-1} \frac{b_\ell}{b_q} \left(\frac{x_\ell}{x_q}\right)^n = 0$$

Hence, we get that

$$g(n) = b_q {x_q}^n (1 + o(1)). \tag{25}$$

Now, b_q can be calculated by solving a linear system for the first q elements of the sequence $g(n)$.

$$\begin{pmatrix} 1 & 1 & \dots & 1 \\ x_1 & x_2 & \dots & x_q \\ \vdots & & & \vdots \\ {x_1}^{q-1} & {x_2}^{q-1} & \dots & {x_q}^{q-1} \end{pmatrix} \begin{pmatrix} b_1 \\ b_2 \\ \vdots \\ b_q \end{pmatrix} = \begin{pmatrix} 0 \\ 0 \\ \vdots \\ 1 \end{pmatrix}$$

The determinant of the above matrix is known as Vandermondé determinant [17]. The general solution of such a system is

$$b_\ell = \prod_{j=1, j\neq \ell}^{q} (x_\ell - x_j)^{-1}$$

Our polynomial is $p(x) = \prod_{\ell=1}^{q}(x - x_\ell)$, and $p'(x_\ell) = \prod_{j=1,j\neq\ell}^{q}(x_\ell - x_j)$, thus $b_\ell = \frac{1}{p'(x_\ell)}$, and in particular, using (20) we have

$$b_q = \frac{1}{q{x_q}^{q-1} - (q-1){x_q}^{q-2}} \tag{26}$$

Substituting into (25) we get that

$$g(n) = \frac{{x_q}^n}{q{x_q}^q \left(\frac{1}{x_q} - \frac{q-1}{q}\frac{1}{{x_q}^2}\right)}(1 + o(1)).$$

This yields the statement of the lemma. ∎