Minimizing Cache Usage in Paging

Alejandro López-Ortiz and Alejandro Salinger

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

Abstract. Traditional paging models seek algorithms that maximize their performance while using the maximum amount of cache resources available. However, in many applications this resource is shared or its usage involves a cost. In this work we introduce the Minimum Cache Usage problem, which is an extension to the classic paging problem that accounts for the efficient use of cache resources by paging algorithms. In this problem, the cost of a paging algorithm is a combination of both its number of faults and the amount of cache it uses, where the relative cost of faults and cache usage can vary with the application. We present a simple family of online paging algorithms that adapt to the ratio α between cache and fault costs, achieving competitive ratios that vary with α, and that are between 2 and the cache size k. Furthermore, for sequences with high locality of reference, we show that the competitive ratio is at most 2, and provide evidence of the competitiveness of our algorithms on real world traces. Finally, we show that the offline problem admits a polynomial time algorithm. In doing so, we define a reduction of paging with cache usage to weighted interval scheduling on identical machines.

1 Introduction

The efficient management of a computer memory hierarchy is a fundamental problem in both computer architecture and software design. A program's data and instructions reside in various levels of the hierarchy, in which memories at higher levels have higher capacities, but slower access times. Simplified to a two-level memory system, the paging problem models a slow memory of infinite size and a fast memory of limited size, usually known as the cache. The input consists of a sequence of page requests. If the page of a request is in the cache then the request is a *hit*; otherwise it is a *miss* or *fault* and the requested page must be brought from slow memory to cache, possibly requiring the eviction of one or more pages currently residing in cache. A paging algorithm must decide which pages to maintain in the cache at each time in order to minimize a defined cost measure. In the classic page fault model the cost of an algorithm is measured in terms of its number of faults and hits have no cost, reflecting the fact that an access to slow memory is orders of magnitude slower than an access to cache.

As computer architectures and applications evolve, other cost models have arisen to reflect, for example, varying fetching costs and sizes in web-caches [11,

18, 29], or multi-threaded applications sharing a cache [4, 9, 16, 17, 21]. In this paper we consider a generalization of the classic page fault model whose performance objective function is a combination of both the number of faults and the amount of cache used by an algorithm. Thus in addition to the fault cost, at each step we charge a cost proportional to the number of pages in cache. In general, the model seeks algorithms with good performance in terms of number of faults while at the same time using available resources efficiently. Naturally, minimizing the number of faults and the cache usage of a paging algorithm are conflicting goals.

Paging strategies that minimize cache usage are relevant in multi-core architectures where multiple cores share some level of cache. In this context, multiple request sequences compete for the use of this shared resource. While traditional models of paging encourage algorithms to use the entire cache so as to minimize the faults incurred, a model that charges for cache usage can make a paging algorithm in a shared cache scenario be "context aware". Varying the parameters of the model for each sequence can be used to achieve a cooperative global strategy with better overall performance.

The cache minimizing model can also be used as an energy efficient paging model. Several applications use caches implemented with Content-Addressable Memories (CAMs), most notably networking routers and switches, and Translation Lookaside Buffers (TLBs). CAMs provide a single clock cycle throughput, making them faster than other hardware alternatives [24]. However, speed comes at a cost of increased power consumption, mainly due to the comparison circuitry. Reducing this power without sacrificing capacity or speed is an important goal of research in circuit design [24]. Power consumption could be reduced if inactive cache lines are turned off, thus our model can provide a framework for paging strategies that achieve good performance in terms of faults while contributing to energy savings.

1.1 Paging and Cost Models

The paging problem has been extensively studied; some well-known page replacement policies are Least-Recently-Used (LRU), which evicts the page in the cache whose last access time is furthest in the past; First-In-First-Out (FIFO), which evicts the page that has been longest in the cache; and Flush-When-Full (FWF), which when required to evict a page evicts all pages from the cache.

The performance of paging algorithms has been traditionally measured using competitive analysis [26]. A paging algorithm A has competitive ratio r or is r-competitive if its cost $A(\mathcal{R})$ over any sequence $\mathcal{R}$ satisfies $A(\mathcal{R}) \leq r \cdot OPT(\mathcal{R}) + \beta$, where $OPT(\mathcal{R})$ is the optimal cost of serving $\mathcal{R}$ offline, and β is a constant. In the page fault model, where a fault has cost 1 and hits have no cost, the algorithms above are k-competitive, where k is the size of the cache, which is optimal for deterministic algorithms [6]. A competitive ratio of k is achieved by all *marking* and *conservative* algorithms. An algorithm is conservative if it incurs at most k faults on any consecutive subsequence of requests that contains at most k distinct pages [6]. A marking algorithm associates a mark with each

page in its cache (either explicitly or implicitly) and marks a page when it is brought to cache or if it is unmarked and requested. Upon a fault with a full cache, it only evicts unmarked pages if there are any, and unmarks all pages in cache otherwise. The latter event marks the start of a phase, which defines a k-phase partition of a request sequence. LRU and FWF are marking algorithms, while LRU and FIFO are conservative algorithms [6]. Randomized algorithms with optimal competitive ratio $\Theta(\log k)$ exist for this problem [6]. The offline problem can be solved optimally by Belady's algorithm [5]: evict the page in cache that is going to be requested furthest in the future (FITF).

Other cost models for paging differ in the assumptions of applications with respect to the cost of bringing a page into the cache, and the size of pages [18, 11, 10, 29]. Unlike these models, which consider only the cost of faults, the *full access cost* model [27] charges a cost of 1 for a hit, and a cost of $s \geq 1$ for a fault. In this model, marking algorithms achieve a competitive ratio of $1+\frac{(k-1)s}{L+s}$, where L is the average phase length in the k-phase partition of a sequence. In the worst case, $L = k$ and the ratio is $k(s+1)/(k+s)$, which is optimal. The model coincides with the classic model when $s \to \infty$, but can yield competitive ratios that are significantly smaller if s is small or if a sequence has high locality [6], properties that, as we shall see, are also shared by our model.

A related paging model that also includes the amount of cache used in the cost of algorithms is described in [14]. In this model an algorithm can purchase cache slots, and the overall cost of the algorithm is the number of faults plus the cost of purchased cache. As cache may only be bought, the cache size can only increase (with no bound on the maximum size). In our model, however, an algorithm is charged for the number of pages it has in the cache at every step, which can both increase or decrease. In this sense our model charges algorithms for renting cache, while keeping the upper bound k on the maximum cache available. Finally, we note that the idea of memory renting for reducing RAM power consumption was previously mentioned in [12].

1.2 Our Contributions

This work introduces a generic model of efficient cache usage in paging that can be applied to any scenario in which it is desirable for a paging algorithm to minimize the amount of cache it uses.

We define a family of online algorithms that combine the eviction policies of traditional marking or conservative algorithms with cache saving policies. The performance of the algorithms adapts to the relative cost of faults and cache. More precisely, they achieve a competitive ratio of 2 if $\alpha < k$, where $\alpha = f/c$ is the ratio between fault and cache cost, and $\min\left\{k, \frac{\alpha(k+1)}{\alpha+k-1}\right\}$ if $\alpha \geq k$, thus matching the performance of classical algorithms when $f \gg c$. We further parametrize the analysis by considering the locality of reference of the sequence, and show that for sequences with high locality of reference the competitive ratio of our algorithms is at most 2. Simulations on real-world inputs show that our

algorithms are close to optimal in terms of the total cost, and both its cache usage and number of faults are close to those of the optimal offline.

Lastly, we show that the offline problem admits a polynomial time algorithm via a reduction to interval weighted interval scheduling on identical machines.

The rest of this paper is organized as follows. Section 2 introduces the Minimum Cache Usage model and problem. We present an optimal offline algorithm in Sect. 3, and present our results related to online algorithms and simulations in Sect. 4. Due to space constraints, we include only some of the proofs and charts, while the rest appear in the full version [20].

2 Paging with Cache Usage

The paging model we consider in this paper extends classic paging to a model in which the cost of a paging algorithm on a request sequence is a weighted function of the number of faults and the total amount of cache used by the algorithm. An instance of paging with minimum cache consists of a sequence $\mathcal{R} = \{r_1, r_2, \ldots, r_n\}$ of page requests and a maximum cache size k. Each request r_i is associated with a page σ_j, for $1 \leq j \leq N$, where N is the size of the universe of pages that can be requested. We denote by $page(r_i)$ the page associated with request r_i. A paging algorithm can hold at most k pages in its cache, but can also choose to hold fewer pages, in order to reduce its cache usage.

Definition 1 (Total cache usage). *Let A be a paging algorithm and $\mathcal{R}$ a request sequence. Let $k(i) \leq k$ denote the number of pages in A's cache immediately before request r_i, where k is the maximum cache size. The total cache usage of A when serving $\mathcal{R}$ is defined as $C_A(\mathcal{R}) = \sum_i k(i)$.*

Given a request sequence $\mathcal{R}$ and maximum cache size k, the cost of an algorithm A on $\mathcal{R}$ is defined as $A(\mathcal{R}) = fF_A(\mathcal{R}) + cC_A(\mathcal{R})$, where $F_A(\mathcal{R})$ and $C_A(\mathcal{R})$ are the number of faults and total cache usage of A when serving $\mathcal{R}$, respectively, and $f \geq 0$ and $c \geq 0$ are parameters. The *Minimum Cache Usage* problem is then the problem of serving a request sequence with minimum cost.

In reality a request sequence is revealed in an online fashion, thus our focus is on the performance of online algorithms in terms of their competitive ratio. An online algorithm has competitive ratio r if, given a maximum cache size k, and parameters f and c, for all request sequences $A(\mathcal{R}) \leq r \cdot OPT(\mathcal{R}) + \beta$, where OPT is the optimal offline, r is a function of k, f and c, and β is a constant that does not depend on $\mathcal{R}$. As in classic paging, the steps involved in serving a request r_i are as follows: the page associated with the request is revealed to the algorithm, after which the algorithm acts by possibly evicting one or more pages, and finally the request is served. Thus, all pages evicted in cache in step i were held in cache up to time $i - 1$. A paging algorithm is said to be *lazy* or *demand paging* if it only evicts a page when a page fault occurs. Observe that unlike classic paging, in which any algorithm can be made demand paging without sacrificing performance [6], in our model algorithms can benefit from evicting pages even when there is no page fault.

The relation between the parameters f and c can vary according to the application to emphasize the importance of minimizing faults or using the cache efficiently, or a combination of both. Naturally, an instance with $c = 0$ and $f > 0$ is an instance of the classical model, in which the cost of an algorithm is its number of faults. On the other hand, if $f < c$ then the problem is trivial: an optimal algorithm always evicts the page of each request immediately after serving it. We assume in general that $f \geq c > 0$.

2.1 Applications

The cost model described above provides incentives for an eviction policy to be efficient not only in terms of its faults but also with respect to the use of the resources that are available to it. Thus, the model can be used in any environment where the latter has significance. We mention the following applications.

Shared Cache Multiprocessors. Multi-core processors are equipped with both private and shared caches, with threads running in each core usually competing for the latter type. While there are schedulers that seek to achieve cooperative use of a shared cache, in general paging strategies for individual threads do not act cooperatively but use as much of the available cache as possible. The cost model we propose provides incentives for paging algorithms to balance their own benefits—a fast execution due to a small number of faults—and the benefits they can provide to concurrently running threads. Depending on the values of f and c, an algorithm will favour one or the other.

Energy Efficient Caching. Content Addressable Memories (CAMs) are used in many applications that require high speed searches, and whose primary applications are in network routers [24]. CAMs are indexed by stored data words instead of memory addresses, as in regular caches. Each cell has a matchline that indicates if the stored word in the cell and the searched word match. A search for an input data word first precharges all matchlines, then each cell compares its bits against the searched bits, and matchlines corresponding to non-matching entries are discharged. The overall missing matchline dynamic power consumption for a system with w matchlines can be modeled as $P = wCV^2f$, where C is the matchline capaticance, V is the supply of a matchline and f is the frequency of misses (the power associated with a matchline in a match is small and can be neglected) [24]. The power involved in this operation can be therefore reduced if matchline precharging is controlled based on the valid bit status of each entry [22]: on a search, only valid entries require the precharging of matchlines, thus the power cost of a search can be proportional to the number of valid entries in the cache. In this scenario, a paging algorithm that uses its cache efficiently will contribute to power savings.

3 Offline Optimum

In the next section we describe a simple family of online algorithms for the cache usage problem and analyze their competitiveness. In order to provide a better

intuition for that analysis we first describe a solution to the offline problem. We recast the paging instance as an instance of weighted interval scheduling on identical machines, and use an algorithm for this problem to obtain an optimal polynomial time paging algorithm.

An instance of Weighted Interval Scheduling on Identical Machines consists of a set J of jobs and a number m of available identical machines. Each job has a starting time, a duration, and a weight. In order to be processed, a job must be assigned to a machine immediately after its start time and cannot be interrupted. A machine can process only one job at a time. The goal is to process a subset $J' \subseteq J$ of jobs such that the total weight of jobs in J' is maximized. Equivalently, each job corresponds to an interval in the real line, and we seek to schedule the maximum weight subset of intervals such that at most m intervals overlap at any time. This problem can be solved in polynomial time by reduction to minimum cost flow [3, 7].

It will be useful to see a paging problem instance as an instance of interval scheduling on identical machines: each pair of consecutive requests to the same page defines an interval whose start and end times are the times of the requests. In each pair of requests, the second request results in a hit if and only if the corresponding page is kept in the cache since the previous request, or equivalently, if the interval is scheduled.

The connection between interval scheduling and paging has been noted before in [28, 8] where it is used to study cache policies in non-standard caches. It is assumed, however, that the reduction applies only when bypassing is allowed. More recently, [13] used this connection to show that offline paging in the fault and bit models is NP-hard by reducing interval packing problems to paging. Unlike our model, these models consider pages (and hence intervals) of different sizes. The reduction we introduce in this paper is from paging to interval scheduling, and it is defined as follows.

Definition 2 (Interval representation of a sequence). *An interval representation of a request $\mathcal{R}$ of length n is a set of intervals $\mathcal{I}(\mathcal{R}) = \{I_1, I_2, \ldots, I_n\}$ where each interval I_i corresponds to request r_i in $\mathcal{R}$. The starting time of each interval I_i is $s(I_i) = i + 1$ and the end time is $e(I_i) = j - 1$, where $j > i$ is the smallest index such that $page(r_j) = page(r_i)$, or $e(I_i) = n$ if no such j exists. We say that an interval I_i is* feasible *if $e(I_i) < n$ and* unfinished *otherwise. Thus the length of interval I_i is $|I_i| = e(I_i) - s(I_i) + 1$.*

An example of a sequence and its interval representation is shown in Fig. 1. Intuitively, an interval corresponding to request r_j represents the time interval in which $page(r_j)$ must reside in the cache in order for the next request to this page to result in a hit. Note that each first request to a page has no preceding interval thus cannot be a hit. Similarly, a page that is requested for the last time in a sequence can be held in cache, but as the interval does not finish in the corresponding page, it cannot result in a hit. Note that intervals do not overlap with the times in which their corresponding pages are requested, thus using this reduction theres is no need to assume that bypassing is allowed. All requests are

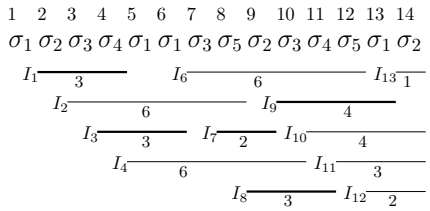

Fig. 1. A request sequence and its interval representation. The length of each interval is shown below the interval (I_5 of length 0 is not shown). Feasible intervals are $\{I_1, I_2, I_3, I_4, I_5, I_6, I_7, I_8, I_9\}$ while $\{I_{10}, I_{11}, I_{12}, I_{13}\}$ are unfinished. The request can be served with a cache of size 3 with 8 faults and a cache usage of 29 by scheduling intervals $\{I_1, I_3, I_5, I_7, I_8, I_9\}$ on 2 machines (thus requests 5,6,7,10,12,14 are hits and the rest are faults), which is the optimal cache cost for the minimum number of faults.

served, but only the ones whose interval was scheduled will result in hits. Note as well that two consecutive requests to the same page define an interval of length 0 that does not overlap any other interval, and thus it is always scheduled. The following Lemma formalizes the reduction[1].

Lemma 1. *Let $\mathcal{R}$ be a request sequence. Let $\mathcal{I}' = \mathcal{I}(\mathcal{R}) \setminus \{I_i : I_i \text{ is unfinished}\}$. Let $S \subset \mathcal{I}'$ be a feasible schedule of $\mathcal{I}'$ on $k-1$ machines. Then $\mathcal{R}$ can be served with a cache of size k such that all requests r_j with $I_i \in S$ and $j = i + |I_i| + 1$ are hits, with a total cache usage of $|\mathcal{R}| + \sum_{I_i \in S} |I_i|$.*

In light of Lemma 1, when describing the actions of an algorithm while serving a request $\mathcal{R}$, we sometimes use the terminology related to interval scheduling. Thus we say that an algorithm schedules an interval I_i to mean that it keeps a page $page(r_i)$ in cache until request r_j with $j = i + |I_i| + 1$ (and $page(r_j) = page(r_i)$). We define the *cache cost* of a request r_j as the number of requests that $page(r_j)$ was kept in cache for after r_i, which equals $|I_i|$ if r_j is a hit, and is smaller otherwise.

If we are only interested in minimizing faults then the problem corresponds to Maximal Interval Scheduling. This problem can be solved by sorting intervals in increasing order of end time, and then greedily scheduling intervals while there are available machines. Minimizing the number of faults while at the same time using the least possible cache can be solved instead by computing the maximum weight schedule in the corresponding interval representation. Weighted interval scheduling on identical machines can in turn be solved by formulating the problem as a minimum cost flow problem [3, 7]. Since we are interested in minimizing cache usage (equivalently, minimizing processing time in the interval schedule), for a given instance $\mathcal{R}$ we assign weights to intervals using the following corollary from [7]:

[1] See full version [20] for full proofs.

Corollary 1. *[7, Cor. 2] For each interval $I_j \in I(\mathcal{R})$ with processing time $|I_j|$, define a weight $w_j = M - |I_j| + 1$ [2], where M is a positive real number such that $M \geq \sum |I_j|$. Then a solution to maximum weight interval scheduling gives an optimal solution to maximal interval scheduling with minimum total processing time.*

Using the above weight assignment and a maximum weight scheduling algorithm we obtain a way of serving request $\mathcal{R}$ with the minimum number of faults, and with minimum cache usage. Recall that in general we seek to minimize the total cost of serving a sequence $\mathcal{R}$, defined as $fF(\mathcal{R}) + cC(\mathcal{R})$, which does not necessarily imply minimizing the number of faults F. However, we can still use the same reduction to interval scheduling and subsequently to minimum cost flow by first eliminating from $\mathcal{I}(\mathcal{R})$ all intervals whose cache cost is higher than the fault cost. It is easy to see that any solution that includes an interval I_i such that $c|I_i| > f$ could be modified to obtain a smaller cost by not scheduling that interval and paying for the fault instead. Hence, an optimal algorithm does not schedule any interval whose cost is higher than that of the fault cost. The resulting optimal offline algorithm is shown in Algorithm 1, where MaxWeightSchedule is an algorithm for maximum weight interval scheduling. Clearly, computing the interval representation of a request $\mathcal{R}$ of n pages (lines 2-13) takes $O(n)$ time, while MaxWeightSchedule takes time $O(m^2 \log m)$ [3], where m is the number of intervals of the weighted interval scheduling problem. Naturally, $m = O(n)$, which yields an $O(n^2 \log n)$ total running time. However, in general m might be much smaller than n, depending on the number of different pages in $\mathcal{R}$ and the number of intervals whose length is greater than f/c.

Theorem 1. *Given a request sequence $\mathcal{R}$ of length n and a cache size k, and constants $f \geq 0$, and $c \geq 0$, an optimal way of serving $\mathcal{R}$ that minimizes $fF(\mathcal{R}) + cC(\mathcal{R})$, where $F(\mathcal{R})$ and $C(\mathcal{R})$ are the number of faults and cache usage when serving $\mathcal{R}$, can be computed in $O(n^2 \log n)$ time.*

4 Online Algorithms

In this section we present a family of online algorithms that adapt to the relative cost of a fault versus the cache cost. These algorithms are k-competitive in the worst case (when $f \gg c$), but can achieve significant cache savings and smaller cost when the cache cost is closer to the fault cost. As a warm-up, we show that while classical optimal paging algorithms are also k-competitive, this ratio does not improve when the cache cost is high relative to the fault cost.

Lemma 2. *Let A be any marking or conservative paging algorithm. The competitive ratio of A is at most k.*

[2] We add 1 to the weight of each interval so that intervals have non-zero weight if all intervals have length 0.

Algorithm 1 Minimum Cache Usage Cost($\mathcal{R}, k, f, c$)

```
1: {Compute interval representation of R without unfinished intervals}
2: I = ∅ ; M ← 0
3: for j = 1 to |R| do
4:    lastRequest[page(r_j)] = −1
5: for j = 1 to |R| do
6:    i ←lastRequest[page(r_j)]
7:    if i ≠ −1 then
8:       s(I_i) ← i + 1 ; e(I_i) ← j − 1
9:       if c · |I_i| ≤ f then
10:         add I_i to I; M ← M + |I_i|
11:   lastRequest[σ] = j
12: for i = 1 to |I| do
13:   w(I_i) = M − |I_i| + 1
14: S ←MaxWeightSchedule(I, k − 1)
15: return f(|R| − |S|) + c(Σ_{I_i∈S} |I_i| + |R|)
```

Proof. Let $\mathcal{R}$ be any sequence and consider its k-phase partition. Since A is marking or conservative, it faults at most k times per phase. In addition, in a phase of m requests any algorithm has a cache cost of at most cmk. On the other hand, any algorithm must fault at least once per phase, and must pay at least cm for a phase of m requests. Thus $A(\mathcal{R})/OPT(\mathcal{R}) \leq (fk+cmk)/(f+cm) = k$.

Lemma 3. *Let A be any lazy paging algorithm. Then the competitive ratio of A is at least k.*

Proof. Let $\alpha = f/c$ and $c \neq 0$. Suppose that α is finite. Let $\mathcal{R} = \{\sigma_1, \sigma_2, \ldots, \sigma_{k-1}, (\sigma_k)^x\}$, with $\sigma_i \neq \sigma_j$ for all $i \neq j$, and $(\sigma)^x$ denotes a sequence of x consecutive requests to σ. Since A is a lazy algorithm, it will not evict any page from the cache, thus only faulting in the first k requests but using the entire cache until the end of the sequence. Hence, $A(\mathcal{R}) \geq fk + xkc$. An optimal algorithm can use only one cell of cache for a cost of $OPT(\mathcal{R}) = fk + xc$. Since x can be made arbitrarily large and f/c is bounded, the result follows. In the case of an unbounded α, the same sequence used in the classic lower bound of k applies: request the page in $\{\sigma_1, \ldots, \sigma_{k+1}\}$ not currently in the cache. Thus, $A(\mathcal{R}) \geq n(f + c)$ and $OPT(\mathcal{R}) \leq (n/k)f + nkc$ and the ratio approaches k as $\alpha \to \infty$.

4.1 A Family of Cost-Sensitive Online Algorithms

Definition 3. *For any online paging algorithm A, we define A_d as the algorithm that acts like A, except that for each r_i, it evicts page(r_i) at time $i + d$ if this page has not been requested by that time and is still in the cache. In this case, we say that page(r_i) expires at time $i + d$. We say that a page suffers an early eviction if it is evicted as a result of a capacity miss, according to A's eviction policy. Thus, if page(r_i) is not requested or evicted early within $[i, i + d]$, it will reside in cache for $d + 1$ requests.*

We restrict our choice of online algorithms in the definition above to marking and conservative algorithms and set $d = \lfloor \alpha \rfloor = \lfloor f/c \rfloor$. Consider A=LRU. For some instances LRU could have a better cost than LRU_α[3]. We now show, however, that the cost of LRU_α is always at most twice the cost of LRU, while there exists a sequence for which the cost of LRU is k times worse than the cost of LRU_α, which is the worst possible ratio for a marking algorithm. This direct comparison of two algorithms can be seen as a variation of *relative interval analysis* [15] that uses the cost ratio instead of the cost difference: for algorithms A and B let $Min(A,B) = \liminf_{n\to\infty}(\min_{|\mathcal{R}|=n}\{A(\mathcal{R})/B(\mathcal{R})\})$ and $Max(A,B) = \limsup_{n\to\infty}(\max_{|\mathcal{R}|=n}\{A(\mathcal{R})/B(\mathcal{R})\})$. Then the relative interval of A and B is $\mathcal{I}(A,B) = [Min(A,B), Max(A,B)]$, and $\mathcal{I}(A,B) \subseteq [\gamma,\delta]$ if $\gamma \leq Min(A,B)$ and $Max(A,B) \leq \delta$. Thus, if $\mathcal{I}(A,B) \subseteq [\gamma \geq 1, \delta > 1]$ we say that B dominates A, since on any sequence B is no worse than A and there is at least one sequence for which B is better than A. Lemma 5[1] and Theorem 2 show that $\mathcal{I}(\text{LRU},\text{LRU}_\alpha) \subseteq [1/2, k]$. Thus, although LRU does not properly dominate LRU_α, the latter is generally preferable to the former. Throughout the proofs in this section we use the following lemma:

Lemma 4. *[25, Cor. 11] Let two vectors $\boldsymbol{x} = (x_1,\ldots,x_n) \geq \mathbf{0}$ and $\boldsymbol{y} = (y_1,\ldots,y_n) > \mathbf{0}$ be given. Let π denote a permutation of $(1,\ldots,n)$. Then*

$$\frac{\sum_{i=1}^n x_i}{\sum_{i=1}^n y_i} \leq \min_\pi \max\left\{\frac{x_i}{y_{\pi(i)}} : 1 \leq i \leq n\right\} \leq \max\left\{\frac{x_i}{y_{\pi(i)}} : 1 \leq i \leq n, \text{ and fixed } \pi\right\}$$

Lemma 5. *Let $\alpha = f/c$ be finite. Then $Max(LRU,LRU_\alpha) = k$.*

Theorem 2. *Assume $k \geq 2$. Then, for all $\mathcal{R}$, $LRU_\alpha(\mathcal{R}) \leq 2\ LRU(\mathcal{R})$, and thus $Min(LRU,LRU_\alpha) \geq 1/2$.*

Proof. Let $\mathcal{R}$ be any sequence. Let F and C denote the faults and cache cost of LRU on $\mathcal{R}$ and let F_α and C_α denote the corresponding costs for LRU_α. Let $C_\alpha = C_{fh} + C_{hh} + C_{ff} + C_{hf} + \gamma$, where C_{fh} is the cache cost of requests that are faults for LRU_α and hits for LRU, and C_{ff}, C_{hh}, and C_{hf} are defined analogously. γ is the cost of keeping unfinished intervals. We will use the following properties: (1) every page of a request sequence is kept in LRU's cache for at least as long as in LRU_α's cache; and (2) any request that is a fault for LRU_α and is a hit for LRU corresponds to a page that expired in LRU_α's cache.

To see that Property (1) holds, note that if LRU evicts a page σ upon request r_i, then either σ has also expired in LRU_α's cache, or it is evicted at this point on request r_i as well. The latter holds because if σ was evicted from LRU's cache, then there are k distinct requests since the last request to σ, and since σ has not expired in LRU_α's cache, there are $k-1$ pages in LRU_α's cache that have not expired either and are younger than σ. Hence upon request r_i, LRU_α evicts σ as well. Property (1) implies that every request that is a hit for LRU_α is a hit

[3] To keep notation simple, we refer to $A_{\lfloor\alpha\rfloor}$ as A_α.

for LRU, and thus $C_{hf} = 0$. Property (2) follows from the fact if LRU_α evicts a page σ due a capacity miss, then its cache is full and since all pages stay longer in LRU's cache, then LRU's cache holds the same pages and evicts σ as well, hence the next request to σ is also a fault for LRU.

Property (1) implies as well that LRU's cache cost is $C \geq C_{fh} + F_\alpha - F + C_{ff} + C_{hh} + \gamma$. Moreover, both properties imply that $C_{fh} = \lfloor \alpha \rfloor (F_\alpha - F)$. Hence,

$$\begin{aligned}
\frac{LRU_\alpha(\mathcal{R})}{LRU(\mathcal{R})} &\leq \frac{fF_\alpha + c(\lfloor \alpha \rfloor (F_\alpha - F) + C_{hh} + C_{ff} + \gamma)}{fF + c(\lfloor \alpha \rfloor (F_\alpha - F) + F_\alpha - F + C_{ff} + C_{hh} + \gamma)} \\
&\leq \frac{fF_\alpha + c\lfloor \alpha \rfloor (F_\alpha - F)}{fF + c(\lfloor \alpha \rfloor (F_\alpha - F) + F_\alpha - F)} \qquad \text{(by Lemma 4)} \\
&= \frac{\alpha F_\alpha + \lfloor \alpha \rfloor (F_\alpha - F)}{\alpha F + \lfloor \alpha \rfloor (F_\alpha - F) + F_\alpha - F}
\end{aligned}$$

The above expression is bounded above by 2 if $\alpha \geq 2$. The case $\alpha < 2$ is covered by the upper bound on the competitive ratio of A_α in Theorem 3. □

4.2 Upper bound on the Competitive Ratio of A_α

We now show that for any marking or conservative algorithm A, the competitive ratio of A_α adapts to the relative costs of faults and hits, being at most 2 when the cost of faults is relatively small, and matching the competitiveness of traditional paging algorithms when the cache cost is negligible.[1]

Theorem 3. *Let A be any marking or conservative algorithm and let $\alpha = f/c$. Assume $k \geq 2$. The competitive ratio of A_α is at most $2 - \frac{1+\alpha-\lfloor \alpha \rfloor}{\alpha+1}$ if $\alpha < k$ and $\min\left\{k, \frac{\alpha(k+1)}{k+\alpha-1}\right\}$ if $\alpha \geq k$.*

Lemma 6 gives a lower bound on the competitive ratio for A_α, which matches the upper bound for $\alpha < k-1$. For larger values of α the gap between upper and lower bounds is reduced as α grows. Lemma 7 gives a straightforward smaller lower bound for any online algorithm.[1]

Lemma 6. *For A marking or conservative, the competitive ratio of A_α is at least $2 - \frac{1+\alpha-\lfloor \alpha \rfloor}{\alpha+1}$ if $\alpha < k-1$ and $\frac{\alpha k + k^2/2}{\alpha + k^2}$ otherwise.*

Lemma 7. *The competitive ratio of any online deterministic algorithm is at least $\frac{k(\alpha+1)}{\alpha+k^2}$.*

The classic paging cost model has been criticized for not being able to capture the benefit of online algorithms on sequences with high locality of reference [6]. Various studies have analyzed the competitiveness of paging algorithms in a parameterized manner, attempting to capture relevant characteristics of sequences such as, for example, locality and typical memory accesses [25], and attack rate [23]. We now give a parameterized competitive ratio for A_α that varies with the locality of reference of the input sequence, for which we use the definition in terms of the average phase length in its k-phase partition.[1]

Theorem 4. *Let A be any marking or conservative algorithm, let $\alpha = f/c$, and let $k \geq 2$. Let $\mathcal{R}$ be any request sequence and let ϕ be the number of phases in $\mathcal{R}$'s k-phase partition. Let $L(\mathcal{R}) = |\mathcal{R}|/\phi$. Then $A_\alpha(\mathcal{R})/OPT(\mathcal{R}) \leq 2$ if $L(\mathcal{R}) > k\alpha(\alpha - 2)$, and $\frac{A_\alpha(\mathcal{R})}{OPT(\mathcal{R})} \leq 1 + \frac{\alpha k + 1 - \alpha}{\alpha + k - 1 + L(\mathcal{R})}$ otherwise.*

4.3 Real World Sequences

We measured the performance of various algorithms on real world cache traces collected from 4 applications using VMTrace (for Linux) and the Etch tool (on Windows NT)[19]. We obtained the traces from [2] and truncated them to 3×10^6 entries. We simulated LRU, LRU$_\alpha$, FWF, FWF$_\alpha$, FIFO, FIFO$_\alpha$, and OPT on these sequences. For each sequence, we used the size of cache that would yield a fault rate of 1% and 0.1% for LRU. Figure 4.3 shows the cost ratio compared to OPT, fault rate, and average cache usage for the *espresso* sequence (a circuit simulator) for two cache sizes. Results for other sequences are shown in the full version [20]. For the total cost we set $c = 1$ and $f = \alpha$. We implemented the optimal offline (Algorithm 1) using the reduction to minimum cost flow in [7], and solved the minimum cost flow instances using the implementation of the cost scaling algorithm from the LEMON C++ library [1]. Results in these practical instances show that the cost of A_α algorithms adapt nicely to the value of α, and that their fault rate and cache usage approaches those ones of the optimal offline. In fact, the ratio A_α/OPT is never more than 2 and in most cases is close to 1. As suggested by Theorem 4, the cost ratio of A_α algorithms improves for sequences with higher locality. Note as well that as α grows, the performance of the traditional marking algorithms gets closer to that of its cost-sensitive counterpart, which is more noticeable for instances with smaller caches.

5 Conclusions

We introduced a model for paging with minimum cache usage and presented a cost-sensitive family of online algorithms whose performance adapts to the relative costs of cache and faults. The cost model that we propose is able to capture locality of reference, yielding a competitive ratio of at most 2 for inputs with high locality. Experiments on request sequences collected from actual programs agree with the theoretical results.

It would be interesting to show a better lower bound for online algorithms, and to propose and analyze other online algorithms, including randomized ones. A natural direction of research would be to evaluate the model in an application, either in theory or in practice. For example, it would be interesting to study and design a global shared caching strategy that varies the relative cache and fault cost for various threads so that the cooperative execution leads to an advantage in overall performance.

Acknowledgements. We would like to thank Francisco Claude, Robert Fraser, Patrick Nicholson, and Hiren Patel for insightful discussions. We are also thankful to the anonymous reviewers of this paper for their useful comments.

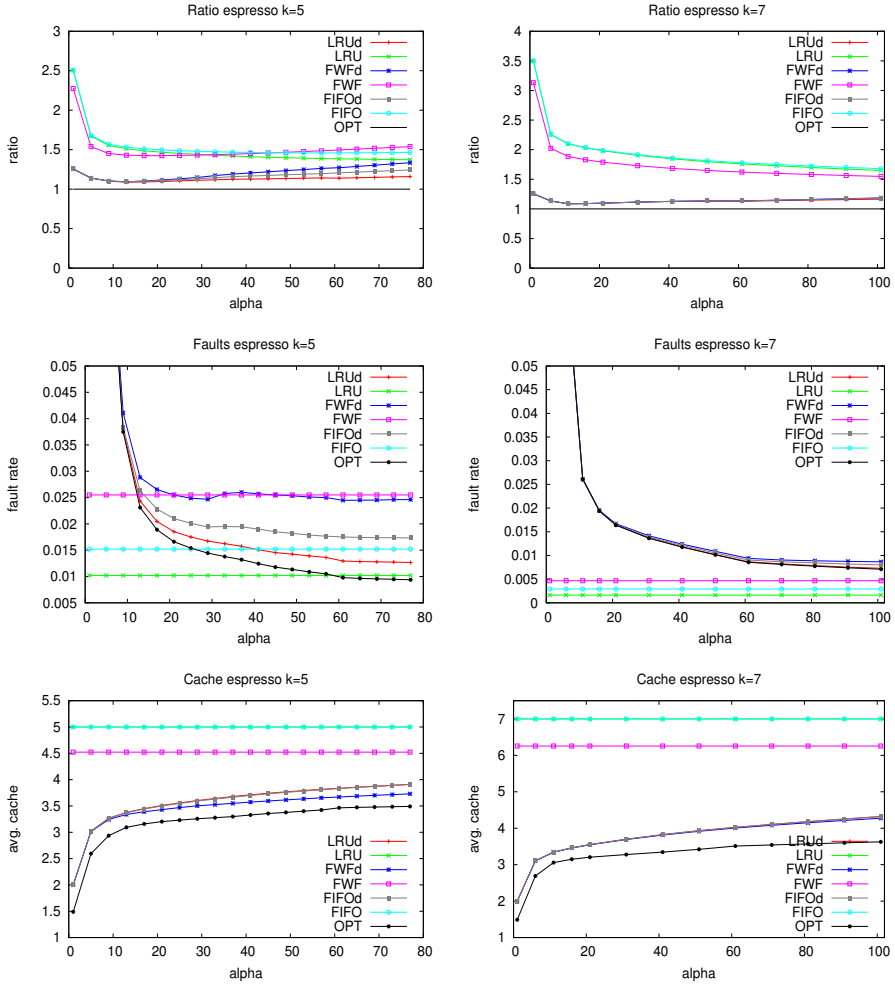

Fig. 2. Cost ratio, fault rate, and average cache used by LRU, LRUd, FWF, FWFd, FIFO, FIFOd, and OPT (with $d = \alpha$) on sequence "espresso" of length 3×10^6 with cache sizes $k = 5$ (average phase length 196) and $k = 7$ (average phase length 1502).

References

1. Lemon graph library. http://lemon.cs.elte.hu/trac/lemon
2. Trace reduction for virtual memory simulation. http://www.cs.amherst.edu/∼sfkaplan/research/trace-reduction/index.html
3. Arkin, E.M., Silverberg, E.B.: Scheduling jobs with fixed start and end times. Discrete Appl. Math. 18(1), 1–8 (1987)
4. Barve, R.D., Grove, E.F., Vitter, J.S.: Application-controlled paging for a shared cache. SIAM J. Comput. 29, 1290–1303 (2000)

5. Belady, L.A.: A study of replacement algorithms for virtual-storage computer. IBM Systems Journal 5(2), 78–101 (1966)
6. Borodin, A., El-Yaniv, R.: Online computation and competitive analysis. Cambridge University Press, New York, NY, USA (1998)
7. Bouzina, K.I., Emmons, H.: Interval scheduling on identical machines. Journal of Global Optimization 9, 379–393 (1996)
8. Brehob, M., Wagner, S., Torng, E., Enbody, R.J.: Optimal replacement is np-hard for nonstandard caches. IEEE Trans. Computers 53(1), 73–76 (2004)
9. Cao, P., Felten, E.W., Li, K.: Application-controlled file caching policies. In: Proceedings of USTC - Volume 1. pp. 171–182 (1994)
10. Cao, P., Irani, S.: Cost-aware www proxy caching algorithms. In: Proceedings of USITS. pp. 18–18 (1997)
11. Chrobak, M., Karloff, H., Payne, T., Vishwanathan, S.: New results on server problems. SIAM J. Discret. Math. 4(2), 172–181 (1991)
12. Chrobak, M.: Sigact news online algorithms column 17. SIGACT News 41(4), 114–121 (2010)
13. Chrobak, M., Woeginger, G.J., Makino, K., Xu, H.: Caching is hard - even in the fault model. Algorithmica 63(4), 781–794 (2012)
14. Csirik, J., Imreh, C., Noga, J., Seiden, S.S., Woeginger, G.J.: Buying a constant competitive ratio for paging. In: Meyer auf der Heide, F. (ed.) ESA. LNCS, vol. 2161, pp. 98–108. Springer (2001)
15. Dorrigiv, R., López-Ortiz, A., Munro, J.I.: On the relative dominance of paging algorithms. Theor. Comput. Sci. 410(38-40), 3694–3701 (2009)
16. Feuerstein, E., Strejilevich de Loma, A.: On-line multi-threaded paging. Algorithmica 32(1), 36–60 (2002)
17. Hassidim, A.: Cache replacement policies for multicore processors. In: Yao, A.C.C. (ed.) ICS. pp. 501–509. Tsinghua University Press (2010)
18. Irani, S.: Page replacement with multi-size pages and applications to web caching. In: Proceedings of STOC. pp. 701–710. ACM (1997)
19. Kaplan, S.F., Smaragdakis, Y., Wilson, P.R.: Flexible reference trace reduction for vm simulations. ACM Trans. Model. Comput. Simul. 13(1), 1–38 (2003)
20. López-Ortiz, A., Salinger, A.: Minimizing cache usage in paging. Tech. Rep. CS-2012-15, University of Waterloo (2012), http://www.cs.uwaterloo.ca/research/tr/2012/CS-2012-15.pdf
21. López-Ortiz, A., Salinger, A.: Paging for multi-core shared caches. In: Goldwasser, S. (ed.) ITCS. pp. 113–127. ACM (2012)
22. Miyatake, H., Tanaka, M., Mori, Y.: A design for high-speed low-power cmos fully parallel content-addressable memory macros. IEEE JSSC 36(6), 956 –968 (2001)
23. Moruz, G., Negoescu, A.: Outperforming lru via competitive analysis on parametrized inputs for paging. In: Proceedings of SODA. pp. 1669–1680 (2012)
24. Pagiamtzis, K., Sheikholeslami, A.: Content-addressable memory (CAM) circuits and architectures: A tutorial and survey. IEEE JSSC 41(3), 712–727 (2006)
25. Panagiotou, K., Souza, A.: On adequate performance measures for paging. In: Proceedings of STOC. pp. 487–496. ACM (2006)
26. Sleator, D.D., Tarjan, R.E.: Amortized efficiency of list update and paging rules. Commun. ACM 28(2), 202–208 (1985)
27. Torng, E.: A unified analysis of paging and caching. Algorithmica 20, 194–203 (1998)
28. Wagner, S.: Restricted Cache Scheduling. Ph.D. thesis, Michigan State Univ. (2001)
29. Young, N.E.: On-line file caching. In: Proc. of SODA. pp. 82–86. SIAM (1998)