SIAM J. COMPUT.
Vol. 29, No. 4, pp. 1290–1303

APPLICATION-CONTROLLED PAGING FOR A SHARED CACHE[*]

RAKESH D. BARVE[†], EDWARD F. GROVE[‡], AND JEFFREY SCOTT VITTER[§]

Abstract. We propose a provably efficient application-controlled global strategy for organizing a cache of size k shared among P application processes. Each application has access to information about its own future page requests, and by using that local information along with randomization in the context of a global caching algorithm, we are able to break through the conventional $H_k \sim \ln k$ lower bound on the competitive ratio for the caching problem. If the P application processes always make good cache replacement decisions, our online application-controlled caching algorithm attains a competitive ratio of $2H_{P-1} + 2 \sim 2 \ln P$. Typically, P is much smaller than k, perhaps by several orders of magnitude. Our competitive ratio improves upon the $2P + 2$ competitive ratio achieved by the deterministic application-controlled strategy of Cao, Felten, and Li. We show that no online application-controlled algorithm can have a competitive ratio better than $\min\{H_{P-1}, H_k\}$, even if each application process has perfect knowledge of its individual page request sequence. Our results are with respect to a worst-case interleaving of the individual page request sequences of the P application processes.

We introduce a notion of fairness in the more realistic situation when application processes do not always make good cache replacement decisions. We show that our algorithm ensures that no application process needs to evict one of its cached pages to service some page fault caused by a mistake of some other application. Our algorithm not only is fair but remains efficient; the global paging performance can be bounded in terms of the number of mistakes that application processes make.

Key words. caching, application-controlled, competitive, online, randomized

AMS subject classifications. 68N25, 68P01, 68P15, 68Q25, 68W20

PII. S0097539797324278

1. Introduction.

Caching is a useful technique for obtaining high performance in these days where the latency of disk access is relatively high. Today's computers typically have several application processes running concurrently on them, by means of time sharing and multiple processors. Some processes have special knowledge of their future access patterns. Cao, Felten, and Li [CFL94a, CFL94b] exploit this special knowledge to develop effective file caching strategies.

An application providing specific information about its future needs is equivalent to the application having its own caching strategy for managing its own pages in cache. We consider the *multiapplication caching* problem, formally defined in section 3, in which P concurrently executing application processes share a common cache of size k. In section 4 we propose an online application-controlled caching scheme in which decisions need to be taken at two levels: when a page needs to be evicted from cache,

[*]Received by the editors July 14, 1997; accepted for publication (in revised form) September 10, 1998; published electronically February 18, 2000. An extended abstract of the results in this paper appeared in *Proceedings of the 36th Annual Symposium on Foundations of Computer Science*, 1995, pp. 204–213.

http://www.siam.org/journals/sicomp/29-4/32427.html

[†]Department of Computer Science, Duke University, Durham, NC 27708-0129 (rbarve@cs.duke.edu). This author was supported in part by an IBM graduate fellowship.

[‡]Department of Computer Science, Duke University, Durham, NC 27708-0129 (efg@cs.duke.edu). This author was supported in part by Army Research Office grant DAAH04-93-G-0076.

[§]Department of Computer Science, Duke University, Durham, NC 27708-0129 (jsv@cs.duke.edu). This author was supported in part by the National Science Foundation under grants CCR-9007851 and CCR-9522047 and by Air Force Office of Scientific Research grants F49620-92-J-0515 and F49620-94-1-0217. Part of this work was done while the author was visiting Lucent Technologies, Bell Laboratories, Murray Hill, NJ.

the global strategy chooses a victim process, but the process itself decides which of its pages will be evicted from cache.

Each application process may use any available information about its future page requests when deciding which of its pages to evict. However, we assume *no global information* about the interleaving of the individual page request sequences; all our bounds are with respect to a *worst-case interleaving* of the individual request sequences.

Competitive ratios smaller than the H_k lower bound for classical caching [FKL$^+$91] are possible for multiapplication caching, because each application may employ future information about its individual page request sequence.[1] The deterministic application-controlled algorithm proposed by Cao, Felten, and Li [CFL94a] achieves a competitive ratio of $2P + 2$, which we prove in the appendix. We show in sections 5–7 that our new randomized online application-controlled caching algorithm improves the competitive ratio to $2H_{P-1} + 2 \sim 2 \ln P$, which is optimal up to a factor of 2 in the realistic scenario when $P < k$. (If we use the algorithm of [FKL$^+$91] for the case $P \geq k$, the resulting bound is optimal up to a factor of 2 for all P.) Our results are significant since P is often *much* smaller than k, perhaps by several orders of magnitude.

In the appendix, we also prove that no deterministic online application-controlled algorithm can have a competitive ratio better than $P+1$. Thus the difference between our competitive ratio and that attained by the algorithm by Cao, Felten, and Li is a further indication of the (well-known) power of randomization while solving online problems in general and online paging in particular.

In the scenario where application processes occasionally make bad page replacement decisions (or "mistakes"), we show in section 8 that our online algorithm incurs very few page faults globally as a function of the number of mistakes. Our algorithm is also *fair*, in the sense that the mistakes made by one processor in its page replacement decisions do not worsen the page fault rate of other processors.

2. Classical caching and competitive analysis. The well-known *classical caching (or paging) problem* deals with a two-level memory hierarchy consisting of a fast *cache* of size k and *slow memory* of arbitrary size. A sequence of requests to pages is to be satisfied in their order of occurrence. In order to satisfy a page request, the page must be in fast memory. When a requested page is not in fast memory, a *page fault* occurs, and some page must be evicted from fast memory to slow memory in order to make room for the new page to be put into fast memory. The caching (or paging) problem is to decide which page must be evicted from the cache. The cost to be minimized is the number of page faults incurred over the course of servicing the page requests.

Belady [Bel66] gives a simple optimum offline algorithm for the caching problem; the page chosen for eviction is the one in cache whose next request is furthest in the future. In order to quantify the performance of an online algorithm, Sleator and Tarjan [ST85] introduce the notion of competitiveness, which in the context of caching can be defined as follows: for a caching algorithm A, let $F_A(\sigma)$ be the number of page faults generated by A while processing page request sequence σ. If A is a randomized algorithm, we let $F_A(\sigma)$ be the *expected* number of page faults generated by A on processing σ, where the expectation is with respect to the random choices made by the algorithm. An online algorithm A is called *c-competitive* if for every

[1] Here H_n represents the nth harmonic number $\sum_{i=1}^{n} 1/i \sim \ln n$.

page request sequence σ, we have $F_A(\sigma) \leq c \cdot F_{OPT}(\sigma) + b$, where b is some fixed constant. The constant c is called the *competitive ratio* of A. Under this measure, an online algorithm's performance needs to be relatively good on worst-case page request sequences in order for the algorithm to be considered good.

For cache size k, Sleator and Tarjan [ST85] show a lower bound of k on the competitive ratio of deterministic caching algorithms. Fiat et al. [FKL$^+$91] prove a lower bound of H_k if randomized algorithms are allowed. They also give a simple and elegant randomized algorithm for the problem that achieves a competitive ratio of $2H_k$. McGeoch and Sleator [MS91] give a rather involved randomized algorithm that attains the theoretically optimal competitive ratio of H_k.

3. Multiapplication caching problem.

In this paper we take up the theoretical issue of how best to use application processes' knowledge about their individual future page requests so as to optimize caching performance. For analysis purposes we use an online framework similar to that of [FKL$^+$91, MS91]. As mentioned before, the caching algorithms in [FKL$^+$91, MS91] use *absolutely no information* about future page requests. Intuitively, knowledge about future page requests can be exploited to decide which page to evict from the cache at the time of a page fault. In practice an application often has advance knowledge of its individual future page requests. Cao, Felten, and Li [CFL94a, CFL94b] introduced strategies that try to combine the advance knowledge of the processors in order to make intelligent page replacement decisions.

In the *multiapplication caching problem* we consider a cache capable of storing k pages that is shared by P different application processes, which we denote $P_1, P_2, \ldots, P_P$. Each page in cache and memory belongs to exactly one process. The individual request sequences of the processes may be interleaved in an arbitrary (worst-case) manner.

Worst-case measure is often criticized when used for evaluating caching algorithms for individual application request sequences [BIRS91, KPR92], but we feel that the worst-case measure is appropriate for considering a global paging strategy for a cache shared by concurrent application processes that have knowledge of their individual page request sequences. The locality of reference within each application's individual request sequence is accounted for in our model by each application process's knowledge of its own future requests. The worst-case nature of our model is that it assumes nothing about the order and duration of time for which application processes are active. In this model our worst-case measure of competitive performance amounts to considering a *worst-case interleaving* of individual sequences.

The approach of Cao, Felten, and Li [CFL94a] is to have the kernel deterministically choose the process owning the least recently used page at the time of a page fault and ask that process to evict a page of its choice (which may be different from the least recently used (LRU) page). In the appendix we show under the assumption that processes always make good page replacement decisions that Cao, Felten, and Li's algorithm has a competitive ratio between $P+1$ and $2P+2$. The algorithm we present in the next section and analyze thereafter improves the competitive ratio to $2H_{P-1} + 2 \sim 2\ln P$.

4. Online algorithm for multiapplication caching.

Our algorithm is an online application-controlled caching strategy for an operating system kernel to manage a shared cache in an efficient and fair manner. We show in the subsequent sections that the competitive ratio of our algorithm is $2H_{P-1}+2 \sim 2\ln P$ and that it is optimal to within a factor of about 2 among all online algorithms. (If $P \geq k$, we can use the algorithm of [FKL$^+$91].)

On a page fault, we first choose a victim process and then ask it to evict a suitable page. Our algorithm can detect mistakes made by application processes, which enables us to reprimand such application processes by having them pay for their mistakes. In our scheme, we mark pages as well as processes in a systematic way while processing the requests that constitute a phase.

Definition 4.1. *The global sequence of page requests is partitioned into a consecutive sequence of* phases*; each phase is a sequence of page requests. At the beginning of each phase, all pages and processes are unmarked. A page gets marked during a phase when it is requested. A process is marked when all of its pages in cache are marked. A new phase begins when a page is requested that is not in cache and all the pages in cache are marked. A page accessed during a phase is called* clean *with respect to that phase if it was not in the online algorithm's cache at the beginning of a phase. A request to a clean page is called a* clean page request*. Each phase always begins with a clean page request.*

Our marking scheme is similar to the one in [FKL$^+$91] for the classical caching problem. However, unlike the algorithm in [FKL$^+$91], the algorithm we develop is a nonmarking algorithm, in the sense that our algorithm may evict marked pages. In addition, our notion of phase in Definintion 4.1 is different from the notion of phase in [FKL$^+$91], which can be looked upon as a special case of our more general notion. We put the differences into perspective in section 4.1.

Our algorithm works as follows when a page p belonging to process P_r is requested:

(1) If p is in cache:
 - (a) If p is not marked, we mark it.
 - (b) If process P_r has no unmarked pages in cache, we mark P_r.

(2) If p is not in cache:
 - (a) If process P_r is unmarked and page p is *not* a clean page with respect to the ongoing phase (i.e., P_r has made a mistake earlier in the phase by evicting p), then
   - (i) We ask process P_r to make a page replacement decision and evict one of its pages from cache in order to bring page p into cache. We mark page p and also mark process P_r if it now has no unmarked pages in cache.
 - (b) Else (process P_r is marked or page p is a clean page, or both)
   - (i) If all pages in cache are marked, we remove marks from all pages and processes, and we start a new phase, beginning with the current request for p.
   - (ii) Let S denote the set of unmarked processes having pages in the cache. We randomly choose a process P_e from S, each process being chosen with a uniform probability $1/|S|$.
   - (iii) We ask process P_e to make a page replacement decision and evict one of its pages from cache in order to bring page p into cache. We mark page p and also mark process P_e if it now has no unmarked page in cache.

Note that in steps 2(a)(i) and 2(b)(iii) our algorithm seeks paging decisions from application processes that are unmarked. Consider an unmarked process P_i that has been asked to evict a page in a phase, and consider P_i's pages in cache at that time. Let u_i denote the *farthest unmarked* page of process P_i; that is, u_i is the unmarked page of process P_i whose next request occurs furthest in the future among all of P_i's unmarked cached pages. Note that process P_i may have marked pages in cache whose next requests occur after the request for u_i.

DEFINITION 4.2. *The* good set *of an unmarked process P_i at the current point in the phase is the set consisting of its farthest unmarked page u_i in cache and every marked page of P_i in cache whose next request occurs* after *the next request for page u_i. A page replacement decision made by an unmarked process P_i in either step* 2(a)(i) *or step* 2(b)(iii) *that evicts a page from its good set is regarded as a* good decision *with respect to the ongoing phase. Any page from the good set of P_i is a* good page *for eviction purposes at the time of the decision. Any decision made by an unmarked process P_i that is not a good decision is regarded as a* mistake *by process P_i.*

If a process P_i makes a mistake by evicting a certain page from cache, we can detect the mistake made by P_i if and when the same page is requested again by P_i in the same phase while P_i is still unmarked.

In sections 6 and 7 we specifically assume that application processes are always able to make good decisions about page replacement. In section 8 we consider fairness properties of our algorithm in the more realistic scenario where processes can make mistakes.

4.1. Relation to previous work on classical caching. Our marking scheme approach is inspired by a similar approach for the classical caching problem in [FKL$^+$91]. However, the phases defined by our algorithm are significantly different in nature from those in [FKL$^+$91]. Our phase ends when there are k distinct marked pages in cache; more than k distinct pages may be requested in the phase. The phases depend on the random choices made by the algorithm and are probabilistic in nature. On the other hand, a phase defined in [FKL$^+$91] ends when exactly k distinct pages have been accessed, so that given the input request sequence, the phases can be determined independently of the caching algorithm being used.

The definition in [FKL$^+$91] is suited to facilitate the analysis of online caching algorithms that never evict marked pages, called *marking algorithms.* In the case of marking algorithms, since marked pages are never evicted, as soon as k distinct pages are requested, there are k distinct marked pages *in* cache. This means that the phases determined by our definition for the special case of marking algorithms are exactly the same as the phases determined by the definition in [FKL$^+$91]. Note that our algorithm is in general *not* a marking algorithm since it may evict marked pages. While marking algorithms always evict unmarked pages, our algorithm always calls on unmarked processes to evict pages; the actual pages evicted may be marked.

4.2. Relation to the algorithm of Cao, Felten, and Li. The algorithm proposed by Cao, Felten, and Li [CFL94a] for the multiapplication caching problem amounts to evicting, at the time of a page fault, the farthest page from cache belonging to the process that owns the LRU page in cache. Thus, for a given interleaving of individual request sequences, the paging decisions made by that algorithm are deterministic. We prove in the appendix that no deterministic online algorithm for the multiapplication caching problem can have a competitive ratio better than $P+1$, and that the competitive ratio attained by the algorithm proposed by Cao, Felten, and Li [CFL94a] attains a competitive ratio of $2P+2$. Thus, the performance of the algorithm in [CFL94a] is within a factor of 2 of the best possible performance by any deterministic online algorithm for the multiapplication caching problem.

Given the notion of good pages that we developed above, it turns out that we can define a slightly more general version of the algorithm in [CFL94a] without changing its paging performance. Basically, in order to attain the competitive ratio of $2P + 2$, it is enough, at the time of a page fault, to evict from cache *any good page* belonging to the process that owns the LRU page in cache. We say that this is a slightly more general version than the algorithm presented in [CFL94a] because it may very often be the case that the good set of the process that owns the LRU page contains several pages other than the farthest page of that process.

5. Lower bounds for *OPT* and competitive ratio.

In this section we prove that the competitive ratio of any online caching algorithm can be no better than $\min\{H_{P-1}, H_k\}$. Let us denote by OPT the optimal offline algorithm for caching that works as follows: when a page fault occurs, OPT evicts the page whose next request is furthest in the future request sequence among all pages in cache.

As in [FKL$^+$91], we will compare the number of page faults generated by our online algorithm during a phase with the number of page faults generated by OPT during that phase. We express the number of page fronts as a function of the number of clean page requests during the phase. Here we state and prove a lower bound on the (amortized) number of page faults generated by OPT in a single phase. The proof is a simple generalization of an analogous proof in [FKL$^+$91], which deals only with the deterministic phases of marking algorithms.

LEMMA 5.1. *Consider any phase σ_i of our online algorithm in which ℓ_i clean pages are requested. Then OPT incurs an amortized cost of at least $\ell_i/2$ on the requests made in that phase.*[2]

Proof. Let d_i be the number of clean pages in OPT's cache at the beginning of phase σ_i; that is, d_i is the number of pages requested in σ_i that are in OPT's cache but not in our algorithm's cache at the beginning of σ_i. Let d_{i+1} represent the same quantity for the next phase σ_{i+1}. Let $d_{i+1} = d_m + d_u$, where d_m of the d_{i+1} clean pages in OPT's cache at the beginning of σ_{i+1} are marked during σ_i and d_u of them are not marked during σ_i. Note that $d_1 = 0$ and $d_i \le k$ for all i.

Of the ℓ_i clean pages requested during σ_i, only d_i are in OPT's cache, so OPT generates at least $\ell_i - d_i$ page faults during σ_i. On the other hand, while processing the requests in σ_i, OPT cannot use d_u of the cache locations, since at the beginning of σ_{i+1} there are d_u pages in OPT's cache that are not marked during σ_i. (These d_u pages would have to be in OPT's cache before σ_i even began.) There are k marked pages in our algorithm's cache at the end of σ_i, and there are d_m other pages marked during σ_i that are out of our algorithm's cache. So the number of distinct pages requested during σ_i is at least $d_m + k$. Hence, OPT serves at least $d_m + k$ requests corresponding to σ_i without using d_u of the cache locations. This means that OPT generates at least $(k + d_m) - (k - d_u) = d_{i+1}$ faults during σ_i. Therefore, the number of faults OPT generates on σ_i is at least

$$\max\{\ell_i - d_i, d_{i+1}\} \ge \frac{\ell_i - d_i + d_{i+1}}{2}. \tag{5.1}$$

Let us consider the number of page faults made by OPT in the first j phases of page request sequence σ. We can use the lower bound (5.1) for phases 2, 3, ..., $j-1$. In the jth phase, OPT makes at least $\ell_j - d_j \ge (\ell_j - d_j)/2$ faults. In the first phase, OPT generates k faults and we have $\ell_1 = k$. Thus the sum of OPT's faults over all

[2]By "amortized" in Lemma 5.1 we mean for each $j \ge 1$ that the number of page faults made by OPT while serving the first j phases is at least $\sum_{i=1}^{j} \ell_i/2$, where ℓ_i is the number of clean page requests in the ith phase.

j phases is at least

$$\ell_1 + \sum_{i=2}^{j-1} \frac{\ell_i - d_i + d_{i+1}}{2} + \frac{\ell_j - d_j}{2} \geq \sum_{i=1}^{j} \ell_i/2,$$

where we use the fact that $d_2 \leq k = \ell_1$. Thus by definition, the amortized number of faults OPT generates over any phase σ_i is at least $\ell_i/2$. ∎

Next we will construct a lower bound for the competitive ratio of *any* randomized online algorithm even when application processes have perfect knowledge of their individual request sequences. The proof is a straightforward adaptation of the proof of the H_k lower bound for classical caching [FKL+91]. However, in the situation at hand, the adversary has more restrictions on the request sequence that he can use to prove the lower bound, thereby resulting in a lowering of the lower bound.

THEOREM 5.2. *The competitive ratio of any randomized algorithm for the multiapplication caching problem is at least* $\min\{H_{P-1}, H_k\}$ *even if application processes have perfect knowledge of their individual request sequences.*

Proof. If $P > k$, the H_k lower bound on the classical caching problem from [FKL+91] is directly applicable by considering the case where each process accesses only one page each. This gives a lower bound of H_k on the competitive ratio.

In the case when $P \leq k$, we construct a multiapplication caching problem based on the nemesis sequence used in [FKL+91] for classical caching. In [FKL+91] a lower bound of $H_{k'}$ is proved for the special case of a cache of size k' and a total of $k'+1$ pages, which we denote $c_1, c_2, \ldots, c_{k'+1}$. All but one of the pages can fit in cache at the same time. Our corresponding multiapplication caching problem consists of $P = k'+1$ application processes $P_1, P_2, \ldots, P_P$ so that there is one process corresponding to each page of the classical caching lower bound instance for a k'-sized cache. Process P_i owns r_i pages $p_{i1}, p_{i2}, \ldots, p_{ir_i}$. The total number $\sum_{i=1}^{P} r_i$ of pages among all the processes is $k+1$, where k is the cache size; that is, all but one of the pages among all the processes can fit in memory simultaneously.

In the instance of the multiapplication caching problem we construct, the request sequence for each process P_i consists of repetitions of the double round-robin sequence

$$p_{i1}, p_{i2}, \ldots, p_{ir_i}, p_{i1}, p_{i2}, \ldots, p_{ir_i} \tag{5.2}$$

of length $2r_i$. We refer to the double round-robin sequence (5.2) as a *touch* of process P_i. When the adversary generates requests corresponding to a touch of process P_i, we say that it "touches process P_i."

Given an arbitrary adversarial sequence for the classical caching problem described above, we construct an adversarial sequence for the multiapplication caching problem by replacing each request for page c_i in the former problem by a touch of process P_i in the latter problem. We can transform an algorithm for this instance of multiapplication caching into one for the classical caching problem by the following correspondence: if the multiapplication algorithm evicts a page from process P_j while servicing the touch of process P_i, the classical caching algorithm evicts page c_j in order to service the request to page c_i. In Lemma 5.3 below, we show that there is an optimum online algorithm for the above instance of multiapplication caching that never evicts a page belonging to process P_i while servicing a fault on a request for a page from process P_i. Thus the transformation is valid, in that page c_i is always resident in cache after the page request to c_i is serviced. This reduction immediately implies that the competitive ratio for this instance of multiapplication caching must be at least $H_{k'} = H_{P-1}$. ∎

LEMMA 5.3. *For the above instance of multiapplication caching, any online algorithm A can be converted into an online algorithm A' that is at least as good in an amortized sense and that has the property that all the pages for process P_i are in cache immediately after a touch of P_i is processed.*

Proof. Intuitively, the double round-robin sequences force an optimal online algorithm to service the touch of a process by evicting a page belonging to *another* process. We construct online algorithm A' from A in an online manner. Suppose that both A and A' fault during a touch of process P_i. If algorithm A evicts a page of P_j, for some $j \neq i$, then A' does the same. If algorithm A evicts a page of P_i during the first round-robin while servicing a touch of P_i, then there will be a page fault during the second round-robin. If A then evicts a page of another process during the second round-robin, then A' evicts that page during the first round-robin and incurs no fault during the second round-robin. The first page fault of A was wasted; the other page could have been evicted instead during the first round-robin. If instead A evicts another page of P_i during the second round-robin, then A' evicts an arbitrary page of another process during the first round-robin, and A' incurs no page fault during the second round-robin. Thus, if A evicts a page of P_i, it incurs at least one more page fault than does A'.

If A faults during a touch of P_i, but A' doesn't, there is no paging decision for A' to make. If A does not fault during a touch of P_i, but A' does fault, then A' evicts the page that is not in A's cache. The page fault for A' is charged to the extra page fault that A incurred earlier when A' evicted one of P_i's pages.

Thus the number of page faults that A' incurs is no more than the number of page faults that A incurs. By construction, all pages of process P_i are in algorithm A''s cache immediately after a touch of process P_i. □

The double round-robin sequences in the above reduction can be replaced by single round-robin sequences by redoing the explicit lower bound argument of [FKL+91].

6. Holes.

In this section, we introduce the notion of holes, which plays a key role in the analysis of our online caching algorithm. In section 6.2, we mention some crucial properties of holes of our algorithm under the assumption that applications always make good page replacement decisions. These properties are also useful in bounding the page faults that can occur in a phase when applications make mistakes in their page replacement decisions.

DEFINITION 6.1. *The eviction of a cached page at the time of a page fault on a clean page request is said to create a* hole *at the evicted page. Intuitively, a hole is the lack of space for some page, so that that page's place in cache contains a hole and not the page. If page p_1 is evicted for servicing the clean page request, page p_1 is said to be associated with the hole. If page p_1 is subsequently requested and another page p_2 is evicted to service the request, the hole is said to* move *to p_2, and now p_2 is said to be associated with the hole, and so on, until the end of the phase. We say that* hole h moves to process P_i *to mean that the hole h moves to some page p belonging to process P_i.*

6.1. General observations about holes.

All requests to clean pages during a phase are page faults and create holes. The number of holes created during a particular phase equals the number of clean pages requested during that phase. Apart from clean page requests, requests to holes also cause page faults to occur. By a *request to a hole* we mean a request for the page associated with that hole. As we proceed down the request sequence during a phase, the page associated with a particular hole varies with time. Consider a hole h that is created at a page p_1 that is evicted to serve a request for clean page p_c. When a request is made for page p_1, some page p_2 is evicted, and h moves to p_2. Similarly when page p_2 is requested, h moves to some p_3, and so on. Let $p_1, p_2, \ldots, p_m$ be the temporal sequence of pages all associated with hole h in a particular phase such that page p_1 is evicted when clean page p_c is requested, page p_i, where $i > 1$, is evicted when p_{i-1} is requested and the request for p_m falls in the next phase. Then the number of faults incurred in the particular phase being considered due to requests to h is $m - 1$.

6.2. Useful properties of holes. In this section we make the following observations about holes under the assumption that application processes make only good decisions.

LEMMA 6.2. *Let u_i be the farthest unmarked page in cache of process P_i at some point in a phase. Then process P_i is a marked process by the time the request for page u_i is served.*

Proof. This follows from the definition of farthest unmarked page and the nature of the marking scheme employed in our algorithm. □

LEMMA 6.3. *Suppose that there is a request for page p_i, which is associated with hole h. Suppose that process P_i owns page p_i. Then process P_i is already marked at the time of the present request for page p_i.*

Proof. Page p_i is associated with hole h because process P_i evicted page p_i when asked to make a page replacement decision in order to serve either a clean request or a page fault at the previous page associated with h. In either case, page p_i was a good page at the time process P_i made the particular paging decision. Since process P_i was unmarked at the time the decision was made, p_i was either the farthest unmarked page of process P_i then or some marked page of process P_i whose next request is after the request for P_i's farthest unmarked page. By Lemma 6.2, process P_i is a marked process at the time of the request for page p_i. □

LEMMA 6.4. *Suppose that page p_i is associated with hole h. Let P_i denote the process owning page p_i. Suppose page p_i is requested at some time during the phase. Then hole h does not move to process P_i subsequently during the current phase.*

Proof. The hole h belongs to process P_i. By Lemma 6.3 when a request is made to h, P_i is already marked and will remain marked until the end of the phase. Since only unmarked processes are chosen to evict pages, a request for h thereafter cannot result in eviction of any page belonging to P_i, so a hole can never move to a process more than once. □

Let there be R unmarked processes at the time of a request to a hole h. For any unmarked process P_j, $1 \leq j \leq R$, let u_j denote the farthest unmarked page of process P_j at the time of the request to hole h. Without loss of generality, let us relabel the processes so that

$$u_1, u_2, u_3, \ldots, u_R \tag{6.1}$$

is the temporal order of the first subsequent appearance of the pages u_j in the global page request sequence.

LEMMA 6.5. *In the situation described in* (6.1) *above, suppose during the page request for hole h that the hole moves to a good page p_i of unmarked process P_i to serve the current request for h. Then h can never move to any of the processes $P_1, P_2, \ldots, P_{i-1}$ during the current phase.*

Proof. The first subsequent request for the good page p_i that P_i evicts, by definition, must be the same as or must be after the first subsequent request for the farthest unmarked page u_i. So process P_i will be marked by the next time hole h is requested, by Lemma 6.3. On the other hand, the first subsequent requests of the respective farthest unmarked pages $u_1, \dots, u_{i-1}$ appear before that of page u_i. Thus, by Lemma 6.2, the processes $P_1, P_2, \dots, P_{i-1}$ are already marked before the next time hole h (page p_i) gets requested and will remain marked for the remainder of the phase. Hence, by the fact that only unmarked processes get chosen, hole h can never move to any of the processes $P_1, P_2, \dots, P_{i-1}$. □

7. Competitive analysis of our online algorithm.

Our main result is Theorem 7.1, which states that our online algorithm for the multiapplication caching problem is roughly $2 \ln P$-competitive, assuming application processes always make good decisions (e.g., if each process knows its own future page requests). By the lower bound of Theorem 5.2, it follows that our algorithm is optimal in terms of competitive ratio up to a factor of 2.

THEOREM 7.1. *The competitive ratio of our online algorithm in section* 4 *for the multiapplication caching problem, assuming that good evictions are always made, is at most* $2H_{P-1} + 2$. *Our competitive ratio is within a factor of about* 2 *of the best possible competitive ratio for this problem.*

The rest of this section is devoted to proving Theorem 7.1. To count the number of faults generated by our algorithm in a phase, we make use of the properties of holes from the previous section. If ℓ requests are made to clean pages during a phase, there are ℓ holes that move about during the phase. We can count the number of faults generated by our algorithm during the phase as

$$\ell + \sum_{i=1}^{\ell} N_i, \tag{7.1}$$

where N_i is the number of times hole h_i is requested during the phase. Assuming good decisions are always made, we will now prove for each phase and for any hole h_i that the expected value of N_i is bounded by H_{P-1}.

Consider the first request to a hole h during the phase. Let R_h be the number of unmarked processes at that point in time. Let C_{R_h} be the random variable associated with the number of page faults due to requests to hole h during the phase.

LEMMA 7.2. *The expected number* $E(C_{R_h})$ *of page faults due to requests to hole h is at most* H_{R_h}.

Proof. We prove this by induction over R_h. We have $E(C_0) = 0$ and $E(C_1) = 1$. Suppose for $0 \le j \le R_h - 1$ that $E(C_j) \le H_j$. Using the same terminology and notation as in Lemma 6.5, we let the farthest unmarked pages of the R_h unmarked processes at the time of the request for h appear in the temporal order

$$u_1, u_2, u_3, \dots, u_{R_h}$$

in the global request sequence. We renumber the R_h unmarked processes for convenience so that page u_i is the farthest unmarked page of unmarked process P_i.

When the hole h is requested, our algorithm randomly chooses one of the R_h unmarked processes, say, process P_i, and asks process P_i to evict a suitable page. Under our assumption, the hole h moves to some good page p_i of process P_i. From Lemmas 6.4 and 6.5, if our algorithm chooses unmarked process P_i so that its good page p_i is evicted, then *at most* $R_h - i$ *processes remain unmarked* the next time h is requested. Since each of the R_h unmarked processes is chosen with a probability of $1/R_h$, we have

$$\begin{aligned} E(C_{R_h}) &\leq 1 + \frac{1}{R_h} \sum_{i=1}^{R_h} E(C_{R_h - i}) \\ &= 1 + \frac{1}{R_h} \sum_{i=0}^{R_h - 1} E(C_i) \\ &\leq 1 + \frac{1}{R_h} \sum_{i=0}^{R_h - 1} H_i \\ &= H_{R_h}. \end{aligned}$$

The last equality follows easily by induction and algebraic manipulations. □

Now let us complete the proof of Theorem 7.1. By Lemma 6.3 the maximum possible number of unmarked processes at the time a hole h is first requested is $P-1$. Lemma 7.2 implies that the average number of times any hole can be requested during a phase is bounded by H_{P-1}. By (7.1), the total number of page faults during the phase is at most $\ell(1 + H_{P-1})$. We have already shown in Lemma 5.1 that the *OPT* algorithm incurs an amortized cost of at least $\ell/2$ for the requests made in the phase. Therefore, the competitive ratio of our algorithm is bounded by $\ell(1+H_{P-1})/(\ell/2) = 2H_{P-1}+2$. Applying the lower bound of Theorem 5.2 completes the proof.

8. Application-controlled caching with fairness. In this section we analyze our algorithm's performance in the realistic scenario where application processes can make mistakes, as defined in Definition 4.2. We bound the number of page faults it incurs in a phase in terms of page faults caused by mistakes made by application processes during that phase. The main idea here is that if an application process P_i commits a mistake by evicting a certain page p and then during the same phase requests page p while process P_i is still unmarked, our algorithm makes process P_i pay for the mistake in step 2(a)(i).

On the other hand, if page p's eviction from process P_i was a mistake, but process P_i is marked when page p is later requested in the same phase, say, at time t, then process P_i's mistake is "not worth detecting" for the following reason: since evicting page p was a mistake, it must mean that at the time t_1 of p's eviction, there existed a set U of one or more unmarked pages of process P_i in cache whose subsequent requests appear *after* the next request for page p. Process P_i is marked at the time of the next request for p, implying that all pages in U were *evicted* by P_i at some times $t_2, t_3, \ldots, t_{|U|+1}$ after the mistake of evicting p. If instead at time $t_1, t_2, \ldots, t_{|U|+1}$ process P_i makes the specific good paging decisions of evicting the farthest unmarked pages, *the same set* $\{p\} \cup U$ *of pages will be out of cache at time* t. In our notion of fairness we choose to ignore all such mistakes and consider them "not worth detecting."

DEFINITION 8.1. *During an ongoing phase, any page fault corresponding to a request for a page p of an* unmarked *process P_i is called an* unfair fault *if the request for page p is not a clean page request. All faults during the phase that are not unfair are called* fair faults.

The unfair faults are precisely those page faults which are caused by mistakes considered "worth detecting." We state the following two lemmas that follow trivially from the definitions of mistakes, good decisions, unfair faults, and fair faults.

LEMMA 8.2. *During a phase, all page requests that get processed in step* 2(a)(i) *of our algorithm are precisely the unfair faults of that phase. That is, unfair faults correspond to mistakes that get caught in step* 2(a)(i) *of our algorithm.*

LEMMA 8.3. *All fair faults are precisely those requests that get processed in step* 2(b)(iii).

We now consider the behavior of holes in the current mistake-prone scenario.

LEMMA 8.4. *The number of holes in a phase equals the number of clean pages requested in the phase.*

LEMMA 8.5. *Consider a hole h associated with a page p of a process P_i. If a request for h is an unfair fault, process P_i is still unmarked and the hole h moves to some other page belonging to process P_i. If a request for hole h is a fair fault, then process P_i is already marked and the hole h can never move to process P_i subsequently during the phase.*

Proof. If the request for hole h is an unfair fault, then by definition process P_i is unmarked and by Lemma 8.2, h moves to some other page p' of process P_i. If the request for h is a fair fault, then by definition and the fact that the request for h is not a clean page request, process P_i is marked. Since our algorithm never chooses a marked process for eviction, it follows that h can never visit process P_i subsequently during the phase. $\square$

During a phase, a hole h is created in some process, say, P_1, by some clean page request. It then moves around zero or more times within process P_1 on account of P_1's mistakes, until a request for hole h is a fair fault, upon which it moves to some other process P_2, never to come back to process P_1 during the phase. It behaves similarly in process P_2, and so on up to the end of the phase. Let T_h denote the total number of faults attributed to requests to hole h during a phase, of which F_h faults are fair faults and U_h faults are unfair faults. We have $T_h = F_h + U_h$.

By Lemma 8.5 and the same proof techniques as those in the proofs of Lemma 7.2 and Theorem 7.1, we can prove the following key lemma.

LEMMA 8.6. *The expected number $E(F_h)$ of page requests to hole h during a phase that result in fair faults is at most H_{P-1}.*

By Lemma 8.4, our algorithm incurs at most $\ell + \sum_{i=1}^{\ell} T_{h_i}$ page faults in a phase with ℓ clean page requests. The expected value of this quantity is at most $\ell(H_{P-1}+1) + \sum_{i=1}^{\ell} U_{h_i}$, by Lemma 8.6.

The expression $\sum_{i=1}^{\ell} U_{h_i}$ is the number of *unfair faults*, that is, the number of mistakes considered "worth detecting." Our algorithm is very efficient in that the number of unfair faults is an additive term. For any phase ϕ with ℓ clean requests, we denote $\sum_{i=1}^{\ell} U_{h_i}$ as M_ϕ.

THEOREM 8.7. *The number of faults in a phase ϕ with ℓ clean page requests and M_ϕ unfair faults is bounded by $\ell(1 + H_{P-1}) + M_\phi$. At the time of each of the M_ϕ unfair faults, the application process that makes the mistake that causes the fault must evict a page from its own cache. No application process is ever asked to evict a page to service an unfair fault caused by some other application process.*

8.1. Extending fairness to the algorithm by Cao, Felten, and Li. It turns out that our notion of fairness extends, without any change, to the generalized version of the deterministic algorithm of [CFL94a] that we mentioned in section 4.2. It is easy to see that in the case of the generalized version of the algorithm of [CFL94a], a process incurs an unfair fault only if, at some time in the past, that process had the LRU page and the page it evicted was not a good page. Consequently, a result similar to Theorem 8.7 with $(1+H_{P-1})$ replaced by $(1+P)$ holds for the generalized version of the algorithm of [CFL94a].

9. Conclusions.

Cache management strategies are of prime importance for high performance computing. We consider the case where there are P independent processes running on the same computer system and sharing a common cache of size k. Applications often have advance knowledge of their page request sequences. In this paper we have addressed the issue of exploiting this advance knowledge to devise intelligent strategies to manage the shared cache, in a theoretical setting. We have presented a simple and elegant randomized application-controlled caching algorithm for the multiapplication caching problem that achieves a competitive ratio of $2H_{P-1}+2$. Our result is a significant improvement over the competitive ratios of $2P+2$ [CFL94a] for deterministic multiapplication caching and $\Theta(H_k)$ for classical caching, since the cache size k is often orders of magnitude greater than P. We have proven that no online algorithm for this problem can have a competitive ratio smaller than $\min\{H_{P-1}, H_k\}$, even if application processes have perfect knowledge of individual request sequences. We conjecture that an upper bound of H_{P-1} can be proven, up to second-order terms, perhaps using techniques from [MS91], although the resulting algorithm is not likely to be practical.

Using our notion of mistakes we are able to consider a more realistic setting when application processes make bad paging decisions and show that our algorithm is a fair and efficient algorithm in such a situation. No application needs to pay for some other application process's mistake, and we can bound the global caching performance of our algorithm in terms of the number of mistakes. Our notions of good page replacement decisions, mistakes, and fairness in this context are new.

One related area of possible future work is to consider alternative models to our model of worst-case interleaving. Another interesting area would be to consider caching in a situation where some applications have good knowledge of future page requests while other applications have no knowledge of future requests. We could also consider pages shared among application processes.

Appendix. The Cao, Felten, and Li algorithm.

The algorithm proposed by Cao, Felten, and Li [CFL94a] for the multiapplication caching problem amounts to evicting, at the time of a page fault, the *farthest* page from cache belonging to the process that owns the LRU page in cache.

THEOREM A.1. *The algorithm of Cao, Felten, and Li is* $(2P+2)$*-competitive. A generalized version of the algorithm of Cao, Felten, and Li, in which, at the time of a page fault, the process owning the LRU page in cache evicts any (deterministically chosen) good page, is also* $(2P+2)$*-competitive.*

Proof. Let there be ℓ clean page requests in a phase. Then there are ℓ faults due to clean page requests resulting in ℓ holes. The algorithm evicts only *good* pages from cache, so holes are associated only with such pages. By Lemma 6.4 we can conclude that each hole can result in at most one page fault per process up to the end of the phase, so that the total number of page faults in the phase is bounded by $\ell + \ell P$. Using Lemma 5.1 gives the above competitive factor. □

THEOREM A.2. *The competitive ratio of any deterministic online algorithm for the multiapplication caching problem is at least* $P+1$.

Proof. Since the algorithm is deterministic, we can construct an interleave that costs the algorithm a factor of $P+1$ times the number of faults that OPT will incur. For instance, consider a single clean request in each phase. On the basis of our knowledge of the deterministic choices made by the algorithm, we can easily make the resulting hole visit each process at least once so that the deterministic online algorithm incurs at least $P + 1$ faults per phase, whereas OPT incurs just one fault. $\square$

REFERENCES

[Bel66] A. L. Belady, *A study of replacement algorithms for virtual storage computers*, IBM Systems J., 5 (1966), pp. 78–101.

[BIRS91] A. Borodin, S. Irani, P. Raghavan, and B. Schieber, *Competitive paging with locality of reference*, in Proceedings of the 23rd Annual ACM Symposium on Theory of Computation, New Orleans, LA, 1991, pp. 249–259.

[CFL94a] P. Cao, E. W. Felten, and K. Li, *Application-controlled file caching policies*, in Proceedings of the Summer USENIX Conference, Boston, MA, 1994, pp. 171–182.

[CFL94b] P. Cao, E. W. Felten, and K. Li, *Implementation and performance of application-contolled file caching*, in Proceedings of the First OSDI Symposium, Monterey, CA, 1994, pp. 165–177.

[FKL$^+$91] A. Fiat, R. M. Karp, M. Luby, L. A. McGeoch, D. D. Sleator, and N. E. Young, *On competitive algorithms for paging problems*, J. Algorithms, 12 (1991), pp. 685–699.

[KPR92] A. R. Karlin, S. J. Phillips, and P. Raghavan, *Markov paging*, in Proceedings of the 33rd Annual IEEE Conference on Foundations of Computer Science, Pittsburgh, PA, 1992, pp. 208–217.

[MS91] L. A. McGeoch and D. D. Sleator, *A strongly competitive randomized paging algorithm*, Algorithmica, 6 (1991), pp. 816–825.

[ST85] D. D. Sleator and R. E. Tarjan, *Amortized efficiency of list update and paging rules*, Comm. ACM, 28 (1985), pp. 202–208.