Knowledge States for the Caching Problem in Shared Memory Multiprocessor Systems

Wolfgang Bein and Lawrence L. Larmore*
School of Computer Science
University of Nevada, Las Vegas
bein@cs.unlv.edu, larmore@cs.unlv.edu

Rüdiger Reischuk
Institut für Theoretische Informatik
Universität zu Lübeck, Germany
reischuk@tcs.uni-luebeck.de

Abstract

Multiprocessor systems with a global shared memory provide logically uniform data access. To hide latencies when accessing global memory each processor makes use of a private cache. Several copies of a data item may exist concurrently in the system. To guarantee consistency when updating an item a processor must invalidate copies of the item in other private caches. To exclude the effect of classical paging faults, one assumes that each processor knows its own data access sequence, but does not know the sequence of future invalidations requested by other processors. Performance of a processor with this restriction can be measured against the optimal behavior of a theoretical omniscient processor, using competitive analysis. A $\frac{4}{3}$-competitive randomized online algorithm for this problem for cache size of 2 is presented. This algorithm is derived with the help of a new concept we call knowledge states. We also prove a matching lower bound, thus this online algorithm is best possible. Finally, a lower bound of $\frac{3}{2}$ on the competitiveness for larger cache sizes is shown.

1 Introduction

Modern computer architectures use memory hierarchies to speedup the data access. There is a large body of work for caching problems for single processors, where a bounded cache size and a sequence of data requests is given, see *e.g.* [8] for a survey. When fresh data has to be loaded into a full cache some of the data stored in the cache has to be evicted in order to make space. Since data requests are typically not known in advance a caching strategy has to speculate about future data requests and decide what data item to evict in order to minimize the number of cache misses. This gives rise to an online problem, and the competitive ratio of different online strategies has been compared to the optimal offline algorithm which knows the complete sequence of future requests.

For similar reasons in a parallel system there have to exist local caches for each processor to mask the delay of accessing global memory, even if this is shared random access. (See also Figure 1.) In fact, it may be necessary to store several copies of a data item in different local caches, otherwise processors might slow each other down significantly.

To make this more precise, let k denote the size of a local cache, that is the maximal number of data items or pages which can be stored. Each processor P manages its own cache. But whenever

*Research of these authors supported by NSF grant CCR-0132093. Wolfgang Bein prepared part of this work as Kyoto University Visiting Professor.

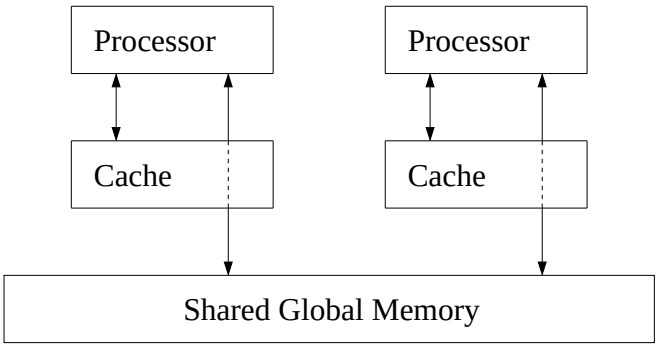

Figure 1. Caching in a Multiprocessor System

another processor P' changes a data item x, which has a copy in the cache of P, this copy of x has to be invalidated in order to guarantee consistency of the computation. Smaller parallel systems may use some sort of snooping such that P learns the new value of x immediately and updates its cache, but for a larger number of processors there is not enough bandwidth for such a procedure. In this case a better strategy is to store pointers to cached copies in global memory and to send invalidation messages to such caches. Hence there can be two reasons for a cache miss: Either, P can experience a "miss" in the usual way, *i.e.*, because data x needed by the processor was not among any of its k cached items, or x has been in the cache, but an invalidation caused by some other processor has occurred before P can access this item. This we call an *invalidation miss*.

The goal of our research is to measure the effect of this parallel setting on the online complexity of caching. With respect to worst case complexity, the competitive ratio cannot increase: It is known that already for the single processor case the ratio equals k and it is not difficult to see that it cannot increase further in a parallel environment. However, this worst case ratio of k is considerably too pessimistic for most real-world applications. The question becomes: How can one investigate the additional complexity caused by invalidation misses?

We propose to consider an intermediate, albeit slightly artificial, model in which private cache misses can be avoided as much as possible. Given the sequence ρ of requested items a processor P knows ρ in its entirety, only invalidations caused by other processors are not known beforehand and are given online. We call this the *OFI* (Oblivious to Future Invalidations) restriction. Thus, an OFI online algorithm can minimize private cache misses to the same extent as an offline algorithm if there were no invalidation misses. Invalidation, however, cannot be foreseen in the OFI setting.

For example, let the cache contain items a and b and the next requests be c, a in that order. Since c produces a cache miss one of the items a or b has to be evicted. If we know that a will be requested again later it is obviously better to keep a. Indeed, this will be the action of the optimal offline algorithm as well as of an OFI strategy. Now assume the request sequence to be $\rho = c, \mathrm{invalidate}(a), a, b$. This means that before another access to a there will be an invalidation of this data item. An OFI caching strategy (which does not know this fact) will still keep a and evict b in order to make space for c. However, an offline algorithm can deduce that it makes no sense to keep a since it will be invalidated before its next access. By keeping b instead, the offline algorithm has only 2 cache misses on ρ, whereas the OFI strategy incurs 3.

Our model gives insight how much this lack of information may increase cache misses: We estimate the competitive ratio of (a) the performance of a single processor caching strategy in a multiprocessor system which knows its own future data requests against (b) an optimal offline algorithm that has complete knowledge of the whole system. Thus we view the *OFI* model as mainly an analytic tool to

differentiate between the two types of cache misses. Other models which involve partial information about the future include *lookahead* for the paging problem [2], [3], [9], [14], and competitive implementation of parallel programs [11]. However, our model is specifically geared to the multiprocessor case.

Genther and Reischuk [12, 13] have already studied different deterministic caching strategies in the *OFI* setting. They give upper and lower bounds for the competitive ratio between deterministic *OFI* caching strategies and the optimal offline algorithm – in particular for every k, a competitive ratio 2 can be achieved.

In this paper, we consider the generalization of this problem to randomized *OFI* caching. Using a novel technique, the *knowledge state approach*, we achieve better bounds in the randomized case. We will briefly describe this technique in Section 2. In Section 3, we give a knowledge state algorithm with a competitive ratio of $\frac{4}{3}$ for $k = 2$. Section 4 and 5 give lower bounds. For $k = 2$ we show that the ratio $\frac{4}{3}$ is best possible, thus establishing that the knowledge state algorithm is optimally competitive. For $k \geq 3$, we show a lower bound of $\frac{3}{2}$. We conjecture that for $k = 3$ there exist a knowledge state algorithm that matches this lower bound.

2 Distributions, Work Functions and Knowledge States

We assume the reader to be familiar with the basic concepts of online algorithms and competitive analysis (see for example [8]). Competitiveness makes sense as a concept when an algorithm lacks timely access to all input data. If $\mathcal{A}$ is an algorithm for a given minimization problem we say that $\mathcal{A}$ is *C-competitive*, where $C \geq 1$ is some constant factor, if for every instance, the cost paid by $\mathcal{A}$ does not exceed C times the minimal cost for that instance plus some fixed constant λ. If $\mathcal{A}$ is randomized the cost of $\mathcal{A}$ is its expected cost on the instance.

We remind the reader that many randomized online algorithms are given in a so called distributional form, including a number of well known paging algorithms, *e.g.* the algorithm EQUITABLE, [1], [7]. For randomized online algorithms against an oblivious adversary, the distribution model is equivalent to another form of description called behavioral model. For the k-cache problem, such an algorithm is essentially a state transition diagram, where each state is a probabilistic distribution of configurations. More precisely, a *configuration* is simply an unordered k-tuple of pages, and represents a possible cache configuration. Then, a transition from one state to the next state is a deterministic transition to a new distribution. Unfortunately, the number of configurations in each state (and hence the number of states) can increase arbitrarily. One way to avoid this is to allow non-deterministic transitions and we note that we have a great degree of freedom in designing such a state transition diagram. In the standard distribution model, the algorithm deterministically chooses a distribution at each step, but in this paper we allow the algorithm to use randomization to choose the distribution. This variation, called the *mixed model* of randomized algorithms is a generalization of both the behavioral model and the distributional model.

On the other hand, so that competitiveness can be determined, estimates on optimal costs have to be maintained. One tool useful in the analysis of online algorithms, and which plays a role in our problem, is the concept of a *work function*. For a request sequence, the work function gives the optimal cost up to the current request and ending in cache configuration $x \in \mathcal{X}$. Thus work functions provide information about the optimal cost of serving the past request sequence and can be used as estimators. More precisely, let $\mathcal{X}$ denote the set of all configurations. Then, for a request sequence ρ, by $\omega^\rho(x)$, we denote the minimum cost of serving ρ and ending in configuration $x \in \mathcal{X}$. When the request sequence is understood, we omit the superscript ρ and simply write $\omega(x)$. We define function

$\omega \wedge r$ as $(\omega \wedge r)(y) = \min_{x \in \mathcal{X}} \{\omega(x) + \textit{cost}(x, r, y)\}$ where $cost(x, r, y)$ denotes the cost of serving r given cache configuration x while resulting in configuration y. The reader is reminded that in order to calculate (and update) the work function one can use dynamic programming. In such a dynamic program, only values for configurations $x \in \mathcal{X}$ with

$$\omega(x) = \min_x \omega(x)$$

need to be kept, since all other values can be reconstructed from these *support configurations*. In this context, it is also convenient to do the following: Let $\textit{MIN} = \min_x \omega(x)$. Then consider the function $(\omega - \textit{MIN})$. This function, which is called the *offset function*, is non-negative and has value 0 on all support configurations. Thus, a work function can be recorded by giving its support set together with the value *MIN*, which we call the *offset*.

With this we introduce a convenient notation, taken from [15], for offset functions for the k-cache problem, which we call the *bar notation*. Let w be a string consisting of at least k page names and exactly k bars, with the condition that at least i page names are to the left of the i^{th} bar. Then w defines an offset function ω as follows: If x is any configuration of pages such that, for each $i = 1, \ldots k$, the names of at least i members of x are written to the left of the i^{th} bar, then $\omega(x) = 0$. We call configurations which satisfy this condition the *support set* of ω. If $y \in \mathcal{X}$ does not satisfy this condition, then $\omega(y)$ is the number of page replacements necessary to change y to a configuration which is a member of the support set. For example, if $k = 2$, $ab||$ denotes the offset function whose support set consists of just the configuration $\{a, b\}$, while if $k = 4$, $ab||cd|ef|$ denotes the offset function whose support consists of the configurations $\{a, b, c, d\}$, $\{a, b, c, e\}$, $\{a, b, c, f\}$, $\{a, b, d, e\}$, and $\{a, b, d, f\}$. From [15], we have:

Lemma 1 *A function ω is an offset function for the k-cache problem if and only if it can be expressed using the bar notation.*

Central to our discussion is a novel technique in competitive analysis: the knowledge state approach. It incorporates the two elements just described: non-deterministic transitions as well as estimates on the offline cost. We summarize that technique briefly in this section. We refer the reader to [4] for a detailed description of the technique.

A *knowledge state algorithm* [4, 6] is a mixed online algorithm that computes an *adjustment* and an *estimator* at each step. The estimator (together with the adjustment) is a real-valued function on configurations that is updated at every step, and which estimates the cost of the optimal offline algorithm. More formally, if $\mathcal{A}$ is a knowledge-state algorithm, then:

1. At any given step, the state of $\mathcal{A}$ is a pair (ω, π), where π is a finite distribution on $\mathcal{X}$, and $\omega : \mathcal{X} \to \mathbb{R}$ is the current estimator. We call this pair the *current knowledge state.*

2. If $S = (\omega, \pi)$ is the knowledge state and the next request is r, then $\mathcal{A}$ computes an adjustment, a number which we call $\textit{adjust}_{\mathcal{A}}(S, r)$, and uses randomization to pick a new knowledge state $S' = (\omega', \pi')$. More precisely, there are subsequent knowledge states $S_i = (\omega_i, \pi_i)$ and subsequent positive weights λ_i for $i = 1, \ldots m$, $\sum_{i=1}^m \lambda_i = 1$, such that

   (a) $(\omega \wedge r)(x) \geq \textit{adjust}_{\mathcal{A}}(S, r) + \sum_{i=1}^m \lambda_i \omega_i(x)$ for each $x \in \mathcal{X}$.

   (b) For each i, $\mathcal{A}$ chooses S' to be S_i with probability λ_i.

In our application, we use offset functions (in bar notation) as estimators. We will also keep a small finite amount of information about the future private request sequence with each offset function[1]. When

[1]This requires generalizing our model slightly; we note that our method is also valid in this case (see [6].)

there is a request, the algorithm computes a new offset function and selects a new distribution, but may also need to "query the future" to decide the new knowledge state. (Note that the algorithm does not obtain new information in this query. The algorithm has the entire private request sequence available, and "querying the future" means simply looking at this memory.)

3 A $\frac{4}{3}$-Competitive Algorithm for $k = 2$

Without loss of generality, each invalidation of a page a takes place immediately before a is requested since this is the worst case scenario for OFI-page faults in multiprocessor systems. The request of a preceded by an invalidation is denoted by $\hat{a}$, while a request of a with no invalidation before we simply write as a. For example, if $\hat{\rho} = a\hat{b}\hat{c}ba\hat{d}\hat{b}a$ is the sequence of future requests, then only the sequence of its private future requests, $\rho = abcbadba$, is known to the OFI-online-algorithm. In the first step serving a, the algorithm must decide on evicting a page without knowing whether the next request is of the form b or $\hat{b}$. If it is $\hat{b}$ it would not make sense to keep b. Thus if we have to make room for a, and b is in the cache at the moment an optimal offline algorithm knowing that $\hat{b}$ comes next will evict b. We require that the request sequence $\hat{\rho}$ must be *consistent* with the private request sequence ρ, meaning that ρ can be obtained from $\hat{\rho}$ by replacing every $\hat{a}$ by a, for any page a.

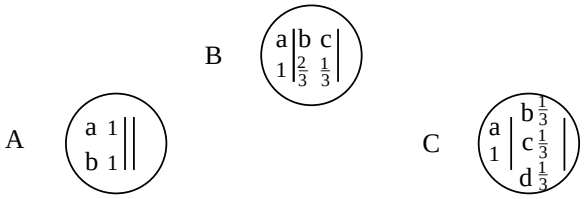

Figure 2. The Three Knowledge States A, B , C

Let us now define the four parameterized knowledge states, which we name A_{ab}, B_{abc}, C_{abcd}, and D_{abcde}, where a, b, c, d, e are arbitrary pages. A_{ab}, B_{abc}, and C_{abcd} are illustrated in Figure 2. The knowledge state D_{abcde}, which is only used as a transitory state, appears in Figure 5.

If a and b are pages, the notation "$a < b$" means that, if a and b both appear in the future private request sequence (which is known to the algorithm), then the next instance of a precedes the next instance of b, or that b never appears in the future.

1. $A_{ab} = (ab||, \pi^A_{ab})$, where π^A_{ab}, is defined to be 1 on the pair $\{a, b\}$. The knowledge state A_{ab} is the same as the knowledge state A_{ba}.

   Our full notation for the knowledge state A_{ab} is $\begin{smallmatrix} a & 1 \\ b & 1 \end{smallmatrix}||$, where we write the page names vertically instead of horizontally to indicate that we do not care whether $a < b$.

2. $B_{abc} = (a|bc|$ and $b < c, \pi^B_{abc})$, where π^B_{abc} is defined to be $\frac{2}{3}$ on $\{a, b\}$ and $\frac{1}{3}$ on $\{a, c\}$. Besides the offset function $a|bc|$, the knowledge state contains the information that $b < c$.

   Thus, $B_{abc} \neq B_{acb}$, despite the fact that they have the same offset function.

   Our full notation for this knowledge state is $B_{abc} = \begin{smallmatrix} a \\ 1 \end{smallmatrix}\left|\begin{smallmatrix} b & c \\ \frac{2}{3} & \frac{1}{3} \end{smallmatrix}\right|$, where the placing of c horizontally after b indicates that $b < c$.

3. $C_{abcd} = (a|bcd|, \pi^C_{abcd})$, where π^C_{abcd} is $\frac{1}{3}$ on each of the three states, $\{a, b\}$, $\{a, c\}$, and $\{a, d\}$. Note that $C_{abcd} = C_{abdc} = C_{acbd} = C_{acdb} = C_{adcb} = C_{adbc}$, since this knowledge state contains no information about the future order of requests.

The full notation for this knowledge state is $C_{abcd} = \begin{matrix} a \\ 1 \end{matrix} \left| \begin{matrix} b\frac{1}{3} \\ c\frac{1}{3} \\ d\frac{1}{3} \end{matrix} \right|$

4. $D_{abcde} = (a|bcde|, \pi^D_{abcde})$, where π^D_{abcde} is $\frac{1}{4}$ on each of the states $\{a, b\}$, $\{a, c\}$, $\{a, d\}$, and $\{a, e\}$.

   The full notation for this knowledge state is $D_{abcde} = \begin{matrix} a \\ 1 \end{matrix} \left| \begin{matrix} b\frac{1}{4} \\ c\frac{1}{4} \\ d\frac{1}{4} \\ e\frac{1}{4} \end{matrix} \right|$ The knowledge state D is transitory, meaning that when our algorithm enters it it immediately leaves it, as shown in Figure 5.

Theorem 1 *For cache size* 2*, there exists an online algorithm* $\mathcal{A}$ *achieving a competitiveness bounded by* $\frac{4}{3}$.

Proof: We will use a standard potential argument to prove competitiveness, and thus we will need to associate a potential Φ with each knowledge state. We now define the *update condition* for a given step. Fix $C > 1$. Let S^{t-1} be the knowledge state after $t-1$ steps, let $\{U_i\}$ be the subsequents for step t, and λ_i be the probability that U_i will be chosen to be S^t. We define *adjust* to be the expected adjustment of this step, the minimum value of the difference between the updated work function and the expected work function after the Las Vegas step, and $cost_{\mathcal{A}}$ to be the expected cost of the algorithm $\mathcal{A}$. Then the update condition is that

$$\Phi(S^{t-1}) \;\geq\; cost_{\mathcal{A}} - C \cdot adjust + \sum_i \lambda_i \, \Phi(U_i) \; .$$

We will make use of the following lemma from [6]:

Lemma 2 *If the update condition holds at every step of an online algorithm then it is* C*-competitive.*

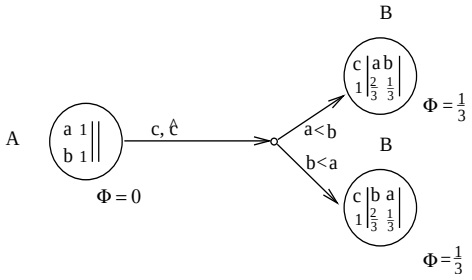

Figure 3. Transition from Knowledge State A

Fix now $C = \frac{4}{3}$. We specify the C-competitive algorithm $\mathcal{A}$, simultaneously verifying the update condition at each step. $\mathcal{A}$ uses the 4 knowledge states defined above with potentials

$$\Phi(A_{ab}) = 0, \;\; \Phi(B_{abc}) = \tfrac{1}{3}, \;\; \Phi(C_{abcd}) = \tfrac{2}{3}, \;\; \phi(D_{abcde}) = 1 \; .$$

The actions of $\mathcal{A}$ in each knowledge state are as follows.

1. If the knowledge state is A_{ab}, B_{abc}, or C_{abcd}, and the request a nothing happens (not illustrated).
2. If the knowledge state is A_{ab}, B_{abc}, or C_{abcd}, and the request $\hat{a}$, the algorithm ejects a and later moves (the updated version of) a back into the cache. It holds that $cost_{\mathcal{A}} = adjust = 1$, and the knowledge state remains the same (not illustrated).

3. If the knowledge state is A_{ab} and the request c or $\hat{c}$, $\mathcal{A}$ considers its private request sequence. If $a < b$ it moves to the knowledge state B_{cab}, else to B_{cba}. In both cases, *cost*$_{\mathcal{A}} =$ *adjust* $= 1$, and Φ increases by $\frac{1}{3}$, thus the update condition is exactly satisfied (see Figure 3).

4. If the knowledge state is B_{abc} and the request is b, the new knowledge state is A_{ab}. Then $cost_{\mathcal{A}} = \frac{1}{3}$, since the probability is $\frac{2}{3}$ that our cache state is already ab, and *adjust* $= 0$. The potential decreases by $\frac{1}{3}$ (see Figure 4).

5. If the knowledge state is B_{abc} and the request $\hat{b}$, $\mathcal{A}$ considers its private request sequence. If $a < c$ it moves to the knowledge state B_{bac}. Then *cost*$_{\mathcal{A}} =$ *adjust* $= 1$, and the potential remains the same. Otherwise, the algorithm moves to the knowledge state B_{bca}. Then *cost*$_{\mathcal{A}} = \frac{4}{3}$, *adjust* $= 1$ without changing the potential (see Figure 4). In either case, the update condition is satisfied.

   Note that in B_{abc}, a request of c or $\hat{c}$ is impossible since $b < c$.

6. If the knowledge state is B_{abc} and the request is a new page d or $\hat{d}$, the algorithm moves to C_{dabc}. Then *cost*$_{\mathcal{A}} =$ *adjust* $= 1$, and Φ increases by $\frac{1}{3}$, thus the update condition is exactly satisfied (see Figure 4).

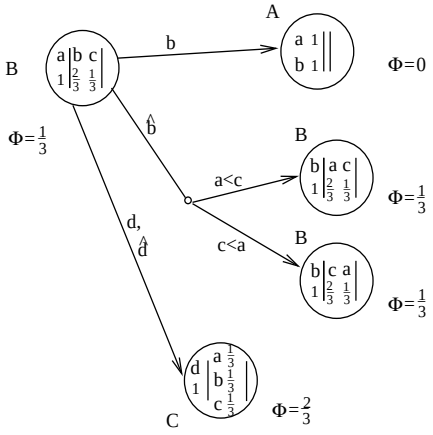

Figure 4. Transition from Knowledge State B

7. If the knowledge state is C_{abcd} and the request is either b, c, d, or $\hat{b}$, $\hat{c}$, or $\hat{d}$, without loss of generality, it is either b or $\hat{b}$, since no special distinguishing information about b, c, d has been recorded in the knowledge state. In the case of b, $\mathcal{A}$ moves to knowledge state A_{ab}. Then *cost*$_{\mathcal{A}} = \frac{2}{3}$, *adjust* $= 0$, and the update condition is exactly satisfied (see Figure 5). If the request is $\hat{b}$, $\mathcal{A}$ moves to knowledge state C_{bacd}. Then *cost*$_{\mathcal{A}} =$ *adjust* $= 1$, the potential remaining the same (see Figure 5).

8. If the knowledge state is C_{abcd} and the request is e or $\hat{e}$, where e is a new page, the algorithm first moves to the transitory knowledge state D_{eabcd}. For this part, *cost*$_{\mathcal{A}} =$ *adjust* $= 1$, and Φ increases by $\frac{1}{3}$. Then the algorithm moves to one of the four knowledge states A_{ae}, A_{be}, A_{ce}, or A_{de}, choosing each with probability $\frac{1}{4}$. For this part, by Lemma 3 below, *cost*$_{\mathcal{A}} = 0$, while, by Lemma 4 below, *adjust* $= -\frac{3}{4}$, and the potential decreases by 1. Thus, the update condition is satisfied exactly (see Figure 5).

Lemma 3 *In Case 8, the cost of algorithm $\mathcal{A}$ is 0.*

Proof: Although in the behavioral model randomization must be used to choose one of the A's, in the distributional model the choice is deterministic. If the algorithm's state is ab it chooses the knowledge state A_{ab}. If the algorithm's state is ac it chooses the knowledge state A_{ac}. If the algorithm's state is ad it chooses the knowledge state A_{ad}. If the algorithm's state is ae it chooses the knowledge state A_{ae}. Thus, the movement cost is 0. □

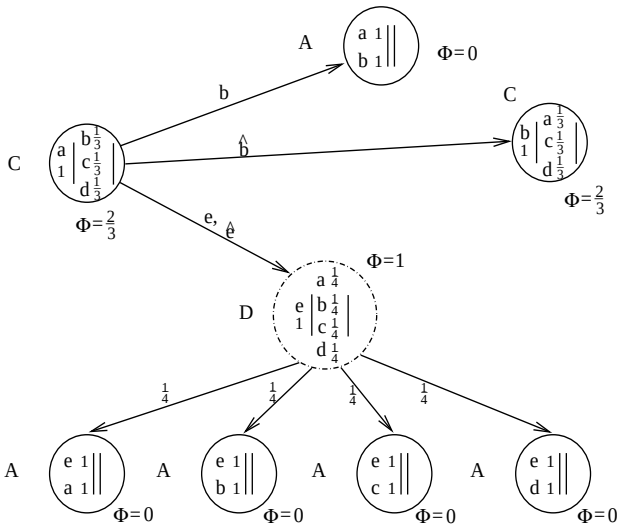

Figure 5. Transition from Knowledge State C

Lemma 4 *In Case 8, the adjustment is* $-\frac{3}{4}$.

Proof: The offset function before the Las Vegas step is $a|bcde|$. The offset function after the Las Vegas step is either $ab||$, $ac||$, $ad||$, or $ae||$, each with probability $\frac{1}{4}$. The adjustment is thus, by definition, the maximum value of the function $a|bcde| - \frac{1}{4}ab|| - \frac{1}{4}ac|| - \frac{1}{4}ad|| - \frac{1}{4}ae||$. This function has the same value $-\frac{3}{4}$ on each of the four pairs. □

Note that in every case in the above list, the cost of $\mathcal{A}$ plus the increase in potential is less than or equal to $\frac{4}{3}$ times the adjustment. This completes the proof of Theorem 1. □

We have only given a distributional description of algorithm $\mathcal{A}$. One has to translate it into a behavioral description in order to implement the algorithm. That this can be done one has to check that the transitions between the different distributions of the knowledge states are indeed possible. We omit these simple calculations here.

4 A Lower Bound for Cache Size 2

Theorem 2 *For cache size* 2*, the competitiveness of any randomized online algorithm for the cache coherent data access problem is at least* $\frac{4}{3}$.

Proof: Let us consider a scenario where there are just three pages altogether, which we call a, b, and c. Let $\mathcal{A}$ be any randomized online algorithm. We give a randomized adversary which forces $\mathcal{A}$'s expected cost to be at least $\frac{4}{3}$ the optimal cost.

Our adversary picks the private request sequence $\rho = (abc)^n$ for some large n, and then chooses a request sequence $\hat{r}ho$ consistent with ρ using randomization, as follows. The adversary first picks a random string Γ of sufficient length (roughly $\frac{3n}{2}$ will be sufficient) over the alphabet $\{2, 3\}$. Each symbol of Γ is picked independently and uniformly. Γ is then used as a guide to pick $\hat{\rho}$.

The sequence $\hat{\rho}$ will consist of *phases*, where each phase has length either 2 or 3 determined by the i^{th} symbol of Γ. The last request in each phase is a simple page load (that is of type a), all other requests are page loads with an invalidation of that page before (that means of type $\hat{a}$). For example, if $\Gamma = 2332322$, then $\hat{\rho} = \hat{a}b,\ \hat{c}\hat{a}b,\ \hat{c}\hat{a}b,\ \hat{c}a,\ \hat{b}\hat{c}a,\ \hat{b}c,\ \hat{a}b$, where the commas separate the phases.

Our particular setting implies that an optimal offline algorithm can always satisfy the page load without a cache miss. Thus, depending on the length of a phase it makes either 1 or 2 pages faults. The expected number of page faults then equals $\frac{3n}{2}$.

By symmetry, let $\hat{a}$ be the first request of a phase. Then the online algorithm $\mathcal{A}$ will be in state ab or state ac after serving it. If $\mathcal{A}$ is in state ac then its cost of serving the whole phase is at least 2 since the next request will generate a page fault for sure. If $\mathcal{A}$ is in state ab its cost of serving the phase is 1 if the phase has length 2, and 3 if the phase is $\hat{a}\hat{b}c$. Thus, the expected cost of servicing one phase is at least 2. This yields a lower bound $\frac{4}{3}$ on the competitiveness of $\mathcal{A}$. □

Combining Theorems 1 and 2, we obtain:

Theorem 3 *For cache size* 2*, the randomized competitiveness of the cache coherent data access problem is* $\frac{4}{3}$.

5 A Lower Bound for Larger Cache Sizes

We say that an online algorithm $\mathcal{A}$ for the cache coherent data access problem is *lazy* if, given a request to any page, $\mathcal{A}$ does not move any page other than the requested page into its cache.

Lemma 5 *Without loss of generality, an online algorithm for the cache coherent data access problem is lazy.*

Proof: Suppose $\mathcal{A}$ is any online algorithm. Define a lazy online algorithm $\hat{\mathcal{A}}$ as follows: $\hat{\mathcal{A}}$ never moves a page into the cache unless it is the page requested at that step. If $\hat{\mathcal{A}}$ is forced to eject a page, it chooses to eject a page in such a way as to match its cache to $\mathcal{A}$'s cache as much as possible subject to the lazy condition. Let D^t be the difference between $\mathcal{A}$'s cache and $\hat{\mathcal{A}}$'s cache after t steps, *i.e.*, the number of pages in $\mathcal{A}$'s cache that are not in $\hat{\mathcal{A}}$'s cache. Let $Cost^t$ and $\widehat{Cost}^t$ be the total costs of $\mathcal{A}$ and $\hat{\mathcal{A}}$, respectively, for the first t steps. The loop invariant $\widehat{Cost}^t \leq Cost^t - D^t$ holds for all t, as can be routinely verified by considering several cases. thus, the expected total cost of $\hat{\mathcal{A}}$ is no greater than the expected total cost of $\mathcal{A}$ for any given request sequence. □

We can also prove a lower bound of $\frac{3}{2}$ on the competitiveness of the problem for $k = 3$, which implies the same lower bound for $k > 3$, by a simple reduction. To prove the result we use a standard potential argument but we note that using a computer program, we established the minimum value of C to be $\frac{3}{2}$, and also to identify a minimal set of variables and inequalities needed to prove the lower bound. The actual proof is short, and requires defining five quantities (similar to α, β, γ, δ) and proving five inequalities, similar to the four inequalities given in the proof of the lower bound for $k = 2$. But that simple proof belies the complexity of the problem especially for values of k larger than 3. However, the resulting proof given below can be understood in its entirety without the use of a computer.

We first assume that there exists an algorithm $\mathcal{A}$ which is C-competitive for some C. By Lemma 5, $\mathcal{A}$ is lazy. We use the distributional model to describe $\mathcal{A}$. In order to prove our lower bound, we need consider only four pages, which we name a, b, c, d. Let the public request sequence be $(abcd)^N$, where N is assumed to be arbitrarily large.[2] We assume that the initial cache is $\{b, c, d\}$. Let s^t be the public request at step t, and let ω^t be the offset function at step t. Let r^t be the actual request at step t;

[2]The proof given below seems to assume that N is infinite. In fact, this problem is an easily resolved technical detail.

therefore either $r^t = s^t$ or $r^t = \hat{s}^t$. Given this public request sequence, there are only twelve possible offset functions, which can be classified into three types:

$$A = \{cab|||, dcb|||, adc|||, bad|||\}, \qquad B = \{ba||dc|, cb||ad|, dc||ba|, ad||cb|\},$$
$$C = \{a|dcb||, b|adc||, c|bad||, d|cba||\}.$$

Let π^t be the distribution of $\mathcal{A}$ at step t. Let Φ be a C-potential for $\mathcal{A}$, and let Φ^t be the potential of $\mathcal{A}$ at step t. Let *cost*^{t} be the cost incurred by $\mathcal{A}$ at step t. Although technically, π^t is a distribution on triples, we abuse notation by writing π^t_s to be the probability that a given page s is in the cache. Thus, $\pi^t_{s^t} = 1$, and $\pi^t_a + \pi^t_b + \pi^t_c + \pi^t_d = 3$ for all t. Furthermore, let *adjust*^{t} be $\max_s \left\{ \left(\omega^{t-1} \wedge r^t\right)(s) - \omega^t(s) \right\}$, where the maximum is taken over $s \in \{a, b, c, d\}$. Then we have

Lemma 6 *Let $t > 0$. Then*

If $r^t = \hat{s}^t$, then $\omega^t \in C$.

If $r^t = s^t$ and $\omega^{t-1} \in A$, then $\omega^t \in C$.

If $r^t = s^t$ and $\omega^{t-1} \in B$, then $\omega^t \in A$.

If $r^t = s^t$ and $\omega^{t-1} \in C$, then $\omega^t \in B$.

For $t \geq 1$, if $r^t = \hat{s}^t$ or ω^{t-1} is of type A, then adjust$^t = 1$. Otherwise, adjust$^t = 0$.

For $t \geq 0$, cost$^{t+1} + \Phi^{t+1} \leq \Phi^t + C \cdot$ adjust^{t+1}.

Proof: The proof is by routine calculation. □

With these preliminaries we are now ready to formulate our potential argument. To this end, we define the following quantities:

$$\alpha = \min_{\omega^t \in A} \left\{ \Phi^t + \pi^t_{s^{t+2}} \right\} \quad \beta = \min_{\omega^t \in B} \left\{ \Phi^t \right\}$$
$$\gamma = \min_{\omega^t \in B} \left\{ \Phi^t + \pi^t_{s^{t+2}} \right\} \quad \delta = \min_{\omega^t \in B} \left\{ \Phi^t - \pi^t_{s^{t+2}} \right\}$$
$$\epsilon = \min_{\omega^t \in C} \left\{ \Phi^t + \pi^t_{s^{t+1}} \right\}$$

The following claims will yield our result immediately. For the proof we simply make use of Lemma 6, as well as the definitions of the quantities themselves:

Claim 1 $\alpha \leq \delta + 1$. **Claim 2** $\beta + 1 \leq \epsilon$. **Claim 3** $\delta + \gamma \leq 2\beta$.
Claim 4 $\epsilon + 1 \leq \alpha + C$. **Claim 5** $\epsilon + 1 \leq \gamma + C$.

Proof: (Of Claim 1). Pick some t such that $\omega^t \in B$ and $\delta = \Phi^t - \pi^t_{s^{t+2}}$. Without loss of generality, $\omega^t = ba||dc|$. Let $r^{t+1} = c$. Then $s^{t+2} = d$, *adjust*$^{t+1} = 0$, and *cost*$^{t+1} = 1 - \pi^t_c$. Thus

$$\begin{aligned} \Phi^t - \pi^t_d &= \delta \\ \alpha &\leq \Phi^{t+1} + \pi^t_a \\ \Phi^{t+1} + 1 - \pi^t_c &\leq \Phi^t + C \\ \pi^{t+1}_a &\leq \pi^t_a \\ \pi^t_a + \pi^t_c + \pi^t_d &= 2 \end{aligned}$$

□

Proof: (Of Claim 2.) Pick some t such that $\omega^t \in C$ and $\epsilon = \Phi^t + \pi^t_{s^{t+1}}$. Without loss of generality, $\omega^t = a|dcb||$. Suppose $r^{t+1} = b$. Then $s^{t+2} = c$, *adjust*$^{t+1} = 0$, and *cost*$^{t+1} = 1 - \pi^t_b$. Thus

$$\begin{aligned} \Phi^t + \pi^t_b &= \epsilon \\ \beta &\leq \Phi^{t+1} \\ \Phi^{t+1} + 1 - \pi^t_b &\leq \Phi^t \end{aligned}$$

Combining the above inequalities, we obtain the result. □
Proof: (Of Claim 3.) Pick t such that $\omega^t \in B$ and $\beta = \Phi^t$. Then

$$\begin{aligned} 2\beta &= \Phi^t + \pi^t_{s^{t+2}} + \Phi^t - \pi^t_{s^{t+2}} \\ &\geq \gamma + \delta \end{aligned}$$

Combining the above inequalities, we obtain the result. □ *Proof:* (Of Claim 4.) Pick some t such that $\omega^t \in A$ and $\alpha = \Phi^t + \pi^t_{s^{t+2}}$. Without loss of generality, $\omega^t = cba|||$. Then $s^{t+2} = a$, *adjust*$^{t+1} = 1$, and *cost*$^{t+1} = 1$. Thus

$$\begin{aligned} \Phi^t + \pi^t_a &= \alpha \\ \epsilon &\leq \Phi^{t+1} + \pi^{t+1}_a \\ \Phi^{t+1} + 1 &\leq \Phi^t + C \\ \pi^{t+1}_a &\leq \pi^t_a \end{aligned}$$

Combining the above inequalities, we obtain the result. □
Proof: (Of Claim 5). Pick some t such that $\omega^t \in B$ and $\gamma = \Phi^t + \pi^t_{s^{t+2}}$. Without loss of generality, $\omega^t = ba||dc|$. Suppose $r^{t+1} = \hat{c}$. Then $s^{t+2} = d$, *adjust*$^{t+1} = 1$, and *cost*$^{t+1} = 1$. Thus

$$\begin{aligned} \Phi^t + \pi^t_d &= \gamma \\ \epsilon &\leq \Phi^{t+1} + \pi^{t+1}_d \\ \Phi^{t+1} + 1 &\leq \Phi^t + C \\ \pi^{t+1}_d &\leq \pi^t_d \end{aligned}$$

Combining the above inequalities, we obtain the result. □

Combining the five claims yields the inequality $3 \leq 2C$. Thus:

Theorem 4 *There is no online algorithm for the cache coherent data access problem for $k = 3$ whose competitiveness is less than $\frac{3}{2}$.*

5.1 Lower Bound for $k > 3$

Theorem 5 *There is no online algorithm for the cache coherent data access problem whose competitiveness is less than $\frac{3}{2}$.*

Proof: Our lower bound proof, as earlier, is obtained by considering a specific public request sequence, where the overall number of pages is exactly $k + 1$. Let $a, b, c, d, x_4, x_5, \ldots x_k$ be pages. We call x_i, for $i \geq 4$, *dummy pages*. We let the initial cache be $\{b, c, d, x_4, x_5, \ldots x_k\}$, and the public request sequence be $(ax_4x_5 \ldots x_k bx_4x_5 \ldots x_k cx_4x_5 \ldots x_k dx_4x_5 \ldots x_k)^N$ for arbitrarily large N. Consider

any randomized adversary which completely ignores the dummy pages; *i.e.*, it behaves just like an optimal adversary for $k = 3$ on the subsequence $(abcd)^N$, and never ejects a dummy page. A *normal* request is defined to be a request which is not a request to a dummy page.

Let $\mathcal{A}$ be a randomized online algorithm. We need to prove that, without loss of generality, $\mathcal{A}$ never ejects a dummy page. Without loss of generality, $\mathcal{A}$ is lazy. Choose an online algorithm $\hat{\mathcal{A}}$ as follows: $\hat{\mathcal{A}}$ is lazy and never ejects a dummy page. If forced to eject a page, $\hat{\mathcal{A}}$ chooses to eject a page in such a way as to match its cache to $\mathcal{A}$'s cache as much as possible subject to the above conditions.

Define D^t to be the difference between $\hat{\mathcal{A}}$'s cache and $\mathcal{A}$'s cache after t steps. Note that D^t must be either zero or one. Let Cost^t and $\widehat{\mathit{Cost}}^t$ be the total costs of $\mathcal{A}$ and $\hat{\mathcal{A}}$, respectively, for the first t steps. Notice that the request sequence can be partitioned into *phases* of length $k-2$ each. Each phase begins with one normal request, followed by $k-3$ requests to dummy pages. We need to prove two loop invariants:

1. Before and after each phase, $\widehat{\mathit{Cost}}^t \leq \mathit{Cost}^t - D^t$.
2. Immediately after each normal request, if $\mathcal{A}$ has all dummy pages in its cache, then $\widehat{\mathit{Cost}}^t \leq \mathit{Cost}^t - D^t$, while if $\mathcal{A}$ does not have all dummy pages in its cache, $\widehat{\mathit{Cost}}^t \leq \mathit{Cost}^t$.

We prove the each of two invariants by assuming the other holds for any $u < t$.

We first prove Invariant 1. It clearly holds initially. Suppose that $t = m(k-2)$ for $m > 0$, and let $u = t - k + 3$. Assume Invariant 2 holds at u. If $\mathcal{A}$ has all dummy pages in its cache at u, both caches are unchanged and both algorithms pay nothing during the dummy requests, so Invariant 1 holds at t. Otherwise, $\mathcal{A}$ pays at least 1 during the dummy requests, while $\widehat{\mathcal{A}}$ pays nothing, so Invariant 1 holds for t.

We now prove Invariant 2. Suppose that $t = m(k-2)+1$ for $m \geq 0$. Let $u = t - 1$. Assume that Invariant 1 holds for u. Suppose that $\mathcal{A}$ has all dummy pages in its cache at t. If s^t is in $\widehat{\mathcal{A}}$'s cache after step u, then $\widehat{\mathcal{A}}$ pays no more than $\mathcal{A}$ for the request at t, while if s^t is not in $\widehat{\mathcal{A}}$'s cache after step u, $\widehat{\mathcal{A}}$ pays at most 1, and can always arrange for D^t to be zero. On the other hand, suppose that $\mathcal{A}$'s cache is missing one dummy page after t steps. If $D^u = 0$, then $\widehat{\mathcal{A}}$ pays no more than $\mathcal{A}$ pays at step t. If $D^u = 1$, then $\widehat{\mathcal{A}}$ pays no more than 1 more than $\mathcal{A}$ pays at step t. Thus. Invariant 2 holds for t.

It follows immediately that $\widehat{\mathcal{A}}$ pays no more than $\mathcal{A}$ for the entire request sequence. We can now assume that the dummy pages are always remain in the cache of both the adversary and the algorithm, and contribute nothing to either's cost. It follows, from Theorem 4, that the competitiveness against this adversary is at least $\frac{3}{2}$. □

6 Conclusions

We mention that a forgiveness online algorithm is a knowledge state algorithm with the special restriction that there is always exactly one subsequent. It is interesting to note that historically, forgiveness came first, so we can think of the knowledge state approach as being a generalization of forgiveness. We mention that very recently, Bein *et al.* [7] have been able to improve the memory requirements of the algorithm EQUITABLE using such a forgiveness technique. The randomized k-paging algorithm EQUITABLE given by Achlioptas *et al.* is H_k-competitive and uses $O(k^2 \log k)$ memory. This competitive ratio is best possible. Borodin and El Yaniv [8] listed as an open question whether there exists an H_k-competitive randomized algorithm which requires $O(k)$ memory for k-paging: Bein *et al.* [7] have answered this question in the affirmative.

We note that the result of Section 3 is specific to the case $k = 2$, but it might be possible to obtain results for larger k using a well designed forgiveness algorithm. Unfortunately, there is not much slack since in the deterministic case a ratio of 2 can be achieved for all k. However, we do conjecture that a knowledge state algorithm with randomized competitiveness $\frac{3}{2}$ exists for cache size of 3.

Since our randomized results show better competitiveness than the deterministic results of [12] and [13], our work suggests that randomization indeed could be beneficial in practice.

References

[1] D. Achlioptas, M. Chrobak, and J. Noga. Competitive analysis of randomized paging algorithms. *Theoretical Computer Science*, 234:203–218, 2000.

[2] S. Albers. The influence of lookahead in competitive paging algorithms. In *Proc. 1st European Symp. on Algorithms*, LNCS 726, pages 1–12. Springer, 1993.

[3] S. Albers. A competitive analysis of the list update problem with lookahead. In *Proc. 19th Symp. on Mathematical Foundations of Computer Science*, LNCS , pages 201–210, Springer Verlag, 1994.

[4] W. Bein and L. Larmore. Trackless and Limited-Bookmark Algorithms for Paging. *ACM SIGACT News*, pages 38–48, 35(1), 2004.

[5] W. Bein, L. Larmore, and R. Reischuk. Knowledge states for the caching problem in shared memory multiprocessor systems. In *Proceedings of the 7th International Symposium on Parallel Architectures, Algorithms and Networks*, pages 307 – 312, IEEE, 2004.

[6] W. Bein, L. Larmore, and R. Reischuk. Knowledge state algorithms: Randomization with limited information. *Arxiv:* `archive.org/cs/0701142`, 2007. Submitted: *Discrete Applied Mathematics*.

[7] W. Bein, J. Noga, and L. Larmore. Equitable revisited. In *Proc. 15th European Symp. on Algorithms (ESA)*. Lecture Notes in Computer Science, Springer Verlag. to appear.

[8] A. Borodin and R. El-Yaniv. *Online Computation and Competitive Analysis*. Cambridge University Press, 1998.

[9] D. Breslauer. On competitive on-line paging with lookahead. In *Proc. 13th Symp. on Theoretical Aspects of Computer Science*, LNCS 1046, pages 593–603, 1996.

[10] M. Chrobak and L. Larmore. Metrical task systems, the server problem, and the work function algorithm. In *Online Algorithms: State of the Art*, pages 74–94, Springer-Verlag, 1998.

[11] Xi. Deng, E. Koutsoupias, and P. MacKenzie. Competitive implementation of parallel programs. *Algorithmica*, 23, 1999.

[12] K. Genther and R. Reischuk. Analyzing data access strategies in cache coherent architectures. Technical Report TR A-98-25, Universität zu Lübeck, Institut für Theoretische Informatik, 1999.

[13] K. Genther and R. Reischuk. Analyzing the competitive ratio of caching in multiprocessor systems. Technical Report TR-A-01-19, 2001, Universität zu Lübeck, Institut für Theoretische Informatik, 2001.

[14] E. Grove. Online binpacking with lookahead. In *Proc. 6th Symp. on Discrete Algorithms*, pages 430–436, 1995.

[15] E. Koutsoupias and C. Papadimitriou. Beyond competitive analysis. In *Proc. 35th Symp. Foundations of Computer Science*, pages 394–400, 1994.