Paging for Multicore Processors

University of Waterloo Technical Report CS-2011-12 * **

Alejandro López-Ortiz and Alejandro Salinger

David R. Cheriton School of Computer Science, University of Waterloo,
200 University Ave. West, Waterloo, ON, Canada, N2L3G1
{alopez-o,ajsalinger}@uwaterloo.ca

Abstract. Paging for multicore processors extends the classical paging problem to a setting in which several processes simultaneously share the cache. Recently, Hassidim [16] studied cache eviction policies for multicores under the traditional competitive analysis metric, showing that LRU is not competitive against an offline policy that has the power of arbitrarily delaying request sequences to its advantage. In this paper we study caching under the more conservative model in which requests must be served as they arrive. We study the problem of minimizing the number of faults, deriving bounds on the competitive ratios of natural strategies to manage the cache. We then study the offline paging problem and show that deciding if a request can be served such that at a given time each sequence has faulted at most a given number of times is NP-complete and that its maximization version is APX-hard (for an unbounded number of sequences). Lastly, we describe offline algorithms for the decision problem and for minimizing the total number of faults that run in polynomial time in the length of the sequences.

1 Introduction

In the last few years, multicore processors have become the dominant processor architecture. While cache eviction policies have been widely studied both in theory and practice for sequential processors, in the case in which various simultaneous processes share a common cache, the performance of even the most common eviction policies is not yet fully understood. In particular, there is almost no theoretical backing for the use of current eviction policies in multicore processors. Recently, a work by Hassidim [16] presented a theoretical study of paging strategies for shared caches in multicores or Chip Multiprocessors (CMPs). In a CMP system with p cores, a shared cache might receive up to p page requests simultaneously. Hassidim proposes a somewhat unconventional model in which the paging strategy can schedule the execution of threads. While in principle there is no reason why this cannot be so, historically the scheduler within the operating system concentrates in fairness and throughput considerations to determine which task should be executed while the paging algorithm focuses on which of the pages currently in cache should be evicted upon a fault.

In this work we assume a more conservative model, in which cache algorithms are not allowed to make any scheduling decisions but must serve all active requests. In this model, a paging strategy serves a set of p request sequences with a shared cache of size K. Requests can be served in parallel, thus various pages can be read from cache or fetched from memory simultaneously, and a fault delays the remaining requests of the sequence involved by τ units of time. We define as FINAL-TOTAL-FAULTS (FTF) the problem of minimizing the total number of faults, and as PARTIAL-INDIVIDUAL-FAULTS (PIF) the problem of deciding, given a request sequence $\mathcal{R}$ and bound vector $\boldsymbol{b} \in \mathbb{N}^p$, whether $\mathcal{R}$ can be served such that at time t the number of faults on each sequence R_i is at most b_i.

* Updated version with citation [12] corrected on June 16, 2011.

** A two page research announcement of this result appeared in the brief announcement session in SPAA'11 [18].

Without loss of generality we define a cache strategy as a combination of a possible partition policy, and an eviction policy, and compare the performance of natural strategies for FTF within this framework. We show that when restricted to static partition strategies, the choice of the partition has more impact than the choice of an eviction policy. We show, however, that strategies that partition the cache cannot be competitive with respect to shared strategies if they do not update the partition often, even for disjoint request sequences. On the other hand, partitions that change often are essentially equivalent to shared strategies.

We then study the offline cache problem and show properties of optimal offline algorithms. We show that PIF is NP-complete and that a maximization version does not admit a Polynomial Time Approximation Scheme (PTAS). We then present optimal offline algorithms for both FTF and PIF that run in polynomial time in the length of the sequences (and exponential in the number of sequences).

The rest of the paper is organized as follows. We review related work in Section 2. In Section 3 we describe the CMP cache model and formally define the problems we address in this paper. In Section 4 we describe natural strategies to minimize the number of faults and derive bounds on their performance. We study the offline problem in Section 5. We provide concluding remarks and future directions of research in Section 6.

2 Related Work

The performance of the cache in the presence of multiple threads has been extensively studied, and research on the subject has increased markedly since the appearance of mainstream multicore architectures. A variety of works have studied cache strategies in practice, developing heuristics to dynamically partition the cache (e.g. [25, 19, 8]) or to manage cache at the operating system level (e.g [11, 28, 21]).

From a theoretical perspective, researchers have studied schedulers and algorithms with good theoretical cache performance (See, e.g., [5, 4, 9] and references therein). More directly related to cache replacement policies, various models have been proposed to analyze the performance of paging algorithms in the presence of multiple request sequences, either modeling multiple applications or multiple threads. In what follows, we briefly review some of these models, which differ mainly in the assumptions they make with respect to the abilities of paging algorithms to schedule requests.

Fiat and Karlin [13] study paging algorithms in the access graph model, in which request sequences are restricted to paths in a graph [7]. They study the multi-pointer case, in which several paths through an access graph might be performed simultaneously, modeling both different applications (the graph could be disconnected) or multithreaded computations (having several paths in one same connected component). In this model, the order of requests is independent of the paging algorithm, which makes the paging problem substantially different from the one we study in this paper.

Barve *et al.* [2] study multiapplication caching in the competitive analysis framework [24]. In this model a set of page requests corresponding to different applications is served with a shared cache, and bounds are given with respect to a worst possible interleaving of the request sequences. As in Fiat and Karlin's model, however, the order of requests is the same for all algorithms.

Feuerstein and Strajilevich introduced Multi-Threaded Paging (MTP) [12]. In this problem, given a set of page requests, an algorithm must decide at each step which request to serve next, and how to serve it. They show that when no fairness restrictions are imposed there exist algorithms

with bounded competitive ratio, but that no competitive algorithms exist when general fairness restrictions are imposed. Results in this model were further extended in [26] and [23]. In this model, a paging algorithm has the capability of scheduling requests, and thus the order in which requests are served is algorithm dependent.

A recent paper by Hassidim [16] introduced a model for cache replacement policies specific to multicore caches and studied the performance of algorithms in the competitive analysis framework with makespan as the performance measure. Hassidim's model includes the fetching time of pages from memory. Thus if the algorithm incurs on a fault on one sequence, it can continue serving other sequences while the faulting sequence's page is fetched from memory. Hassidim shows that the competitive ratio of LRU with a cache of size K is $\Omega(\tau/\alpha)$, where τ is the ratio between miss and hit times, and the offline optimal has a cache of size K/α. He also shows that computing the optimal offline schedule is NP-complete, and presents a PTAS for constants p (number of cores) and τ.

As in the MTP model, Hassidim's model assumes that the paging strategy can choose to serve requests of some sequences and delay others. In particular, the offline strategy is able to modify the schedule of requests, and hence is more powerful than a regular cache eviction algorithm. We discard this possibility, assuming that the order in which requests of different processors arrive to the cache is given by a scheduler over which the caching strategy has no influence. Given a request, the algorithm must serve the request either from cache or slow memory, and it cannot be delayed. Hence, our model is different from previous models in that we assume no explicit scheduling capabilities of the paging strategy, while at the same time faults introduce delays in sequences, thus changing the order of requests in the input.

3 The Cache Model

The model we use in this paper is broadly based on Hassidim's model [16]. We have a multicore processor with p cores $\{1, \ldots, p\}$, and a shared cache of size K pages. The input is a multiset of request sequences $\mathcal{R} = \{R_1, \ldots, R_p\}$, where $R_j = \sigma^j_{s_1}, \ldots, \sigma^j_{s_{n_j}}$ is the request sequence of core j of length n_j. $\sigma^j_{s_i}$ is the identifier of the i-th page of the request, with $1 \leq s_i \leq N$, where N is the size of the universe of pages. The total number of page requests is $n = \sum_{j=1}^{p} n_j$. We assume $K \gg p$ and $n_j \gg K$, for all $1 \leq j \leq p$. In particular, we assume that $K \geq p^2$, which can be regarded as a CMP variant of the tall cache assumption. We say that a request $\mathcal{R}$ is *disjoint* if $\bigcap_{j=1}^{p} R_j = \emptyset$ and *non-disjoint* otherwise. In practice, a single instruction of a core can involve more than one page. We treat each request as a request for one page, which models the case of separate data and instruction caches.

A parallel request is served in one parallel step. This assumes that requested pages from different cores can be read in parallel from cache. We assume as well that fetching can be done in parallel, i.e. pages from memory corresponding to requests of different cores can be brought simultaneously from memory to cache.

In our model, when a page request arrives it must be serviced. The only choice the paging algorithm has is in which page to evict shall the request be a fault. A cache miss delays the remaining requests of the corresponding processor by an additive term τ[1]. In other words, if request $\sigma^j_{i^*}$ is a miss, then for all $i > i^*$, the earliest time at which σ^j_i can be served is increased by τ.

[1] Note that in [16] τ is defined as the fetching time, which in this paper is $\tau + 1$.

To be consistent with [16], we adhere to the convention than when a page needs to be evicted to make space, first the page is evicted and the cache cell is unused until the fetching of the new page is finished. We also assume that cache coherency is provided at no cost to the algorithms. Finally, we adopt the convention that simultaneous requests are served logically in a fixed order (e.g. by increasing number of processor), which for online algorithms means that requests are served without knowledge of simultaneous requests that are served later according to the logical order.

Under this model, various natural choices of objective functions may be considered. We define and address the following problems in this paper:

Definition 1 (FINAL-TOTAL-FAULTS (FTF)).
Given a set of requests $\mathcal{R} = \{R_1, \ldots, R_p\}$, a cache size K, an integer $\tau \geq 0$, minimize the total number of faults when serving $\mathcal{R}$ with a size of cache K.

Definition 2 (PARTIAL-INDIVIDUAL-FAULTS (PIF)).
Given a set of requests $\mathcal{R} = \{R_1, \ldots, R_p\}$, a cache size K, a time t, an integer $\tau \geq 0$, and $\boldsymbol{b} \in \mathbb{N}^p$, can $\mathcal{R}$ be served with a cache of size K such that at time t the number of faults on each sequence R_i is at most b_i?

Definition 3 (MAX-PARTIAL-INDIVIDUAL-FAULTS (MAX-PIF)).
Given an instance of PARTIAL-INDIVIDUAL-FAULTS, maximize the number of sequences whose number of faults at a given time is within the given bound.

Intuitively PIF is harder than FTF, since the former poses more restrictions on feasible solutions. Posing a bound on individual faults might be required to ensure fairness, and furthermore, doing so at arbitrary times can be used to ensure fairness throughout the execution of an algorithm.

4 Bounds of Online Strategies for Minimizing Faults

Natural strategies to manage the cache in the multicore cache model can be classified in two families: shared and partition. In the first one, the entire cache is shared by all processors, and a cache cell can hold a page corresponding to any processor. In the second one, the cache is partitioned in p parts, with each part destined exclusively to store pages of requests from one processor. A partition strategy is static if the size of all parts remain constant during an execution and dynamic otherwise.

Both shared and partition strategies are accompanied by an eviction policy A. We use S_A to denote the algorithm that uses a shared cache with eviction policy A. For partition strategies the partition needs to be specified as well. Thus, sP_A^B and dP_A^D are the static partition and dynamic partition algorithms that use eviction policy A and partitions B and D, respectively.

A partition is a function $k : \{P, \mathbb{N}\} \to \{\Pi(K, p)\}$, where $P = \{1, \ldots, p\}$ is the set of processors, and $\Pi(K, p) = \{\{k_1, k_2, \ldots, k_p\} \mid k_i \in \{0, \ldots, K\} \wedge \sum_{i=1}^{p} k_i = K\}$ is the set of possible partitions of K with p nonnegative integers. Thus, $k(i, t)$ is the size of the cache for processor i at time t. We make the restriction that all partitions must assign at least one unit of cache to all processors whose requests are active. For dynamic partitions, a reduction in the size of a part of a processor might involve evicting pages of that processor. If at any time t the cache contains $c(j, t)$ pages in the part of processor j, then if at time $t + 1$, $k(j, t + 1) < k(j, t)$, then $\max\{c(j, t) - k(j, t + 1), 0\}$ pages are evicted from the cache according to the eviction policy.

For example, according to the notation defined above, S_{LRU} evicts the least recently used page in the entire cache and sP_{LRU}^{OPT} performs LRU on each part of the partition, which is determined offline so as to minimize the total number of faults.

In the remainder of this section we compare the performance of partition and shared strategies for FTF. We denote the number of faults of a strategy Alg on a sequence $\mathcal{R}$ as $Alg(\mathcal{R})$.

Partitioning the cache may desirable to avoid costs of managing concurrency issues that arise when different processors are allowed to access the same cells. A static partition allows the execution of regular paging algorithms in each part, oblivious to the presence of other threads. Let us first restrict global strategies to static partitions, and consider a fixed partition of the cache, independent of the input sequences. In this scenario, results analogous to the ones of sequential paging apply. For example, the following Lemma implies that in this setting the competitive ratio of LRU, FIFO, or any deterministic marking or conservative[2] algorithm is K.

Lemma 1 (Online vs. offline eviction policies with a fixed static partition). *Let A be any deterministic online cache eviction algorithm and let $B = \{k_1, k_2, ..., k_p\}$ be any online static partition. There exists a sequence $\mathcal{R}$ such that $sP_A^B(\mathcal{R})/sP_{OPT}^B(\mathcal{R}) = \Omega(\max_j\{k_j\})$. When A is any marking or conservative algorithm (e.g. LRU), there is a matching upper bound, i.e., $\forall\ \mathcal{R}$, $sP_A^B(\mathcal{R})/sP_{OPT}^B(\mathcal{R}) \leq \max_j\{k_j\}$.*

Proof (lower bound). Let $j^* = argmax_j\{k_j\}$. The sequence $\mathcal{R}$ is such that for $j \neq j^*$, $R_j = (\sigma_1^j)^{n/p}$, i.e., the same page is requested n/p times, while R_j consists of requesting, among pages $\{\sigma_1, \sigma_2, ..., \sigma_{k_{j^*}+1}\}$, the page just evicted by A, where $\sigma_{i_1} \neq \sigma_{i_2}$ for $i_1 \neq i_2$, and all sequences are disjoint. $sP_A^B(\mathcal{R}) = n/p + p - 1$, since it faults on every request of R_{j^*} and once on each other sequence. On the other hand, since sP_{OPT}^B only evicts a page of sequence R_{j^*} if it is not requested in the following k_{j^*} requests, we have $sP_{OPT}^B(\mathcal{R}) \leq (n/p)/k_{j^*} + p - 1$ and the lemma follows.

Proof (upper bound). Divide sequence R_j in phases such that a new phase starts every time there is a request for the $(k_j + 1)$-th distinct page since the beginning of the previous phase, and the first phase begins at the first page of R_j. sP_{LRU}^B faults at most k_j times in each phase of R_j, while any algorithm must fault at least once in each phase. Let ϕ_j denote the number of phases of sequence R_j, then $sP_{LRU}^B(\mathcal{R}) \leq \sum_{j=1}^p \phi_j k_j \leq \max_j\{k_j\} \sum_{j=1}^p \phi_j$. On the other hand, $sP_{OPT}^B(\mathcal{R}) \geq \sum_{j=1}^p \phi_j$, and thus $sP_{LRU}^B(\mathcal{R})/sP_{OPT}^B(\mathcal{R}) \leq \max_j\{k_j\}$. □

Although traditional sequential eviction policies are competitive for given static partitions, when the partition can take into account the input sequences, then no deterministic online algorithm is competitive, as shown in the following Lemma:

Lemma 2 (Online static partition strategies are not competitive). *Let $B = \{k_1, ..., k_p\}$ be any online static partition. Let $k^* = \min_j\{k_j | k_j \geq 2\}$. $\exists \mathcal{R}$, s.t. for all A, $sP_A^B(\mathcal{R})/sP_{LRU}^{OPT}(\mathcal{R}) \geq \min\{k^*, p-1\}\frac{n}{K^2 p} = \Omega(n)$.*

Proof. Consider first $A = LRU$. Let $j^* = argmin_j\{k_j | k_j \geq 2\}$ (i.e. $k_{j^*} = k^*$). Let P denote the set of the first k_{j^*} processors in decreasing order of part of the cache according to B. Note that if $k_{j^*} \geq (p-1)$ then P is equal to the set of all processors. Let $P' = P \backslash \{j^*\}$. Let $R_j = (\sigma_1^j \sigma_2^j ... \sigma_{k_j+1}^j)^{x_j}$ with x_j such that $x_j(k_j+1) = n/p$ for all $j \in P'$, and let $R_j = (\sigma_1^j \sigma_2^j ... \sigma_{k_j}^j)^{x_j}$ $j \notin P'$ and $j \neq j^*$, where x_j is such that $x_j k_j = n/p$. Let $R_{j^*} = (\sigma_1^{j^*})^{n/p}$. sP_{LRU}^B faults on every request of $|P'|$ processors and faults only on the first request of processor j^*. Hence, $sP_{LRU}^B(\mathcal{R}) \geq \min\{k_{j^*}, (p-1)\}n/p$.

[2] See [6, Ch. 3] for the definitions of marking and conservative algorithms.

On the other hand, an optimal partition for $\mathcal{R}$ would be one such that all different pages of each request R_j fit in the cache. Intuitively, an optimal partition takes units of cache from j^* and assigns them to other processors. Let k_j^{OPT} denote the size of the cache for processor j according to the optimal partition, then $k_j^{OPT} = k_j + 1$ if $j \in P'$ and $k_j^{OPT} = \min\{1, k_j - (p-1)\}$ for $j = j^*$. The number of faults of sP_{LRU}^{OPT} on $\mathcal{R}$ is K, since it only faults on the first request to each different page. Hence $sP_{LRU}^{B}(\mathcal{R})/sP_{LRU}^{OPT}(\mathcal{R}) \geq \min\{k_{j^*}, (p-1)\}n/Kp = \Omega(n)$. Now, by Lemma 1 $sP_{OPT}^{B}(\mathcal{R}) \geq sP_{LRU}^{B}(\mathcal{R})/K$, and the lemma follows. □

Although strategies that partition the cache only once or that even allow a small number of changes during the execution might be simple to manage compared to general strategies, the performance of these strategies is not competitive when shared strategies are allowed. While this is perhaps to be expected for non-disjoint sequences, surprisingly this holds even for disjoint sequences. Theorem 1 shows that static partitions are preferable over static partitions (even if the partition is computed offline) and dynamic partitions that do not change often.

Theorem 1. *Let A be any deterministic online cache eviction policy, and let D be any online dynamic partition strategy that changes the sizes of the parts $o(n)$ times. The following statements hold:*

1. *There exists a sequence $\mathcal{R}$ such that $sP_{OPT}^{OPT}(\mathcal{R})/S_{LRU}(\mathcal{R}) = \Omega(n)$.*
2. *For all $\mathcal{R}$ $S_{LRU}(\mathcal{R})/sP_{OPT}^{OPT}(\mathcal{R}) \leq K$.*
3. *There exists a sequence $\mathcal{R}$ s.t. $dP_A^B(\mathcal{R})/S_{LRU}(\mathcal{R}) = \omega(1)$. Furthermore, if D varies the partition a constant number of times, $dP_A^D(\mathcal{R})/S_{LRU}(\mathcal{R}) = \Omega(n)$.*

Proof. 1. Let $K_1 = 0$ and $K_j = \sum_{i=1}^{j-1} k_i + 1$, for all $2 \leq j \leq p$. Consider a sequence of requests $\mathcal{R}$, in which processor j requests the following pages, for all j simultaneously:

$$(\sigma_1^j)^{(j-1)(K/p+1)(\tau+x)}(\sigma_1^j\sigma_2^j\ldots\sigma_{K/p+1}^j)^x(\sigma_1^j)^{(K+p-j(K/p+1))(\tau+x)}$$

where $\sigma_{i_1}^j \neq \sigma_{i_2}^j$ for all $i_1 \neq i_2$, and x is a parameter. In other words, processor j requests the same page for a while, then repeatedly requests $K/p+1$ distinct pages (call this the *distinct period*), and then goes back to requesting the same page again. All processors do the same, taking turns to be the processor currently in the distinct period: when one processor is in the distinct period, all other processors request repeatedly the same page. Given the request sequence, an optimal partition assigns $K/p+1$ units of cache to $p-1$ processors, and the rest to one processor: assigning more than $K/p+1$ units of cache to any processor does not result in fewer faults, and assigning less than $K/p+1$ to more than one processor increases the number of faults. Let j^* be the processor whose partition is $k_{j^*} = K/p-(p-1)$. Consider the distinct period of this processor. Let A be any eviction policy. No matter what the eviction policy A is, even the optimal offline, sP_A^{OPT} will fault at least once every k_{j^*} requests. Hence $sP_A^{OPT}(\mathcal{R}) \geq x(K/p+1)/k_{j^*}$. On the other hand, $S_{LRU}(\mathcal{R})$ faults only on the first $K/p+1$ requests of the distinct period of each processor, for a total of $K+p$ faults. Hence $sP_A^{OPT}(\mathcal{R})/S_{LRU}(\mathcal{R}) \geq x/(pk_{j^*})$. x can be made arbitrarily large, in fact $n = \tau(K+p)(p-1)+xp(K+p)$ and thus $x = n/(p(K+p))+\tau(p-1)/p$, and thus $x/(pk_{j^*}) = \Omega(n)$.

2. Divide a sequence R_j of processor j in phases such that in a sequential traversal of pages, a new phase begins either on the first page, or at the (k_j+1)-th different page since the beginning

of the current phase, where k_j is the size of the cache assigned by OPT to processor j. Define a phase for the entire sequence $\mathcal{R}$ equivalently for the cache size K. Call this phase a shared phase. We claim that a shared phase cannot start and end without at least one sequence changing phase. In other words, the phase of at least one sequence must end before the end of a shared phase. If this was not the case, within the shared phase, the number of different pages in the sequence of each processor j would be at most k_j, and therefore the total number of different pages in the shared phase would be at most K, which is a contradiction. Let ϕ denote the number of shared phases of sequence $\mathcal{R}$ and ϕ_j denote the number of phases of sequence R_j. The above claim implies that $\phi \leq \sum_{j=1}^{p} \phi_j$. Since S_{LRU} will fault at most K times per shared phase, and any cache eviction algorithm must fault at least once per phase, it follows that $S_{LRU}(\mathcal{R}) \leq K\phi \leq K \sum_{j=1}^{p} \phi_j \leq K s P_{OPT}^{OPT}(\mathcal{R})$.
Note that the arguments holds for any $\tau \geq 0$: although during the execution of the algorithm the effect of τ changes the length of the phases of each sequence and therefore it changes the phases of the entire sequence, it still holds that a phase for the entire sequence cannot end before a change of phase of at least one sequence.

3. Let a stage of D denote a period in which the sizes of the partition are constant. If the number of stages of D is $o(n)$, then at least one stage has non-constant length $\ell = \omega(1)$ (in number of parallel page requests). We then apply the same argument as in the proof of statement 1. Let $\mathcal{R}$ in this stage consist of a sequence in the form of the sequence in that proof: each processor's sequence has three periods: (1) only one page σ_1^j is requested repeatedly, (2) the page requested is any page not in the cache of processor j (the *distinct period*), and (3) again only one page σ_1^j is requested. The length of period (2) is m pages, and each processor takes turns to be in the distinct period. Hence the total number of requests in the stage is $mp^2 = \ell p$. Let t be the time where the long stage begins. During the distinct period of processor j, R_j consists of repeatedly requesting the page not in j's cache, among the pages $\{\sigma_1^j, ..., \sigma_{k(j,t)+1}\}$. dP_A^D faults on every request of the distinct period of all processors, and hence in this stage $dP_A^D(\mathcal{R}) = pm = \ell$. On the other hand, in this stage, S_{LRU} faults only on the first request to a distinct page in the distinct period of each processor, and thus in this stage $S_{LRU}(\mathcal{R}) = K + p$ (recall we assume $k_j \geq 1$ for all $1 \leq j \leq p$ at all times). Let the rest of $\mathcal{R}$ be such that neither algorithm faults. Then $dP_A^D(\mathcal{R})/S_{LRU}(\mathcal{R}) \geq \ell/(K+p) = \omega(1)$. Note that if partitions are allowed only a constant number of stages, then $dP_A^D(\mathcal{R})/S_{LRU}(\mathcal{R}) = \Omega(n)$.

□

Theorem 1 suggests that competitive strategies must either be shared or have a partition that changes often. We show that in fact these types of strategies are equivalent for disjoint sequences. Although a dynamic partition strategy executes an eviction policy in each part separately, if the variation in the partition can be determined globally, then any shared strategy can be simulated by a dynamic partition on disjoint sequences. The following lemma shows this for shared LRU:

Lemma 3 (Dynamic partitions equal shared strategies for disjoint sequences). *There exists a dynamic partition D such that for all disjoint $\mathcal{R}$, $dP_{LRU}^D(\mathcal{R}) = S_{LRU}(\mathcal{R})$.*

Proof. Let D be the following strategy. D starts by assigning an equal share of the cache to all processors. On a request to page σ_i^j, if this page is a fault, let j^* be any processor whose cache is not full, or if all caches are full, the processor whose least recently page contained in its cache partition is the least recently used overall. D modifies k_{j^*} to be $k_{j^*} - 1$ (evicting one page in this

cache according to LRU if the cache is full), and assigns that cache cell to j, the processor of the new request. If, on the other hand, σ_i^j is a hit, no change in the partition is made, and only the priority of the pages in the cache of processor j is updated according to the eviction policy. It is not difficult to see that at all times, the caches of dP^D_{LRU} and S_{LRU} contain the same pages: if the entire cache is not full, no pages are evicted; if the cache is full, both algorithms evict the overall least recently used page. □

The above observation implies that a dynamic partition strategy can be as effective as any other strategy. In fact, Hassidim [16] showed in his model that an optimal algorithm evicts the page that is furthest in the future for some processor. This is equivalent to having a dynamic partition in which upon a fault, the part of one processor is reduced and the page that is furthest in the future in that processor's sequence is the one which should be evicted. We show in Section 5 that the same result holds in our model.

Multicore paging differs from sequential paging in that the actions of algorithms modify the order of future requests. Hence paging strategies must decide which page to evict not only with the goal of delaying further faults, but at the same time trying to properly align the demand periods of future requests. An online strategy, however, is oblivious to future request, and hence in general its strategy cannot work toward the second goal. In this sense, in multicore paging an optimal offline strategy has significantly more advantage over an online strategy than in sequential paging. While in the latter setting any online marking algorithm has a bounded competitive ratio of K [17, 27], in multicore paging the competitive ratio might be larger. Although the optimal offline strategy does not have the ability of scheduling requests, it can effectively delay the sequence of one processor and serve the rest of the sequences having enough cache space. In particular, if $\tau = \Omega(n)$, then the competitive ratio of S_{LRU} can be arbitrarily large.

Lemma 4 (Lower bound on the competitive ratio of LRU). *There exists a sequence $\mathcal{R}$ such that $S_{LRU}(\mathcal{R})/S_{OPT}(\mathcal{R}) = \Omega(p(\tau+1))$.*

Proof. Consider a disjoint sequence $\mathcal{R}$ where each R_j consists of repeatedly requesting pages $(\sigma_1^j \dots \sigma_{K/p+1}^j)$, where all pages are different, and $|R_j| = n/p$. S_{LRU} will fault on every single request. An offline algorithm S_{OFF}, after the initial K requests (all faults) can evict the pages of one sequence only, say R_p. Thus pages $\sigma_{K/p+1}^j$ for $j = 1, \dots, p-1$ will replace $p-1$ pages currently in the cache of R_p. Since $K \geq p^2$, $(p-1)(K/p+1) < K$ and hence all the pages of R_j, $j \neq p$ fit in the cache and thus S_{OFF} will incur in no more faults on these sequences. The total number of faults on sequences $R_1, \dots, R_{p-1}$ will then be $(p-1)(K/p+1)$. On requests of R_p, S_{OFF} evicts the next page to be requested in R_p (note that since there is space in S_{OFF}'s cache to store at least one page of R_p, S_{OFF} does not have to evict any page of the other sequences), and therefore S_{OFF} will fault on every request of sequence R_p while other sequences are being served. Once the other sequences are completely served, the rest of R_p will be served with all the cache. The total number of faults on R_p will be $K/p+1$ for the initial requests plus $(n/p - K/p - 1)/(\tau+1)$ for the requests that are served while the other sequences are served, plus a final $K/p+1$ faults before all pages of R_p fit in the cache. The total number of faults of S_{OFF} is then $(p-1)(K/p+1) + 2(K/p+1) + (n/p - K/p - 1)/(\tau+1) = O(n/p(\tau+1))$, and hence $S_{LRU}(\mathcal{R})/S_{OFF}(\mathcal{R}) = \Omega(p(\tau+1))$. □

The sequence in the proof of Lemma 4 shows that in general Furthest-In-The-Future is not optimal: it is not hard to verify that when $\tau > K/p$, $S_{FITF}(\mathcal{R}) > S_{OFF}(\mathcal{R})$

5 The Offline Problem

In reality requests sequences are not known in advance, and thus paging is an online problem. In general, however, in any online problem setting deriving efficient optimal offline solutions is both of theoretical interest as well as useful in evaluating online algorithms in practice in the competitive analysis framework. Furthermore, an online solution can be designed based on properties of the offline solution. For example, in traditional paging, LRU approximates the optimal Furthest-In-The-Future (FITF) algorithm [3] using the past as the best approximation of the future. An example of an inherently online problem for which the offline problem has been extensively studied is the List Update problem (See, e.g., [22, 20, 1, 15]).

5.1 Hardness of Multicore Paging

In this section we show that even if the entire sequence of requests is known in advance, the multicore paging is hard. More specifically, we show that PARTIAL-INDIVIDUAL-FAULTS is NP-complete and that there is no PTAS for its maximization version. This is in contrast to sequential paging. If there is only one processor, both FTF and PIF are solvable by FITF, which is not optimal in the multicore setting.As in the proof of Hassidim's makespan problem [16], the proof of NP-completeness of PIF uses a reduction from 3-PARTITION. Hassidim's proof relies on the fact that sequences can be scheduled in his model. This is not possible in our model, and hence our reduction is quite different.

Theorem 2. *PARTIAL-INDIVIDUAL-FAULTS is NP-complete.*

Proof sketch[3]**:** We reduce from 3-PARTITION. Recall that an instance of 3-PARTITION consists of a set of n integers $S = \{s_1, \ldots, s_n\}$, and a bound B, such that $B/4 < s_i < B/2$ for all $1 \leq i \leq n$. The problem is to determine if S can be partitioned into $n/3$ sets $A_1, \ldots, A_{n/3}$ such that for all $1 \leq j \leq n/3$, $\sum_{i \in A_j} s_i = B$. Note that the restrictions of the problem imply that each subset A_i must have exactly 3 elements [14]. Given an instance $\mathcal{J}$ of 3-PARTITION, we build an instance $\mathcal{I}$ of PARTIAL-INDIVIDUAL-FAULTS as follows. There are $p = |S|$ sequences. Each sequence R_i consists of alternating requests to 2 distinct pages α^i and β^i, i.e. $R_i = \alpha^i\beta^i\alpha^i\beta^i \ldots$, and all sequences are disjoint. The length of R_i is $|R_i| = B(\tau+1)+4\tau+5$, where $\tau \geq 1$ is any integer. The size of the cache is $K = (4/3)p$, and the maximum allowed number of faults in each sequence R_i is $b_i = B - s_i + 4$, at time $t = B(\tau + 1) + 4\tau + 5$. Since 3-PARTITION is strongly NP-complete (it remains NP-complete if the input is encoded in unary), a reduction from a unary-encoded instance takes polynomial time.

A solution to an instance J of 3-PARTITION gives a solution to $\mathcal{I}$ as follows. Let $A_1, \ldots, A_{n/3}$ be a solution to J. The idea is that each group of 3 sequences corresponding to a group A_j will share a group of 4 cells of the cache. Each sequence in a group will have a dedicated cell at all times, while the extra 4-th cell is assigned to each sequence at different times. It is not difficult to show that if each sequence R_i uses the extra cell continuously so that it incurs in exactly $h_i = s_i(\tau + 1) + 1$, then at time $t = B(\tau + 1) + 4\tau + 5$ the number of faults of this sequence is exactly $(t - h_i)/(\tau + 1) = B - s_i + 4$.

We argue now that a solution to the instance of PIF gives a solution to $\mathcal{J}$. It is clear that by time t a sequence R_i must have at least $h_i = s_i(\tau + 1) + 1$ hits in order to satisfy the bound on

[3] See Appendix A for a complete proof.

the number of faults. Given the alternating pages in a sequence, a sequence can have a hit only if it has two cells for consecutive timesteps. Each sequence uses at least one cell until time t, and hence there are only $p/3$ extra cells. Therefore there can be at most $p/3$ hits in any timestep. In addition, it is not hard to see that any change in the sequence to which a cell is assigned implies at least τ timesteps without hits for the two sequences involved. Considering the total number of hits possible until time t and the minimum total hits required, we can show that the maximum number of changes in the partition is $2p/3$ and hence each sequence must have 2 cells for a continuous period of time. Furthermore, it can be shown that exactly 3 sequences must share one extra cell, otherwise at least one sequence will exceed the allowed faults before time t. Finally, it is not hard to show that the 3 sequences $R_{i_1}, R_{i_2}, R_{i_3}$ that share one cell must be the ones whose corresponding s_i's satisfy $s_{i_1} + s_{i_2} + s_{i_3} = B$. Hence each group of 3 sequences that share the extra cell define a solution for the instance $\mathcal{J}$ of 3-PARTITION. □

Observe that PIF remains NP-complete even when $\tau = 0$, i.e. when sequences are not delayed due to faults. This means that this problem is hard also in the multiapplication caching model of Barve *et al.* [2]. Note that this is not the case for FINAL-TOTAL-FAULTS, for which FITF is optimal when $\tau = 0$. This confirms that the goal of achieving a fair distribution of faults is more difficult to attain than merely minimizing the number of overall faults.

We also show that MAX-PIF is APX-hard, i.e. there is no polynomial time algorithm that can approximate MAX-PIF within a factor of $(1 - \epsilon)$ for any ϵ, unless P=NP. In order to show this, we describe a gap-preserving reduction from MAX-4-PARTITION (shown to be APX-hard in [10]). Therefore unless P=NP, given an instance of PIF there is no efficient way of serving the request sequences ensuring that a large part of them will fault within the allowed bounds.

Theorem 3. *MAX-PARTIAL-INDIVIDUAL-FAULTS is APX-hard.*

Proof. We describe a gap preserving reduction from MAX-4-PARTITION to MAX-PIF. The 4-PARTITION [14] problem is an analog of 3-PARTITION in which the goal is to partition a set $S = \{s_1, \ldots, s_n\}$ in subsets $A_1, \ldots, A_{n/4}$ such that for all $1 \leq j \leq n/4$, $\sum_{i \in A_j} s_i = B$, where $B = (4/n) \sum_{i=1}^{n} s_i$. Each element s_i satisfies $B/5 < s_i < B/3$ and thus each subset must have 4 elements. 4-PARTITION is also NP-complete [14], and a reduction to PIF can be built by modifying the proof of Theorem 2 in a straightforward way: the cache size is now $K = (5/4)p$, the length of each sequence is $B(\tau + 1) + 5\tau + 6$ and the goal is to serve the sequences such that at time $t = B(\tau + 1) + 5\tau + 6$ sequence i has incurred in at most $b_i = B - s_i + 5$. It is not hard to see that the same arguments in the proof of Theorem 2 apply to argue that an instance of 4-PARTITION admits a solution if an only if the instance of PIF admits a solution.

The MAX-4-PARTITION problem (as defined in [10]) is: given a set S and B as in the 4-PARTITION problem, find a maximum number of disjoint subsets whose elements add up to B. This problem is APX-hard, i.e. it does not admit a PTAS (assuming P $\neq$ NP)[10]. Given an instance $\mathcal{J}$ ($\mathcal{I}$) of MAX-4-PARTITION (MAX-PIF), let $OPT_{4PART}(\mathcal{J})$ ($OPT_{PIF}(\mathcal{I})$) denote the value of the optimal solution to $\mathcal{J}$ ($\mathcal{I}$). Let $n = |S|$ in $\mathcal{J}$. In order to show that MAX-PIF is APX-hard, we build a reduction to an instance $\mathcal{I}$ of MAX-PIF and show:

1. $OPT_{4PART}(\mathcal{J}) \geq n/4 \Rightarrow OPT_{PIF}(\mathcal{I}) \geq n$
2. $OPT_{4PART}(\mathcal{J}) < (1 - \epsilon)n/4 \Rightarrow OPT_{PIF}(\mathcal{I}) < (1 - \epsilon/4)n$

The reduction from an instance $\mathcal{J}$ of MAX-4-PARTITION to an instance $\mathcal{I}$ of MAX-PIF is exactly the same as the reduction from 4-PARTITION described above. Since a solution to $\mathcal{J}$ gives a

solution to $\mathcal{I}$, if $OPT_{4PART}(\mathcal{J}) \geq n/4$ (and thus equal to $n/4$) then all sequences of $\mathcal{I}$ can be served with a number of faults within the given bounds, proving statement 1.

For statement 2., note that in the reduction from 4-PARTITION to PIF (adapted from the proof of Theorem 2), the only way of serving all sequences within the fault bounds is by partitioning them in groups of 4 that shared 5 cells of cache. Furthermore, the 4 sequences in a group can be served within the faults bounds if and only if the corresponding elements in S add up to B. Otherwise, at least one of the sequences will have to incur in more faults than the allowed bound. Therefore, $OPT_{PIF}(\mathcal{I}) \leq 4OPT_{4PART}(\mathcal{J}) + 3(n/4 - OPT_{4PART}(\mathcal{J})) = OPT_{4PART}(\mathcal{J}) + 3n/4$. Since $OPT_{4PART}(\mathcal{J}) < (1-\epsilon)n/4$, we have $OPT_{PIF}(\mathcal{I}) < (1-\epsilon)n/4 + 3n/4 = (1-\epsilon/4)n$. Thus the reduction is gap-preserving, proving the theorem. □

5.2 Properties of Offline Algorithms for FINAL-TOTAL-FAULTS

The changes in the relative alignment of sequences can significantly affect the performance of an algorithm (See the proof of Lemma 4 for an example). Offline algorithms can benefit from properly aligning the demand periods of future requests. Even without explicit scheduling, a strategy can try to schedule sequences to its convenience by means of faults and their corresponding delays. For this purpose, an algorithm could evict a page voluntarily (i.e. not forced by a fault on another page) before it is requested in order to force a fault. We show, however, that forcing faults for the purpose of changing the alignments in this way is not beneficial for minimizing the number of faults. We say that an algorithm is honest if it does not evict a page unless there is a fault, and show that there exist an optimal algorithm that is honest.

Theorem 4. *Let Alg be an offline optimal algorithm that is capable of forcing faults. There exists an offline algorithm Alg' that is honest such that $\forall$ disjoint $\mathcal{R}$, $Alg'(\mathcal{R}) = Alg(\mathcal{R})$.*

Proof sketch[3]**:** We follow an inductive argument similar to the proof of optimality of Furthest-In-The-Future in the sequential setting [6]. Let Alg be any offline algorithm. The proof is based on the following claim: for each timestep i we can build an algorithm Alg_i that behaves exactly like Alg until $t = i-1$ and that at time $t = i$, if Alg forces a fault, then Alg_i does not. Furthermore, $Alg_i(\mathcal{R}) \leq Alg(\mathcal{R})$. If this claim is true, then we can build an optimal algorithm that does not force faults: given an optimal algorithm OPT, apply the claim with $i = 1$ to obtain OPT_1, then apply the claim with $i = 2$ to obtain OPT_2 and so on. $OPT_{t'}$ is an optimal algorithm, where t' is the maximum execution time.

In order to prove that the claim is true, we consider the execution of Alg and Alg_i after Alg has forced a fault at $t = i$. Both executions can differ in the contents of the cache, the relative alignment of the sequences, and the number of current faults. Based on these, we define 5 possible states in which the executions can be, for which all satisfy $Alg_i(\mathcal{R}, t) \leq Alg(\mathcal{R}, t)$ at the current timestep, where $Alg(\mathcal{R}, t)$ is the number of faults of Alg at time t. We show that Alg_i manages to always keep the executions in one of these 5 states, possibly coming back to total synchronization with Alg, or reaching the end of the sequence with $Alg_i(\mathcal{R}) \leq Alg(\mathcal{R})$. □

Hassidim shows that there is an optimal solution for minimizing the makespan that on each fault it evicts the page that is furthest in the future for some core. In other words, if the sequence whose page should be evicted is known, the page to be evicted is the furthest in the future in that sequence. We show that the same result holds in our model for minimizing the number of faults.

Theorem 5. *There exists an optimal offline algorithm for FTF on disjoint sequences that upon each fault evicts a page $\sigma \in R_j$ whose next request time is maximal in R_j, for some j.*

Proof. As in the proof of Theorem 4, we claim that for any offline algorithm Alg there exists an algorithm Alg_i that behaves exactly like Alg until time $t = i - 1$. If at time i Alg evicts a page from sequence R_j, Alg_i evicts the page from the same sequence that is furthest in the future, and $Alg_i(\mathcal{R}) \leq Alg(R)$. Applying this claim on an optimal algorithm successively at each timestep gives an optimal algorithm.

We now prove the claim. Suppose that at time $t = i$ Alg evicts a page $\sigma_1 \in R_j$. Alg_i behaves exactly like Alg until $t = i - 1$ but at $t = i$ it evicts a page $\sigma_2 \in R_j$, which is the page furthest in the future in this sequence. Assume $\sigma_1 \neq \sigma_2$ otherwise the claim is trivially true. Let $C_{Alg}(t)$ and $Alg(\mathcal{R}, t)$ denote the contents of Alg's cache and the number of faults before serving $R(t)$. We have $C_{Alg}(t) = C_{Alg_i}(t)$ and $C_{Alg}(t + \tau) = (C_{Alg_i}(t + \tau) \cup \sigma_2) \setminus \{\sigma_1\}$, and the sequences have the same alignment in both algorithms.

Before the request for σ_1, if upon a fault Alg evicts σ_2, Alg_i evicts σ_1, thus the caches are equal and from then on Alg_i behaves exactly like Alg, thus $Alg_i(\mathcal{R}) \leq Alg(\mathcal{R})$. If Alg instead evicts a page $\alpha \neq \sigma_2$, Alg_i evicts the same page.

If σ_2 was not evicted, when σ_1 is requested Alg faults and evicts a page α from some sequence. σ_1 is a hit for Alg_i and thus $Alg(\mathcal{R}, t) = Alg_i(\mathcal{R}, t) + 1$, $C_{Alg}(t) = (C_{Alg_i}(t) \cup \sigma_2) \setminus \{\alpha\}$, and Alg_i is ahead in R_j by τ timesteps. Assume first that σ_2 is not evicted by Alg before its request. Suppose Alg_i gets to σ_2 with this configuration (for some α). σ_2 is a fault for Alg_i and a hit for Alg. Alg_i evicts α and now $Alg(\mathcal{R}, t) = Alg_i(\mathcal{R}, t)$, $C_{Alg}(t) = C_{Alg_i}$ and the sequences are aligned equally. From then on both algorithms are equivalent and the claim is true. After the request for σ_1, Alg_i evicts whatever Alg evicts (assume σ_2 is not evicted by Alg during this period). If α is requested (a fault for Alg but a hit for Alg_i), Alg_i forces a fault and hence $Alg(\mathcal{R}, t) = Alg_i(\mathcal{R}, t) + 1$ and $C_{Alg}(t) = (C_{Alg_i}(t) \cup \sigma_2) \setminus \{\alpha'\}$ still holds, with α' being the page evicted by Alg. By Theorem 4, we can build another algorithm Alg_i' with the same number of faults that does not force faults, and hence the claim is true in this case. Now, if Alg evicts σ_2 before getting to its request Alg_i evicts α, and both caches are the same. When σ_2 is requested both algorithms will fault and $Alg(\mathcal{R}, t) = Alg_i(\mathcal{R}, t) + 1$ holds. However, sequences R_j in both algorithms do not have the same alignment with respect to the rest of the sequences, with Alg_i's sequence being ahead by τ timesteps. This setting is the same as the one in the proof of Theorem 4 after Alg has forced a fault. Applying that proof we can show that Alg_i can keep the execution within the 5 states defined in the proof, and hence $Alg_i(\mathcal{R}) \leq Alg(\mathcal{R})$ at the end of the execution. Again, if at any point Alg_i forces a fault, then by the same Theorem 4 we can obtain an honest algorithm that does not exceed Alg_i's faults. Since in all cases the claim is true, this proves the theorem. □

5.3 Optimal Algorithms for FINAL-TOTAL-FAULTS and PARTIAL-INDIVIDUAL-FAULTS

Theorem 5 implies an $O(p^n)$ time optimal optimal algorithm for FTF that upon each fault chooses the sequence to evict from optimally by trying all possibilities. Using dynamic programming, however, we can obtain a faster algorithm that is exponential in the number of sequences, but polynomial in the length of the sequences (recall we assume $n \gg p$). This algorithm can be extended to solve PIF as well. We describe next these algorithms, showing that if the number of sequences is constant, then both FTF and PIF admit polynomial time algorithms.

Algorithm 1 Minimum Final Total Faults(R,K)

for all configurations C **do**
  $F[C, 1, \ldots, 1] = 0$
  for each $(x_1, \ldots, x_p) \in \{2, \ldots, n_i(\tau+1)+1\}^p$ **do**
    $F[C, x_1, \ldots, x_p] = \infty$
for each $(x_1, \ldots, x_p) \in \{1, \ldots, n_i(\tau+1)\}^p$ **do**
  for all configurations C **do**
    if $F[C, x_1, \ldots, x_p] \neq \infty$ **then**
      for $i = 1$ to p **do**
        if $x_i = (j-1)(\tau+1)+1$ for some j, and $R_i(x_i) \in C$ **then** $\{R_i(x_i)$ is a hit$\}$
          $x'_i = x_i + \tau + 1$
        else $\{R_i(x_i)$ is being fetched or it is a fault$\}$
          $x'_i = x_i + 1$
      for all configurations C' s.t. $\mathcal{R}(\boldsymbol{x}) \in C'$ **do**
        $\{|\mathcal{R}(\boldsymbol{x}) \setminus C|$ is the number of faults in $\mathcal{R}(\boldsymbol{x})$ among pages not being fetched$\}$
        if $F[C, x_1, \ldots, x_p] + |\mathcal{R}(\boldsymbol{x}) \setminus C| < F[C', x'_1, \ldots, x'_p]$ **then**
          $F[C', x'_1, \ldots, x'_p] = F[C, x_1, \ldots, x_p] + |\mathcal{R}(\boldsymbol{x}) \setminus C|$
return $\min_C\{F[C, n_1(\tau+1)+1, \ldots, n_p(\tau+1)+1]\}$

Minimizing the number of faults Let $\boldsymbol{x} = (x_1, \ldots, x_p)$, where each x_i, $1 \leq x_i \leq n_i(\tau+1)+1$, is an index of a place in sequence i when serving it, either at a page, or at a time when a page is being fetched. If x_i is of the form $x_i = (j-1)(\tau+1)+1$, then x_i is the index of the j-th page in R_i. In this case, we say x_i points to a page (denoted as $R_i(x_i)$). Otherwise, x_i points to the fetching period of page $\lceil x_i/(\tau+1) \rceil$, and $R_i(x_i)$ denotes the page being fetched. Let $\mathcal{R}(\boldsymbol{x})$ denote the set of pages p indexed by $\boldsymbol{x}$, including pages in the fetching period.

Given a request $\mathcal{R}$ we want to compute, for each possible cache configuration C and possible vector of positions $\boldsymbol{x}$, the minimum number of faults required to serve $\mathcal{R}$ up to $\boldsymbol{x}$, and arriving at a cache configuration C. If x_i is in a page, this cost does not include serving $R_i(x_i)$. If x_i points to a fetching period, this cost includes the fault on $R_i(x_i)$. We compute and store these values in a $(p+1)$-dimensional table storing the minimum cost for each configuration and position. A cell $F[C, x_1, \ldots, x_p]$ can contribute to a cell $F[C', x'_1, \ldots, x'_p]$, where C' is any configuration that contains $\mathcal{R}(\boldsymbol{x})$ and $x'_i > x_i$ is the next index on sequence i. If x_i points to a page and $R_i(x_i)$ is a hit, then $x'_i = x_i + \tau + 1$, i.e. the index jumps to the next page. If $R_i(x_i)$ is being fetched or is a miss, then $x'_i = x_i + 1$. Note that requiring that C' contains the pages being fetched ensures that no pages that are being fetched can be evicted when changing configurations. We fill the table in a bottom up fashion, updating from a cell c only the cells that c can contribute to with a lower number of faults. The total minimum number of faults is then the minimum among all cache configurations C of $F[C, n_1(\tau+1)+1, \ldots, n_p(\tau+1)+1]$. Algorithm 1 shows this procedure in pseudocode.

The running time of Algorithm 1 is exponential the number of sequences p and the size of the cache K, but it is polynomial in the length of the sequences, which in practice is much larger than both p and K. Let w be the total number of different pages requested in an instance. The number of possible cache configurations is $\sum_{i=0}^{K} \binom{w}{i} \leq (w+1)^K$. Hence, the total number of cells in table F is $O((w+1)^K)(n(\tau+1)+1)^p)$. $|\mathcal{R}(\boldsymbol{x})| = p$ and thus at most $\binom{K}{K-p} = O(K^p)$ cache configurations can contain $\mathcal{R}(\boldsymbol{x})$. Therefore, the time to update the costs of all cells that one cell can contribute to is at most $O(K^p)$. Since $w \leq n$, when K and p are constants, the total running time is $O(n^{K+p}(\tau+1)^p)$.

Algorithm 2 Partial Individual Faults($\mathcal{R}, K, time, \tau, \boldsymbol{b}$)

```
for all configurations C do
    F[C, 1, ..., 1] = {({0}^p, 0)}
    for each (x_1, ..., x_p) ∈ {2, ..., n_i(τ+1)+1}^p do
        F[C, x_1, ..., x_p] = ∅
for each (x_1, ..., x_p) ∈ {1, ..., n_i(τ+1)}^p do
    for all configurations C do
        if F[C, x_1, ..., x_p] ≠ ∅ then
            for i = 1 to p do
                if x_i = (j−1)(τ+1)+1 for some j, and R_i(x_i) ∈ C then {R_i(x_i) is a hit}
                    x'_i = x_i + τ + 1
                else {R_i(x_i) is being fetched or it is a fault}
                    x'_i = x_i + 1
            P = ∅ {P is the list of updated fault vectors}
            for each (f, t) in F[C, x_1, ..., x_p] do
                validVector = true
                for i = 1 to p do
                    f'_i = f_i {f_i is the i-th element of f}
                    if R_i(x_i) ∉ C then {R_i(x_i) is a fault}
                        f'_i = f'_i + 1
                    if f'_i > b_i then {this path exceeded the maximum faults for sequence i}
                        validVector = false
                if validVector and t + 1 ≤ time then
                    P = P ∪ (f', t + 1)
                    if t + 1 = time or x'_i = n_i(τ+1)+1 for all i = 1..p then {we reached the checkpoint time or the end
                    of all sequences}
                        return TRUE
            for all configurations C' s.t. R(x) ∈ C' do
                F[C', x'_1, ..., x'_p] = F[C', x'_1, ..., x'_p] ∪ P
{we reached the end of the table before finding a feasible solution}
return FALSE
```

Theorem 6. *Given a set $\mathcal{R}$ of p sequences of total length n, a cache of size K, and $\tau \geq 0$, with $p = O(1)$ and $K = O(1)$, the minimum number of faults to serve $\mathcal{R}$ can be determined in $O(n^{K+p}(\tau+1)^p)$ time.*

Deciding PARTIAL-INDIVIDUAL-FAULTS The algorithm for FTF can be extended to solve PIF as follows. For each cache configuration C and positions $(x_1, \ldots, x_p)$ we store a set of pairs $(\boldsymbol{f}, t)$, where $\boldsymbol{f} = (f_1, f_2, \ldots, f_p)$ specifies the faults on each sequence when reaching configuration $(C, x_1, \ldots, x_p)$ at time t. Thus, each pair $(\boldsymbol{f}, t)$ associated with $F[C, x_1, \ldots, x_p]$ represents the number of faults in each sequence for a possible way of serving sequence $\mathcal{R}$ up to time t. Algorithm 2 shows the algorithm in pseudocode. The number of entries of the table F in Algorithm 2 is $O(n^{K+p}(\tau+1)^p)$ as in the algorithm for FTF. However, now each entry stores a list of pairs of fault vectors and time. Since at any time the number of faults in a sequence is at most n, the total number of different fault vectors is $O((n+1)^p)$. The time component of each pair can have at most $n(\tau+1)$ values, and hence each set can have at most $O(n^{p+1}(\tau+1))$ pairs. For each entry in F we have to go through the list of pairs and compute the new vectors, and hence processing an entry takes $O(pn^{p+1}(\tau+1))$ time. Once the new vectors are computed, these might have to be added to at most at most $O(K^p)$ other entries. Hence the total time to process one entry is $O(pn^{p+1}(\tau+1) + K^p)$, and therefore the total time is $O(n^{K+2p+1}(\tau+1)^{p+1})$.

Theorem 7. *Given a set $\mathcal{R}$ of p sequences of total length n, a cache of size K, $\tau \geq 0$, a checkpoint time t, and a bound vector $\boldsymbol{b} = \{b_1, \ldots, b_p\}$, with $p = O(1)$ and $K = O(1)$, it can be decided if $\mathcal{R}$ can be served such that at time t each sequence R_i has incurred in at most b_i faults in $O(n^{K+2p+1}(\tau+1)^{p+1})$ time.*

6 Conclusions

The fact that faults change the relative alignment of request sequences plays a key factor in the multicore cache problem, making it significantly more difficult than the traditional sequential problem and even counterintuitive when trying to apply the reasoning that works in the sequential case. An offline algorithm can and should take advantage of its knowledge of future cache demand distributions to achieve alignments that combine periods of high demand of some sequences with low demands of others. Although algorithms are not allowed to explicitly schedule requests, they can do so by taking this as a consideration when choosing what page to evict.

A natural direction of further research is to obtain competitive online algorithms. Given the apparent excessive advantage of an offline algorithm over an online strategy that cannot do anything about future alignments, perhaps comparing online strategies to an optimal offline algorithm that can align sequences to its advantage might not lead to interesting online strategies. Hence the definition of a good evaluation framework for online strategies is open for debate, and perhaps other measures such as fairness or relative progress of sequences should be considered over minimizing faults globally.

7 Acknowledgments

The authors would like to thank Reza Dorrigiv, Robert Fraser, Patrick Nicholson, and Francisco Claude for important insights during discussions of the ideas in this paper.

References

1. C. Ambühl. Offline list update is np-hard. In *In Proceedings of the 8th Annual European Symposium (ESA 2000), volume 1879 of LNCS*, pages 42–51. Springer, 2000.
2. R. D. Barve, E. F. Grove, and J. S. Vitter. Application-controlled paging for a shared cache. *SIAM J. Comput.*, 29:1290–1303, February 2000.
3. L. A. Belady. A study of replacement algorithms for virtual-storage computer. *IBM Systems Journal*, 5(2):78–101, 1966.
4. G. E. Blelloch and P. B. Gibbons. Effectively sharing a cache among threads. In *SPAA '04: Proceedings of the sixteenth annual ACM symposium on Parallelism in algorithms and architectures*, pages 235–244, New York, NY, USA, 2004. ACM.
5. G. E. Blelloch, P. B. Gibbons, and H. V. Simhadri. Low depth cache-oblivious algorithms. In F. M. auf der Heide and C. A. Phillips, editors, *SPAA*, pages 189–199. ACM, 2010.
6. A. Borodin and R. El-Yaniv. *Online computation and competitive analysis.* Cambridge University Press, New York, NY, USA, 1998.
7. A. Borodin, S. Irani, P. Raghavan, and B. Schieber. Competitive paging with locality of reference. *J. Comput. Syst. Sci.*, 50:244–258, April 1995.
8. J. Chang and G. S. Sohi. Cooperative cache partitioning for chip multiprocessors. In *ICS '07: Proceedings of the 21st annual international conference on Supercomputing*, pages 242–252, New York, NY, USA, 2007. ACM.
9. R. A. Chowdhury, F. Silvestri, B. Blakeley, and V. Ramachandran. Oblivious algorithms for multicores and network of processors. In *IPDPS*, pages 1–12. IEEE, 2010.

10. M. Cieliebak, S. Eidenbenz, and G. Woeginger. Double digest revisited: Complexity and approximability in the presence of noisy data. In T. Warnow and B. Zhu, editors, *Computing and Combinatorics*, volume 2697 of *Lecture Notes in Computer Science*, pages 519–527. Springer Berlin / Heidelberg, 2003. 10.1007/3-540-45071-8_52.
11. R. Fedorova, M. Seltzer, and M. D. Smith. Cache-fair thread scheduling for multicore processors. Technical report, Harvard University, 2006.
12. E. Feuerstein and A. Strejilevich de Loma. On-line multi-threaded paging. *Algorithmica*, 32(1):36–60, 2002.
13. A. Fiat and A. R. Karlin. Randomized and multipointer paging with locality of reference. In *Proceedings of the twenty-seventh annual ACM symposium on Theory of computing*, STOC '95, pages 626–634, New York, NY, USA, 1995. ACM.
14. M. R. Garey and D. S. Johnson. *Computers and Intractability: A Guide to the Theory of NP-Completeness*. W. H. Freeman, 1979.
15. T. Hagerup. Online and offline access to short lists. In L. Kucera and A. Kucera, editors, *MFCS*, volume 4708 of *Lecture Notes in Computer Science*, pages 691–702. Springer, 2007.
16. A. Hassidim. Cache replacement policies for multicore processors. In *Proceedings of The First Symposium on Innovations in Computer Science*. Tsinghua University Press, 2010.
17. A. R. Karlin, M. S. Manasse, L. Rudolph, and D. D. Sleator. Competitive snoopy caching. *Algorithmica*, 3:77–119, 1988.
18. A. López-Ortiz and A. Salinger. Paging for multicore processors. In *SPAA '11: Proceedings of the 23rd annual ACM symposium on Parallelism in algorithms and architectures (to appear)*.
19. A. M. Molnos, S. D. Cotofana, M. J. M. Heijligers, and J. T. J. van Eijndhoven. Throughput optimization via cache partitioning for embedded multiprocessors. In G. Gaydadjiev, C. J. Glossner, J. Takala, and S. Vassiliadis, editors, *ICSAMOS*, pages 185–192. IEEE, 2006.
20. J. I. Munro. On the competitiveness of linear search. In *Proceedings of the 8th Annual European Symposium on Algorithms*, ESA '00, pages 338–345, London, UK, 2000. Springer-Verlag.
21. M. K. Qureshi, A. Jaleel, Y. N. Patt, S. C. Steely, and J. Emer. Adaptive insertion policies for high performance caching. In *ISCA '07: Proceedings of the 34th annual international symposium on Computer architecture*, pages 381–391, New York, NY, USA, 2007. ACM.
22. N. Reingold and J. Westbrook. Off-line algorithms for the list update problem. *Inf. Process. Lett.*, 60(2):75–80, 1996.
23. S. S. Seiden. Randomized online multi-threaded paging. *Nord. J. Comput.*, 6(2):148–161, 1999.
24. D. D. Sleator and R. E. Tarjan. Amortized efficiency of list update and paging rules. *Commun. ACM*, 28(2):202–208, 1985.
25. H. Stone, J. Turek, and J. Wolf. Optimal partitioning of cache memory. *IEEE Transactions on Computers*, 41:1054–1068, 1992.
26. A. Strejilevich de Loma. New results on fair multi-threaded paging. *Electronic Journal of SADIO*, 1(1):21–36, 1998.
27. E. Torng. A unified analysis of paging and caching. In *Proceedings of the 36th Annual Symposium on Foundations of Computer Science*, FOCS '95, pages 194–, Washington, DC, USA, 1995. IEEE Computer Society.
28. Y. Xie and G. H. Loh. Pipp: promotion/insertion pseudo-partitioning of multi-core shared caches. *SIGARCH Comput. Archit. News*, 37(3):174–183, 2009.

A Proofs for Section 5

Theorem 2. *PARTIAL-INDIVIDUAL-FAULTS is NP-complete.*

Proof. It is easy to see that the problem is in NP: given an instance $\mathcal{I} = \{\mathcal{R}, K, t, \tau, \boldsymbol{b}\}$ with a "yes" answer, and a certificate consisting of the pages to be evicted after each fault, it can be verified in time $O(tp)$ that the number of faults in each sequence R_i is at most b_i. Note that $t \leq \max_i\{|R_i|\}(\tau + 1)$, and $\tau = O(\max_i\{|R_i|\})$, therefore the verifier runs in time polynomial in the size of the input.

In order to show that the problem is NP-complete, we build a reduction from 3-PARTITION. Recall that an instance of 3-PARTITION consists of a set of n integers $S = \{s_1, \ldots, s_n\}$, and a bound B, such that $B/4 < s_i < B/2$ for all $1 \leq i \leq n$. The problem is to determine if S can be

partitioned into $n/3$ sets $A_1, \ldots, A_{n/3}$ such that for all $1 \leq j \leq n/3$, $\sum_{i \in A_j} s_i = B$. Note that the restrictions of the problem imply that each subset A_i must have exactly 3 elements [14].

Let $\mathcal{J}$ be an instance of 3-PARTITION. We build an instance $\mathcal{I}$ of PARTIAL-INDIVIDUAL-FAULTS as follows. There are $p = |S|$ sequences. Each sequence R_i consists of alternating requests to 2 pages α^i and β^i, where $\alpha^i \neq \beta^i$, and $\alpha^i \neq \alpha^j$ and $\beta^i \neq \beta^j$ for all $i \neq j$, and $\alpha^i \neq \beta^j$ for all i, j. In other words, $R_i = \alpha^i \beta^i \alpha^i \beta^i \ldots$, and all sequences are disjoint. The length of R_i is $|R_i| = B(\tau + 1) + 4\tau + 5$, where $\tau \geq 0$ is any integer. The size of the cache is $K = (4/3)p$, and we want to know if the number of faults in each sequence R_i is at most $b_i = B - s_i + 4$, at time $t = B(\tau + 1) + 4\tau + 5$. Note that since 3-PARTITION is strongly NP-complete (it remains NP-complete if the input is encoded in unary), the reduction can be done from an instance encoded in unary, and hence it can be done in time polynomial in the size of $\mathcal{J}$.

We show now that there exists a solution for $\mathcal{J}$ if and only if we can serve each R_i with at most b_i faults.

($\Rightarrow$) We show first that if $\mathcal{J}$ admits a solution then we can serve each R_i with at most b_i faults. Let $A_1, \ldots, A_{n/3}$ be the partition for $\mathcal{J}$. Divide the sequences in groups according to the partition, so that the sequences corresponding to A_j will share a group of 4 cells of the cache. Let R_{i_1}, R_{i_2}, and R_{i_3} be the sequences in group j. Each of these sequences will be assigned one cell for some time and two at other times. In other words, the three sequences will have one dedicated cell at least until time t and will share the extra cell of the group. Sequence R_i will use the extra cell continuously for enough time so it incurs in exactly $h_i = s_i(\tau + 1) + 1$ hits.

Say R_{i_1}, R_{i_2}, and R_{i_3} use the extra cell in that order. The first request to each sequence results in a fault and it is fetched to the dedicated cell of the corresponding sequence. The second request of R_{i_1} (also a fault) is fetched to the extra cell. Now both pages of R_{i_1} are in the cache, and they are kept there for the next h_{i_1} requests of R_{i_1}. Meanwhile, every request of R_{i_2} and R_{i_3} results in a fault and the eviction of the page in their corresponding dedicated cell. The last hit of R_{i_1} occurs at time $(2 + s_i)(\tau + 1) + 1$, which coincides with a new request σ for R_{i_2}, since all pages have been faults for R_{i_2}. Instead of fetching σ to this sequence's dedicated cell, σ is fetched into the extra cell or R_{i_1}'s dedicated cell, depending on which page can be evicted at the time (if σ is fetched into R_{i_1}'s dedicated cell then this cell becomes the shared cell and the former shared cell becomes R_{i_1}'s dedicated cell). Now R_{i_2} has the extra cell and the remaining requests of R_{i_1} will result in faults. After h_{i_2} hits of R_{i_2}, the extra cell is now passed to R_{i_3} (again the last hit of R_{i_2} coincides with a request of R_{i_3}), and this sequences keeps this cell until it completes h_{i_3} hits. At this point, the time elapsed is the sum of the hits of each sequence, plus 2τ for the transitions of the extra cell from R_{i_1} to R_{i_2} and from R_{i_2} to R_{i_3}, plus the initial $2(\tau + 1)$ time corresponding to the first 2 faults of the three sequences. Hence the time is $t = h_{i_1} + h_{i_2} + h_{i_3} + 4\tau + 2 = (s_{i_1} + s_{i_2} + s_{i_3})(\tau + 1) + 4\tau + 5 = B(\tau + 1) + 4\tau + 5$. The same strategy is used for each group in the partition, and the number of faults of sequence R_i is exactly $(t - h_i)/(\tau + 1) = B - s_i + 4$. Thus, if there is a solution to $\mathcal{J}$ $\mathcal{R}$ can be served such that at time $t = B(\tau + 1) + 4\tau + 5$, each sequence has incurred in $b_i = B - s_i + 4$ faults.

($\Leftarrow$) We show now that a solution to the instance $\mathcal{I}$ of PIF gives a solution to the instance $\mathcal{J}$ of 3-PARTITION. If $\mathcal{R}$ can be served so that each sequence faults at most $B - s_i + 4$ times by time t, then at least $h_i = s_i(\tau + 1) + 1$ of R_i's requests must be hits. Note that for all i, $|R_i| = t$, and hence each sequence uses at least one cell until time at least t. A request can only be a hit if a sequence has two cells of the cache for consecutive timesteps. Since there are only $(4/3)p$ cells and

each sequence uses at least one cell, there are only $p/3$ extra cells which can be used to store the second page of a sequence, and hence there can be at most $p/3$ hits in one timestep.

In addition, any change in the partition that removes one cell from a sequence that had 2 cells and gives one more cell to another sequence implies at least τ time without hits for the sequences involved. To see this, let R_i and R_j be two sequences such that $k(i,t_1) = 2$ and $k(j,t_1) = 1$, and $k(i,t_1+1) = 1$ and $k(j,t_1+1) = 2$ for some $t_1 < t$. Say page α^i was a hit in R_i. Then at t_1+1, β^i and α^j are requested in R_i and R_j, respectively[4]. α^i is evicted from the cache, and α^j is fetched in to the cell previously used by α^i. Sequence R_j must wait τ more timesteps before having its first hit, while sequence R_j's next request (α^i) will result in a fault. Hence, there are no hits in these two sequences in the period $[t_1+2, t_1+1+\tau]$, i.e. τ timesteps without hits.

Let C denote the number of all such changes in partition between a pair of sequences. C does not count the initial assignment of 2 cells to a sequence when starting to serve $\mathcal{R}$, but only when a sequence acquires an extra cell that held a page of another sequence. Note that no hits can happen until time $2(\tau+1)$, and hence the total number of possible hits before time t can be at most $H_1 = (p/3)(t-2(\tau+1)) - C\tau = (p/3)(B-2)(\tau+1)+4\tau+5) - C\tau$. On the other hand, the minimum number of required hits is $H_2 = \sum_{i=1}^{p} h_i = \sum_{i=1}^{p} s_i(\tau+1)+1 = (p/3)B(\tau+1)+p$. We require $H_1 \geq H_2$, which implies that $C \leq 2p/3$, i.e. we can have at most $2p/3$ changes in partitions. Note that initially at most $p/3$ sequences can be assigned two cells, therefore every sequence must have 2 cells during a continuous period of time. We call this period the hit period of a sequence.

Consider a group of sequences whose hit periods are consecutive, i.e. a group $I = \{i_1, i_2, ...i_\ell\}$ of the sequences such that a request in sequence i_{j+1} evicts a page from sequence i_j to start its hit period. These sequences can be served using $\ell+1$ cells: one cell is dedicated for each sequence, while the other extra cell is used to assign 2 cells to some sequence during its hit period (this extra cell need not to be the same one during the entire execution). We claim that $\ell \leq 3$. Too see this, assume $\ell > 3$, then since hit periods have no interruptions, the total time to serve these sequences is at least the total number of hits, plus the time for each change in partition, plus 2 initial faults, for a total of $T = \big(\sum_{i\in I} h_i\big)+(\ell-1)\tau+2(\tau+1) = \big(\sum_{i\in I} s_i(\tau+1)+1\big)+(\ell+1)\tau+2 > (\ell/4)B(\tau+1)+\ell+(\ell+1)\tau+2$, since $s_i > B/4$ for all $1 \leq i \leq p$. Taking $\ell = 4$, $T > B(\tau+1)+5\tau+6$, which is strictly greater than t, and thus one sequence will not have all its required hits before t and hence it will exceed the allowed faults. Hence $\ell \leq 3$. Furthermore, ℓ is exactly 3. Assume, otherwise, that sequences are served in groups of 1, 2, and 3 sequences. Let n_ℓ be the number of groups of ℓ sequences, for $\ell \in \{1,2,3\}$. In order to satisfy the minimum number of hits for each sequence, it must be the case that a group of ℓ sequences must use at least $\ell+1$ cells. Hence, the following must be satisfied: $n_1+2n_2+3n_3 = p$, and $2n_1+3n_2+4n_3 \leq (4/3)p$, with $n_1, n_2, n_3 \geq 0$. It is not difficult to see that a feasible solution must have $n_1 = n_2 = 0$, and R must be served only with groups of 3 sequences.

Finally, it must be the case that every group of sequences $I = i_1, i_2, i_3$ must satisfy $s_{i_1}+s_{i_2}+s_{i_3} = B$. Suppose that a group I_1 is such that $\sum_{i\in I_1} s_i < B$. Then since $B = (3/p)\sum_{i=1}^{p} s_i$ there must exist another group I_2 such that $\sum_{i\in I_2} s_i > B$. Then, since the minimum number of hits per sequence is $h_i = s_i(\tau+1)$, the total time to serve the group would be $T = \big(\sum_{i\in I_2} h_i\big) + 2\tau + 2(\tau+1) > B(\tau+1)+4\tau+5 = t$, and thus at least one sequence in the group would have to fault more than its maximum number of allowed faults by time t. Therefore, the only possible way to serve the requests satisfying the faults requirement for each sequence is to divide them in groups of 3

[4] Note that R_j might be fetching a page instead, but the case when a new request comes is the one that minimizes the time that elapses from t_1 until R_j's next hit.

sequences $I_1, \ldots, I_{p/3}$ such that $\sum_{i \in I_j} s_i = B$ for all $1 \leq j \leq p/3$, which is a solution for the instance of 3-PARTITION. □

Theorem 4. *Let Alg be an offline optimal algorithm that is capable of forcing faults. There exists an offline algorithm Alg' that is honest such that $\forall$ disjoint $\mathcal{R}$, $Alg'(\mathcal{R}) = Alg(\mathcal{R})$.*

Proof. We follow an inductive argument similar to the proof of optimality of Furthest-In-The-Future in the sequential setting [6]. The proof relies on the following claim: let Alg be any paging algorithm and R be any request sequence. Then, for all timesteps i it is possible to construct an algorithm Alg_i such that (i) for all $t = 1, \ldots, i-1$, it behaves exactly like Alg, (ii) if Alg forces a fault on $t = i$, Alg_i does not, and (iii) $Alg_i(R) \leq Alg(R)$.

If the claim is correct, this implies that it is possible to obtain an optimal algorithm that does not force faults: for a given sequence R, start from any optimal algorithm OPT, and apply the claim with $i = 1$ to obtain OPT_1, then apply the claim with $i = 2$ to OPT_1 to obtain OPT_2, and so on and so forth. $OPT_{t'}$ is an optimal algorithm that does not force faults, where t' is the maximum timestep of the execution of the algorithm on R.

Let us prove the claim. Both algorithms start with an empty cache and hence Alg_i can do exactly as Alg does up to step $i-1$. If at timestep i Alg does not force a fault, then Alg_i continues behaving like Alg until the end of the request, and the number of faults of both algorithms is the same. Now, assume that Alg_i forces a fault on $t = i$ on a page p_1 on sequence R_s. Let $C_{Alg}(t)$ and $C_{Alg_i}(t)$ denote the caches of Alg and Alg_i right before serving $R(t)$. Since both algorithms behaved exactly the same up to $t = i-1$, $C_{Alg}(i) = C_{Alg_i}(i)$. Also, let $A(R,t)$ denote the number of faults of algorithm A right before serving request $R(t)$.

We argue that from that point on Alg_i can be such that both algorithms will fault on exactly the same pages in the rest of the sequences (and hence keeping the same alignment with respect to both algorithms), and that their caches will differ by at most one page. Furthermore, $Alg_i(R) \leq Alg(R)$ at all times. We show this by defining a set of states that describe the differences between the algorithms sequences, caches, and number of faults for each subsequent request in R_s for Alg. We define the following states at time t:

(A) All sequences in both algorithms have the same alignment, $C_{Alg}(t) = C_{Alg_i}(t)$, and $Alg(R,t) = Alg_i(R,t)$.
(B) All sequences other than R_s have the same alignment in both algorithms, $C_{Alg}(t) = C_{Alg_i}(t)$, and $Alg(R,t) = Alg_i(R,t) + 1$.
(C) All sequences other than R_s have the same alignment in both algorithms, $C_{Alg}(t) = C_{Alg_i}(t) \cup \{p\}$, and $Alg(R,t) = Alg_i(R,t)$, where p is a page previously requested in R_s, and the remaining cell of Alg_i cache is fetching a page $p' \neq p$.
(D) All sequences other than R_s have the same alignment in both algorithms, $C_{Alg}(t) = (C_{Alg_i}(t) \cup \{p\}) \setminus \{\alpha\}$, and $Alg(R,t) = Alg_i(R,t) + 1$, where p is a page previously requested in R_s, and α is a page from a sequence other than R_s.
(E) All sequences other than R_s have the same alignment in both algorithms, $C_{Alg}(t) = C_{Alg_i}(t) \cup \{p\}$, and $Alg(R,t) = Alg_i(R,t)$, where p is a page previously requested in R_s, and the remaining cell of Alg_i cache is fetching p.

Let $p_2, p_3, \ldots$ be the pages in R_s after p_1. We define the request period of each page $p_j \in R_s$ as the timesteps that include its request and possible fetching for algorithm Alg, i.e. if p_j is request at time t_j by Alg, then its request period is $[t_j, t_j + \tau]$ if p_j is a fault, and just t_j if it is a hit. We will

now show that the above are all the possible states that describe the algorithms in each period. We will prove this by induction on the request periods of pages in R_s.

Before p_1, the algorithms are in state (A). Consider the request period for p_1. Recall that $C_{Alg}(t_1) = C_{Alg_i}(t_1)$. Alg forces a fault on p_1 and hence the cell corresponding to p_1 in the cache is being used to fetch this page until the end of the period. Upon any request σ during the period, if Alg evicts α, Alg_i evicts α as well. In this period up to τ pages $p_2, ..., p_{1+\tau}$ after p_1 might be requested for Alg_i in R_s. If none of these are faults, then at the end of the period $Alg(R, t_2) = Alg_i(R, t_2) + 1$ and the caches of both algorithms are equal, hence the algorithms are in state (B)[5]. On the other hand, if any of the pages $p_2, ..., p_{1+\tau}$ is a fault for Alg_i, then Alg_i evicts any of the previous hit pages in R_s (there is a least one, p_1). Hence at the end of the period the number of faults of both algorithms is the same, and both caches have the same pages, but for the evicted page by Alg_i, thus arriving at state (C).

Assume the algorithms are now in one of the states (A),(B),(C),(D), or (E), on period p_j, for $j > 1$. We show now that Alg_i can be such that the state of the next period is one of these as well.

[$s(j) = A$] Suppose we have reached state (A). Since Alg_i cannot force faults only on request i, but it can do so on later requests, it just behaves exactly like Alg for the rest of the sequence, maintaining state (A).

[$s(j) = B$] We can only arrive to state (B) if Alg_i had no faults for requests in R_s in the previous period. Since p_j was requested for Alg_i in the previous period (sequences R_s in both algorithms are misaligned by τ at most), p_j is a hit for Alg. If there is any fault on a request of another sequence, Alg_i evicts whatever Alg evicts. At time t_j, $p_{j+\tau}$ is requested for Alg_i. If this page is a hit, then we stay in state (B). If this page is a fault, Alg_i evicts some page $p_{j'}$ with $j' < j + \tau$. At least one page of R_s is in Alg_i's cache since they were all hits in the previous period (and we assume they are not evicted during this period by Alg). Hence, at the end of the period $Alg(R, t_{j+1}) = Alg_i(R, t_{j+1})$ and $C_{Alg}(t_{j+1}) = C_{Alg_i}(t_{j+1}) + \{p\}$, for some $p \in R_s$, arriving at state (C).

[$s(j) = C$] At the beginning of this period p_j is requested for Alg and Alg_i is fetching some page $p_{j'}$ that resulted in a fault in the previous period. This is the case if we arrive from states (A), (B), and we will see that it holds if we stay in (C), or arrive from (D) or (E) as well. For all faults in sequences other than R_s, Alg_i evicts the same page that Alg evicts, unless Alg evicts p, in which case Alg_i evicts another page in R_s. If p_j is a hit for Alg then nothing changes and we stay in (C) as well. If p_j is a fault, then Alg evicts a page α. Recall that $C_{Alg}(t_{j+1}) = C_{Alg_i}(t_{j+1}) + \{p\}$. Assume first that $\alpha \neq p$. Then, if all the subsequent pages of R_s requested for Alg_i are hits and α is not requested during this period, then we have one more fault for Alg and at the end of the period Alg's cache still has p but not α, while Alg_i does not have p but has α, hence reaching state (D). If α is requested during the period, then Alg_i forces a fault on this page and hence we remain in state (C). Now, if one of the requests from R_s for Alg_i results in a fault, say $p_{j'}$, then, again if α was not requested before $p_{j'}$, then Alg_i evicts α for this request. If α is requested before $p_{j'}$, Alg_i forces a fault on α and evicts for $p_{j'}$ whatever Alg evicted for α (or some page $p' \in R_s$ if Alg evicts p). At the end of the period the difference in number of faults remains the same. Now, if $p_{j'} \neq p$,

[5] We assume here and for the rest of the analysis that Alg_i lets Alg run ahead at least until its next fault in R_s, so that if it evicts a page p_j after the time this page is requested for Alg_i, Alg_i forces the fault on this page as well. In other words, if p_j is a hit for Alg_i at time t, it will be a hit for Alg at time $t + \tau$.

then we stay in state (C). However, if $p_{j'} = p$, i.e. Alg_i faulted in the page that it did not have but that Alg had, then we reach state (E).

Now, we analyze the case $\alpha = p$. The first page in R_s requested for Alg_i is p_{j+1}. If this page is a fault, then Alg_i evicts another page $p' \in R_s$, for example p_j. In this case the number of faults increases by 1 for both algorithms and we remain in state (C). If on the other hand, p_{j+1} is in Alg_i's cache (and hence in Alg's cache), Alg_i forces a fault on this page, reaching state (E).

[$s(j) = D$] We could only reach this state if p_j is a hit. Let $p_{j'}$ be the page of R_s requested for Alg_i. Suppose α is not requested in $R(t_j)$. If $p_{j'}$ is a hit, then nothing changes, and we remain in state (D). If $p_{j'}$ is a fault, Alg_i evicts α and we reach state (C). If α is requested, since $\alpha \notin C_{Alg}(t_j)$, Alg evicts a page α'. Alg_i forces a fault on α. If $p_{j'}$ is a hit, then we remain in state (D). If $p_{j'}$ is a fault, Alg_i evicts α' if $\alpha' \neq p$, or another page $p' \in R_s$ if $\alpha' = p$, reaching state (C).

[$s(j) = E$] This state is reached when the number of faults of both algorithms is the same and Alg_i is fetching the page p that is missing with respect to Alg's cache. Suppose first that $p_j \neq p$ (i.e. this is not the timestep in which Alg_i finishes fetching p[6]). p_j is a hit for Alg (this is the case in the two cases that we can arrive to this state from (C), and the one from (E)). Upon any fault on another sequence Alg_i evicts whatever Alg evicts (with the assumption that Alg will not evict a page of R_s that was a hit in the previous period for Alg_i). If one of these evictions is for page p, then Alg_i evicts another page $p' \in R_s$ and we reach state (C). If p was not evicted, then we remain in state (E). Now, if $p_j = p$, this page is in Alg's cache. Again, Alg_i evicts what Alg evicts for other requests. If, however, in any of these evictions is the page evicted is p[7], Alg_i evicts another page $p' \in R_s$. In this case Alg faults in p_j and we are back in state (C). On the other hand, if $p_j = p$ is a hit, Alg_i finishes at the same time to fetch p, therefore both caches are equal, sequences R_s in both algorithms are aligned, and the number of faults of both algorithms is the same, thus we reach state (A).

Therefore, the algorithms can only be in these states until the end of the execution. Since in any state $Alg_i(R) \leq Alg(R)$, this will hold until the end of the sequence, proving the claim. □

[6] Note that the sequences R_s in the execution of both algorithms is either aligned or misaligned by exactly τ, with Alg's sequence being behind Alg_i's. Thus if the page p_j being requested for Alg is the one being fetched by Alg_i, it must be the case that this is the timestep in which Alg_i finishes fetching p_j.

[7] Since we assume that a parallel request is served independently, it could be the case that p is evicted despite being requested in the same parallel request.