Integrated Prefetching and Caching with Read and Write Requests

Susanne Albers[1] and Markus Büttner[1]

Institute of Computer Science, Freiburg University,
Georges-Köhler-Allee 79, 79110 Freiburg, Germany.
{salbers,buettner}@informatik.uni-freiburg.de

Abstract. All previous work on integrated prefetching/caching assumes that memory reference strings consist of read requests only. In this paper we present the first study of integrated prefetching/caching with both read and write requests. For single disk systems we analyze popular algorithms such as *Conservative* and *Aggressive* and give tight bounds on their approximation ratios. We also develop a new algorithm that performs better than *Conservative* and *Aggressive*. For parallel disk systems we present a general technique to construct feasible schedules. The technique achieves a load balancing among the disks. Finally we show that it is NP-complete to decide if an input can be served with f fetch and w write operations, even in the single disk setting.

1 Introduction

Prefetching and caching are powerful and extensively studied techniques to improve the performance of storage hierarchies. In prefetching missing memory blocks are loaded from slow memory, e.g. a disk, into cache before their actual reference. Caching strategies try to keep actively referenced blocks in cache. Both techniques aim at reducing processor stall times that occur when requested data is not available in cache. Most of the previous work investigated prefetching and caching in isolation although they are strongly related: When prefetching a block, one has to evict a block from cache in order to make room for the incoming block. Prefetch operations initiated too early can harm the cache configuration. Prefetch operations started too late diminish the effect of prefetching. Therefore, there has recently been considerable research interest in integrated prefetching and caching [1, 2, 4–9]. The goal is to develop strategies that coordinate prefetching and caching decisions.

All the previous work on integrated prefetching/caching assumes that memory reference strings consist of read requests only, i.e. we only wish to read data blocks. In other words, memory blocks are read-only and do not have to be written back to disk when they are evicted from cache. However, in practice reference strings consist of both read and write requests. In a write request we wish to modify and update a given data block. Of course, modified blocks must be written to disk when they are evicted from cache.

In this paper we present the first study of integrated prefetching/caching with read and write requests. It turns out that integrated prefetching/caching is considerably more complicated in the presence of write requests. The problem is that prefetch and write operations compete with each other and it is not clear when to schedule which disk operation. Moreover, compared to the read-only case, it is not true anymore that in a prefetch operation we always evict the block from cache whose next request is furthest in the future. To save a write-back operation it might be better to evict an unmodified block, even if it is requested again soon. Finally, even if it were known when to initiate write operations, there is no simple rule that determines which blocks to write to disk.

Cao et al. [4] introduced a formal model for integrated prefetching/caching. We also use this model but generalize it to take into account read and write requests. We are given a request sequence $\sigma = r_1, \ldots, r_n$ consisting of n requests. Each request specifies the block to be accessed and the type of reference. If $r_i = b_i$, then r_i is a read request to block b_i. If $r_i = b_i^*$, then the reference is a write request where we want to modify b_i. We first assume that all blocks reside on a single disk. To serve a request, the requested block must be in cache. The cache can simultaneously hold k blocks. Serving a request to a block in cache takes 1 time unit. If a requested block is not in cache, then it must be fetched from disk, which takes F time units. A fetch operation may overlap with the service of requests to blocks already in cache. If a fetch, i.e. a prefetch, of a block is initiated at least F requests before the reference to the block, then the block is in cache at the time of the request and no processor stall time is incurred. If the fetch is started only i, $i < F$, requests before the reference, then the processor has to stall for $F - i$ time units until the fetch is finished. When a fetch operation is initiated, a block must be evicted from cache to make room for the incoming block. A block that was modified since the last time it was brought into cache can only be evicted if it has been written back to disk after its last write request. Such a write-back operation takes W time units and can be scheduled any time before the eviction. If the operation overlaps with the service of $i \leq W$ requests, then $W - i$ units of processor stall time are incurred to complete the write operation. In this submission, unless otherwise stated we assume for simplicity that $W = F$. The goal is to minimize the total processor stall time incurred on the entire request sequence. This is equivalent to minimizing the elapsed time, which is the sum of the processor stall time and the length of the request sequence. We emphasize here that the input σ is completely known in advance.

To illustrate the problem, consider a small example. Let $\sigma = b_1, b_2^*, b_2, b_3, b_4^*, b_3, b_4, b_3, b_5, b_1, b_4, b_2$. Assume that we have a cache of size $k = 4$ and that initially blocks b_1, b_2, b_3 and b_4 reside in cache. Let $F = W = 3$. The first missing block is b_5. We could initiate the fetch for b_5 when starting the service of the request b_2^*. The fetch would be executed while serving requests b_2^*, b_2 and b_3. When starting this fetch, we can only evict b_1, which is requested again after b_5. We could initiate the fetch for b_1 when serving request b_5 and evict b_3. Two units of stall time would be incurred before request b_1, so that the total elapsed time is equal to 14 time units. A better option is to write b_2 back to disk after request b_2^* and then to initiate a fetch for b_5 by evicting b_2. Both disk operations finish in time before request b_5 because the write operation may overlap with the service of the read request to b_2. When serving request b_5 we could start fetching b_2 by evicting b_3. Again this operation would be finished in time so that the elapsed time of this schedule is equal to 12 time units.

Integrated prefetching and caching is also interesting in parallel disk systems. Suppose that we have D disks and that each memory block always resides on exactly one of the disks. Fetch and write operations on different disks may be executed in parallel. Of course we can take advantage of the parallelism given by a multiple disk system. If the processor incurs stall time to wait for the completion of a fetch or write operation, then fetch and write operations executed in parallel on other disks also make progress towards completion during that time. Again we wish to minimize the total elapsed time.

Previous work: As mentioned before, all previous work on integrated prefetching/caching [1, 2, 4–9] assumes that request sequences consist of read request only. Cao et al. [4], who initiated the research on integrated prefetching/caching, introduced two popular algorithms called *Conservative* and *Aggressive* for the single disk problem. *Conservative* performs exactly the same cache replacements as the optimum offline paging algorithm [3] but starts each fetch at the earliest possible point in time. Cao et al. showed that *Conservative* achieves an approximation ratio of 2, i.e. for any request sequence the elapsed time of *Conservative*'s schedule is at most twice the elapsed time of an optimal schedule. This bound is tight. The *Aggressive* algorithm starts prefetch operations at the earliest reasonable point in time. Cao et al. proved that *Aggressive* has an approximation ratio of at most $\min\{1 + F/k, 2\}$ and showed that this bound is tight for $F = k$. Kimbrel and Karlin [7] analyzed *Conservative* and *Aggressive* in parallel disk systems and showed that the approximation guarantees are essentially equal to D. They also presented an algorithm called *Reserve Aggressive* and proved an approximation guarantee of $1 + DF/k$.

In [1] it was shown, that an optimal prefetching/caching schedule for a single disk can be computed in polynominal time based on a linear programming approach. The approach was extended to parallel disk systems and gave a D-approximation algorithm for the problem of minimizing the stall time of a schedule. The algorithm uses $D - 1$ extra memory locations in cache. The complexity of the parallel disk problem is still unknown.

Our contribution: This paper is an in-depth study of integrated prefetching/caching with read an write requests. We first address the single disk problem. In Section 2 we investigate implementations of *Conservative* and *Aggressive* and prove that *Conservative* has an approximation ratio of 3. We show that this bound is tight. We also show that *Aggressive* achieves an approximation guarantee of $\min\{2 + 2F/k, 4\}$ and that this bound is tight for $F = k$. Hence, surprisingly, for large ratios of F/k *Conservative* performs better than *Aggressive*. This is in contrast to the algorithms' relative performance in the read-only case.

In Section 3 we develop a new prefetching/caching algorithm that has an approximation ratio of 2 and hence performs better than *Conservative* and *Ag-*

gressive for all F and k. The basic idea of the new strategy is to delay cache replacements for a few time units.

The complexity of integrated prefetching/caching in the presence of write requests is unknown. However, Section 4 indicates that the problem is probably NP-hard. More precisely we prove that it is NP-complete to decide if a given request sequence can be served with at most f fetch and w write operations.

In Section 5 we study systems with D parallel disks. To speed up write operations, many parallel disk systems have the option of writing memory blocks back to an arbitrary disk and not necessarily to the disk where the block was stored previously. Of course, old copies of a block become invalid. Hence the disk where a given block resides may change over time. We present a general technique for constructing feasible prefetching/caching schedules in two steps. In the first step an algorithm determines fetch and write operations without considering on which disks the involved blocks reside. The second step assigns disks to all the fetch and write operations so that a load balancing is achieved for all the disks. Using a parallel, synchronized implementation of the *Conservative* algorithm in step 1 we obtain schedules whose elapsed time is at most 5 times the elapsed time of an optimal schedule plus an additive term that depends on the initial disk configuration. Replacing *Conservative* by *Aggressive* and investing $\lceil D/2 \rceil$ additional memory locations in cache the ratio of 5 drops to 4.

2 Analysis of *Conservative* and *Aggressive*

In this section we study the single disk setting. We extend the algorithms *Conservative* and *Aggressive* to request sequences consisting of both read and write requests and analyze their performance. *Conservative* executes exactly the same cache replacements as the optimum offline paging algorithm MIN [3] while initiating a fetch at the earliest reasonable point in time, i.e. the block to be evicted should not be requested before the block to be fetched. Modified blocks to be evicted may be written back to disk anytime before their eviction.

Theorem 1. *For any request sequence σ, the elapsed time of Conservative's schedule is at most 3 times the elapsed time of an optimal schedule. This bound is nearly tight, i.e. there are request sequences for which the ratio of Conservative's elapsed time to OPT's elapsed time is at least* $(3F+2)/(F+2)$.

Proof. The upper bound of 3 is easy to see. Consider an arbitrary request sequence σ and suppose that *Conservative* performs m cache replacements. In the worst case each replacements takes $2F$ time units: The algorithm may need $W = F$ time units to write the block to be evicted to disk; F time units are incurred to fetch the new block. Let $Cons(\sigma)$ be the total elapsed time of *Conservative*'s schedule. Then $Cons(\sigma) \leq |\sigma| + 2Fm$. *Conservative*'s cache replacements are determined by the MIN algorithm, which incurs the minimum number of cache replacements for any request sequence. Thus the optimum algorithm performs at least m cache replacements on σ, each of which takes at least F time units. We have $OPT(\sigma) \geq \max\{|\sigma|, Fm\}$ and hence $Cons(\sigma) \leq 3 \cdot OPT(\sigma)$.

For the construction of the lower bound we assume $k \geq 3$ and use $k-2$ blocks $A_1, \ldots, A_{k-2}$ as well as $k-2$ block $B_1, \ldots, B_{k-2}$ and three auxiliary blocks X, Y and Z. The requests to blocks $A_1, \ldots, A_{k-2}$, X, Y and Z will always be read requests whereas the requests to $B_1, \ldots, B_{k-2}$ will always be write requests. We use the asterisk to denote write requests, i.e. B_i^* is a write request modifying block B_i, $1 \leq i \leq k-2$. The request sequence is composed of subsequences σ_A and σ_B, where $\sigma_A = Z^F, A_1, Z^F, A_2, \ldots, Z^F, A_{k-2}$ and $\sigma_B = B_1^*, \ldots, B_{k-2}^*$. Let $\sigma' = \sigma_A, \sigma_B, Z, X, \sigma_A, \sigma_B, Z, Y$. The request sequence σ is an arbitrary number of repetitions of σ', i.e. $\sigma = (\sigma')^i$, for some positive integer i. To establish the lower bound we compare *Conservative*'s elapsed time on σ' to OPT's elapsed time on σ'. In the analysis the two algorithms start with different cache configurations but at the end of σ' the algorithms are again in their initial configuration.

We assume that initially *Conservative* has blocks $A_1, \ldots, A_{k-2}$, Y and Z in cache. During the service of the first σ_A in σ' *Conservative* first evicts Y to load B_1. This fetch overlaps with the service of requests. While serving the first σ_B, *Conservative* evicts B_i to load B_{i+1}, for $i = 1, \ldots, k-3$. Each operation generates $2F$ units of stall time because the evicted block has to be written to disk and the fetch cannot overlap with the service of requests. Then *Conservative* evicts B_{k-2} to fetch X. Again the operation takes $2F$ time units but can overlap with the service of the request to Z. The algorithm now has $A_1, \ldots, A_{k-2}$, X and Z in cache. It serves the second part of σ' in the same way as the first part except that in the beginning X is evicted to load B_1 and in the end B_{k-2} is evicted to load Y so that the final cache configuration is again $A_1, \ldots, A_{k-2}$, Y and Z. To serve σ', *Conservative* needs $Cons(\sigma') = 2((k-2)(F+1)+1+(k-2)(2F+1)) = 2((k-2)(3F+2)+1)$ time units.

For the analysis of OPT on σ' we assume that OPT has initially $B_1, \ldots, B_{k-2}$, Y and Z in cache. Blocks $B_1, \ldots, B_{k-2}$ and Z are never evicted. In the first part of σ' OPT evicts Y to load A_1 and then evicts A_i to load A_{i+1}, for $i = 1, \ldots, k-3$. These fetches are executed during the sevice of the requests to Z. While serving σ_B OPT evicts A_{k-2} to load X and the cache then contains $B_1, \ldots, B_{k-2}$, X and Z. In the second part of σ' the operations are the same except the roles of X and Y interchange. OPT's cache configuration at the end of σ' is again $B_1, \ldots, B_{k-2}$, Y and Z. The elapsed time is $OPT(\sigma') = 2((k-2)(F+1) + \max\{F, k-1\} + 1)$.

Hence, for $F < k$, the ratio of *Conservative*'s elapsed time to OPT's elapsed time on σ' is

$$\frac{Cons(\sigma')}{OPT(\sigma')} = \frac{(k-2)(3F+2)+1}{(k-2)(F+1)+k} \geq \frac{3F+2}{F+2}$$

and the desired bound follows by repeating σ' often enough. □

The *Aggressive* algorithm proposed by Cao et al. [4] works as follows. Whenever the algorithm is not in the middle of a fetch, it determines the next block b in the request sequence missing in cache as well as the block b' in cache whose next request is furthest in the future. If the next request to b is before the next request to b', then *Aggressive* initiates a fetch for b evicting b' from cache. We consider two extension of this algorithm to request sequences with read and write requests. If b' has to be written back to disk, then *Aggressive1* executes the write operation immediately before initiating the fetch for b and incurs F units of stall time before that fetch operation. *Aggressive2* on the the other hand overlaps the write-back operation as much as possible with the service of past and future requests at the expense of delaying the fetch for b. More formally, assume that *Aggressive2* finished the last fetch operation immediately before reqeust r_i and that r_j, $j \geq i$ is the first request such that the next request to b is before the next request to b'. If b' has to be written back to disk, start the wirte operation at the earliest $r_{i'}$, $i' \geq i$, such that b' is not requested between $r_{i'}$ and r_j. Overlap the operation as much as possible with the service of request.

While *Aggressive1* is very easy to analyze, *Aggressive2* is a more intuitive implementation of an aggressive strategy. We show that the approximation ratios of *Aggressive1* and *Aggressive2* increase by a factor of 2 relative to the approximation ratio of the standard *Aggressive* strategy. For *Aggressive1* this is easy to see. The algorithm performs exactly the same fetches and evictions as the *Aggressive* algorithm if all references were read requests. In the worst case each cache replacement takes $2F$ instead of F time units as the evicted block has to be written to disk. For *Aggressive2* the bound is not obvious. The problem is that *Aggressive2* finishes fetch operations on read/write request sequences later than *Aggressive* if all requests were read references. This affects the blocks to be evicted in future fetches and hence the cache replacements are different. The proof of the following theorem is omitted due to space limitations.

Theorem 2. *For any request sequence σ, the elapsed time of Aggressive1 and Aggressive2 on σ is at most $2\min\{1 + F/k, 2\}$ times the elapsed time of OPT on σ.*

Cao et al. [4] showed that for $F = k - 2$, the approximation ratio of *Aggressive* on request sequences consisting of read requests is not smaller than 2. We prove a corresponding bound for *Aggressive1* and *Aggressive2*.

Theorem 3. *For $F = k$, the approximation ratios of Aggressive1 and* Aggressive2 *are not smaller than 4.*

Proof. Let $k \geq 4$. For the construction of the lower bound we use $k - 3$ blocks $A_1, \ldots, A_{k-3}$, two blocks B_1 and B_2 as well as two blocks C_1 and C_2. Hence we work with a universe of size $k + 1$ so that there is always one block missing in cache. The reference to $A_1, \ldots, A_{k-3}, C_1$ and C_2 will always be write requests. The references to B_1 and B_2 will always be read requests.

Let $\sigma' = \sigma_1, \sigma_2$, where $\sigma_1 = A_1^*, B_1, A_2^*, \ldots, A_{k-3}^*, C_1^*, B_2, C_2^*$ and $\sigma_2 = A_1^*, B_2, A_2^*, \ldots, A_{k-3}^*, C_2^*, B_1, C_1^*$. The sequence σ_1 and σ_2 are identical except that the positions of B_1 and B_2 as well as C_1 and C_2 are interchanged. Let $\sigma = (\sigma')^i$, for some $i \geq 1$, i.e. σ' is repeated an arbitrary number of times. We compare the elapsed time of *Aggressive1* and *Aggressive2* on σ' to the elapsed time of OPT on σ' and assume that our approximation algorithms initially have $A_1, \ldots, A_{k-3}, B_1, B_2$ and C_1 in cache. We first consider *Aggressive1*. At the beginning of σ_1 all blocks in cache are requested before the missing block C_2. Hence *Aggressive1* can start the fetch for C_2 only after the service of the request to A_1

in σ_1. It incurs F units of stall time before the request to B_1 in order to write A_1 to disk and then evicts A_1 to load C_2. The fetch is completed immediately before the request to C_2, where 1 unit of stall time must be incurred. To load the missing block A_1, which is first requested in σ_2, *Aggressive1* writes C_1 to disk immediately before the request to C_2, generating F additional units of stall time before that request. Then C_1 is evicted to load A_1 and $F-1$ units of stall time must be incurred before the request to A_1. At that point *Aggressive1* has blocks $A_1, \ldots, A_{k-3}, B_1, B_2$ and C_2 in cache. The cache replacements in σ_2 are the as as in σ_1, except that the roles of C_1 and C_2 change. At the end of σ' *Aggressive1* has again blocks $A_1, \ldots, A_{k-3}, B_1, B_2$ and C_1 in cache, which is identical to the initial configuration.

Aggressive2's schedule on σ' is the same except that (a) $F+1$ units of stall time are incurred before the last request in σ_1 and σ_2 and (b) $2F-1$ units of stall time are generated before the first requests in σ_1 and σ_2. Hence both algorithms need $2(4F+1)$ time units to serve a subsequence σ'. The optimal algorithm always keeps $A_1, \ldots, A_{k-3}, C_1$ and C_2 in cache and only swaps B_1 and B_2. It needs $2(F+4)$ time units to serve σ'. Since $F = k$, we obtain a performance ratio of $(4k+1)/(k+4)$, which can be arbitrarily close to 4. $\square$

3 New algorithms

We present an algorithm that achieves an approximation ratio of 2 and hence performs better than *Conservative* and *Aggressive*. Intuitively, the following strategy delays the next fetch operation for F time units and then determines the best block to be evicted.

Algorithm Wait: Whenever the algorithm is not in the middle of a fetch or write operation, it works as follows. Let r_i be the next request to be served and r_j, $j \geq i$, be the next request where the referenced block is not in cache at the moment. If all the k blocks currently in cache are requested before r_j, then the algorithm serves r_i without initiating a write or fetch operation. Otherwise let $d = \min\{F, j-i\}$ and let S be the set of blocks referenced by write requests in $r_i, \ldots, r_{i+d-1}$. Immediately before serving r_{i+d} the algorithm initiates a fetch for the block requested by r_j. It evicts the block b whose next request is furthest in the future among blocks in cache that are not contained in S. If b has been modified since the last time it was brought into cache, the algorithm writes b to disk while serving $r_i, \ldots, r_{i+d-1}$, incurring $F-d$ units of stall time. Otherwise $r_i, \ldots, r_{i+d-1}$ are served without executing a write or fetch operation.

Theorem 4. *The Wait algorithm achieves an approximation ratio of 2.*

For the analysis of *Wait* (and *Aggressive2*) we need a dominance concept introduced by Cao et al. [4]. Given a request sequence σ, let $c_A(t)$ be the index of the next request at time t when A processes σ. Suppose that $c_A(t) = i$. For any j with $1 \leq j \leq n-k$, let $h_A(t,j)$ be the smallest index such that the subsequence $\sigma(i), \ldots, \sigma(h_A(t,j))$ contains j distinct block not in cache at time t. We also refer to $h_A(t,j)$ as *A's jth hole*. Given two prefetching/caching algorithms A and B, *A's cursor at time t dominates B's cursor at time t'* if $c_a(t) \geq c_B(t')$. Moreover, *A's holes at time t dominate B's holes at time t'* if $h_A(t,j) \geq h_B(t',j)$, for all $1 \leq j \leq n-k$. Finally *A's state at time t dominates B's state at time t'* if A's cursor at time t dominates B's cursor at time t' and A's holes at time t dominate B's holes at time t'. Cao et al. proved the following lemma.

Lemma 1. [4] *Suppose that A (resp. B) initiates a fetch at time t (resp. t') and that both algorithms fetch the next missing block. Suppose that A replaces the block whose next request is furthest in the future. If A's state at time t dominates B's state at time t', then A's state at time $t+F$ dominates B's state at time $t'+F$.*

Proof (of Theorem 4). We construct time sequences $t_0, t_1, t_2, \ldots$ and $t'_0, t'_1, t'_2, \ldots$ such that (a) *Wait*'s state at time t_l dominates OPT's state at time t'_l, (b) *Wait* is not in the middle of a fetch or write operation at time t_l and (c) $t_{l+1} - t_l \leq 2(t'_{l+1} - t'_l)$, for all $l \geq 0$. Condition (c) then implies the theorem.

Setting $t_0 = t'_0 = 0$, conditions (a–c) hold initially. Suppose that they hold at times t_l and t'_l and let r_i the next request to be served by *Wait*. If at time t_l all blocks in *Wait*'s cache are requested before the next missing block, then *Wait* serves r_i without initiating a write or fetch operation. We set $t_{l+1} = t_l + 1$ and $t'_{l+1} = t_{l+1} + 1$. Conditions (b) and (c) hold. Since at time t_{l+1} *Wait*'s holes occur at the latest possible positions, *Wait*'s state at time t_{l+1} dominates OPT's state at time t'_{l+1}. In the remainder of this proof we assume that at time t_l there is a block in *Wait*'s cache whose next request is after r_j, where r_j is the reference of the next missing block.

Let t_{l+1} be the time when *Wait* completes the next fetch and let $t'_{l+1} = t'_l + F$. We have $t_{l+1} - t_l \leq 2F$ and hence condition (c) holds. Also, *Wait* is not in the middle of a fetch or write operation at time t_{l+1}. We have to argue that *Wait*'s state at time t_{l+1} dominates OPT's state at time t'_{l+1}. First, *Wait*'s cursor at time t_{l+1} dominates OPT's cursor at time t'_{l+1}. This is obvious if *Wait* does not incur stall time to complete the fetch. If *Wait* does incur stall time, then OPT's cursor cannot pass *Wait*'s cursor because the index of *Wait*'s next hole at time t_l is at least as large as the index of OPT's next hole at time t'_l and OPT needs at least F time units to complete the next fetch.

If OPT does not initiate a fetch before t'_{l+1}, we are easily done. The indices of *Wait*'s $n-k$ holes increase when moving from t_l to t_{l+1} while OPT's holes do not change between t'_l and t'_{l+1}. Hence *Wait*'s holes at time t_{l+1} dominate OPT's holes at time t'_{l+1} and we have the desired domination for the states. If OPT does initiate a fetch before t'_{l+1}, then the analysis is more involved. Let a be the block evicted by OPT during the fetch and let b be the block evicted by *Wait* during the first fetch after t_l. If the next request to b is not earlier than the next request to a, then *Wait*'s holes at time t_{l+1} dominate OPT's holes at time t'_{l+1} and we have again domination for the states. Otherwise, let $d = \min\{F, j-i\}$. *Wait* initiates the next fetch after t_l immediately before serving r_{i+d}. OPT cannot initiate the first fetch after t'_l after r_{i+d}. If $d = F$, this follows from the fact that *Wait*'s cursor at time t_l dominates OPT's cursor at time t'_l and OPT initiates the fetch before $t'_l + F$. If $d < F$, then the statement holds because the index of *Wait*'s next hole at time t_l is at least as large as the index of OPT's next hole at time t'_l and r_{i+d} is the next missing block for *Wait*.

Recall that we study the case that the next request to block b is before the next request to a. Block a is not in the set S of blocks referenced by write requests in $r_i, \ldots, r_{i+d-1}$ because a would have to be written back to disk after its last write reference in $r_i, \ldots, i_{i+d-1}$. This write opertion would take F time units after t_l and could not be completed before t_{l+1}. As argued at the end of the last paragraph, *Wait*'s cursor at the time when *Wait* initiates the fetch dominates OPT's cursor when OPT initiates the fetch. By the definition of the algorithm, *Wait* evicts the block whose next request is furthest in the future among blocks not in S. We have $a \notin S$. Since *Wait* does not evict block a but the next request to a is after the next request to b it must be the case that a is not in *Wait*'s cache at the time when the algorithm initiated the first fetch after t_l. Hence a is not in *Wait*'s cache at time t_l and corresponds to one of *Wait*'s holes at time t_l.

Consider OPT's holes at time t'_l that are after *Wait*'s first hole $h_W(t_l, 1)$ at time t_l. If these holes are a subset of *Wait*'s holes at time t_l, then OPT's holes at time t'_{l+1} with index larger than $h_W(t_l, 1)$ are a subset of *Wait*'s holes at time t_{l+1}. The reason is that, as argued above, *Wait* also has a hole at the next request to a, the block evicted by OPT during the fetch. Note that all of *Wait*'s holes at time t'_l have index larger than $h_W(t_l, 1)$. Hence *Wait*'s holes at time t_{l+1} dominate OPT's holes at time t'_{l+1}.

If OPT's holes at time t'_l with index larger than $h_W(t_l, 1)$ are not a subset of *Wait*'s holes at time t_l, then let $h_{OPT}(t'_l, s')$ be the largest index such that $h_{OPT}(t'_l, s') > h_W(t_l, 1)$ and *Wait* does not have a hole at the request indexed $h_{OPT}(t'_l, s')$. The block referenced by that request cannot be in S because OPT would not be able to write the block back to disk before $t_l + F$. Hence the next request to the block b evicted by *Wait* cannot be before $h_{OPT}(t'_l, s')$. At time t_l let s be the number of *Wait*'s holes with index smaller than $h_{OPT}(t'_l, s')$. At time t_{l+1}, the first hole is filled. Hence *Wait*'s first $s-1$ holes at time t_{l+1} dominate OPT's first holes at time t'_{l+1}. *Wait*'s remaining holes at time t'_{l+1} have an index of at least $h_{OPT}(t'_l, s')$ and OPT's holes at time t'_{l+1} with an index larger than $h_{OPT}(t'_l, s')$ are a subset of *Wait*'s holes because, as mentioned before, the next request to block a evicted by OPT is a hole for *Wait*. Hence *Wait*'s last $n-k-(s-1)$ holes at time t_{l+1} dominate OPT's last $n-k-(s-1)$ holes at time t'_{l+1}. Thus *Wait*'s state at time t_{l+1} dominates OPT's state at time t'_{l+1}. □

4 Complexity

Theorem 5. *Given a request sequence σ, it is NP-complete to decide if there exists a prefetching/caching schedule for σ that initiates at most f fetch and at most w write operations.*

The proof is omitted due to space limitations.

5 Algorithms for parallel disk systems

In this we study integrated prefetching and caching in systems with D parallel disks. To speed up write operations, many parallel disk systems have the option of writing a memory block to an arbitrary location in the disk systems and not necessarily to the location where the block was stored previously. In particular, blocks may be written to arbitrary disks. As an example, suppose that block b has to be written to disk and that only disk d is idle at the moment. Now disk d can simply write b to the available location closest to the current head position. Of course, if a block is written to a location different from the one where the block was stored previously, the old copy of the block becomes invalid and cannot be used in future fetch operations. We assume that at any time, for any block there exists exactly one valid copy in the parallel disk system.

Given the ability to write blocks to arbitrary disks, we are able to design prefetching/caching algorithms that achieve a constant performance ratio independent of D. In particular we are able to construct efficient prefetching/caching schedules in two steps. Given a request sequence σ, we first build up a schedule S without considering from which disks blocks have to be fetched and to which disks they have to be written back. The algorithm *Loadbalance* described below then assigns fetch and write operations to the different disks. The algorithm works as long as S is synchronized and executes at most $\lceil D/2 \rceil$ parallel disk operations at any time. Moreover blocks evicted from cache must be written back to disk every time, even if they have not been modified since the last time they were brought into cache.

A schedule is *synchronized* if any two disk operations either are executed in exactly the same time interval or do not overlap at all. Formally, for any two disk operations executed from time t_1 to t'_1 and from time t_2 to t'_2, with $t_1 \leq t_2$ we require (1) $t_1 = t_2$ and $t'_1 = t'_2$ or (2) $t'_1 < t_2$.

Algorithm Loadbalance: The algorithm takes as input a synchronized prefetching/caching schedule S in which at most $\lceil D/2 \rceil$ disk operations are performed at any time. Blocks are written back to disk each time they are evicted from cache. The schedule is feasible except that disk operations have not yet been assigned to disks. The assignment is now done as follows. The initial disk configuration specifies from which disk to load a block when it is fetched for the first time in S. As for the other assignments, the algorithm considers the write operations in S in order of increasing time when they are initiated; ties are broken arbitrarily. Let w be the write operation just considered and b be the block written back. Let f be the operation in S that fetches b back the next time. Assign w and f to a disk that is not yet used by operations executed in parallel with w and f. Such a disk must exist because a total of $2(\lceil D/2 \rceil - 1)$ disk operations are performed in parallel with w and f.

We next present algorithms for computing schedules S that have the properties required by *Loadbalance*. We first develop a parallel implementation of the *Conservative* algorithm.

Algorithm Conservative: Consider the requests in the given sequence σ one by one. Let r_i be the next request for which the referenced block is not in cache.

The algorithm schedules up to $\lceil D/2 \rceil$ cache replacements immediately before r_i as follows. In each step let a be the next block missing in cache and b be the block in cache whose next request is further in the future. If the next request to a is before the next request is to b, then evict b in order to load a. Suppose that $d \leq \lceil D/2 \rceil$ cache replacements are determined in this way. Let $a_1, \ldots, a_n$ be the blocks loaded and $b_1, \ldots, b_n$ be the blocks evicted. Schedule a set of d synchronized write operations in which $b_1, \ldots, b_d$ are written, followed by a set of d synchronized fetch operations in which $a_1, \ldots, a_n$ are loaded immediately before r_i. These disk operations do not overlap with the service of requests. In the following we refer to such a combination of write and fetch operations as an access interval.

Applying *Loadbalance* to a schedule constructed by *Conservative*, we obtain a feasible prefetching/caching schedule for a given σ, provided that we modify the schedule as follows. If an access interval fetches two blocks that are loaded for the first time in the schedule and reside on the same disk in the initial disk configuration, then schedule an additional fetch operation before the given request r_i.

Theorem 6. *For any σ, the elapsed time of the schedule constructed by Conservative and Loadbalance is at most 5 times the elapsed time of an optimal schedule plus FB. Here B is the number of distinct blocks requested in σ.*

Proof. Given an arbitrary request sequence σ, let I be the number of access intervals generated by *Conservative*. The total elapsed time of the schedule constructed by *Conservative* and *Loadbalance* is bounded by $|\sigma| + (W+F)I + FB$. The additive FB is necessary to bound the fetch time for blocks loaded for the first time in the schedule. Because of initial disk configuration, it might not be possible to execute these fetch operations in parallel with other fetches. We will show that the elapsed time of an optimal schedule is at least $\max\{|\sigma|, F\lceil I/2 \rceil\}$. Since $W \leq F$, the theorem then follows.

It suffices to show that $F\lceil I/2 \rceil$ is a lower bound on the elapsed time of an optimal schedule because the lower bound of $|\sigma|$ is obvious. Let S be an optimal schedule for $|\sigma|$. We partition the fetch operations in σ into *sets of fetches*. For this purpose we sort the fetch operations in S by increasing starting times; ties are broken arbitrarily. The first set of fetches contains the first fetch operation f and all the fetches that are initiated before f is finished. In general, suppose that $i-1$ sets of fetches have been constructed so far. The ith set of fetches contains fetch operations that are not yet contained in the $i-1$ first sets. It contains the first such fetch f as well as all fetch operations that are initiated before f terminates. Let J be the number of sets thus created. The first fetches in these J sets are non-overlapping and hence the optimum algorithm spends at least FJ time units fetching blocks.

Lemma 2. *It is possible to modify the schedule S such that it is identical to* Conservative*'s schedule and the total fetch time is at most $2FJ$.*

The proof is omitted. Since the total fetch time of *Conservative*'s schedule is IF, the desired bound then follows. □

We next give an implementation of the *Aggressive* algorithm. It uses $\lceil D/2 \rceil$ extra memory locations in cache.

Algorithm Aggressive+: Let r_i be the next request to be served and r_j be the next request where the referenced block is not in cache. Let $d = \min\{j - i, F\}$. Determine the largest number d, $d \leq \lceil D/2 \rceil$, such that there exist d blocks in cache whose next requests after r_{i+d-1} are later than the first references of the next d blocks missing in cache. If $d = 0$, then serve r_i without initiating a fetch. Otherwise, when serving r_i, initiate d synchronized fetch operations in which the next d missing blocks are loaded into $\lceil D/2 \rceil$ extra cache locations. When these fetches are complete, evict the d blocks from cache whose next requests are furthest in the future and write them back to disk in a synchronized write operation. The $\lceil D/2 \rceil$ extra cache locations are available again. Note that the write operations start with the service of r_{i+d}.

Again we apply *Loadbalance* to a schedule constructed by *Aggressive+*. The proof of the next theorem is omitted.

Theorem 7. *Given a request sequence σ, the elapsed time of the schedule constructed by Aggressive+ and Loadbalance is at most 4 time the elapsed time of an optimal schedule plus FB, where B is the number of distinct blocks requested in σ.*

References

1. S. Albers, N. Garg and S. Leonardi. Minimizing stall time in single and parallel disk systems. *Journal of the ACM*, 47:969–986, 2000. Preliminary version STOC98.
2. S. Albers and C. Witt. Minimizing stall time in single and parallel disk systems using multicommodity network flows. *Proc. 4th International Workshop on Approximation Algorithms for Combinatorial Optimization Problems (APPROX)*, Springer LNCS 2129, 12–23, 2001.
3. L.A. Belady. A study of replacement algorithms for virtual storage computers. *IBM Systems Journal*, 5:78–101, 1966.
4. P. Cao, E.W. Felten, A.R. Karlin and K. Li. A study of integrated prefetching and caching strategies. *Proc. ACM International Conference on Measurement and Modeling of Computer Systems (SIGMETRICS)*, 188–196, 1995.
5. 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 (TOCS)*, 14:311–343, 1996.
6. A. Gaysinsky, A. Itai, and H. Shachnai. Strongly competitive algorithms for caching with pipelined prefetching. *Proc. of the 9th Annual European Symposium on Algorithms (ESA01)*, Springer LNCS 2161, 49–61, 2001.
7. T. Kimbrel and A.R. Karlin. Near-optimal parallel prefetching and caching. *SIAM Journal on Computing*, 29:1051 – 1082, 2000. Preliminary version in FOCS96.
8. T. Kimbrel, P. Cao, E.W. Felten, A.R. Karlin and K. Li. Integrated parallel prefetching and caching. *Proc. ACM International Conference on Measurement and Modeling of Computer Systems (SIGMETRICS)*, 1996.
9. T. Kimbrel, A. Tomkins, R.H. Patterson, B. Bershad, P. Cao, E.W. Felten, G.A. Gibson, A.R. Karlin and K. Li. A trace-driven comparison of algorithms for parallel prefetching and caching. *Proc. of the ACM SIGOPS/USENIX Association Symposium on Operating System Design and Implementation*, 1996.