Minimizing Stall Time in Single and Parallel Disk Systems Using Multicommodity Network Flows

Susanne Albers and Carsten Witt

Dept. of Computer Science, Dortmund University, 44221 Dortmund, Germany,
albers@ls2.cs.uni-dortmund.de, carsten.witt@udo.edu

Abstract. We study integrated prefetching and caching in single and parallel disk systems. A recent approach used linear programming to solve the problem. We show that integrated prefetching and caching can also be formulated as a min-cost multicommodity flow problem and, exploiting special properties of our network, can be solved using combinatorial techniques. Moreover, for parallel disk systems, we develop improved approximation algorithms, trading performance guarantee for running time. If the number of disks is constant, we achieve a 2-approximation.

1 Introduction

In today's computer systems there is a large gap between processor speeds and memory access times, the latter usually being the limiting factor in the performance of the overall system. Therefore, computer designers devote a lot of attention to building improved memory systems, which typically consist of hard disks and associated caches. Caching and prefetching are two very well-known techniques for improving the performance of memory systems and, separately, have been the subject of extensive studies. Caching strategies try to keep actively referenced memory blocks in cache, ignoring the possibility of reducing processor stall times by prefetching blocks into cache before their actual reference. On the other hand, most of the previous work on prefetching tries to predict the memory blocks requested next, not taking into account that blocks must be evicted from cache in order to make room for the prefetched blocks. Only recently researchers have been working on an integration of both techniques [1–5, 7].

Cao et al. [3] and Kimbrel and Karlin [7] introduced a theoretical model for studying "Integrated Prefetching and Caching" (IPC) that we will also use in this paper. We first consider single disk systems. A set S of memory blocks resides on one disk. At any time a cache can store k of these blocks. The system must serve a request sequence $\sigma = \sigma(1), \ldots, \sigma(n)$, where each request $\sigma(i)$, $1 \leq i \leq n$, specifies a memory block. The service of a request takes one time unit and can only be accomplished if the requested block is in cache. If a requested block is not in cache, it must be fetched from disk, which takes F time units, where $F \in \mathbb{N}$. If a missing block is fetched immediately before its reference, then the processor has to stall for F time units. However, a fetch may also overlap with the service of requests. If a fetch is started i time units before the next reference

to the block, then the processor has to stall for only $\max\{0, F-i\}$ time units. In case $i \geq 1$, we have a real prefetch. Of course, at most one fetch operation may be executed at any time. Once a fetch is initiated, a block must be evicted from cache in order to make room for the incoming block. The goal is to minimize the processor stall time, or equivalently the *elapsed time*, which is the sum of the processor stall time and the length n of the request sequence.

In parallel disk systems with D disks we have D sets of memory blocks $S_1, \ldots, S_D$, where S_d is the set of blocks that reside on disk d, $1 \leq d \leq D$. We assume that each block in the system is located on only one of the disks. The main advantage of parallel disk systems is that blocks from different disks may be fetched in parallel. Thus if the processor has to stall at some point in time, then all the fetches currently being active advance towards completion. If a fetch is initiated, we may evict any block from cache, which corresponds to the model that blocks are read-only. Again the goal is to minimize the processor stall time.

Cao et al. [3, 4] studied IPC in single disk systems. They presented simple combinatorial algorithms, called *conservative* and *aggressive*, that run in polynomial time and approximate the elapsed time. *Conservative* achieves an approximation factor of 2, whereas *aggressive* achieves a better factor of $\min\{2, 1+F/k\}$. Karlin and Kimbrel [7] investigated IPC in parallel disk systems and presented a polynomial-time algorithm whose approximation guarantee on the elapsed time is $(1 + DF/k)$. In [1] Albers, Garg and Leonardi developed a polynomial-time algorithm that computes an optimal prefetching/caching schedule for single disk systems. For parallel disk systems they developed a polynomial-time algorithm that approximates the stall time. The algorithm achieves an approximation factor of D, using at most $D-1$ extra memory locations in cache. All the results presented in [1] are based on a linear program formulation.

In this paper we show that IPC in single and parallel disk systems can be formulated as a min-cost multicommodity flow problem and, exploiting special properties of the network, can be solved using combinatorial methods. These results are presented in Sec. 2. We first investigate the single disk problem. We describe the construction of the network and establish relationships between min-cost multicommodity flows and prefetching/caching schedules. We prove that a combinatorial approximation algorithm by Kamath et al. [6] for computing min-cost multicommodity flows, when applied to our network, computes an optimal prefetching/caching schedule in polynomial time. We then generalize our multicommodity flow formulation to parallel disk systems. With minor modifications of the original network we are able to apply the algorithm by Kamath et al. [6] again. We derive a combinatorial algorithm that achieves a D-approximation on the stall time, using at most $D-1$ extra memory location in cache. Thus, the results presented in [1] can also be obtained using combinatorial techniques.

For parallel disk systems, D is the best approximation factor on the stall time currently known. This factor D is caused by the fact that the approach in [1] heavily overestimates the stall times in prefetching/caching schedules: Stall time is counted separately on each disk, i. e. no advantage is taken of the fact that prefetches executed in parallel simultaneously benefit from a processor

stall time. In Sec. 3 we develop improved approximation guarantees that are bounded away from D. We are able to formulate a trade-off. For any $z \in \mathbb{N}$, we achieve an approximation factor of $2(D/z)$ at the expense of a running time that grows exponentially with z. If the number D of disks is constant, we obtain a 2-approximation. For the special case $D = 2$ we also give a better 1.5-approximation. Again, our solutions need $D-1$ extra memory locations in cache. The improved approximation algorithms can also be obtained using min-cost multicommodity flows. However, for the sake of clarity and due to space limitations we present an LP-formulation in this extended abstract.

2 Modeling IPC by Network Flows

We first consider single disk systems. We build up our combinatorial algorithm in several steps. Given a request sequence σ, we first construct a network $G = (V, E)$ with several commodities such that an integral min-cost flow corresponds to an optimal prefetching/caching schedule for σ, and vice versa. Of course, an algorithm for computing min-cost multicommodity flows does not necessarily return an integral flow when applied to our network. We show that a non-integral flow corresponds to a *fractional* prefetching/caching schedule in which we can identify an integral schedule using a technique from [1].

The main problem we are faced with is that we know of no combinatorial polynomial-time algorithm for computing a (non-integral) min-cost flow in our network. We solve this problem by applying a combinatorial approximation algorithm by Kamath et al. [6]. For any $\varepsilon \geq 0$, $\delta \geq 0$, the algorithm computes a flow such that a fraction of at least $1 - \varepsilon$ of each demand in the network is satisfied and the cost of the flow is at most $(1 + \delta)$ times the optimum. Unfortunately, the flow computed by the algorithm, when applied to our network, does not correspond to a feasible fractional prefetching/caching schedule: It is possible that (a) more than one block is fetched from disk at any time and (b) blocks are not completely in cache when requested. We first reduce the flow in the network to resolve (a). This reduces the extent to which blocks are in cache at the time of their request even further. We then show that, given such flow, we can still derive an optimal prefetching/caching schedule, provided that ε and δ are chosen properly.

2.1 The Network

Let σ be a request sequence consisting of n requests. We construct a network $G = (V, E)$ with $n + 1$ commodities. Associated with each request $\sigma(i)$ is a commodity i, $1 \leq i \leq n$. This commodity has a source s_i, a sink t_i and demand $d_i = 1$. Let a_i be the block requested by $\sigma(i)$. For each request $\sigma(i)$, we introduce vertices x_i and x'_i. These vertices are linearly linked, i. e. there are edges (x_i, x'_i), $1 \leq i \leq n$, and edges (x'_i, x_{i+1}), $1 \leq i \leq n - 1$, each with capacity k and cost 0. Intuitively, this sequence of vertices and edges represents the cache. If commodity i flows through (x_j, x'_j), then block a_i is in cache when $\sigma(j)$ is served.

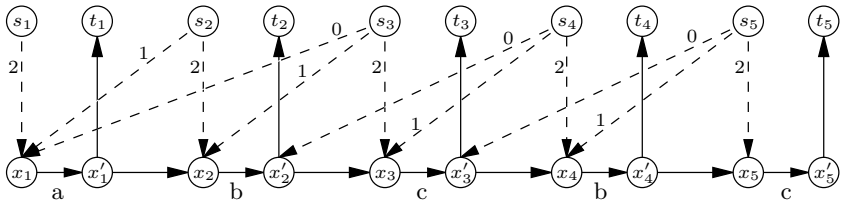

Fig. 1. Sketch of the network for request sequence *abcbc* and $F = 2$

To ensure that a_i is in cache when $\sigma(i)$ is served, we insert an edge (x'_i, t_i) with capacity 1 and cost 0, and there are no other edges into t_i or x'_i, i. e. commodity i must pass through (x_i, x'_i).

Let p_i be the time of the previous request to a_i, i. e. p_i is the largest j, $j < i$, such that a_i was requested by $\sigma(j)$. If a_i is requested for the first time in σ, then we set $p_i = 0$. To serve $\sigma(i)$, block a_i can (1) remain in cache after $\sigma(p_i)$ until request $\sigma(i)$, provided that $p_i > 0$, or can (2) be fetched into cache at some time before $\sigma(i)$. To model case (1) we introduce an edge (s_i, x'_{p_i}), if $p_i > 0$, with capacity 1 and cost 0. To model case (2) we essentially add edges (s_i, x_j), for $j = p_i+1, \ldots, i$, indicating that a fetch for a_i is initiated starting at the service of $\sigma(j)$. For the special case $j = i$ the edge represents a fetch executed immediately before $\sigma(i)$. If $i-j < F$, then the processor has to stall for $F-(i-j)$ time units and hence we assign a cost of $F-(i-j)$ to edge (s_i, x_j). Figure 1 illustrates this construction for the examplary request sequence $\sigma = abcbc$ and fetch time $F = 2$. Edges outgoing of a source s_i, $i \in \{1, \ldots, 5\}$, are labeled with their cost.

So far our construction allows a flow algorithm to saturate more than one of the edges that correspond to fetches executed simultaneously (consider, e. g. the edges (s_i, x_{i-1}) and (s_{i-1}, x_{i-2}) for some i such that $\sigma(i-2)$, $\sigma(i-1)$ and $\sigma(i)$ are pairwise distinct). However, we have to make sure that at most one fetch operation is executed at any time. Therefore, in our construction we split the "super edge" (s_i, x_j) into several parts. For any ℓ, $1 \le \ell \le n-1$, let $[\ell, \ell+1)$ be the time interval starting at the service of $\sigma(\ell)$ and ending immediately before the service of $\sigma(\ell+1)$. Interval $[0, 1)$ is the time before the service of $\sigma(1)$.

We have to consider all the fetches being active at some time in $[\ell, \ell+1)$, for any fixed ℓ. A fetch for a_i starting at $\sigma(j)$, $j < i$, is active during $[\ell, \ell+1)$ for $\ell = j, \ldots, \min\{j+F, i\} - 1$. For any fixed i and j with $1 \le i \le n$ and $p_i+1 \le j < i$ we introduce vertices v^ℓ_{ij} and w^ℓ_{ij} where $\ell = j, \ldots, \min\{j+F, i\}-1$. These vertices are linked by edges of capacity 1 and cost 0. More specifically, we have edges $(v^\ell_{ij}, w^\ell_{ij})$, $\ell = j, \ldots, \min\{j+F, i\} - 1$, and edges $(w^\ell_{ij}, v^{\ell-1}_{ij})$, $\ell = j+1, \ldots, \min\{j+F, i\}-1$. The last vertex in this sequence, w^j_{ij}, is linked to x_j with an edge of cost 0 and capacity 1. Finally we add an edge (s_i, v^ℓ_{ij}), where $\ell = \min\{j+F, i\} - 1$, to the first vertex in this sequence with cost $F-(i-j)$ and capacity 1. In this construction we excluded the case $j = i$ because a fetch for a_i initiated at $\sigma(i)$ is somewhat special: The fetch is performed completely before $\sigma(i)$, i. e. it does not overlap with any request, and the processor stalls

for F time units. The fetch is active at some time during $[i-1, i)$. We introduce vertices $v_{i,i}^{i-1}$ and w_{ii}^{i-1} linked by an edge of capacity 1 and cost 0. Vertex w_{ii}^{i-1} is linked to x_i with an edge of the same capacity and cost. Finally, we have an edge (s_i, v_{ii}^{i-1}) of capacity 1 and cost F.

Next we describe the role of the $(n+1)$-st commodity, which is used to ensure that no two prefetches are performed at the same time. More precisely, we ensure that at most one prefetch is executed in any fixed interval $[\ell, \ell+1)$, $1 \leq \ell \leq n-1$. For any fixed ℓ, let f_ℓ be the number of prefetches whose execution overlaps with $[\ell, \ell+1)$, i. e.

$$f_\ell = |\{v_{ij}^\ell \mid 1 \leq i \leq n,\ p_i + 1 \leq j \leq i\}| \ . \tag{1}$$

Commodity $n+1$ has a source s_{n+1}, a sink t_{n+1} and a demand of $d_{n+1} = \sum_{\ell=1}^{n-1}(f_\ell - 1)$. The flow from s_{n+1} to t_{n+1} is routed through the edges $(v_{ij}^\ell, w_{ij}^\ell)$ and newly introduced "subsinks" t_{n+1}^ℓ, $1 \leq \ell \leq n-1$. For any pair of vertices v_{ij}^ℓ and w_{ij}^ℓ we introduce edges (s_{n+1}, v_{ij}^ℓ) and $(w_{ij}^\ell, t_{n+1}^\ell)$ with capacity 1 and cost 0. Additionally, we insert edges (t_{n+1}^ℓ, t_{n+1}) with capacity $f_\ell - 1$ and cost 0. Now consider a fixed interval $[\ell, \ell+1)$, $1 \leq \ell \leq n-1$. Every prefetch for some a_i initiated at $\sigma(j)$ that is active at some time during $[\ell, \ell+1)$ is represented by a "super edge" (s_i, x_j) and contains an edge $(v_{ij}^\ell, w_{ij}^\ell)$. For fixed ℓ the network contains f_ℓ such edges. The capacities $f_\ell - 1$ of the edges (t_{n+1}^ℓ, t_{n+1}) ensure that only one of the edges $(v_{ij}^\ell, w_{ij}^\ell)$ can carry a flow of commodity i, $i \leq n$. If two or more such edges were carrying flow of commodity $i \leq n$, then the capacity constraint would be violated at some edge $(t_{n+1}^{\ell'}, t_{n+1})$, for some $\ell' \neq \ell$, or demand d_{n+1} would not be satisfied.

The following lemma states that our network correctly models IPC on a single disk. Its proof is omitted in this extended abstract.

Lemma 1. *Any feasible integral flow of cost C in G corresponds to a feasible prefetching/caching schedule with stall time C for σ, and vice versa.*

2.2 Properties of Optimal Flows

We show that a non-integral flow in our network corresponds to a fractional prefetching/caching schedule, defined in the following way.

Definition 2. *Given an instance of the problem IPC, we define the set of* fractional solutions *as a superset of the set of integral solutions to the instance. A fractional solution may deviate from an integral solution in the following way:*

- *The amount to which a block resides in cache may take a fractional value between* 0 *and* 1*. However, this amount must be* 1 *while the block is requested.*
- *Fractional parts of blocks in cache arise due to* partial evictions *or* partial fetches*. For each time interval, the net amount of blocks fetched must not be larger than the net amount of blocks evicted, and the net amount of blocks fetched must not exceed* 1*.*
- *Stall times are accounted as follows: If a fetch to $\delta \in [0, 1]$ units of block $\sigma(j)$ is initiated starting at the service of reference $\sigma(i)$ and $j - i < F$ holds, we incur a stall time of $\delta(F - (j - i))$ time units.*

Loosely speaking, the main difference between integral and fractional solutions lies in the possibility to interrupt fetches and to leave parts of a block in cache between consecutive requests to it. Regarding the second item in the above definition we may assume w. l. o. g. that between any two consecutive references to a specific block, the points of time where the block is evicted from cache precede the ones where the block is fetched back.

Lemma 3. *Let G be the network obtained by transforming a request sequence σ of the problem IPC according to the construction in Sec. 2.1. A valid multi-commodity flow with cost C within the network G corresponds to a fractional prefetching/caching schedule with stall time C.*

The next lemma follows immeditately from [1]; it was shown that a fractional solution is a convex combination of polynomially many integral solutions.

Lemma 4. *Let L be a fractional solution to an input for IPC. There is a polynomial-time algorithm that computes an integral prefetching/caching schedule L^* from L where the stall time of L^* is less than or equal to the one of L.*

2.3 Applying the Approximation Algorithm

We show how to compute a flow in our network and how to derive an optimal prefetching/caching schedule. We apply the algorithm by Kamath et al. [6] by setting $\varepsilon := 1/(4F^2n^3)$ and $\delta := 1/(3nF)$. These settings have been derived from the easy-to-see upper bound $d_{n+1} \leq n^2F$ on the demand of commodity $n+1$.

As the approximation algorithm only satisfies a fraction of $1-\varepsilon$ of each commodity, the flow out of each source s_i, $i \in \{1,\ldots,n\}$, is lower bounded by $1-\varepsilon$. Moreover, commodity $n+1$ might lack an amount of $\varepsilon d_{n+1} \leq \varepsilon Fn^2$. We assume pessimistically that this leads to an additional "illegal" flow with value εFn^2 during a time interval $[\ell, \ell+1)$, $\ell \in \{1,\ldots,n-1\}$, in so far as edges representing fetches in that interval are not "congested" properly by commodity $n+1$.

Let $\varrho := 1 - \varepsilon d_{n+1} - \varepsilon$ be a crucial lower bound on the flow of commodities $1,\ldots,n$. We can transform the flow ϕ output by the approximation algorithm into a uniform flow ϕ' which directs exactly ϱ units of flow from s_i to t_i for any commodity $i \in \{1,\ldots,n\}$. The main idea is to reduce, for each edge, the flow of commodity i proportionally to the relative amount of flow of commodity i on the considered edge. Then we end up with a uniform flow ϕ' which does not "overflow" any interval $[\ell, \ell+1)$ and delivers the same amount for each commodity.

In view of Definition 2 and the equivalence described in Lemma 3, the flow ϕ' corresponds to a fractional solution to IPC in which all blocks have size ϱ. Flow ϕ' corresponds to a fractional solution to IPC in which all blocks have size ϱ and the number of cache slots is upper bounded by k/ϱ—hereinafter we call such a solution a *ϱ-solution*. According to Lemma 4, we may interpret the fractional solution which corresponds to ϕ' as a convex combination of *integral* ϱ-solutions.

In order to analyze the quality of the above-described convex combination of integral ϱ-solutions, we have to establish a lower bound on ϱ. As $d_{n+1} \leq Fn^2$,

we obtain $\varrho \geq 1 - \varepsilon d_{n+1} - \varepsilon \geq 1 - 2\varepsilon d_{n+1} \geq 1 - 1/2nF$. Next we estimate the cost C of the convex combination of ϱ-solutions. Since the approximation algorithm outputs flows with cost at most $(1+\delta)\,\mathrm{OPT}$, where OPT is the cost of an optimal schedule, and reducing flows to ϱ does never increase cost, the following upper bound on C holds:

$$C \leq (1+\delta)\,\mathrm{OPT} = \mathrm{OPT}\,/(3nF) + \mathrm{OPT} \leq \mathrm{OPT} + 1/3 \quad \text{since } \mathrm{OPT} \leq nF \ . \tag{2}$$

We underestimated the cost C of the convex combination of ϱ-solutions by an additive term of at most $n(1-\varrho)F$. This is due to the fact that each block corresponding to a specific commodity has size 1 in reality, but size ϱ in the convex combination. By increasing the block size (or, equivalently, the flow of the corresponding commodity), the cost can rise by at most $(1-\varrho)F$. Hence, the cost C' of the convex combination of integral solutions is at most

$$C' \leq C + n(1-\varrho)F \leq \mathrm{OPT} + 1/3 + nF/2nF < \mathrm{OPT} + 1 \ . \tag{3}$$

From $C' < \mathrm{OPT} + 1$ we conclude that the convex combination contains at least one integral solution with optimal costs. As the number of possible integral solutions is bounded by Fn^2 (see [1]), an optimal component, i. e. integral solution, within the convex composition can be computed in polynomial time. However, each integral solution originates from a ϱ-solution where a block has size ϱ. Since the cache is still k large, it remains to prove that no integral component of the convex composition does hold more than k blocks in cache concurrently.

Since, w. l. o. g., $k/(nF) < 1$ holds, the number of blocks of size ϱ held concurrently in cache is at most

$$\frac{k}{\varrho} \leq \frac{k}{1 - \frac{1}{2nF}} \leq k\left(1 + \frac{1}{nF}\right) < k + 1 \tag{4}$$

because $(1-\varepsilon')^{-1} \leq 1 + 2\varepsilon'$ for any $\varepsilon' \in [0, 1/2]$. Therefore, $(k+1)\varrho > k$ holds, and we would obtain a contradiction if an integral solution held more than k pages in cache concurrently. Finally, this implies that we have found a feasible and optimal prefetching/caching schedule. The overall running time of the approximation algorithm is $O^*(\varepsilon^{-3}\delta^{-3}c|E||V|^2)$, where c denotes the number of commodities and O^* means "up to logarithmic factors". As $|V| = O(n^2)$ and $|E| = O(n^2)$, we obtain the polynomial upper bound $O^*((nF)^3(n^3F^2)^3(n+1)n^2n^4) = O^*(n^{18})$. Now we state the main result of this section.

Theorem 5. *An optimal solution to an input for IPC can be computed by a combinatorial algorithm in polynomial time.*

2.4 Generalization to Multiple Disks

The solution developed for single disk systems can be generalized to multiple disks. Due to space limitations, we only state the main result here.

Theorem 6. *There is a combinatorial polynomial-time algorithm which computes a D-approximation to an input for IPC if the number of disks is D and there are $D-1$ slots of extra cache available.*

3 Improving the Approximation Factor

In this section we return to the linear program by Albers, Garg and Leonardi [1] for the multiple disk case and improve its approximation factor. If the number D of disks is constant, we achieve a 2-approximation. We know that our approach leads to a linear program which can also be stated as a min-cost multicommodity flow problem. We omit that representation as we consider the improved approximation guarantee to be the most important contribution.

3.1 Bundling Intervals

The drawback of the LP formulation by Albers, Garg and Leonardi [1] is that it overestimates the stall time of prefetching/caching schedules. We present an LP that counts stall time more accurately. As in [1] we represent time periods in which fetch operations are executed by open intervals $I = (i, j)$, with $i = 0, \ldots, n-1$ and $j = i+1, \ldots, n$, where $n = |\sigma|$ is the length of the given request sequence. Such an interval $I = (i, j)$ corresponds to the time period starting *after* the service of $\sigma(i)$ and ending before the service of $\sigma(j)$. Its length is $|I| = j - i - 1$. If $|I| < F$, then $F - |I|$ units of stall time must be scheduled in the fetch operation. Since fetches take F time units, we can restrict ourselves to intervals with $j \leq i + F + 1$. For each potential interval I we introduce a copy I^d for each disk $d \in \{1, \ldots, D\}$. Let $\mathcal{I}$ be the resulting set of all these intervals. The LP in [1] determines which intervals of $\mathcal{I}$ should execute prefetches. Stall times are counted separately for the intervals and disks, which causes the overestimate.

The main idea of our LP is to form *bundles* of intervals and treat each bundle as a unit: In any bundle either all the intervals or no interval will execute a fetch. We next introduce the notion of bundles and need one property of optimal prefetching/caching schedules. An interval $I = (i_1, i_2)$ *properly contains* interval $J = (j_1, j_2)$ (which is not necessarily associated with the same disk) if $i_1 < j_1$ and $j_2 < i_2$ hold. The proof of the next lemma is omitted.

Lemma 7. *An optimal (fractional or integral) prefetching/caching schedule for a system with D disks does not include fetch intervals properly containing each other.*

Definition 8. *A set of intervals B, $|B| \neq \emptyset$, is called a* bundle *if B contains at most one interval from each disk and is* overlapping. *A set of intervals B is called overlapping if it includes no intervals properly containing each other but has for all but one $I = (i_1, i_2) \in B$ some interval $J = (j_1, j_2) \in B$, $J \neq I$, such that $j_1 \geq i_1$ and J overlaps with I. Two intervals $I = (i_1, i_2)$ and $J = (j_1, j_2)$, $i_1 \leq j_1$, are called overlapping if either $j_1 < i_2 - 1$ is valid, or $j_1 = i_2 - 1$ and additionally $i_2 - i_1 - 1 < F$ hold.*

Fix a $z \in \mathbb{N}$ with $z \leq D$. We will bundle intervals from up to z disks. In this extended abstract we assume for simplicity that $D/z \in \mathbb{N}$. We partition the disk set into D/z sets $\{1, \ldots, z\}, \{z+1, \ldots, 2z\}, \ldots, \{D-z+1, \ldots, D\}$. Now let $\mathcal{B}_z$ be the set of all the bundles composed of intervals from $\mathcal{I}$, with the additional

restriction that the intervals of a bundle must come from the same subset of the disk partition. One can show that $|\mathcal{B}_z| \leq n(F+1)^{2z}(D/z)z!$.

We are nearly ready to state the extended linear program for IPC and $D > 1$. For each bundle $B \in \mathcal{B}_z$, we introduce a variable $x(B)$ which is set to 1 if a prefetch is performed in all intervals in bundle B, and is set to 0 otherwise. In order to specify which blocks are fetched and evicted we use variables $f_{I^d,a}$ and $e_{I^d,a}$ for all $I^d \in \mathcal{I}$ and all blocks a. Variable $f_{I^d,a}$ (respectively $e_{I^d,a}$) is equal to 1 if a is fetched (respectively evicted) in I^d. Of course $e_{I^d,a} = f_{I^d,a} = 0$ if a does not reside on disk d. For a bundle $B \in \mathcal{B}_z$, let $s(B)$ be the minimum stall time needed to execute fetches in all the intervals of B assuming that no other fetch operations are performed in the schedule. The value $s(B)$ can be computed as follows. Let $(a_1, b_1), \ldots, (a_m, b_m)$ be the sequence of all intervals in B obtained by sorting them by increasing end index, where intervals with the same end index are sorted by increasing start index breaking ties arbitrarily. One can easily verify that in an optimal schedule for B, stall times occur at the end of intervals, the fetch in (a_1, b_1) is started at the latest point in time (i. e. immediately before request a_2 if $b_1 \neq a_1$ and after a_1 otherwise) whereas the fetches in (a_i, b_i), $i \geq 2$, are started at the earliest point in time. We determine the amounts of stall times needed at the end of intervals. Let $i_1, i_2, \ldots, i_{m'}$ with $m' \leq m$ be the sequence obtained from $b_1, \ldots, b_m$ by eliminating multiple occurrences of the same value and keeping only the indices i_j such that $b_{i_j+1} \neq b_{i_j}$. By definition, $i_0 := 0$ and $b_{i_0} := 0$. For $j = 1, \ldots, m'$, interval (a_{i_j}, b_{i_j}) is the shortest interval with end index b_{i_j} and determines the stall time to be inserted before that request. The function $h\colon \{b_{i_1}, \ldots, b_{i_{m'}}\} \to \mathbb{N}$ that indicates the actual stall time needed before request b_{i_j} is defined inductively, for $j = 1, \ldots, m'$, as follows:

$$h(b_{i_j}) := \max\Big\{0, F - (b_{i_j} - a_{i_j} - 1) - \sum_{r \in \{b_{i_1}, \ldots, b_{i_{j-1}}\}\colon r \in \{a_{i_j}+1, \ldots, b_{i_j}-1\}} h(r)\Big\} \ .$$

Using this definition we have $s(B) := \sum_{j=1}^{m'} h(b_{i_j})$.

In order to refer to individial disks, we need for $d \in \{1, \ldots, D\}$ the projections $\pi_d\colon \mathcal{B}_z \to \mathcal{I}$, where $\pi_d(B) = I$ if $I \in B$ and I resides on disk d, and $\pi_d(B) = \emptyset$ if B contains no interval associated with disk d. The value of π_d is well defined since at most one interval from each disk is part of a bundle. Now the **extended linear program** reads as follows. Minimize the objective function

$$\sum_{B \in \mathcal{B}_z} x(B)s(B) \tag{5}$$

subject to

$$\forall i \in \{1, \ldots, n\}, \forall d \sum_{B \in \mathcal{B}_z\colon \pi_d(B) \supseteq (i-1, i+1)} x(B) \leq 1 \tag{6}$$

$$\forall d, \forall I^d \sum_a f_{I^d,a} = \sum_a e_{I^d,a} \leq \sum_{B \in \mathcal{B}_z\colon \pi_d(B) = I^d} x(B) \tag{7}$$

$$\forall a, \forall i \in \{1, \ldots, n_a\} \quad \sum_{I \in \mathcal{I}:\ I \subseteq (a_i, a_{i+1})} f_{I,a} = \sum_{I \in \mathcal{I}:\ I \subseteq (a_i, a_{i+1})} e_{I,a} \leq 1 \tag{8}$$

$$\forall a \sum_{I \in \mathcal{I}:\ I \subseteq (0, a_1)} f_{I,a} = 1, \qquad \forall a \sum_{I \in \mathcal{I}:\ I \subseteq (0, a_1)} e_{I,a} = 0 \tag{9}$$

$$\forall a, \forall i \in \{1, \ldots, n_a\} \quad \sum_{I \in \mathcal{I}:\ I \supseteq (a_i - 1, a_i + 1)} f_{I,a} = \sum_{I \in \mathcal{I}:\ I \supseteq (a_i - 1, a_i + 1)} e_{I,a} = 0 \tag{10}$$

$$\forall I \in \mathcal{I}, \forall a \quad f_{I,a}, e_{I,a} \in \{0, 1\} \tag{11}$$

$$\forall B \in \mathcal{B}_z \quad x(B) \in \{0, 1\} \ . \tag{12}$$

Here we have taken over some terminology from the original formulation in [1]. The first set of contraints ensures that for each disk and each point of time, the amount of fetch is at most 1. The second set of constraints guarantees for each interval on every disk that the amount of blocks fetched in the interval is at most the overall amount of blocks evicted in that interval. For a specific interval I^d, we allow a bundle variable $x(B)$, where B contains I^d, to take value 1; observe that B might consist of I^d as the only element. If a bundle variable $x(B)$ is 1, the second set of constraints allows fetches in all intervals belonging to the bundle. Please note that constraint (6) only ensures that at most one prefetch operation may be executed while serving a request. Especially, it allows prefetches to be started in the midst of stall times, such the exact point of time where a prefetch is started may be unspecified if there is stall time at the beginning of an interval. We will argue later that this freedom is justified. Constraints (8)–(11) have been adapted from the LP formulation in [1] and ensure that a block is in cache at the time of its reference. The objective function finally counts the s-values, which are related to parallel stall times, for bundles whose variables are 1. It remains to prove that a solution to the extended linear program induces a valid schedule whose stall time is counted at least once by the value of the objective function.

Consider an arbitrary integer solution to the extended LP, which specifies an assignment to the variables $f_{I,a}, e_{I,a}$ and $x(B)$. Using $f_{I,a}$ and $e_{I,a}$, we know between which requests a prefetch operation must be started, but may choose the exact point of time of the start if the related requests are intermitted by stall time. We inductively construct a schedule whose stall time is bounded from above by the value of the objective function. First, we sort the bundles B for which $x(B) = 1$ holds by increasing maximum end index (of the intervals in the bundle) and, if equality holds, by increasing minimum start index. Let $B_1, \ldots, B_m$ be the resulting sequence. Suppose that we have already constructed a schedule for $B_1, \ldots, B_{r-1}$. For bundle B_r, we have to schedule fetches and evictions for those blocks whose variables $f_{I,a}$ and $e_{I,a}$ have been set to 1 and $I \in B_r$. We use the notation introduced for defining the stall times $s(B)$ on page 9. Let $(a_1, b_1), \ldots, (a_m, b_m)$ be the sequence of intervals in B_r. We first insert $h(b_{i_\ell})$ units of stall time before b_{i_ℓ}, $\ell \in \{1, \ldots, m'\}$, and then schedule the fetches in (a_i, b_i), $i \in \{1, \ldots, m\}$, as follows. The fetch in (a_1, b_1) is started at the latest possible point in time. More precisely, if $a_1 < b_1$, we start the fetch with the service of request $a_1 + 1$; otherwise the fetch is scheduled immediately before the

service of b_1. The fetches in (a_i, b_i), $i \geq 2$, are started at the earliest point in time after a_i such that the required disk is available. The definition of the h-values ensures that we reserve at least F time units for each fetch irrespectively of stall times which are caused by fetches in intervals from bundles $B_1, \ldots, B_{r-1}$. In fact, a reserved time interval might even be longer. However, this is no problem. The fetch simply completes after F time units and the corresponding disk is then idle for the rest of the interval. Thus, the constructed schedule is feasible and we insert exactly $\sum_{j=1}^{m} s(B_j)$ units of stall time.

3.2 Achieving the $(2D/z)$-Approximation

Lemma 9. *The extended linear program for D disks has an integral solution of cost at most $(2D/z)\,\mathrm{OPT}$.*

Proof. Suppose we have been given an optimal integral prefetching/caching schedule of stall time OPT. We restrict ourselves to an arbitrary subset of the partition

$$\bigcup_{i=0}^{D/z-1} \{iz+1, iz+2, \ldots, iz+z\} \tag{13}$$

of the disk set $\{1, \ldots, D\}$. W. l. o. g., this subset is $\{1, \ldots, z\}$. We consider only stall times caused by fetches in intervals associated with disks $\{1, \ldots, z\}$. In the following, we specify an assignment to the variables associated with disks $\{1, \ldots, z\}$ such that the stall time that arises by executing only the fetches on disks $\{1, \ldots, z\}$ is counted at most twice in the objective funtion of the linear program. Repeating this process for all the subsets of the above partition, we obtain an assignment to all the variables $x(B)$ for $B \in \mathcal{B}_z$. As the objective function is separable with respect to the bundles in $\mathcal{B}_z$ and therefore with respect to the (D/z) subsets of the above partition, we count a specific stall time in the optimal prefetching/caching schedule at most $2(D/z)$ times.

By $\mathcal{I}' \subseteq \mathcal{I}$ we denote the set of all intervals associated with the disk set $\{1, \ldots, z\}$ in which prefetches are performed. According to Lemma 7, we have no intervals properly containing each other in the set $\mathcal{I}'$. Therefore, we can order the intervals in $\mathcal{I}'$ by increasing start points and (if these are equal) by increasing end points. Let $I_1, \ldots, I_m$ be the resulting sequence. We partition $\mathcal{I}'$ into bundles according to the following greedy algorithm.

$B := \emptyset$
for $j = 1, \ldots, m$ **do**
  if interval $I_j \cup B$ is a bundle **then** set $B := I_j \cup B$
  else output B as an element of the partition and set $B := \emptyset$.

Let $B_1 \cup \cdots \cup B_\ell$, $\ell \leq m$, be the partition of $\mathcal{I}'$ obtained by this process. Our solution to the linear program is constructed by setting $x(B_j)$ to 1 for $j \in \{1, \ldots, \ell\}$. The variables $f_{I,a}$ and $e_{I,a}$ are set according to which blocks are fetched in the intervals of the considered bundle. This process is repeated for each subset of the partition (13) of the disk set. All remaining variables are zero.

In the following, we revert to the subset $\{1,\ldots,z\}$ and denote by OPT^* the stall time incurred if counting only fetch intervals associated with disks $\{1,\ldots,z\}$. For bundles B_j, $j \in \{1,\ldots,\ell\}$, let $s'(B_j)$ be the sum of the stall times in the optimal schedule between the start of the first fetch in B_j and the completion of the last fetch in B_j. Obviously, $s(B_j) \leq s'(B_j)$. One can show that $\sum_{j=1}^{\ell} s'(B_j) \leq 2 \cdot \mathrm{OPT}^*$, which implies that the value of the objective function is at most $2\,\mathrm{OPT}$. □

Lemma 9 implies that there is also a fractional solution to the extended LP with cost at most $(2D/z)\,\mathrm{OPT}$. Using techniques from [1], we can convert a fractional solution to the extended LP with cost C to an integral prefetching/caching schedule with stall time C if $D-1$ extra memory locations in cache are available. Thus we obtain an approximation algorithm of factor $2D/z$.

Theorem 10. *There is a polynomial $p(n)$ such that for each $z \in \{1,\ldots,D\}$ with $D/z \in \mathbb{N}$ there is a algorithm with running time $O(p(n) \cdot n \cdot (F+1)^z z!)$ that computes a $2D/z$-approximation for IPC provided that $D-1$ extra memory locations are available in cache.*

Finally, for $D = 2$ the extended linear program does not seem to give an improved approximation. However, in this special case, we can show an even better approximation factor of 1.5.

Lemma 11. *If $D = 2$, the extended linear program with $z = 2$ has an integral solution of cost at most $1.5 \cdot \mathrm{OPT}$.*

References

1. S. Albers, N. Garg, and S. Leonardi. Minimizing stall time in single and parallel disk systems. In *Proc. 30th Annual ACM Symp. on Theory of Computing*, pages 454–462, 1998.
2. B. Bershad, P. Cao, E. W. Felten, G. A. Gibson, A. R. Karlin, T. Kimbrel, K. Li, R. H. Patterson, and A. Tomkins. A trace-driven comparison of algorithms for parallel prefetching and caching. In *Proc. ACM SIGOPS/USENIX Assoc. Symp. on Operating System Design and Implementation (OSDI)*, 1996.
3. P. Cao, E. W. Felten, A. R. Karlin, and K. Li. A study of integrated prefetching and caching strategies. In *Proc. ACM Int. Conf. on Measurement and Modeling of Computer Systems (SIGMETRICS)*, pages 188–196, 1995.
4. P. Cao, E. W. Felten, A. R. Karlin, and K. Li. Implementation and performance of integrated application-controlled caching, prefetching and disk scheduling. *ACM Transaction on Computer Systems*, 14(4):311–343, 1996.
5. G. A. Gibson, E. Ginting, R. H. Patterson, D. Stodolsky, and J. Zelenka. Informed prefetching and caching. In *Proc. 17th Int. Conf. on Operating Systems Principles*, pages 79–95, 1995.
6. A. Kamath, O. Palmon, and S. Plotkin. Fast approximation algorithm for minimum cost multicommodity flow. In *Proc. 6th Annual ACM-SIAM Symp. on Discrete Algorithms*, pages 493–501, 1995.
7. R. Karlin and T. Kimbrel. Near-optimal parallel prefetching and caching. In *Proc. 37th Annual Symp. on Foundations of Computer Science*, pages 540–549. IEEE Society, 1996.