Paging for Multicore (CMP) Caches
University of Waterloo Technical Report CS-2010-15

Alejandro López-Ortiz Alejandro Salinger *

Abstract

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 use a shared cache the performance of even the most common eviction policies is not yet fully understood, nor do we know if current best strategies in practice are optimal. In particular, there is almost no theoretical backing for the use of current eviction policies in multicore processors. Recently, a work by Hassidim [14] initiated the theoretical study of cache eviction policies for multicore processors under the traditional competitive analysis, showing that LRU is not competitive against an offline policy that has the power of arbitrarily delaying requests 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 perform a thorough all-to-all comparison of strategies providing lower and upper bounds for the ratios between performances. We show that if the cache is partitioned, the partition policy has a greater influence on performance than the eviction policy. On the other hand, we show that sharing the cache among cores is in general better than partitioning the cache, unless the partition is dynamic and changes frequently, in which case shared cache strategies and dynamic partitions are essentially equivalent when serving disjoint requests.

*David R. Cheriton School of Computer Science, University of Waterloo, 200 University Ave. W., N2L 3G1, Waterloo, Ontario, Canada, e-mail: {alopez-o,ajsalinger}@uwaterloo.ca.

1 Introduction

Multicore processors have become the dominant processor architecture. Currently two, four and eight core processors are generally available, and this number is expected to grow rapidly. While the specific details of these architectures are still in flux all of them so far include a cache area shared across multiple cores. The performance of this cache has been extensively studied within the field of systems research e.g. [19, 1, 23] where they are known as Chip Multiprocessors (CMPs). Recently, Hassidim initiated the theoretical study of paging strategies for shared caches in CMPs [14], where in a system with p cores, a shared cache might receive up to p page requests simultaneously. Hassidim's work uses 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 operating system has kept the scheduling of execution and the paging tasks separate.

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. We study various algorithms in the CMP model. We divide the family of algorithms in two: algorithms that completely share the cache among cores, and algorithms that divide the cache among cores and serve the requests of each core in each part. In the former, the algorithm is specified by the eviction policy. In the latter, both an eviction policy and a partition strategy define the algorithms. Observe that the partition strategy can be static or dynamic.

We compare the performance of various algorithms, and derive lower and upper bounds on the worst case ratio between the number of faults between each pair of algorithms. We show that partitioning the cache statically is not competitive against sharing the cache, even when requests of different processors are disjoint. We also show that if a static partition is required, then the choice of the partition has more impact than the choice of the eviction policy. On the other hand, we show that when different processors request non-disjoint sets of pages, dynamic partitions can perform arbitrarily badly when compared to a shared strategy. Even when restricted to disjoint sequences, we show that dynamic partitions that do not make frequent changes are also not competitive against shared strategies, while dynamic partitions that do, can match the performance of shared strategies.

The rest of the paper is organized as follows. We review related work in Section 2. In Section 3 we describe the cache and cost models we use throughout the paper. In Section 4 we describe the algorithms that we then compare in Section 5. We describe an algorithm to compute the optimal static partition in Appendix D. 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 the research in the subject has increased notably since the appearance of multicore architectures. A variety of works have studied cache strategies in practice, developing strategies to dynamically partition the cache, or to manage cache at the operating system level.

In the context of one processor with multiple threads running either interleaved or by time slices, [21] shows how to compute an offline optimal partition of the cache, and show that for some examples LRU produces cache allocations that are close to optimal. They show as well that a modified version of LRU that evicts only pages form other threads sometimes performs better. Other works propose dynamic partition strategies for simultaneous multithreading systems or CMPs with the goal of improving various performance measures, e.g. hit rates [22], throughput

[17], fair speedup (a global performance fairness metric) [20], or both throughput and fairness [7]. Other works propose approaches to manage shared cache by the operating system through memory address mapping [16] or combining process scheduling and data mapping [8]. [12] proposes an operating system scheduling algorithm that adjusts the CPU time each thread gets to account for unequal cache sharing aiming to minimize the effects of unfair cache sharing. [24] studies a cache management approach that combines dynamic insertion and promotion policies to provide effects similar to cache partitioning. Dynamic insertion policies [18] was further extended in [15].

A recent work [25] measures the influence of various factors related to the types of programs (e.g. number of threads), operating system (e.g. assignment of threads to cores), and architecture (e.g. number of cores) and shows that this influence is rather limited on a benchmark which is representative of shared-memory programs for CMP [2]. They argue that a reason for this is the mismatch between programs and compilers of multithreaded applications and CMP architectures, and show experimentally that a series of cache-sharing-aware program transformations can improve performance. These works motivate us to study the fundamental properties of programs running in multiple cores, in our case with respect to cache efficiency, so that cache allocation and eviction policies take advantage of the underlying architecture.

On the other hand, other works with a theoretical focus have addressed the cache performance of multithreaded systems, mostly from the point of view of schedules and algorithms with good cache performance. See, for example, [5, 3, 4]. The work in [13] proposes an analytical model which predicts the performance of different cache replacement policies for a particular application, and thus it can be used compare the performance of cache replacement policies, however, only in an application-by-application basis. Our work aims to compare cache policies and strategies in general, for arbitrary input sequences.

A recent paper by Hassidim [14] is the first to address cache eviction policies in the multicore setting from the traditional competitive analysis point of view. This work shows that when serving parallel disjoint requests in a single shared cache, Least-Recently-Used (LRU) performs badly with respect to the optimal offline (see [6] for the definitions of common paging algorithms). More specifically, the competitive ratio of LRU 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.

However, Hassidim's model assumes that an offline cache strategy has the power to serve sequences of some cores and delay others. In other words, the offline strategy is able to modify the schedule of requests, and hence has extra advantage over regular cache eviction online algorithms. 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 cache must serve the request, and it cannot be delayed. Secondly, [14] studies only sequences of disjoint pages. It should be no surprise that under this assumption the advantage of a shared cache is diminished. We study and compare both the settings with shared and disjoint pages.

The traditional measure for the performance of sequential paging algorithms is the competitive ratio, in which algorithms are compared to the optimal offline algorithm. Several other measures have been studied, many of which compare online algorithms directly to each other, as we do in this work. (See e.g. [11], and [9, 10] for a survey of performance measures for online algorithms.)

3 The Cache Model

The model we use in this paper is broadly based on Hassidim's model [14]. We have a multicore processor with p cores $\{1, \ldots, p\}$, and a shared L2 cache of size K pages. The input is a multiset

of request sequences $\mathcal{R} = \{R_1, \ldots, R_p\}$, where $R_j = r_1^j, \ldots, r_{n_j}^j$ is the request sequence of core j of length n_j. Let $n = \sum_{j=1}^{p} n_j$, i.e., n is the total number of requests. 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. A request r_i^j is a tuple (t_i^j, σ_i^j) where σ_i^j is the identifier of the i-th page requested by processor j and t_i^j indicates the earliest possible time at which the request σ_j^i could be made shall there be no page faults on earlier requests of process j. Thus, at any given time, the shared cache might receive up to p requests simultaneously. 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. Note also that the above definition of requests accounts for the fact that not all processors request pages at all times. To simplify analysis and notation, we assume that all processors do have requests at all times, but that requests can be for an empty page $\perp$, which does not have any effect on the cache. We then describe requests in terms of the page components of each individual request $r = (t, \sigma)$. Thus, for processor j, we will specify R_j as $R_j = \sigma_1^j \ldots \sigma_{n_j}^j$, where σ_i^j could be empty. When we talk about the length of a sequence n_j, this length includes empty pages. We say that a request $\mathcal{R}$ is *disjoint* if $\bigcap_{j=1}^{p} R_j = \emptyset$ and *non-disjoint* otherwise.

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 $\sigma_{i^*}^j$ is a miss, then R_j is updated by making $t_i^j \leftarrow t_i^j + \tau$, for all $i > i^*$.

To be consistent with [14], 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. In addition, since we allow requests to the same page by different processors, we use the convention that when there is a request by processor j, of a page that is currently in the process of being fetched, then the sequence of processor j is only delayed until the page is fetched into the cache τ units of time after the initial request. We also assume that cache coherency is provided at no cost to the algorithms.

As in [14], we assume that 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.

Finally, we assume that simultaneous requests are served independently. Let $\mathcal{R}(t)$ denote the requests of all processors at time t. We assume that decisions made one each page of $\mathcal{R}(t)$ do not depend directly on other pages in $\mathcal{R}(t)$. Decisions on some pages in $\mathcal{R}(t)$ can affect the decision on other pages indirectly, but only as a result of applying the eviction policy when serving those pages. In addition, and w.l.o.g., we adopt the convention that individual requests within a simultaneous request are served logically in a fixed priority order (e.g. by increasing number of processor), and that this order is consistent with the eviction decisions of the algorithm. For example, for LRU, if two pages σ_1 and σ_2 are part of the same request $\mathcal{R}(t)$ but σ_1 is served before σ_2 in terms of the logical order, then the last access to σ_1 is earlier than the one to σ_2.

3.1 Cost model

Traditionally, the cost of a paging algorithm on a request sequence has been measured in terms of the number of faults it incurs when serving the sequence. An alternative measure is the time that the algorithm takes to serve the request. In the sequential setting both measures are equivalent so

[1]Note that in [14], τ is defined as the fetching time, which would be $\tau + 1$ in this paper.

long as the time for a fault is greater than the time for a hit. In the case of parallel requests, it is not immediately clear that both measures are equivalent. In our model, a fault delays the remaining requests of only one processor, while the requests of other processors continue to be served. Hence a sensible goal is to minimize the total time of serving. However, in this work we measure the performance of a strategy in terms of the total number of faults, since it allows us to compare the results of the parallel setting with the traditional results in the sequential setting, which are usually expressed in terms of this cost measure. It is not difficult to see that both measures are equivalent when comparing the performance on sequences of the same length.

Let $A(\mathcal{R})$ be the number of faults that algorithm A incurs on sequence $\mathcal{R}$ and let $T_A(\mathcal{R})$ be the total time for serving $\mathcal{R}$ under policy A. Let H and F be the hit and fault times when serving a page. An empty page request spans one unit of time. For simplicity we assume $H = 1$, and we let $\tau = F - 1$. Thus, $T_A(\mathcal{R}) \leq A(\mathcal{R})F + |\mathcal{R}| - A(\mathcal{R}) = |\mathcal{R}| + A(\mathcal{R})(F-1) = |\mathcal{R}| + A(\mathcal{R})\tau$, where $|R| = \sum_{j=1}^{p} n_j$, and the inequality is explained by the fact that simultaneous faults on the same page only have to wait for the page to be fetched due to the first fault. When requests from different processors are disjoint, we have equality. It is straightforward to note that for all algorithms A and A', and all parallel disjoint requests $\mathcal{R}$ and $\mathcal{R}'$ such that $|\mathcal{R}| = |\mathcal{R}'|$, $T_A(\mathcal{R}) < T_{A'}(\mathcal{R}') \Leftrightarrow A(\mathcal{R}) < A'(\mathcal{R}')$. The same holds for the equality. Both for requests that are disjoint and non-disjoint, the number of faults places an upper bound on the time that it takes to serve the request.

Another reasonable goal is to minimize the makespan, i.e., the maximum among the times that it takes to serve the requests of all processors. In this work we focus entirely on total time (through the number of faults), in the understanding that it is of interest not only that the parallel request be finished as soon as possible, but that the total work be minimized and hence resources are freed for serving further requests.

4 Caching Algorithms

We consider two families of strategies for managing the cache: shared and partitioned. 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. Part i is destined exclusively to store pages that belong to requests from processor i. Among partition strategies, we consider static and dynamic partitions. In a static partition the sizes of the parts for each processor are determined at the beginning of the execution of the algorithm and remain fixed until the last page is served. In a dynamic partition, the sizes of the parts can vary throughout the execution.

Any of these strategies is accompanied by an eviction policy A. We call S_A 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^B are the static partition and dynamic partition algorithms that use eviction policy A and partition B, respectively.

An offline partition is dependent on the eviction policy A, the request sequence $\mathcal{R}$, and the size of the cache K. It defines a function $k : \{P, \mathbb{N}\} \rightarrow \{\text{part}(K, p)\}$, where $P = \{1, \ldots, p\}$ is the set of processors, and $part(K, p) = \{\{k_1, k_2, \ldots, k_p\} \mid k_i \in \{0, \ldots, K\} \text{ and } \sum_{i=1}^{p} k_i = K\}$ 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 current request is not empty.

For example, according to the notation defined above, S_{LRU} is the strategy that evicts the current page in the entire cache that was least recently used. Examples of static partition strategies are sP_{LRU}^{OPT}: LRU is performed on each sequence in each part of the cache, which is partitioned offline to minimize the total number of faults; and sP_{OPT}^{EQ}: each processor is assigned an equal part

of the cache, and the algorithm performs the optimal offline strategy on each sequence in each part of the cache. We show an example of the execution of a shared strategy and a static partition strategy in Appendix A.

4.1 Dynamic Partition Algorithms

Dynamic partition strategies involve possible changes in the cache partition during the service of a request. A reduction in the size of the cache for a processor can have different consequences for the pages currently in the cache for that processor. The convention we use is the following. If at any time t, the part of the cache of processor j has size $k(j,t)$ and contains $c(j,t)$ pages, then if at time $t+1$, $k(j,t+1) < k(j,t)$, then $\min\{0, c(j,t) - k(j,t+1)\}$ pages are evicted from the cache according to the eviction policy.

We define a *stage* of a dynamic partition strategy to be each period of time during an execution of the algorithm in which the sizes of the caches are constant. The convention about transitions described above simplifies the analysis by allowing us to consider each stage independently of the previous ones: we need only know about the new cache size and the current pages in the cache at the beginning of the stage. We define a restriction on dynamic partition strategies that will be useful in proving some of our results.

Definition 1 (Stable partition) Let B be a dynamic partition algorithm and k be the partition function that it defines. Let m_j^s be the number of non-empty pages requested by processor j in a stage s. We say that a stage s is long if for all processors j, $m_j^s > k_j^s$, where $k_j^s = k(j,t)$ for some t during s, is the size of the cache of processor j in stage s. In other words, the length of the stage (in number of requests) is larger than the sizes of all caches in the partition. We say that a dynamic partition is *stable* if all its stages are long.

5 Comparison of strategies

Table 1 shows an all-vs-all comparison of caching strategies. Two entries in a cell $T(i,j)$ correspond to a lower bound and upper bound in the number of faults between algorithms Alg_i and Alg_j in the following sense. Let $T(i,j) = f_1(n,K,p), f_2(n,K,p)$. Then $\exists \mathcal{R}$ such that $Alg_i(\mathcal{R})/Alg_j(\mathcal{R}) \geq f_1(n,K,p)$, and $\forall \mathcal{R}$ $Alg_i(\mathcal{R})/Alg_j(\mathcal{R}) \leq f_2(n,K,p)$. If a cell has only one entry, and this entry is of the form Ω or ω, this corresponds to a lower bound, otherwise it corresponds to a matching lower and upper bound. In the case of a generic partition strategy B and an eviction policy A, when in the numerator, these should be interpreted as "for all online partition strategies B", and "for all marking or conservative deterministic online eviction policies A"[2] When in the denominator, the quantifier is existential for the lower bound and universal for the upper bound. If both in the numerator and denominator, generic strategies should be regarded as equal. For adversarial sequences with optimal offline partitions, the sequence is fixed before the optimal partition is decided.

[2]Recall that a k-phase partition of a request $\mathcal{R}$ sequence is a partition of $\mathcal{R}$ in subsequences such that the first phase starts at the beginning of $\mathcal{R}$, and a new phase starts every time there is a request for a $(k+1)$-th different page since the end of the previous phase. A *marking* algorithm associates (explicitly or implicitly) a bit to each page. At the beginning of each phase all pages in the cache are unmarked and each page is marked when it is first requested during the phase. A marking algorithm never evicts a marked page and therefore it makes at most k faults in a phase. LRU and FWF are marking algorithms. A *conservative* algorithm incurs in at most ℓ faults on any subsequence of at most ℓ distinct pages. LRU and FIFO are conservative algorithms[6].

In the rest of the section we highlight and discuss the most important results in Table 1. Due to space constraints, we include only sketches of the proofs in this section, while full proofs are provided in Appendix B. We also provide short justifications for the values of all entries in Table 1 in Appendix C.

5.1 Static partition strategies

The main observation from our results when comparing different partition strategies is that the choice of a good partition is more important than a good eviction policy algorithm. For equal static partitions, we observe that both lower bounds and upper bounds are close to the size of the cache, as in the sequential setting. A representative entry in Table 1 is the comparison between static partition with an online eviction policy and a fixed partition B, and static partition with an optimal offline eviction policy and the same partition B.

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 $sP_A^B(\mathcal{R})/sP_{OPT}^B(\mathcal{R}) \geq \max_j\{k_j\}$. When A is any marking or conservative algorithm (e.g. LRU), there is a matching upper bound, i.e., $\forall\ \mathcal{R}$, $sP_A^B(\mathcal{R})/sP_{OPT}^B(\mathcal{R}) \leq \max_j\{k_j\}$, which also holds if B is offline.*

Proof sketch[3]: The proofs for both the lower bound and upper bound are analogous the proofs of the competitive ratio of the most common algorithms in the sequential setting (e.g. LRU, FIFO).

An adversary with the power of an offline eviction policy can only be K times better than algorithms with reasonable eviction policies. However, if the adversary is given the power to partition the cache offline, no algorithm with an online partition can compete with it. The following lemma shows this, which holds even for an optimal offline eviction policy:

Lemma 2 (Online static partition is not competitive against an offline static partition) *Let $B = \{k_1, ..., k_p\}$ be any static online partition. Let $j^* = argmin_j\{k_j | k_j \geq 2\}$. There exist a sequence $\mathcal{R}$, such that for all A, $sP_A^B(\mathcal{R})/sP_{LRU}^{OPT}(\mathcal{R}) \geq \min\{k_{j^*}, p-1\}\frac{n}{K^2 p} = \Omega(n)$.*

Proof sketch[3]: Consider $A = LRU$. All processors but one request repeatedly $k_j + 1$ different pages, and the other one requests repeatedly one page. Thus, sP_A^B faults on every request of $p-1$ processors. An optimal partition allocates enough cache for all processors and hence it faults a constant number of times. The result for any algorithm A follows by Lemma 1.

Observe that the optimal offline static partition for any given paging strategy A can be computed in time $O(p(nK + K^2))$ using dynamic programming (see Appendix D). For the special case when miss rate functions are convex faster algorithms are known [21].

5.2 Shared cache is better than static partition

Table 1 shows that no static partition strategy is competitive against shared strategies. It is to be expected that a shared strategy should perform better when serving non-disjoint requests. We show here that the same holds even when the intersection between processor requests is empty. The following theorem shows that a static partition strategy, even with an optimal partition and the optimal offline eviction policy, is not competitive against S_{LRU}.

[3]See Appendix B for full proofs.

	1	2	3	4	5	6
A_i/A_j	sP^B_{LRU}	sP^{OPT}_{LRU}	sP^B_A	sP^{OPT}_A	sP^B_{OPT}	sP^{OPT}_{OPT}
sP^B_{LRU}	1	$\Omega(n)$	$\max\{k_j\}$	$\Omega(n)$	$\max\{k_j\}$	$\Omega(n)$
sP^{OPT}_{LRU}	1	1	K,$\max\{k_j\}$	K,$\max\{k_j\}$	K,$\max\{k_j\}$	K,$\max\{k_j\}$
sP^B_A	1,$\max\{k_j\}$	$\Omega(n)$	1	$\Omega(n)$	$\max\{k_j\}$	$\Omega(n)$
sP^{OPT}_A	1,$\max\{k_j\}$	1,$\max\{k_j\}$	1	1	K,$\max\{k_j\}$	K,$\max\{k_j\}$
sP^B_{OPT}	1	$\Omega(n)$	1	$\Omega(n)$	1	$\Omega(n)$
sP^{OPT}_{OPT}	1	1	1	1	1	1
S_{LRU}	$\Omega(p)$,K	$\Omega(p)$,K	$K-p^2+p$, K	$K-p^2+p$, K	$K-p^2+p$, K	$K-p^2+p$, K
S_A	$\Omega(p)$,K	$\Omega(p)$,K	$\Omega(p)$,K	$\Omega(p)$,K	$K-p^2+p$, K	$K-p^2+p$, K
S_{OPT}	1	1	1	1	1	1
dP^D_A	$\omega(1)^d$	$\omega(1)^d$	$\omega(1)^d$	$\omega(1)^d$	$\omega(1)^d$	$\omega(1)^d$
dP^D_{OPT}	$\omega(1)^d$	$\omega(1)^d$	$\omega(1)^d$	$\omega(1)^d$	$\omega(1)^d$	$\omega(1)^d$
dP^D_{LRU}	$\omega(1)^d$	$\omega(1)^d$	$\omega(1)^d$	$\omega(1)^d$	$\omega(1)^d$	$\omega(1)^d$

	7	8	9	10	11	12
A_i/A_j	S_{LRU}	S_A	S_{OPT}	dP^D_A	dP^D_{OPT}	dP^D_{LRU}
sP^B_{LRU}	$\Omega(n)$	$\Omega(n)$	$\Omega(n)$	$\Omega(n)$	$\Omega(n)$	$\Omega(n)$
sP^{OPT}_{LRU}	$\Omega(n)$	$\Omega(n)$	$\Omega(n)$	$\Omega(n)$	$\Omega(n)$	$\Omega(n)$
sP^B_A	$\Omega(n)$	$\Omega(n)$	$\Omega(n)$	$\Omega(n)$	$\Omega(n)$	$\Omega(n)$
sP^{OPT}_A	$\Omega(n)$	$\Omega(n)$	$\Omega(n)$	$\Omega(n)$	$\Omega(n)$	$\Omega(n)$
sP^B_{OPT}	$\Omega(n)$	$\Omega(n)$	$\Omega(n)$	$\Omega(n)$	$\Omega(n)$	$\Omega(n)$
sP^{OPT}_{OPT}	$\Omega(n)$	$\Omega(n)$	$\Omega(n)$	$\Omega(n)$	$\Omega(n)$	$\Omega(n)$
S_{LRU}	1	$\Omega(p(\tau+1))$, K^a	$\Omega(p(\tau+1))$, K^a	f_1, K^a	f_1, K^a	$\Omega(p(\tau+1))$, K^a
S_A	1,K^a	1	$\Omega(p(\tau+1))$, K^a	$\Omega(p(\tau+1))$, K^a	f_1, K^a	$\Omega(p(\tau+1))$, K^a
S_{OPT}	1	1	1	1	1	1
dP^D_A	$\Omega(n)^b$	$\Omega(n)^b$	$\Omega(n)^b$	1	K/p^c,$2K^c$	1,$2K^c$
dP^D_{OPT}	$\Omega(n)^b$	$\Omega(n)^b$	$\Omega(n)^b$	1	1	1
dP^D_{LRU}	$\Omega(n)^b$	$\Omega(n)^b$	$\Omega(n)^b$	K/p^c,$2K^c$	K/p^c,$2K^c$	1

Table 1: Comparison of performances of cache strategies. Each entry $T[i,j]$ shows a lower bound and upper bound for the ratio $Alg_i(\mathcal{R})/Alg_j(\mathcal{R})$, where $Alg(\mathcal{R})$ is the number of faults of Alg on $\mathcal{R}$. $f_1 = \Omega(p(\tau+1)(K-p^2+p))$. *a*: for $\tau = 0$. *b*: for non-disjoint sequences. For disjoint sequences, lower bound of $w(1)$ applies with restriction *d*. *c*: for stable partitions. *d*: for partitions with sublinear number of stages.

Theorem 1 (Static partition is not competitive against a shared strategy) *There exists a sequence $\mathcal{R}$ such that $sP^{OPT}_{OPT}(\mathcal{R})/S_{LRU}(\mathcal{R}) = \Omega(n)$.*

Proof sketch[3]: Given any static partition (even the optimal offline), there is always a sequence that can demand more than the given part for each processor at different times, while a shared strategy can use the entire cache to serve each sequence.

A static partition strategy can be arbitrarily bad compared to a shared one. On the other hand, there are sequences for which a static partition can perform better than a shared one. However, the number of faults of a shared strategy can be no worse than K times the number of faults of a

static one. Theorem 2 and Lemma 3 show this.

Theorem 2 (Shared cache can be worse than static partition) *Let A be any online eviction policy and A' be any marking or conservative eviction policy. $\exists \mathcal{R}$ such that $\frac{S_A(\mathcal{R})}{sP_{A'}^{OPT}(\mathcal{R})} = \Omega(p)$.*

Proof sketch[3]: The adversarial sequence in this proof consists of a sequence that makes S_A fault in every request, while at the same time allows a partition such that $sP_{A'}^{OPT}$ will fault only on the requests of one processor. If $A = LRU$ it is enough to request pages $(\sigma_1^j \dots \sigma_{K/p+1}^j)$ for each processor j, hence OPT is such that the requests of all processors but one fit in their part of the cache. For an arbitrary online eviction policy we need to be more careful to ensure a suitable partition.

Lemma 3 (Shared cache with LRU vs optimal static partition) *For all requests $\mathcal{R}$ $S_{LRU}(\mathcal{R})/sP_{OPT}^{OPT}(\mathcal{R}) \leq K$.*

Proof sketch[3]: Consider the phases of each sequence R_j (each phase has at most k_j different pages), and the shared K-phases of $\mathcal{R}$. Since a shared phase cannot start and end without the sequence of some processor changing phases, there are no more shared phases than phases in the sequences themselves.

5.3 Shared strategies

In this section we show that the lower bound of the competitive ratio of τ of a shared LRU strategy shown by Hassidim [14] also holds in our model, although the optimal offline strategy does not have the ability of scheduling the requests. Nevertheless, an offline strategy can effectively delay the sequence of one processor and serve the rest of the sequences having enough cache space. The larger τ, the more a sequence can be delayed and hence an offline strategy can spend more time serving only $p-1$ sequences. Note that if $\tau = \Omega(n)$, then the competitive ratio of S_{LRU} can be arbitrarily large. On the other hand, if $\tau = 0$, the competitive ratio of S_{LRU} is K, as in the traditional sequential setting.

Theorem 3 (Lower bound for the competitive ratio of online shared strategies) *Let A be an online deterministic cache eviction policy. There exists a sequence $\mathcal{R}$ and an offline eviction policy OFF such that $S_A(\mathcal{R})/S_{OFF}(\mathcal{R}) = \Omega(p(\tau+1))$.*

Proof sketch[3]: The adversary requests a sequence such that S_A faults on every request. The offline strategy S_{OFF} forces a fault on one of the sequences, so that the pages of the rest of the sequences fit in the cache, and S_{OFF} faults once every $(\tau+1)$ requests on the delayed sequence.

The proof of Theorem 3 confirms that if $\tau = \omega(K/p)$, Furthest-In-The-Future (FITF) is not an optimal strategy in our model: it is not hard to verify if serving the request in said proof, a shared strategy that were to evict the page that is furthest in the future would make n/K faults. Eviction policy OFF in the proof makes $O(n/p(\tau+1))$, and it is different from FITF. Note that FITF is not the optimal strategy in Hassidim's model either. In fact, it is shown in [14] that the optimal offline schedule is NP-complete.

5.4 Dynamic partitions

In this section we discuss results related to partitions that can change over time. When comparing strategies with different eviction policies but that have the same partition, we obtain bounds that are analogous as those for strategies with the same static partition (see Lemma 1). A dynamic partition strategy with a deterministic online eviction policy can be at least K/p times worse than one with an optimal offline eviction policy, but no more than $2K$ times worse.

Lemma 4 (Online vs offline eviction policies: lower bound) *Let A be any online cache eviction algorithm and let $B = B(t)$ be any stable dynamic partition (see Definition 1). Assume $k_j \geq p$, $\forall\ 1 \leq j \leq p\ \forall t$. There exists a sequence R such that $dP_A^B(R)/dP_{OPT}^B(R) \geq K/p$.*

Lemma 5 (Online vs offline eviction policies: upper bound) *Let B be any stable dynamic partition strategy. For all sequences R, $dP_{LRU}^B(R)/dP_{OPT}^B(R) \leq 2K$.*

Proof sketch[3]: The proofs for both lemmas work essentially by applying the arguments of the proof of Lemma 1 to each stage of the dynamic partition.

As expected, a strategy that utilizes a dynamic partition can be more powerful than one which must respect the initial partition. However, these can still perform badly when compared to shared strategies. As we see in Table 1, the ratio between the number of faults of any dynamic partition strategy to static partition strategies or shared strategies can be non-constant. In fact, when considering non-disjoint requests, it should be no surprise that a dynamic partition strategy is not competitive against a shared strategy, as shown in the following lemma:

Lemma 6 (Dynamic partition is not competitive for non-disjoint requests) *Let A be any eviction policy and B be any dynamic partition strategy. $\exists \mathcal{R}$ s.t. $dP_A^B(\mathcal{R})/S_{LRU}(\mathcal{R}) = \Omega(n/K^2p)$.*

Proof sketch[3]: The adversary requests the same pages for all processors so that all pages fit in the cache of S_{LRU} but not in the cache of dP_A^B, since pages are different in each part for this algorithm. Regardless of the partition, dP_A^B must fault at least once every K requests.

Even if we restrict ourselves to disjoint requests, if a dynamic partition does not change the partition enough, it is subject to the same adversarial sequences than static partitions. To illustrate this, we show that the number of stages of a dynamic strategy is sublinear (w.r.t the length of the sequences), then such strategy is not competitive against shared LRU:

Lemma 7 (Dynamic partitions that do not change enough are not competitive) *Let A be any deterministic online cache eviction policy, and let B be any online dynamic partition whose number of stages is $o(n)$. There exists a sequence $\mathcal{R}$ such that $dP_A^B(\mathcal{R})/S_{LRU}(\mathcal{R}) = \omega(1)$.*

Proof sketch[3]: Since the number of stages is $o(n)$, at least one stage of the partition has non-constant length. Then, we can apply the same argument as in the proof of Theorem 1 for static partitions in this stage. Each processor requests more pages than the size of its part in this stage, but at different times, allowing a shared strategy to use all the cache to serve the requests of each processor.

The argument of Lemma 7 can be extended to show that such partitions are not competitive against static partitions as well, as shown in Table 1. Moreover, if the number of stages is constant, then the ratio becomes arbitrarily large. However, when considering general dynamic partitions and if sequences are disjoint, we cannot hope to show an equivalent result to that of Theorem 1, since there exists a dynamic partition that behaves exactly like S_{LRU}, as shown in Theorem 4:

Theorem 4 (Dynamic partitions equal shared strategies for disjoint sequences) *There exists a dynamic partition B such that for all disjoint requests $\mathcal{R}, dP^B_{LRU}(\mathcal{R}) = S_{LRU}(\mathcal{R})$.*

Proof sketch[3]: dP^B_{LRU} can emulate the behaviour of S_{LRU} by reducing the partition of the processor whose least recently used page is the least recently used among the pages in all parts of the cache. The reduction implies the eviction of the overall least recently used page.

Theorem 4 implies that any static partition can perform arbitrarily badly when compared to dynamic partitions in the worst case, as these can simulate S_{LRU}, and we have already shown that static partitions are not competitive against S_{LRU}. Not only can a dynamic partition have the same performance that a shared strategy, but it can also perform better:

Lemma 8 (Shared cache with LRU can be worse than dynamic partition) *Let A be any online eviction policy. $\exists \mathcal{R}$ such that $\frac{S_A(\mathcal{R})}{dP^{OPT}_{LRU}(\mathcal{R})} = \Omega(p(\tau+1))$.*

Proof sketch[3]: As in Theorem 2, there exists a sequence that makes S_A fault on every request but that allows a good partition to serve $p-1$ requests with no faults. As the partition is dynamic, as soon as these requests are served, the partition can change to give enough cache to the other processor.

Finally, we show that no dynamic or static partition strategy can outperform S_{OPT}, since the latter can emulate any strategy, as the following lemma shows:

Lemma 9 (Offline shared is no worse than dynamic partition) *For all sequences R, $S_{OPT}(R) \leq dP^{OPT}_{OPT}(R)$.*

Proof sketch[3]: An offline shared strategy can compute the optimal dynamic partition and emulate dP^{OPT}_{OPT} by maintaining a logical division of the cache, making decisions depending on the number of pages of each processor residing in the cache at any time and the sizes of the parts of each processor at that time.

6 Conclusions and Future Work

We have presented a thorough comparison between paging algorithms for multicore caches. We have shown that statically partitioning the cache is not competitive against sharing the cache, even when requests of different processors are disjoint. As expected, we show that shared strategies outperform dynamic partition strategies when requests from different processors share pages. However, if requests are disjoint, a dynamic partition can emulate any shared strategy with an online eviction policy.

The fact that sequences may be delayed by a factor τ plays a key role in the multicore caching problem. Delaying some sequences can affect the performance of an algorithm negatively or positively, and sometimes it can be preferable for an algorithm to deliberately incur in faults to delay some sequences, and thus use the cache to serve the rest of the sequences. We have shown this effect when showing, for example, that the competitive ratio of shared LRU is bounded below by the fetching time. In addition, delaying sequences changes the alignments of remaining pages of different processors to the future, which, depending on the contents of the sequences, can increase or decrease the length of phases. A possible direction of research is to study the effect of delaying sequences in the performance of algorithms. In particular, an important open question is to determine an upper bound on the competitive ratio of shared LRU.

Other lines of future research include determining the optimal offline shared strategy, as well as competitive online shared strategies and dynamic partition strategies.

References

[1] A. Agarwal. Performance tradeoffs in multithreaded processors. *IEEE Trans. Parallel Distrib. Syst.*, 3(5):525–539, 1992.

[2] C. Bienia, S. Kumar, J. P. Singh, and K. Li. The parsec benchmark suite: characterization and architectural implications. In *PACT '08: Proceedings of the 17th international conference on Parallel architectures and compilation techniques*, pages 72–81, New York, NY, USA, 2008. ACM.

[3] G. E. Blelloch, R. A. Chowdhury, P. B. Gibbons, V. Ramachandrana, S. Chen, and M. Kozuch. Provably good multicore cache performance for divide-and-conquer algorithms. In *Proceedings of the 2008 ACM-SIAM Symposium on Discrete Algorithms*, January 2008.

[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. Brief announcement: low depth cache-oblivious sorting. In *SPAA '09: Proceedings of the twenty-first annual symposium on Parallelism in algorithms and architectures*, pages 121–123, New York, NY, USA, 2009. ACM.

[6] A. Borodin and R. El-Yaniv. *Online computation and competitive analysis*. Cambridge University Press, New York, NY, USA, 1998.

[7] 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.

[8] S. Cho and L. Jin. Managing distributed, shared l2 caches through os-level page allocation. In *MICRO 39: Proceedings of the 39th Annual IEEE/ACM International Symposium on Microarchitecture*, pages 455–468, Washington, DC, USA, 2006. IEEE Computer Society.

[9] R. Dorrigiv and A. López-Ortiz. A survey of performance measures for on-line algorithms. *SIGACT News*, 36(3):67–81, 2005.

[10] R. Dorrigiv and A. López-Ortiz. On developing new models, with paging as a case study. *SIGACT News*, 40(4):98–123, 2009.

[11] R. Dorrigiv, A. López-Ortiz, and J. I. Munro. On the relative dominance of paging algorithms. *Theor. Comput. Sci.*, 410(38-40):3694–3701, 2009.

[12] R. Fedorova, M. Seltzer, and M. D. Smith. Cache-fair thread scheduling for multicore processors. Technical report, Harvard University, 2006.

[13] F. Guo and Y. Solihin. An analytical model for cache replacement policy performance. In *SIGMETRICS '06/Performance '06: Proceedings of the joint international conference on Measurement and modeling of computer systems*, pages 228–239, New York, NY, USA, 2006. ACM.

[14] A. Hassidim. Cache replacement policies for multicore processors. In *Proceedings of The First Symposium on Innovations in Computer Science*. Tsinghua University Press, 2010.

[15] A. Jaleel, W. Hasenplaugh, M. Qureshi, J. Sebot, S. Steely, Jr., and J. Emer. Adaptive insertion policies for managing shared caches. In *PACT '08: Proceedings of the 17th international conference on Parallel architectures and compilation techniques*, pages 208–219, New York, NY, USA, 2008. ACM.

[16] J. Lin, Q. Lu, X. Ding, Z. Zhang, X. Zhang, and P. Sadayappan. Gaining insights into multicore cache partitioning: Bridging the gap between simulation and real systems. In *INTERNATIONAL SYMPOSIUM ON HIGH PERFORMANCE COMPUTER ARCHITECTURE*, pages 367–378, 2008.

[17] 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.

[18] 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.

[19] A. Settle, D. Connors, E. Gibert, and A. González. A dynamically reconfigurable cache for multithreaded processors. *J. Embedded Comput.*, 2(2):221–233, 2006.

[20] S. Srikantaiah, M. Kandemir, and Q. Wang. Sharp control: controlled shared cache management in chip multiprocessors. In *MICRO 42: Proceedings of the 42nd Annual IEEE/ACM International Symposium on Microarchitecture*, pages 517–528, New York, NY, USA, 2009. ACM.

[21] H. Stone, J. Turek, and J. Wolf. Optimal partitioning of cache memory. *IEEE Transactions on Computers*, 41:1054–1068, 1992.

[22] G. E. Suh, L. Rudolph, and S. Devadas. Dynamic cache partitioning for simultaneous multithreading systems. In *Proceedings of the IASTED International Conference on Parallel and Distributed Computing and Systems*, pages 116–127, 2001.

[23] D. Tam, R. Azimi, L. Soares, and M. Stumm. Managing shared l2 caches on multicore systems in software. In *Workshop on the Interaction between Operating Systems and Computer Architecture*, 2007.

[24] 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.

[25] E. Z. Zhang, Y. Jiang, and X. Shen. Does cache sharing on modern cmp matter to the performance of contemporary multithreaded programs? In R. Govindarajan, D. A. Padua, and M. W. Hall, editors, *PPOPP*, pages 203–212. ACM, 2010.

A Example of execution of shared and static partition strategies

Consider the following sequence $\mathcal{R} = \{R_1, R_2\}$, with $R_1 = (\sigma_1, 1), (\sigma_2, 2), (\sigma_4, 3), (\sigma_2, 4), (\sigma_3, 5), (\sigma_4, 6)$, and $R_2 = (\sigma_5, 1), (\sigma_6, 2), (\sigma_2, 3), (\sigma_4, 4), (\sigma_5, 5), (\sigma_5, 7)$. Let $K = 5$ and let B be a static partition such that $k_1 = 3$ and $k_2 = 2$. Table 2 shows the execution of S_{LRU} and sP^B_{LRU} on $\mathcal{R}$ with $\tau = 2$, starting with a populated cache. The total time (work) of execution of S_{LRU} is $T_{S_{LRU}}(\mathcal{R}) = 10 + 8 = 18$, the makespan is 10, and the number of faults $S_{LRU}(\mathcal{R}) = 3$. For static partition we have $T_{sP^B_{LRU}}(\mathcal{R}) = 10 + 13 = 23$, the makespan is 12 and the number of faults is $sP^B_{LRU}(\mathcal{R}) = 5$.

B Proofs of Lemmas

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 $sP^B_A(\mathcal{R})/sP^B_{OPT}(\mathcal{R}) \geq \max_j\{k_j\}$. When A is any marking or conservative algorithm (e.g. LRU), there is a matching upper bound, i.e., $\forall\ \mathcal{R}$, $sP^B_A(\mathcal{R})/sP^B_{OPT}(\mathcal{R}) \leq \max_j\{k_j\}$, which also holds if B is offline.*

Proof: [lower bound] Let $j^* = argmax_j\{k_j\}$. The sequence $\mathcal{R}$ is such that no pages are requested for R_j with $j \neq j^*$, 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 $i_1 \neq i_2$. Naturally, $sP^B_A(\mathcal{R}) = n/p$. On the other hand, since sP^B_{OPT} only evicts a page of sequence R_{j^*} if it is not requested in the following k_{j^*} requests, we have $sP^B_{OPT}(\mathcal{R}) \leq (n/p)/k_{j^*}$ and the lemma follows. □

Proof: [upper bound] Given the sequence R_j of processor j, divide it 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

S_{LRU}			sP^B_{LRU}	
t	Remaining sequence	Cache	Remaining sequence	Cache
0	$\sigma_1, \sigma_2, \sigma_4, \sigma_2, \sigma_3, \sigma_4$ $\sigma_5, \sigma_6, \sigma_2, \sigma_4, \sigma_5, \perp, \sigma_5$	$\sigma_1\sigma_2\sigma_3\sigma_5\sigma_6$	$\sigma_1, \sigma_2, \sigma_4, \sigma_2, \sigma_3, \sigma_4$ $\sigma_5, \sigma_6, \sigma_2, \sigma_4, \sigma_5, \perp, \sigma_5$	$\sigma_1\sigma_2\sigma_3$ $\sigma_5\sigma_6$
1	$\sigma_2, \sigma_4, \sigma_2, \sigma_3, \sigma_4$ $\sigma_6, \sigma_2, \sigma_4, \sigma_5, \perp, \sigma_5$	$\sigma_5\sigma_1\sigma_2\sigma_3\sigma_6$	$\sigma_2, \sigma_4, \sigma_2, \sigma_3, \sigma_4$ $\sigma_6, \sigma_2, \sigma_4, \sigma_5, \perp, \sigma_5$	$\sigma_1\sigma_2\sigma_3$ $\sigma_5\sigma_6$
2	$\underline{\sigma_4}, \sigma_2, \sigma_3, \sigma_4$ $\sigma_2, \sigma_4, \sigma_5, \perp, \sigma_5$	$\sigma_6\sigma_2\sigma_5\sigma_1\sigma_3$	$\underline{\sigma_4}, \sigma_2, \sigma_3, \sigma_4$ $\underline{\sigma_2}, \sigma_4, \sigma_5, \perp, \sigma_5$	$\sigma_2\sigma_1\sigma_3$ $\sigma_6\sigma_5$
3	$\perp, \perp, \sigma_2, \sigma_3, \sigma_4$ $\underline{\sigma_4}, \sigma_5, \perp, \sigma_5$	$\sigma_2\sigma_6\sigma_5\sigma_1\perp$	$\perp, \perp, \sigma_2, \sigma_3, \sigma_4$ $\perp, \perp, \sigma_4, \sigma_5, \perp, \sigma_5$	$\sigma_2\sigma_1\perp$ $\sigma_6\sigma_5$
4	$\perp, \sigma_2, \sigma_3, \sigma_4$ $\perp, \perp, \sigma_5, \perp, \sigma_5$	$\sigma_2\sigma_6\sigma_5\sigma_1\perp$	$\perp, \sigma_2, \sigma_3, \sigma_4$ $\perp, \sigma_4, \sigma_5, \perp, \sigma_5$	$\sigma_2\sigma_1\perp$ $\sigma_6\sigma_5$
5	$\sigma_2, \sigma_3, \sigma_4$ $\sigma_5, \perp, \sigma_5$	$\sigma_4\sigma_2\sigma_6\sigma_5\sigma_1$	$\sigma_2, \sigma_3, \sigma_4$ $\underline{\sigma_4}, \sigma_5, \perp, \sigma_5$	$\sigma_4\sigma_2\sigma_1$ $\sigma_6\sigma_5$
6	$\underline{\sigma_3}, \sigma_4$ $\perp, \sigma_5$	$\sigma_5\sigma_2\sigma_4\sigma_6\sigma_1$	$\underline{\sigma_3}, \sigma_4$ $\perp, \perp, \sigma_5, \perp, \sigma_5$	$\sigma_2\sigma_4\sigma_1$ $\sigma_6\perp$
7	$\perp, \perp, \sigma_4$ σ_5	$\sigma_5\sigma_2\sigma_4\sigma_6\perp$	$\perp, \perp, \sigma_4$ $\perp, \sigma_5, \perp, \sigma_5$	$\sigma_2\sigma_4\perp$ $\sigma_6\perp$
8	$\perp, \sigma_4$	$\sigma_5\sigma_2\sigma_4\sigma_6\perp$	$\perp, \sigma_4$ $\underline{\sigma_5}, \perp, \sigma_5$	$\sigma_2\sigma_4\perp$ $\sigma_4\sigma_6$
9	σ_4	$\sigma_3\sigma_5\sigma_2\sigma_4\sigma_6$	σ_4 $\perp, \perp, \perp, \sigma_5$	$\sigma_3\sigma_2\sigma_4$ $\sigma_4\perp$
10		$\sigma_4\sigma_3\sigma_5\sigma_2\sigma_6$	$\perp, \perp, \sigma_5$	$\sigma_4\sigma_3\sigma_2$ $\sigma_4\perp$
11			$\perp, \sigma_5$	$\sigma_4\sigma_3\sigma_2$ $\sigma_5\sigma_4$
12			σ_5	$\sigma_4\sigma_3\sigma_2$ $\sigma_5\sigma_4$
13				$\sigma_4\sigma_3\sigma_2$ $\sigma_5\sigma_4$

Table 2: Example of execution of shared LRU with $K = 5$, and static partition with LRU and partition $\{3, 2\}$. $\tau = 2$ in this example. Pages in the caches are shown from left to right in order of most recent use, and an empty page ($\perp$) in the cache indicates that the cell will be used by a page currently being fetched. Underlined pages denote faults.

previous phase, and the first phase begins at the first page of R_j. sP^B_{LRU} 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 $sP^B_{LRU}(\mathcal{R}) \leq \sum_{j=1}^{p} \phi_j k_j \leq \max_j\{k_j\} \sum_{j=1}^{p} \phi_j$. On the other hand, $sP^B_{OPT}(\mathcal{R}) \geq \sum_{j=1}^{p} \phi_j$, and thus $sP^B_{LRU}(\mathcal{R})/sP^B_{OPT}(\mathcal{R}) \leq \max_j\{k_j\}$. □

Lemma 2 (Online static partition is not competitive against an offline static partition) *Let $B = \{k_1, ..., k_p\}$ be any static online partition. Let $j^* = argmin_j\{k_j | k_j \geq 2\}$. There exist a sequence $\mathcal{R}$, such that for all A, $sP^B_A(\mathcal{R})/sP^{OPT}_{LRU}(\mathcal{R}) \geq \min\{k_{j^*}, p-1\}\frac{n}{K^2 p} = \Omega(n)$.*

Proof: Consider first $A = LRU$. 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 $P' = P \setminus \{j^*\}$. Let $R_j = (\sigma^j_1 \sigma^j_2 ... \sigma^j_{k_j+1})^{x_j}$ with x_j such that $x_j(k_j + 1) = n/p$ for all $j \in P'$, and let $R_j = (\sigma^j_1 \sigma^j_2 ... \sigma^j_{k_j})^{x_j}$ $j \notin P'$ 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^B_{LRU} faults on every request of $|P'|$ processors and faults only on the first request of processor j^*. Hence, $sP^B_{LRU}(\mathcal{R}) \geq \min\{k_{j^*}, (p-1)\}n/p$.

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^{OPT}_{LRU} on $\mathcal{R}$ is K, since it only faults on the first request to each different page. Hence $sP^B_{LRU}(\mathcal{R})/sP^{OPT}_{LRU}(\mathcal{R}) \geq \min\{k_{j^*}, (p-1)\}n/Kp = \Omega(n)$. Now, by Lemma 1 $sP^B_{OPT}(\mathcal{R}) \geq sP^B_{LRU}(\mathcal{R})/K$, and the lemma follows. □

Theorem 1 (Static partition is not competitive against a shared strategy) *There exists a sequence $\mathcal{R}$ such that $sP^{OPT}_{OPT}(\mathcal{R})/S_{LRU}(\mathcal{R}) = \Omega(n)$.*

Proof: 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 phase*), 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 phase: when one processor is in the distinct phase, 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 phase 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 phase 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 $x/(pk_{j^*}) = \Omega(n)$. □

Theorem 2 (Shared cache can be worse than static partition) *Let A be any online eviction policy and A' be any marking or conservative eviction policy. $\exists\mathcal{R}$ such that $\frac{S_A(\mathcal{R})}{sP^{OPT}_{A'}(\mathcal{R})} = \Omega(p)$.*

Proof: Let B an offline static partition strategy. We will use an adversarial sequence that makes S_A fault on every request, while at the same is suitable for a partition that allows restricting the faults incurred by $sP^B_{A'}$ exclusively to the requests of one processor (plus a constant number of faults for the first pages of other processors). The sequence $\mathcal{R}$ that the adversary uses is the following. Each processor j will request pages among $\{\sigma_1^j, \sigma_2^j, \ldots, \sigma_{K/p+1}^j\}$, where the set of pages is disjoint among processors. The adversary keeps a cyclic order π of processors ordered by, for example, increasing processor id. While no evictions are made by S_A the adversary requests, in turn for each processor according to π, the page that is not in the cache. Whenever S_A evicts a page σ_i^j, the adversary makes processor j request the page among $\{\sigma_1^j, \ldots, \sigma_{K/p+1}^j\}$ that is not in the cache. This might require the introduction of empty requests among the requests of the rest of the processors

so that the next page requested effectively corresponds to processor j. If again there is no eviction, $\mathcal{R}$ requests a page corresponding to the next processor according to π.

Let C_j be the number of pages in the cache of S_A that belong to processor j. For the request $\mathcal{R}$ to be plausible, it must be the case that at all times $C_j \leq K/p$ for all j (otherwise $\mathcal{R}$ would not be able to request the page not in the cache for some processor). Initially $C_j = 0$ for all j. If upon a request A does not make an eviction, then C_j is increased by one, where j is the next processor according to π. If A evicts a page from processor j, then $\mathcal{R}$ requests a page from processor j and C_j remains the same. It is not hard to see that if $\mathcal{R}$ follows the rules for requests described above, when the cache becomes full then $C_j = K/p$ for all j. After the cache is full, S_A incurs in a fault that requires eviction for all subsequent requests, and hence the value of C_j remains equal to K/p until the end of the request. Hence $\mathcal{R}$ is plausible and each processor does not request more than $K/p + 1$ different pages.

Let m_j be the total number of (non-empty) pages requested by processor j in $\mathcal{R}$ (recall that n counts the empty pages as well), and let $M = \sum_{j=1}^{p} m_j$. Since $\mathcal{R}$ always requests a page that is not in the cache, $S_A(\mathcal{R}) = M$. On the other hand, since the number of different pages requested by each processor is at most $K/p+1$, the partition B can assign $K/p+1$ units of cache to $p-1$ processors and the rest to one processor. Let $j^* = argmin_j\{m_j\}$ (j^* is the processor whose sequence ended up with the least number of non-empty requests among all processors). $sP^B_{A'}$ gives $K/p+1$ units of cache to all processors $j \neq j^*$ and gives $K/p-(p-1)$ to j^*. The total number of faults of $sP^B_{A'}$ is at most $m_{j^*} + (p-1)(K/p+1) \leq M/p + (p-1)(K/p+1)$ (recall that A' is marking or conservative). Since M can be made arbitrarily large, we ignore constant additive factors and since $sP^{OPT}_{A'}(\mathcal{R}) \leq sP^B_{A'}(\mathcal{R})$ for all B and $\mathcal{R}$, it follows that $S_A(\mathcal{R})/sP^{OPT}_{A'}(\mathcal{R}) = \Omega(p)$. □

Lemma 3 (Shared cache with LRU vs optimal static partition) *For all requests* $\mathcal{R}$ $S_{LRU}(\mathcal{R})/sP^{OPT}_{OPT}(\mathcal{R}) \leq K$.

Proof: 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 $S_{LRU}(\mathcal{R}) \leq K\phi \leq K\sum_{j=1}^{p} \phi_j \leq K sP^{OPT}_{OPT}(\mathcal{R})$.

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. □

Theorem 3 (Lower bound for the competitive ratio of online shared strategies) *Let A be an online deterministic cache eviction policy. There exists a sequence $\mathcal{R}$ and an offline eviction policy OFF such that $S_A(\mathcal{R})/S_{OFF}(\mathcal{R}) = \Omega(p(\tau+1))$.*

Proof: Let us first consider $A = LRU$. Consider a disjoint sequence $\mathcal{R}$ such that 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. Consider an algorithm OFF that, after the first $K/p+1$ parallel requests (all faults), forces a fault only on the request of one processor, say p. In other words, for all processors $j = 1, \dots, p-1$, OFF evicts page $\sigma_{K/p}^j$ when serving $\sigma_{K/p+1}^j$, but for processor p, OFF evicts σ_1^p. Hence the second request for σ_1^j will be a hit for all processors $j = 1, \dots, p-1$, and a fault for $j = p$. The sequence R_p gets delayed by τ, while all other continue to be served. Because of the delay of sequence R_p, all pages currently in the cache that belong R_p will be replaced by pages of other sequences. 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. Since all pages of R_p will be a fault and this sequence will get further delayed, eventually (after only one more fault per sequences) all pages of sequences R_j, $j \neq p$ will be in the cache and OFF will incur in no more faults on these sequences. The total number of faults on the sequences $R_1, \dots, R_{p-1}$ will then be $(p-1)(K/p+2)$. On the other hand, S_{OFF} will fault on every request of sequence R_p while other sequences are being served, and since there is space in S_{OFF}'s cache to store one page of R_p, S_{OFF} does not have to evict any page of the other sequences. Once the other sequences are completely served, the rest of R_p will be served with all the cache for this purpose. 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 faults of S_A is then $(p-1)(K/p+2) + 2(K/p+1) + (n/p - K/p - 1)/(\tau+1) = O(n/p(\tau+1))$, and hence $S_{LRU}(\mathcal{R})/S_{OFF}(\mathcal{R}) = \Omega(p(\tau+1))$ Now, the argument holds for any deterministic online eviction policy A by modifying the sequences so that the next page requested is the one currently not in the cache, and modifying the offline algorithm OFF accordingly. □

Lemma 4 (Online vs offline eviction policies: lower bound) *Let A be any online cache eviction algorithm and let $B = B(t)$ be any stable dynamic partition (see Definition 1). Assume $k_j \geq p$, $\forall\ 1 \leq j \leq p\ \forall t$. There exists a sequence R such that $dP_A^B(R)/dP_{OPT}^B(R) \geq K/p$.*

Proof: We divide $\mathcal{R}$ in stages (see Section 4.1) and apply the same argument as in Lemma 1. Let $n = |\mathcal{R}|$ and n_j^s be the number of pages requested in R_j in the s-th stage and k_j^s be the size of the cache of processor j in this stage. We set the length of the sequence of each processor to $n_j = n/p$ and thus $n_{j_1}^s = n_{j_2}^s$ for all j_1, j_2.

As before, the sequence of each processor requests the page just evicted by A. As in the static case, $dP_A^B(\mathcal{R}) = n$. Now, for each stage s, we have $dP_{OPT}^B(R_j) \leq n_j^s/k_j^s$. This follows from the fact that in this stage, OPT evicts a page only if it will not be requested in the next k_j^s requests, and also from the fact that B is stable and hence none of the pages evicted in a possible reduction of cache size between stages will be among the first k_j^s requests of the stage. Let $n_j^s = n^s$ for stage s and all processors j, then $dP_{OPT}^B(\mathcal{R}) \leq \sum_{j=1}^p \sum_s n_j^s/k_j^s = \sum_s n_s \sum_{j=1}^p 1/k_j^s$. Again, since $k_j^s > p$, for all s and j, $\sum_s n_s \sum_{j=1}^p 1/k_j^s \leq \sum_s n_s p^2/K$, and since $\sum_s n_s = n/p$, it follows that $dP_{OPT}^B(\mathcal{R}) \leq np/K$ and the lemma follows. □

Lemma 5 (Online vs offline eviction policies: upper bound) *Let B be any stable dynamic partition strategy. For all sequences R, $dP_{LRU}^B(R)/dP_{OPT}^B(R) \leq 2K$.*

Proof: Similarly to the case of a static partition in the proof of the upper bound in Lemma 1, divide each stage of B in phases. Let ϕ_j^s the number of phases of R_j in stage s, and let $\phi = \sum_s \sum_{j=1}^p \phi_j^s$. Let k_j^s be size of the cache of processor j in phase s. The number of faults of dP_{LRU}^B on the sequence R_j in

stage s is at most k_j^s per phase, and hence $dP^B_{LRU}(\mathcal{R}) \leq \sum_{j=1}^{p} \sum_s \phi_j^s k_j^s \leq \sum_s \bar{\phi}^s \sum_{j=1}^{p} k_j^s = \sum_s \bar{\phi}^s K$, where $\bar{\phi}^s = max_j\{\phi_j^s\}$. On the other hand, dP^B_{OPT} faults at least once per phase of a stage, except possibly for the first phase. Thus, $dP^B_{OPT}(\mathcal{R}) \geq \sum_s \sum_{j=1}^{p} \phi_j^s - 1 \geq \phi/2$, since B is stable and hence each stage has at least two phases for each sequence. Since $\sum_s \bar{\phi}^s \leq \phi$, it follows that $dP^B_{LRU}(\mathcal{R}) \leq \phi K \leq 2KsP^B_{OPT}(\mathcal{R})$. □

Lemma 6 (Dynamic partition is not competitive for non-disjoint requests) *Let A be any eviction policy and B be any dynamic partition strategy. $\exists \mathcal{R}$ s.t. $dP^B_A(\mathcal{R})/S_{LRU}(\mathcal{R}) = \Omega(n/K^2p)$.*

Proof: Let $\mathcal{R}$ be the sequence in which each processor repeatedly requests pages $(\sigma_1 \ldots \sigma_K)$. Clearly, $S_{LRU}(\mathcal{R}) = K$. On the other hand, each page of a different processor is different for $dP^B_A(\mathcal{R})$, and thus the sequence has Kp different pages. Let us divide the request $\mathcal{R}$ in phases such that each phase has K different pages in total. We will consider only the period when all sequences are alive, i.e., none of the sequences has been entirely consumed. Consider the sequence R_j such that the time to serve this sequence is the minimum among all processors. Let ℓ_i be the number of pages of R_j in phase i of this period. Given the request sequence, $\ell_i \leq K$, and thus $L = \sum_{i=1}^{\phi} \ell_i \leq K\phi$, where ϕ is the number of phases in the period. Since $L = n/p$ it follows that $\phi \geq n/(pK)$. Since any dynamic partition strategy, regardless of the partition and the eviction policy, must fault at least once in a phase, the lemma follows. □

Lemma 7 (Dynamic partitions that do not change enough are not competitive) *Let A be any deterministic online cache eviction policy, and let B be any online dynamic partition whose number of stages is $o(n)$. There exists a sequence $\mathcal{R}$ such that $dP^B_A(\mathcal{R})/S_{LRU}(\mathcal{R}) = \omega(1)$.*

Proof: If the number of stages of B is $o(n)$, then at least one stage has non-constant length $\ell = \omega(1)$ (in number of parallel page requests). Let $\mathcal{R}$ in this stage consist of a sequence in the form of the sequence of Theorem 1: each processor's sequence has three phases: (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 phase*), and (3) again only one page σ_1^j is requested. The length of phase (2) is m pages, and each processor takes turns to be in the distinct phase. 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 phase 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^B_A faults on every request of the distinct phase of all processors, and hence in this stage $dP^B_A(\mathcal{R}) = pm = \ell$. On the other hand, in this stage, S_{LRU} faults only on the first request to a distinct page in the distinct phase 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 $dP^B_A(\mathcal{R})/S_{LRU}(\mathcal{R}) \geq \ell/(K + p) = \omega(1)$. Note that if partitions are allowed only a constant number of stages, then $dP^B_A(\mathcal{R})/S_{LRU}(\mathcal{R}) = \Omega(n)$. □

Theorem 4 (Dynamic partitions equal shared strategies for disjoint sequences) *There exists a dynamic partition B such that for all disjoint requests $\mathcal{R}, dP^B_{LRU}(\mathcal{R}) = S_{LRU}(\mathcal{R})$.*

Proof: Let B be the following strategy. B 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. B 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^B_{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. □

Lemma 8 (Shared cache with LRU can be worse than dynamic partition) *Let A be any online eviction policy. $\exists \mathcal{R}$ such that $\frac{S_A(\mathcal{R})}{dP^{OPT}_{LRU}(\mathcal{R})} = \Omega(p(\tau+1))$.*

Proof: The argument is the same as in Theorem 2: using the request $\mathcal{R}$ in this lemma, $S_A(\mathcal{R})$ faults on every page, and hence $S_A(R) = n$. On the other hand, an offline partition can give $K/p+1$ units of cache to all processors but one, and give the rest to the last processor, say processor p. However, as the partition is dynamic, as soon as the requests of processors 1 to $p-1$ are served, the partition can change to give enough cache to processor p. Hence, dP^{OPT}_{LRU} will fault in the initial $(K/p+1)$ pages of requests $R_1, \ldots, R_{p-1}$, and in the first $(n/p)/(\tau+1) + K/p + 1$ requests of R_p, for a total of $(n/p)(\tau+1) + K + p$ faults. Hence $\frac{S_{LRU}(R)}{dP^{OPT}_{LRU}(R)} = \Omega(p(\tau+1))$. □

Lemma 9 (Offline shared is no worse than dynamic partition) *For all sequences R, $S_{OPT}(R) \leq dP^{OPT}_{OPT}(R)$.*

Proof: [Proof of Lemma 9] Consider an offline strategy A. A computes the optimal dynamic partition. Let k_j^t denote the size of the cache for processor j at time t in this optimal partition. Let C_j be the set of pages corresponding to requests of processor j that reside currently in the cache. S_A emulates dP_{OPT} working as follows: let σ_j be the request of processor j at time t. If σ_j is in the cache, then nothing changes. Otherwise, if $|C_j| < k_j^t$ then σ_j is added to C_j, whereas if $|C_j| = k_j^t$ then a page from C_j is evicted from the cache optimally (minimizing the number of faults overall). If at any point in the execution the optimal dynamic partition requires the size of the cache of processor j to be reduced, i.e. $k_j^{t+1} < k_j^t$, then S_A evicts $k_j^t - k_j^{t+1}$ pages from C_j in the way that minimizes the total number of faults. Hence S_A behaves like dP^{OPT}_{OPT} and thus they make the same number of faults. Since $S_{OPT}(R) \leq S_A(R)$ for all R, the lemma follows. □

C Proofs for all entries of Table 1

We briefly justify all entries of the table, organized by row.

sP^B_{LRU}: Columns 2, 4, and 6 follow from Lemma 2. Columns 3 and 5 follow from Lemma 1. Columns 7 to 9 follow from Theorem 1, while columns 10 to 12 follow from Theorems 4 and 1, with $A = LRU$ for column 10.

sP^{OPT}_{LRU}: Column 1: lower bound for $B = OPT$, upper bound from optimality of the partition. Columns 3 to 6 follow by letting $A = OPT$ and $B = OPT$ and modifying the proof of Lemma 1 so the sequence requests pages $(\sigma_1 \ldots \sigma_{K+1})$ repeatedly for one processor and nothing for other processors. Then, the optimal partition will give all cache to this processor. The lower bound is the same as in the sequential case. The upper bound follows directly from Lemma 1. Note that these are not contradicting, since for the sequence of the lower bound $\max\{k_j\} = K$. Columns 7

to 9 follow from Theorem 1, while columns 10 to 12 follow from Theorems 4 and 1, with $A = LRU$ for column 10.

sP_A^B: Column 1: Since A can be LRU, we cannot prove a larger lower bound than 1. For the upper bound, for any marking or conservative algorithm A, the performance on each partition cannot be worse than k_j times the one of LRU. The proof is the same as the one of Lemma 1. Columns 2, 4, and 6 follow from Lemma 2. Note that for column 4 (sP_A^{OPT}), A is the same in both algorithms, but the bound still holds because A a marking or conservative algorithm and hence the number of faults of sP_A^{OPT} is K. Column 5 follows from Lemma 1. Columns 7 to 9 follow from Theorem 1. Column 10 follows from Theorem 4 but making the partition D in dP_A^D reduce the cache of the page evicted by A instead of LRU, and Theorem 1. Columns 11 and 12 follow from Theorems 4 and 1.

sP_A^{OPT}: For columns 1 and 2, the same argument used for column 1 of the row corresponding sP_A^B applies here, with $B = OPT$ for sP_{LRU}^B. Column 3: lower bound for $B = OPT$, upper bound from the optimality of the partition. Columns 5 and 6 can be proved using the same argument from the row for sP_{LRU}^{OPT} (columns 3 to 6). Again, columns 7 to 9 follow from Theorem 1. The bounds in columns 10 to 12 follow by applying the arguments of columns 10 to 12 of the row corresponding to sP_A^B.

sP_{OPT}^B: For column 1, a lower bound of 1 can be achieved with a request $\mathcal{R}$ such that neither algorithm faults. The upper bound follows from the optimality of the eviction policy. Column 2, 4, and 6 follow from Lemma 2. Column 3: lower bound for $A = OPT$, upper bound follows from the optimality of the eviction policy. Columns 7 to 9 follow from Theorem 1, while columns 10 to 12 follow from Theorems 4 and 1, with $A = LRU$ for column 10.

sP_{OPT}^{OPT}: For columns 1 and 2, a lower bound of 1 can be achieved with a request $\mathcal{R}$ such that neither algorithm faults. The upper bound follows from the optimality of sP_{OPT}^{OPT}. Columns 3 to 5: lower bound for $A = OPT$ and $B = OPT$, upper bound from the optimality of sP_{OPT}^{OPT}. Columns 7 to 9 follow from Theorem 1, while columns 10 to 12 follow from Theorems 4 and 1, with $A = LRU$ for column 10.

S_{LRU}: Columns 1 and 2: lower bound follows from Theorem 2 with $B = OPT$. The lower bound of Theorem 2 applies as well for columns 3 to 6, but since the eviction policy is the optimal offline, we can show a greater lower bound. The optimal offline strategy faults once every $k_{j^*} = K/p - (p-1)$ requests (see proof of Theorem 2), from which the lower bound can be easily derived. Note that since we assume $K \geq p^2$, this bound is always at least p. The upper bound for columns 1 to 6 follows from Lemma 3. The lower bound in columns 8 and 9 follow from Theorem 3 and the upper bound is the same as in the sequential case when $\tau = 0$. The lower bound in columns 10 and 11 follow from Lemma 8 with $A = OPT$, and the fact that if we replace dP_{LRU}^{OPT} for dP_{OPT}^{OPT} then the number of faults decreases by a multiplicative factor equal to the size of the cache of processor p, which is $K - p^2 + p$, and thus $\frac{S_{LRU}(R)}{dP_{OPT}^{OPT}(R)} = \Omega(p(\tau+1)(K - p^2 + p))$. The lower bound in column 12 follows from Lemma 8. The upper bound in columns 10 to 12 holds when $\tau = 0$, and it follows from the fact that dP_{OPT}^{OPT} must fault at least once for every shared phase, regardless of how the partition changes. For $\tau = 0$, the phases of S_{LRU} and dP_{OPT}^{OPT} are the same. LRU (or any marking or conservative algorithm) will fault at most K times per phase.

S_A**:** Columns 1 to 4: lower bound follows from Theorem 2 with $B = OPT$ and A being marking or conservative. The lower bounds of Columns 5 and 6 follows from the same argument as in columns 3 to 6 in the row for S_{LRU}. The upper bound for columns 1 to 6 follows from Lemma 3 and A being a marking or conservative algorithm. Column 7: A could be LRU and hence no larger upper bound can be proved. The upper bound is the same as in the sequential case when $\tau = 0$, and this applies to column 9 as well. The lower bound of column 9 follows from Theorem 3. The lower bounds for columns 10 and 12 follow from Lemma 8 and A being a conservative or marking algorithm (for column 10). For the lower bound of column 11 we can apply the same argument used for row S_{LRU} in columns 10 and 11. Again, the upper bound for columns 10 to 12 follows from the same argument used for row S_{LRU} in columns 10 to 12, and the fact that A is a conservative or marking.

S_{OPT}**:** Columns 1 to 12: For the lower bound, let $A = OPT$ and $B = D = OPT$, there is a sequence $\mathcal{R}$ such that all the algorithms in these columns make the same number of faults as S_{OPT} (e.g. each processor requests repeatedly the same page). Columns 1 to 6: the upper bound follows from Lemma 9 and the fact that $dP_{OPT}^{OPT}(\mathcal{R}) \leq sP_{OPT}^{OPT}$ for all $\mathcal{R}$. Columns 7 and 8: upper bound from optimality of S_{OPT}. Columns 10 to 12: upper bound follows from Lemma 9.

dP_A^D**:** Columns 1 to 6: let $B = OPT$, and apply Lemma 2 on the long stage of D. Since D is online and does not change in the long stage, it will fault a non-constant number of times (in the length of the sequence), while an optimal offline partition will be tailored for this stage and hence sP_A^{OPT} will fault at most K times (since A is marking or conservative). Columns 7 to 9: lower bound of $\Omega(n)$ for the case of non-disjoint requests follows from Lemma 6 and A being a marking or conservative algorithm (for column 8). Lower bound of $w(1)$ follows from Lemma 7 and A being a marking or conservative algorithm (for column 8). Column 11: lower bound follows from Lemma 4. Column 12: lower bound if $A = LRU$. The upper bounds for columns 11 and 12 follow from Lemma 5 and A being a conservative or marking algorithm.

dP_{OPT}^D**:** Columns 1 to 6: bounds can be derived applying the argument as in columns 1 to 6 for row for dP_A^D. The optimal eviction policy reduces the number of faults by at most K. Columns 7 to 9: lower bound of $\Omega(n)$ for the case of non-disjoint requests follows from Lemma 6. Lower bound of $w(1)$ follows from the same argument as in Lemma 7, however, the adversary cannot chose the page that is not in the cache, since the eviction policy of the dynamic partition algorithm is optimal. It is enough to request, for example, repeatedly pages $(\sigma_1^j, ..., \sigma_{k(j,t)+1}^j)$, where t is the time when the long stage begins. The number of faults of dP_{OPT}^D is at least ℓ/K, which can be arbitrarily large. The number of faults of S_{LRU} is at most $K + p$. Columns 10: lower bound for $A = OPT$. Column 12: lower bound for a sequence such that dP_{OPT}^D and dP_{LRU}^D fault the same number of times. Upper bounds for columns 10 and 12 follow from the optimality of the eviction policy.

dP_{LRU}^D**:** Columns 1 to 9: bounds follow from the same arguments as in columns 1 to 9 for row for dP_A^D. Columns 10 and 11, for $A = OPT$, lower bounds follows from Lemma 4 and upper bounds from Lemma 5.

D Optimal static partition

If we are able to process a request offline, the optimal partition of the cache can be computed by a simple dynamic program. If we have one sequence, then we can simply compute the cost of serving the sequence by running algorithm A on that sequence with a given cache size. If the size of the cache is zero, then the cost is the sum of the length of the sequences. Otherwise, the optimal cache partition is the minimum over all K choices of dividing the cache into two parts, giving the first to the first core, and the rest to an optimal partition of the rest $p-1$ cores.

Let $\mathcal{R} = \{R_1, \ldots, R_r\}$ denote a set of r request sequences and let $\mathcal{R}[i,j] = \{R_i, R_{i+1}, \ldots, R_j\}$. Let $C(\mathcal{R}, K)$ denote the optimal cost of serving $\mathcal{R}$ with a cache of size K, and let $A(R, K)$ be the number of faults when serving a sequence $R \in \mathcal{R}$ with algorithm A. The dynamic program that computes the optimal cost (number of faults) of serving a set of sequences with algorithm A with the optimal partition corresponds to the following recurrence:

$$C(\mathcal{R}, K) = \begin{cases} \sum_{i=1}^{r} |R_i| & \text{if } K = 0 \\ A(R_1, K) & \text{if } K > 0 \text{ and } r = 1 \\ \min_{i=0..K}\{C(R_1, i) + C(\mathcal{R}[2, r], K - i)\} & \text{otherwise} \end{cases}$$

The above dynamic program computes the cost of the optimal partition of the cache. The actual partition can be easily computed by backtracking in the table keeping track of the choices made for each cell. Note that the recurrence allows requests to be served with a cache of size zero. This recurrence can be easily modified for the case when we require non-empty caches for all requests.

Lemma 10 *Let A be an eviction policy. Assume that A takes constant time to decide which page to evict. Let $\mathcal{R}$ be a sequence of requests such that $|R| \leq n$ for all $R \in \mathcal{R}$. Given the size of cache K and number of processors p, we can compute the static partition B that minimizes the number of faults of $sP_A^B(\mathcal{R})$ in $O(p(nK + K^2))$ time.*

Proof: The table used to solve the dynamic program is a $(2p-1) \times (K+1)$ table. The entries corresponding to $K = 0$ can be computed in $O(p)$ time by computing the prefix sums of the lengths of the sequences. There are p rows corresponding to the base case when $r = 1$. Each of these pK entries can be computed by running the algorithm on a single sequence of length at most n, and hence the cost to compute these base cases is $O(npK)$. Finally, the rest of the entries, of which there are $(p-1)K$ can be computed in $O(K)$ time. The lemma follows. □