Strongly Competitive Algorithms for Caching with Pipelined Prefetching[*]

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

Computer Science Department, The Technion, Haifa 32000, Israel

Abstract

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, $\min\{k, d\}$ block fetches can be in progress simultaneously. Any sequence of block references is modeled as a walk on the *access graph* of the program. The goal is to find a policy for prefetching and caching, which minimizes the overall execution time of a given reference sequence. This 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 proposed by Cao et al. [6] is optimal for this problem. In the online case, we give an algorithm that 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 for several classes of access graphs which arise in applications, including *complete graphs* and *directed acyclic graphs (DAG)*.

Key words: Caching, pipelined prefetching, access graphs, online algorithms.

1 Introduction

Caching and prefetching have been studied extensively in the past decades; however, the interaction between the two was not well understood until the seminal 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, 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

[*]A preliminary version of this paper appeared in proceedings of *ESA* 2001.

[†]`csalex@tx.technion.ac.il`

[‡]`itai@cs.technion.ac.il`

[§]Part of this work was done while the author was 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: +972-4-829-3900.

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,[1] 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. Following Borodin et al. [4], we capture the locality of reference in memory access patterns of real programs by the access graph model (see below). 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 $V = \{b_1, b_2, \ldots, b_n\}$ is held in secondary storage. The access graph for the program that reads/writes into V 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) b_i, the next request has to be either to the block b_i itself or to a block b_j, such that $(b_i, b_j) \in E$. To allow consecutive accesses to a block b_i, a self-loop is added to the corresponding vertex in G. We denote by $Paths(G)$ the set of paths in G.

Pipelined prefetching allows to initiate a prefetch one time unit after the previous prefetch. Note that if block b_i is replaced in order to bring block b_j, then b_i becomes unavailable for access when the fetch is initiated; b_j can be accessed only when the fetch terminates, i.e., after d time units. This allows for $\min\{k, d\}$ fetches to be in progress, at any given time.

Let the reference sequence $\sigma = (r_1, \ldots, r_m)$, and suppose that at time t block r_i is referenced. To complete the reference, block r_i should be in the cache. If at time t block r_i is already in the cache, the reference is satisfied immediately, incurring one time unit; otherwise, a prefetch of r_i was initiated by algorithm $\mathcal{A}$ at time $t_i \leq t$, and a *stall* for $d_{\mathcal{A}}(r_i) = d - (t - t_i)$ time units incurs. The *total stall time* of σ is $\sum_{i=1}^{m} d_{\mathcal{A}}(r_i)$. The *total execution time* of σ is the time to access the m blocks plus the total stall time, i.e., $time(\mathcal{A}, \sigma) = m + \sum_{i=1}^{m} d_{\mathcal{A}}(r_i)$. 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 σ, find an algorithm $\mathcal{A}$ for pipelined prefetching and caching, such that $time(\mathcal{A}, \sigma)$ is minimized.

The *adversary* is an optimal offline algorithm, OPT, together with a request sequence. We use competitive analysis (see e.g. in [5]) to establish performance bounds for online algorithms for our problem. The competitive ratio of an online algorithm $\mathcal{A}$ on a graph G, for fixed cache size k and delivery time d, is given by

$$c_{\mathcal{A}}(G, k, d) = \sup_{\sigma \in Paths(G)} \frac{time(\mathcal{A}, \sigma)}{time(OPT, \sigma)} \ . \qquad (1)$$

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. We say that $\mathcal{A}$ is *strongly competitive*, if $c_{\mathcal{A}}(G, k, d) = O(c(G, k, d))$. 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 delivery time d. 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. Branch trees model program execution on fast processors (see [11]), in which case, the cache size, k, is the length of the pipe. Since each instruction goes

[1] A detailed survey of these applications is given in [11].

through $k-1$ stages in the pipe before it reaches the *execution* phase, in our discussion of branch trees we assume that $d = k-1$.

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 2), 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 3), we give an algorithm, Lazy-OL-AGG, which is strongly competitive in the set of deterministic online algorithms for CLPP. In particular, we show that $c_{\mathsf{Lazy\text{-}OL\text{-}AGG}}(G,k,d) \leq 2 \cdot c^{det}(G,k,d)$ for *any* access graph G and $k, d \geq 1$. Better ratios are derived (in Section 4) for several classes of access graphs that arise in applications. Specifically, we show that if G is a *directed acyclic graph (DAG)* then $c_{\mathsf{Lazy\text{-}OL\text{-}AGG}}(G,k,d) \leq \min\{1+k/d, 2\} \cdot c^{det}(G,k,d)$, for any $k, d \geq 1$, and if G is a branch tree then $c_{\mathsf{Lazy\text{-}OL\text{-}AGG}}(G,k,k-1) \leq (1+o(1))c(G,k,k-1)$, where the $o(1)$ term refers to a function of k. For complete graphs, we show that any deterministic *marking* algorithm[2] $\mathcal{A}$ satisfies $c_{\mathcal{A}}(G,k,d) \leq \min\{d+1, k+1\}$. We also show that this is almost the best possible, since for any complete graph G and $k, d \geq 1$, $c^{det}(G,k,d) \geq \min\{d+1, k\}$.

For deriving upper bounds on the competitive ratios of our algorithms, we develop (in Section 3) a general proof technique that we use also for special classes of access graphs (in Section 4). 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, i.e., if at time t, d' prefetches are in progress, upto $d-d'$ additional prefetches can be initiated at time t. The technique may be useful for tackling other problems in which *pipelined* service is granted to a set of requests in an online fashion.

Related Work: The concept of integrated 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. Algorithm AGG was shown to yield a $min\{1+d/k, 2\}$-approximation to the optimal. Albers et al. showed in [2] that the CP problem can be optimally solved using linear programming. Later, Albers and Witt [3] presented an optimal combinatorial algorithm for the problem. Recently, Albers and Büttner [1] gave a refined analysis for AGG on a single disk and showed that it achieves the ratio $min\{1+d/(k+\lfloor \frac{k}{d} \rfloor -1), 2\}$, which is tight.

Kimbrel and Karlin studied in [15] storage systems that 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. An algorithm that operates similar to AGG was shown to achieve a competitive ratio of $(1+rd/k)$. The papers [2], [3] and [1] give approximation algorithms for the problem of minimizing the *total stall time* in a system that consists of r units, for any $r > 1$. Other papers (see e.g. [17, 18]) present experimental results for *cooperative* prefetching and caching, in the presence of optional program-provided hints of future accesses.

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 $d \gg 1$.[3] There is a wide literature on the caching (paging) problem. (Comprehensive surveys appear e.g. in [13, 16, 5, 7].) Borodin et al. [4] introduced the *access graph* model. The paper presents an online algorithm that is strongly competitive on any

[2]We give the precise definition of this class of algorithms in Section 4.

[3]Thus, when normalizing (by factor d) we get that the delivery time equals to one, while the access time, $1/d$, asymptotically tends to 0.

access graph. Later works (e.g., [8, 14, 10]) consider extensions of the access graph model, or give experimental results for some heuristics for paging in this model [9].

2 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 2.1 *[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.*[4] (iii) *$\mathcal{A}$ never replaces a block B by a block C if B will be referenced before C.*

The proof of the lemma follows from the next claim, which can be shown by induction on i, $i \geq 1$. (A detailed proof is given in [11].)

Claim 2.2 *Any algorithm $\mathcal{A}$ can be transformed to an algorithm $\bar{A}$ which satisfies the three "no harm" rules in the first i references, for any $i \geq 1$, without incurring extra cost.*

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. As stated in our next result, this algorithm is the best possible for CLPP.

Theorem 2.3 *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 i-th reference:

(i) $\mathcal{A}$ initiates a fetch of some block r_x. 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_x 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_x and discards r_y. Indeed, this implies that r_x is the next block in the reference sequence that is missing in the cache, for both AGG and $\mathcal{A}$. We define an algorithm $\mathcal{A}'$ that operates like AGG in step i and then proceeds like $\mathcal{A}$, until the time $t > i$ in which $\mathcal{A}$ fetches r_x and discards some block r_d. If $r_d \neq r_y$ then $\mathcal{A}'$ fetches r_y and discards r_d; otherwise, the contents of $\mathcal{A}'$'s cache remain unchanged. Clearly, the above occurs before the next reference to r_x (which precedes the next reference to r_y). Thus, the cost of $\mathcal{A}'$ is equal to that of $\mathcal{A}$, and $\mathcal{A}'$ acts like AGG in the first i steps.

[4]If there is no future reference to two blocks A and B they are evicted from the cache in lexicographic order.

Note that algorithm $\mathcal{A}'$ may not follow the "no harm" rules after step i. Indeed, r_y may not be the first block that is missing in the cache at the time it is fetched by $\mathcal{A}'$. However, by Claim 2.2, we can modify $\mathcal{A}'$ so that it satisfies the "no harm" rules at any time. (As shown in the proof of Claim 2.2 [11], if $\mathcal{A}'$ violates a "no harm" rule for the first time at $t > i$, then we modify the steps of the algorithm at time t or later, therefore $\mathcal{A}'$ still acts like AGG in the first i steps). ∎

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

Corollary 2.4 *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, 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 r_e and initiate a fetch of r_{i+d}. Hence the claim holds also after the ith reference. ∎

3 The Online CLPP Problem

In this section we study the online CLPP on general access graphs. Recall that in the online case r_{i+1} is revealed to the algorithm only when r_i is accessed, for $1 \leq i \leq m-1$. Given the parameters $k, d \geq 1$ and the access graph G, our goal is to develop an online algorithm $\mathcal{A}$ that minimizes the ratio $c_{\mathcal{A}}(G, k, d)$, as given in (1). We start by investigating the case where the length of σ is bounded by $\min\{k, d\}$; later we extend the discussion to sequences of arbitrary length.

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. Then σ is a path in G which – 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. A vertex $v \in V \setminus V'$ is called a *hole*. If s is the length of the shortest path from r to a hole 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.

Towards obtaining a competitive algorithm for the online CLPP, we consider first 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*). Given a reference sequence, σ, and the subgraph $G_{\mathcal{A}} \subseteq G$ selected by 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 S_AGG below mimics the operation of AGG in solving S_CLPP, i.e., in maximizing the benefit of the selected subgraph. Denote by $dist(v, u)$ the length of the shortest path from v to u in G.[5] 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 S_AGG.

```
Algorithm S_AGG
for $t = 1, \ldots, \ell$ do
  Let $u = \arg\min\{dist(r_1, v) : v \in OUT_t\}$ .
  Let $w = \arg\max\{dist(r_1, v) : v \in IN_t\}$ .
  If $dist(r_1, u) < dist(r_1, w)$
    Evict $w$ from the cache and initiate a fetch for $u$.
```

Lemma 3.1 *S_AGG is an optimal algorithm 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 S_AGG always selects the hole that is closest to r_1, $G_{\text{S\_AGG}}$ maximizes the length of the shortest path from r_1 to a hole. Hence, $B(r_1, G_{\text{S\_AGG}})$, the benefit of S_AGG, is maximum. ■

The above lemma facilitates the derivation of our main result for the online CLPP. Assume now that σ has *arbitrary* length $m \geq 1$. Let Lazy-OL-AGG be an algorithm that breaks σ into smaller sequences and solves for each the S_CLPP problem. More specifically, whenever a block is missing in the cache, Lazy-OL-AGG initiates the fetches of $\ell = \min\{k, d\}$ blocks, using S_AGG; then, it stalls for d time units (until the first of these block is in the cache) and starts to access blocks, until another block is missing.

Theorem 3.2 *For any graph G and $k, d \geq 1$, Algorithm Lazy-OL-AGG satisfies*

$$c_{\text{Lazy-OL-AGG}}(G, k, d) \leq 2 \cdot c^{det}(G, k, d) \ .$$

Proof: We compare two algorithms, Algorithm Lazy-OL-AGG and Algorithm Par-OL-AGG, which 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 Lazy-OL-AGG. In the analysis below, we use an alternative description of the algorithm, in which the operation of Lazy-OL-AGG is partitioned into *phases.* Phase i, $i \geq 1$, starts at some time t_i, with a stall of d time units, for fetching a missing block – r_i. (The first phase starts at $t_1 = 1$.) 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, Lazy-OL-AGG invokes Algorithm S_AGG to select a subset, $V_{i,j}$, of $\ell = \min\{d, k\}$ vertices. Some of these vertices (=blocks) are already in the cache: Lazy-OL-AGG 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 $g_{i,j} = |Good(i, j)|$. (Note that $g_{i,j}$ is known only at time $t + d + g_{i,j}$.)

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

- If $g_{i,j} = d$ then Lazy-OL-AGG waits until d blocks were accessed in the cache; at time $t_{i,j} + 2d$ the j-th sub-phase terminates, and Lazy-OL-AGG starts sub-phase $j + 1$ of phase i.
- If $g_{i,j} < d$ then at time $t_{i,j} + d + g_{i,j}$ phase i terminates and the first missing block in the cache becomes r_{i+1}.

Consider now Algorithm Par-OL-AGG, which operates like Lazy-OL-AGG, except that Par-OL-AGG 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 Par-OL-AGG *knows* the value of $g_{i,j}$ and the first missing block in the cache; then, all the vertices in $Good(i, j)$ can be accessed *in parallel* $d + 1$ time units after the start of this sub-phase. As in Lazy-OL-AGG, if $g_{i,j} = d$ then Par-OL-AGG proceeds to the next sub-phase of phase i; if $g_{i,j} < d$ phase i terminates. Note that, combining Lemma 3.1 with the parallel fetching property, we get that Par-OL-AGG outperforms any deterministic online algorithm for CLPP.

We now show that Lazy-OL-AGG is close to the optimal in the set of deterministic algorithms for the online CLPP. To compute $c_{\text{Lazy-OL-AGG}}(G, k, d)/c^{det}(G, k, d)$, it suffices to compare the length of phase i of Lazy-OL-AGG and Par-OL-AGG, for any $i \geq 1$. Suppose that there are $sp(i)$ sub-phases in phase i. For Lazy-OL-AGG, 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 Par-OL-AGG, each sub-phase (including the last one) incurs d time units; thus, for any sequence σ, we get the ratio

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

■

Remark 3.1 *We note that the above result relies strongly on the assumption that r_{i+1} is revealed to the algorithm only after r_i is accessed. In a slightly different model (used e.g. in the k-server problem [5]), r_i is revealed to the algorithm at time i. In this model, we can easily obtain for our problem a* competitive ratio *of* 2*. This is done by waiting until the whole sequence is known (at time m) and then solving optimally the offline CLPP. In fact, for instances in which $d < k$, we can apply AGG to the online problem and get a* single *stall (when r_1 is fetched). This yields a the competitive ratio of* 1*, by Corollary 2.4.*

4 Online CLPP on DAGs and Complete Graphs

In this section we discuss several classes of graphs that arise in applications. In Sections 4.1 and 4.2 we study the class of DAGs and the more restricted subclass of branch trees, for which we can improve the bound obtained in the previous section for Lazy-OL-AGG. In Section 4.3 we study complete graphs. We show that on such graphs pipelined prefetching is not helpful, as *marking* algorithms (that do not use prefetching) achieve almost the optimal competitive ratio in the set of deterministic algorithms.

4.1 Directed Acyclic Access Graphs

Consider now the subclass of DAGs. Our next result improves the bound in Theorem 3.2 in the case where $k < d$.

Theorem 4.1 *If G is a DAG then, for any cache size $k \geq 1$ and delivery time $d \geq 1$,*

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

Proof: Recall that in the proof of Theorem 3.2 we showed that

$$c_{\textsf{Lazy-OL-AGG}}(G, k, d) \leq 2 \cdot c^{det}(G, k, d) \ .$$

When $k < d$, each phase consists of a single sub-phase. Indeed, since G is a DAG (i.e., no self-loops), consecutive accesses to the same block cannot occur; thus, each block can be accessed at most once along the execution of the program. It follows that, in each phase, both Par-OL-AGG and Lazy-OL-AGG have at most $k < d$ 'good' blocks in the cache (i.e., $g_{i,1} < d$), and phase i terminates. The ratio between the length of phase i for Lazy-OL-AGG and Par-OL-AGG is then at most $(d + k)/d$. This yields the statement of the theorem. ∎

4.2 Branch Trees

The case where G is a branch tree is of particular interest in the application of CLPP to pipeline execution of programs on fast processors (see in [11]). For this case, we show that the bound in Theorem 4.1 can be further improved. Specifically, we show that the competitive ratio of Lazy-OL-AGG is within factor $1 + o(1)$ of the optimal in the set of online (deterministic or randomized) algorithms on branch trees.

Theorem 4.2 *If G is a branch tree then $c_{\textsf{Lazy-OL-AGG}}(G, k, k-1) \leq (1 + o(1))c(G, k, k-1)$, where the $o(1)$ term refers to a function of k.*

For the proof we need the following lemma.

Lemma 4.3 *If G is a branch tree then $c(G, k, k-1) \geq k/\lg k$.*

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 [19].

Recall that on branch trees $d = k-1$. Suppose that the tree T is rooted at $r_1 = r$. The adversary generates the reference sequence σ as follows. At any vertex $v \in T$, the adversary proceeds to the left child with probability 1/2. Denote by *depth*(v) the depth of v in T. (The depth of r_1 is 0). Let $p_a(v)$ denote 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. At time k, $\mathcal{A}$ accesses *simultaneously* all the good blocks, i.e., the fetched blocks that are a prefix of σ. At this time, $\mathcal{A}$ knows the last correct block in the fetched subtree. Then, at time $k + 1$, $\mathcal{A}$ starts fetching another set of k blocks. Thus, $\mathcal{A}$ partitions the reference sequence to phases, each consists of k time units; in each phase $\mathcal{A}$ solves S_CLPP.

Recall that in 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 Algorithm BAL, that proceeds as follows. First, BAL sorts the vertices in T in decreasing order by the values of $p_a(v)$, and then BAL fetches the first k vertices in the list. In our case, for any $i \geq 2$, in phase i BAL takes a balanced subtree of k vertices, rooted at the last correct block of phase $i - 1$ (this block becomes the root of T in phase i; $r_1 = r$).

We now calculate the expected benefit of any online algorithm, $\mathcal{A}$, in a single phase. Let v_j be the jth vertex fetched to the cache. Then the expected benefit of $\mathcal{A}$ from selecting the subtree $T_\mathcal{A}$ 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 BAL maximizes this value, since it selects k vertices with the highest probabilities. Hence, the expected benefit of *any* deterministic algorithm in solving S_CLPP is bounded by the height of a balanced tree of k vertices, that is, $\lg k$.

Assume that $\mathcal{A}$ uses BAL in each phase, then clearly $\mathcal{A}$ outperforms any deterministic online algorithm, since it fetches in each phase a subtree that maximizes the benefit; also, for fetching (and accessing) a subset of k blocks, $\mathcal{A}$ stalls only for k time units: this is the time required for handling the *first* missing block in the cache.

Since $d < k$, by Corollary 2.4, the total execution time of any optimal algorithm is $|\sigma| + d$, while the expected execution time of $\mathcal{A}$ is at least $|\sigma| k / \lg k$. ∎

Proof of Theorem 4.2: Consider the following simple variant of Lazy-OL-AGG: 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.

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

Using Lemma 4.3 we get the statement of the theorem. ∎

4.3 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 derive a lower bound on the competitive ratio of any deterministic algorithm.

Theorem 4.4 *If G is a complete graph then, for any $d \ge 1$ and cache size $k \ge 1$,*

$$c^{det}(G, k, d) \ge \min\{d + 1, k\} \ . \tag{2}$$

Proof: W.l.o.g. we assume that $|V| \ge k + 1$ (otherwise, any online algorithm that never evicts blocks from the cache is optimal). Consider first the case where $d \le k - 1$. Recall that when $d < k$, the total execution time of any optimal algorithm is $|\sigma| + d$. Let $\mathcal{A}$ be an online deterministic algorithm. For any $t \ge 1$, 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 d 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 time units, and for sufficiently long reference sequences,

$$c_{\mathcal{A}}(G, k, d) = \frac{|\sigma|(d + 1)}{|\sigma| + d} \to d + 1 \ .$$

Assume now that $d \ge k$; then, in the worst case, OPT has to fetch a block every k accesses, and its total execution time is $|\sigma| \cdot (d + 1)/k$. Since $\mathcal{A}$'s execution time is $(d + 1)|\sigma|$, we get a ratio of k. This completes the proof. ∎

In the following we show that the lower bound derived in Theorem 4.4 cannot be substantially improved: the ratio in (2) can be achieved, to within an additive factor of 1, by a caching-by-demand algorithm. Consider the set of *marking* algorithms proposed for the classical caching (paging) problem (see, e.g., in [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.

Theorem 4.5 *For any access graph G, cache size k and delivery time $d \geq 1$, if $\mathcal{A}$ is a marking algorithm then* $c_{\mathcal{A}}(G,k,d) \leq \min\{d+1, k+1\}$.

Proof: Let $\mathcal{A}$ be a deterministic marking algorithm. 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 ends once we access the $(k+1)$-st distinct block; 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 accessing the blocks. 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:

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

If $d < k$ then $\max\{n_j, d\} = n_j$, hence

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

If $d \geq k$ then

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

■

¿From Theorems 4.4 and 4.5 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.

Corollary 4.6 *Let G be a complete graph and $\mathcal{A}$ a marking algorithm, then if $d < k$ $\mathcal{A}$ is optimal in the set of deterministic algorithms for CLPP; otherwise, $\mathcal{A}$ is within factor of* $1 + 1/k$ *from any optimal deterministic online algorithm.*

Note that our results for complete graphs apply also to the case where the access graph is *unknown* to the algorithm, in which no deterministic algorithm can make 'good choices' in *prefetching* blocks. Thus, marking algorithms are almost the best possible in the set of deterministic algorithms.

Acknowledgments: We thank Anna Karlin, Rajeev Motwani and Prabhakar Raghavan, for valuable discussions on this paper. Thanks also to two anonymous referees for helpful comments and suggestions.

References

[1] S. Albers and M. Büttner, "Integrated Prefetching and Caching in Single and Parallel Disk Systems", Proc. of *the Fifteenth Annual ACM Symposium on Parallel Algorithms*, pp. 109–117, 2003.

[2] 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.

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

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

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

[6] P. Cao, E. Felton, A.R. Karlin, and K. Li, "A Study of Integrated Prefetching and Caching Strategies", in Proc. of *SIGMETRICS*, 1995, pp. 188–197.

[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 *27th 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?", Proc. of *8th Annual ACM-SIAM Symposium on Discrete Algorithms*, 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*, 1997, pp. 326–335.

[11] A. Gaysinsky, A. Itai, and H. Shachnai, "Strongly Competitive Algorithms for Caching with Pipelined Prefetching", `http://www.cs.technion.ac.il/∼hadas/PUB/clpp.ps`.

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

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

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

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

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

[17] A. Tomkins, R.H. Patterson, and G. Gibson, "Informed Multi-Process Prefetching and Caching", *SIGMETRICS*, 1997, pp. 100–114.

[18] 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, pp. 33–43.

[19] A.C.-C. Yao, "Probabilistic Computations: towards a unified measure of complexity", *Proc. of the 18th IEEE Symposium on Foundations of Computer Science*, 1977, pp. 222–227.