Experimental Results on Statistical Approaches to Page Replacement Policies

Vitus Leung[1]* and Sandy Irani[2]**

[1] Sandia National Laboratories, Albuquerque, NM 87185-1110, USA,
vjleung@cs.sandia.gov

[2] University of California, Irvine, CA 92697-3425, USA,
irani@ics.uci.edu

Abstract. This paper investigates the questions of what statistical information about a memory request sequence is useful to have in making page replacement decisions. Our starting point is the Markov Request Model for page request sequences. Although the utility of modeling page request sequences by the Markov model has been recently put into doubt ([13]), we find that two previously suggested algorithms (Maximum Hitting Time [11] and Dominating Distribution [14]) which are based on the Markov model work well on the trace data used in this study. Interestingly, both of these algorithms perform equally well despite the fact that the theoretical results for these two algorithms differ dramatically. We then develop succinct characteristics of memory access patterns in an attempt to approximate the simpler of the two algorithms. Finally, we investigate how to collect these characteristics in an online manner in order to have a purely online algorithm.

1 Introduction

Motivated by increasing gaps at all levels in the hierarchy from CPU speed to network I/O speed, researchers have investigated page replacement algorithms in the hopes of improving upon the time-honored Least-Recently-Used (LRU). LRU is a purely *online* algorithm in that it requires no information about the future requests in order to make a decision about page replacement. Many of the recent approaches to page replacement policies have sought to use some extra information about the future request sequence in order to guide page replacement algorithms.

The assumption that the application has perfect knowledge about its page request behavior has been the starting point of recent work related to reducing memory access time. The issue of how to partition memory between different applications sharing a common cache has been examined by [6] and [1]. Both

* Sandia is a multiprogram laboratory operated by Sandia Corporation, a Lockheed Martin Company, for the United States Department of Energy under contract DE-AC04-94AL85000. Part of this work was done while at the University of California, Irvine and supported by NSF Grant CCR-9309456.

** Supported in part by NSF Grant CCR-9309456.

these papers assume that each individual application has perfect knowledge of its request sequence and only the interleaving of these request sequences is unknown in advance. Cao, Felten, Karlin and Li investigate the question of how to integrate page replacement and prefetching decisions so as to optimize total memory access time [4]. They also use perfect knowledge of the request sequence in their algorithms.

While offline algorithms are a very reasonable starting point for these investigations, an important next step is to address the question of how to use imperfect information about a request sequence and what are reasonable models for partial knowledge. A very natural approach to modeling imperfect information is to use a statistical model for request access sequences which can be gathered on previous executions of the program or in an online fashion as the execution of a program progresses.

The goals of this paper are two-fold: the first set of experiments are designed to empirically test algorithms that have been designed using the Markov model for paging. The Markov model has been a popular method of modeling paging request sequences in recent theoretical work [16, 11]. We use traces data to simulate the behavior of each algorithm on real paging sequences and report the fault rate as the amount of main memory varies. We also give the fault rate of the optimal offline algorithm as a point of comparison. Interestingly, although recent work [13] suggests that the Markov Request Model may not be an accurate model for paging sequences, we find that both algorithms in this study which were developed with the Markov model in mind perform near optimal.

Unfortunately, these Markov-based algorithms are too complex to implement in practice. Therefore, the second goal of this paper is to investigate the question of what statistical information about a memory request sequence is useful to have in making page replacement decisions. We use the Markov-based algorithms as a starting place and develop more practical algorithms which approximate the behavior of these algorithms. Again, we use trace data to empirically evaluate the algorithms we develop, reporting the number of faults incurred by each algorithms as the amount of main memory varies.

The remainder of Section 1 describes related work and reviews some of the results on the Markov model for paging. Section 2 describes the set of trace files which we use in this empirical investigation. This paper focuses on the memory hierarchy between main memory and disk. Here the unit of memory is a page. It would also be very useful to extend this study to examine the memory hierarchy between the cache and main memory. In this case, the unit of memory would be a cache line. The traces we use are detailed enough for studying cache line replacement as well.

In Section 3, we discuss our implementation of two Markov-based algorithms described in Section 1.2 and present our empirical results for these algorithms. In Section 4, we develop succinct characteristics of memory request sequences which significantly improve paging performance on the set of trace files. The characteristics we develop are based on MHT, the simpler of the two algorithms, but require maintaining only a small amount of information with each page.

Finally, in Section 5, we investigate how to collect these characteristics in an online manner in order to have a purely online algorithm.

1.1 Related Work

A popular recent approach to page replacement policies has been to provide mechanisms for the applications programmer to provide information about the request sequence using knowledge about the program [18, 15, 10]. Although these schemes can be effective, they place an additional burden on the programmer to provide this extra information. Recently, Glass and Cao have devised a scheme which switches between LRU and MRU depending on whether the preceding references indicate that the program is referencing a long sequence of contiguous virtual memory addresses [9].

The literature of computer performance modeling and analysis contains much work on statistical approaches to page replacement, both theoretical and empirical. Shedler and Tung [16] were the first to propose a Markovian model of locality of reference. Spirn [17] gives a comprehensive survey of models for program behavior, focusing on semi-Markovian methods for generating sequences exhibiting locality of reference. Denning [7] (and references therein) develops the *working set* model of program behavior for capturing locality of reference. A recent approach similar to the Markov model is the access-graph model in which patterns in the reference streams are characterized using a graph-theoretical model [3, 12]. These methods require more extensive data about the program, although a recent empirical study of Fiat and Rosen [8] indicates that some of these ideas can be successfully simplified.

Recently, Liberatore has conducted an empirical investigation of the validity of the Markov Request Model for paging sequences [13]. His results use several different measures to show that the Markov Request Model is not an accurate model for paging sequences. His study is based on the traces used in the study by Fiat and Rosen [8]. We choose not to include those traces in this study. The main reason for this choice is that the Fiat-Rosen traces are only composed of the subsequence of requests to instruction pages. As a result, the number of distinct pages is so small that the total amount of memory accessed in any individual trace is never more than 352 kB which would clearly fit into main memory. While these traces may have been appropriate for the studies in which they were used, we felt that our study was better suited to a set of traces which represent a more complete account of the total memory access patterns.

1.2 A First-Order Markov Model for Request Sequences

Karlin, Phillips and Raghavan examine the question of how to devise provably good page replacement algorithms under the assumption that page request sequences are generated by first-order Markov chains [11]. The state space is the set of all pages referenced in the course of the program execution. If a memory request sequence is generated by a first order Markov chain, the probability that a page is referenced at a given point in the request sequence depends only on the previous request. Seen another way, given that the i^{th} reference is to a particular page p, the distribution over the next request is determined.

If a page replacement algorithm knows the Markov chain which generates a memory request sequence, it is theoretically possible to determine the optimal online algorithm which will minimize the expected number of faults [11]. Unfortunately, determining this strategy requires solving a linear program with $n\binom{n}{k}$ pages, where n is the number of pages referenced in the course of the program's execution and k is the number of pages which fit in the cache. Since this is clearly infeasible, they turn to devising an approximation algorithm whose expected page fault rate can be proven to be within a constant factor of optimal.

This paper contains a series of negative results which show that a number of intuitively reasonable algorithms do not attain this theoretical goal. Most of these algorithms are designed to use the knowledge provided by the Markov chain to approximate the behavior of the optimal offline algorithm. The optimal offline algorithm, devised by Belady [2], will evict the page whose next reference is farthest in the future on a fault. The approximation of the optimal which we use here, called *Maximum Hitting Time* (MHT), determines the expected next request of each of the pages in the cache and evicts the page whose expected next request time is farthest in the future. Although this algorithm in the worst case can be as bad as $\Omega(k)$ times the optimal, it proves to be very effective in our empirical study. Note that a lower bound entails showing a single Markov chain for which the expected fault rate of MHT is $\Omega(k)$ times that of the optimal online algorithm. It does not necessarily mean that MHT will perform that badly on all or even typical Markov sequences. Unfortunately, MHT is not very practical since it requires determining for each pair of pages p and q, the expected time to travel in the Markov chain from p to q which requires $O(n^2)$ space.

A subsequent paper by Lund, Phillips, and Reingold [14] addresses more general distributions over memory request sequences. They show an algorithm called the *dominating distribution algorithm* (DD) which is conceptually very elegant. The algorithm is provably close to optimal in its expected fault rate, but impractical to implement. At each point in time that a fault occurs, the algorithm determines for each pair of pages p and q in memory, the probability that p is requested before q. Given these values, it is possible to determine what is called a *dominating distribution* over the set of pages in memory. A dominating distribution has the property that if a page r is chosen according to this distribution, then for any other page in memory p, the probability that p is requested before r is at most $1/2$. They prove that when the page to be evicted is chosen according to the dominating distribution, the expected number of faults is within a factor of four of optimal. The algorithm is impractical for two reasons: if the request sequence is generated by a Markov chain, then for each ordered triplet of pages q, p and r, we must have determined the probability that r is requested before p given that the last request was to page q. This is $O(n^3)$ amount of data to maintain. The other reason for the impracticality of the algorithm is that determining the dominating distribution algorithm requires solving a linear program in k variables. Furthermore, this linear program must be solved at *every* page fault.

Despite the fact that recent work indicates that first-order Markov chains may not model page request sequences very accurately ([13]), the two algorithms described here work very well on the traces used in this study. Although these algorithms are not practical page replacement algorithms, they give important intuition for developing more feasible algorithms.

2 Trace Methodology

We use I/O traces collected from a number of applications by Cao, Felten, and Li [5]. They instrumented the Ultrix 4.3 kernel to collect the traces. When tracing is turned on, file I/O system calls for every process are recorded in a log. They gathered traces for a number of applications. The applications are

dinero	dinero simulating about 8 MB of cc trace.
cscope	cscope searching for eight C symbol names in about 18 MB of kernel source.
glimpse	glimpse searching for 5 sets of keywords in about 40 MB of newsgroup text.
ld	ld building the Ultrix 4.3 kernel from about 25 MB of object files.
sort	sorting numerically on the first field of a 200,000 line, 17 MB text file.
postgres join	postgres joining 200,000 tuple and 20,000 tuple relational databases.
postgres select	postgres doing a selection.
rendering	3-D volume rendering software rendering 22 slices with stride 10.

For more detailed information on these applications, see [6].

The table below shows the number of distinct pages requested and the length of the sequence for each of the traces.

Trace	Pages	Requests
dinero	1972	17715
cscope	2143	17255
glimpse	10310	49828
ld	6583	10383
sort	16578	36938
postgres join	8781	27170
postgres selection	7265	14410
volume rendering	11007	46703

We used a uniform page size of 4 kB throughout our study. For read and write system calls, the device and inode numbers of the file accessed and the integer part of the offset in the file divided by 4000 is stored in the linked list. If the count in the read or write extends over multiple pages, each of those pages is stored in the linked list. Consecutive reads and writes to the same page are stored as one entry in the linked list.

Each of our programs reads in a trace file and stores the relevant page information in a linked list. When time information is necessary, as in LRU and the optimal offline algorithm, a counter is incremented for each page in the linked list. Each algorithm can then be run on various cache sizes with the page requests in the linked list. Of course, the online algorithms do not "remember" anything from one cache size to another.

2.1 Experimental Design

In each of the studies for this paper, we evaluate different algorithms measuring the number of faults they incur on a given request sequence. The amount of main memory available is varied, to test the algorithms under different conditions. For simplicity, the amount of main memory is measured in terms of the number of memory pages it can hold. Naturally, as this number increases, all the algorithms improve. In each graph, we give the fault rate of the optimal offline algorithm for comparison.

The fault rate versus memory size is represented as a continuous curve, although it is really the results of many discrete trials. In Figures 1 and 2, each line is composed of roughly one hundred discrete points. The amount of main memory is increased linearly from one page to the maximum number used. For the Dominating Distribution algorithm, we only had computational resources to test the algorithm for a small number of main memory sizes on a few traces. This data is represented by discrete points, instead of a continuous curve. In Figures 3, 4, 5, and 6, the size of main memory is increased by 100 pages in each trial.

In all the graphs, when the fault rate for a particular algorithm reaches the total number of distinct pages requested in the sequence, we do not perform any further trials with more main memory. Any algorithm must incur a fault the first time a page is requested, so increasing the amount of main memory beyond this value would not decrease the fault rate. Thus, you will notice that the curve that corresponds to the optimal offline algorithm is always the shortest curve since it reaches the minimum number of faults first.

3 Markov-Based Algorithms

3.1 Gathering Statistics from the Traces

The Maximum Hitting Time algorithm requires keeping for each ordered pair of pages (p, q), the expected number of requests after a request to p until a request to q. Since this information is expensive to derive from the trace exactly (in terms of available memory, see Section 3.2), we made some approximations in our preprocessing. A counter is kept for every ordered pair (p, q). This counter is always in one of two states: *active* or *inactive*. The counter starts out inactive.

If a request to p is encountered, the counter is put in the active state if it is not already in the active state. Whenever a request to q is encountered, the counter is put in the inactive state if it is not already in the inactive state. While the counter is active, it is incremented once for every request in the sequence. The value of the counter is taken to be the total length of all hitting times from p to q. This value is divided by the number of times a request to p is followed by a request to q without an intervening request to p. Naturally, it is possible that there is never a request to p followed by a request to q. This case is noted by a flag indicating an infinite hitting time from p to q. Consider the example below

$$x\ x\ p\ \mathbf{x\ x\ q}\ x\ x\ x\ p\ \mathbf{x\ p\ x\ x\ x\ q}\ x\ p\ \mathbf{x\ x\ p\ x}$$

The x charcters represent requests to pages other than p or q. The characters in bold indicate when the counter for (p, q) is active. The value of the counter at the end of the sequence is 13 which is the number of request which arrive while the counter is active. This value is divided by two to get the expected hitting time from p to q. Note that the divisor is the number of times the counter is switched from the active to the inactive state which is the same as the number times a request to p is followed by a request to q without an intervening request to p.

For the Dominating distribution algorithm, we need to know for every triple of pages (p, q, r), the probability that r is requested before q after a request to p. A counter is maintained for this triple which can either be active or inactive. The counter is maintained as follows:

– If a request to p is encountered, it is put in the active state if it is not already in the active state.
– If a request to r is encountered and the counter is active, then the counter is incremented and put in the inactive state.
– If a request q is encountered, the counter is put in the inactive state if it is not already in the inactive state.

The value of the counter is then divided by the number of times a request to p is followed by a request to q or r without an intervening request to p. Consider the example below

$$x\ x\ p\ \mathbf{x\ x\ r}\ x\ x\ x\ p\ \mathbf{x\ q}\ x\ p\ \mathbf{x\ p\ x\ x\ x\ r}\ x\ p\ \mathbf{x\ x\ p\ x}$$

The x characters represent requests to pages other than p, q and r. The characters in bold represent the requests which come in while the counter for (p, q, r) is active. At the end of the sequence, the counter is two, since two of the active subsequences end in an r. This is then divided by three to get the probability that r is requested before q after a request to p. Again, the divisor represents the number of times the counter is switched from the active to the inactive state which is the same as the number of times a request to p is followed by a request to r or q without an intervening request to p.

3.2 Implementation

Simulating MHT and DD is problematic because of both space and the computational requirements. Fortunately, the number of pages was small enough that each page could be identified by a short integer even for the largest page set. This enabled us to maintain the entire $O(n^2)$ data structure for MHT. However the $O(n^3)$ space required by DD was still insurmountable. In order to reduce the space required by DD to $O(n^2)$, we had to reprocess the trace data for each page fault. Reprocessing the trace data and solving a linear program at every page fault made the simulation of DD a very time consuming process. We only simulated a few cache sizes for a few traces to spot check DD against the other algorithms.

3.3 Experimental Results

Figures 1 and 2 show the results of running four page replacement algorithms on the traces. These are: LRU, Belady's optimal offline algorithm (OPT), Maximum Hitting Time (MHT) and Dominating Distribution. As you can see from the figures, MHT does very well on all of these traces, and DD does about as well as OPT and MHT for the points that we simulated. This is despite the fact that DD requires $O(n^3)$ data and solving a linear program at every page fault while MHT only requires $O(n^2)$ data and a linear search at every page fault. This suggests that the extra resources used by DD to improve the page fault rate is unnecessary for the programs used in these experiments.

4 A More Practical Algorithm

Motivated by the theoretical results on Markov paging, we develop page replacement policies which are more efficient, both in their time and space requirements. The algorithms described here are closest in spirit to the Maximum Hitting Time described above in Section 1.2. As in MHT, the goal of the algorithm is to evict the page whose expected next request time is farthest in the future. However, instead of basing this expectation on the last request seen, we make a cruder estimate which requires only $O(1)$ space per page. We determine the average *round-trip time* for each page which is the average number of requests in between consecutive requests to that page. For each page in the cache, we estimate its next request time by the last time it was requested plus the expected round-trip time for that page. This algorithm is called *Average Round-Trip Time* (ARTT).

In an offline pass of the data, we determine the average round trip time for each page. Pages which are only requested once are given an arbitrarily large round-trip time. Then when the cache is simulated, every time a page is requested, its estimated next request time is set to be the current time plus its average round-trip time. On a fault, the page with the largest estimated next request time is evicted. In order to prevent pages from staying in memory for a very long time, if a page's estimated next request time has passed, its priority for eviction is set arbitrarily high.

The algorithm is relatively simple to implement. Each page has its average round trip time which it keeps throughout the execution of the program. This information is static. The pages in memory are kept in a priority queue according to their estimated next request time. One update to the priority queue must be performed at each memory request (although updating the priority queue could be done less frequently). A separate priority queue is maintained which keeps the pages in reverse order to detect when a page's next request time has expired.

A second version of this algorithm is based on the observation that typically a page is requested with relatively short round-trip times and then may not be requested for a long time. Presumably, we want to use the short round-trip time for the estimated next request time for the page and when this time passes, mark the page for eviction. Thus, we have the *Conditional Average Round-Trip Time* (CARTT) algorithm which is the same as ARTT except that it only takes the average of the round-trip times which are shorter than a predefined threshold. The threshold we use here is $3k$ where k is the number of pages which can fit in memory.

The three is an arbitrarily chosen small constant. Varying this value slightly does not significantly alter the results. The factor of k comes from the reasoning that under LRU, a page will sit in memory for a length of time that depends on the size of the cache. The fact that round-trip times have somewhat bi-modal distributions, indicates that the choice of threshold is not so important. It does begin to matter as the size of the cache approaches one of the humps in the distribution, which would explain the occasionally erratic behavior in CARTT seen below.

We compared these two algorithms to the offline optimal and LRU as well as to a slightly altered version of LRU which we call *Augmented-LRU*. In the augmented LRU, we give the algorithm this single bit of offline information that tells whether the page ever has a round-trip time of less than three times the capacity of the cache. This is to determine whether the advantage that ARTT and CARTT gain are from this bit of information or from the more refined data given by the round-trip times. The results are shown in Figures 3, and 4. Although it is hard to tell in some of the figures below, Augmented-LRU and LRU often behave exactly the same and worse than CARTT and ARTT. This indicates that the round-trip time values used by CARTT and ARTT do provide valuable information. As expected, CARTT out-performs ARTT in almost all cases. Both algorithms obtain significant gains over LRU on most of the traces and are never significantly worse.

5 Purely Online Algorithms

We use the intuition from the partially offline algorithms discussed in Section 4 to develop purely online algorithms which have no prior knowledge of future requests except what they have seen in the past. The algorithms we describe here attempt to behave like CARTT except that they are forced to approximate the conditional average round-trip time for each page based on what they have seen so far in the sequence. Each page maintains the sum of the round trip times under $3k$ which have occurred so far and the number of these "small" round trips. These values are updated whenever the page is requested. Whenever the average round-trip time is needed, the ratio of these two values is used.

For the first three requests to a page, we use a default value for the conditional round-trip time until more data has been gathered for that page. The algorithms only differ in what is used for the default value. We present data from two algorithms below. The first one uses a default of k. We called this one *online:default-k*. The second uses a dynamic default value which is the number of requests since the last request to that page. The goal here is to get the algorithm to behave in an LRU-like manner until enough statistical information is available to improve page replacement performance. If the time since the last request is always used as the estimated round-trip time, then the algorithm behaves exactly as LRU. We called this algorithm *online:default-LRU*. We also tested an algorithm which set the default to a very high value, which would lead to quickly evicting pages which had not been requested more than three times. This algorithm did much worse than the other two.

The results are shown in Figures 5, and 6. Surprisingly, the default-k algorithm was almost always the better of the two. One reason for this is that the relative values of the default round-trip times and the estimated round-trip time may have been off so that the algorithm unfairly favored one set over the other. This perhaps could be adjusted by weighting the two values differently. The online:default-k algorithm improved upon LRU on most of the traces. Interestingly, when the default-k was significantly worse than LRU, the default-LRU algorithm was better than LRU.

References

1. Barve, R., Grove, E., Vitter, J.: Application-controlled paging for a shared cache. Proc. 36th Symposium on Foundations of Computer Science (1995) 204–213
2. Belady, L.: A study of replacement algorithms for virtual storage computers. IBM Systems Journal **5** (1966) 78–101
3. Borodin, A., Irani, S., Raghavan, P., Schieber, B.: Competitive paging with locality of reference. Journal of Computer and System Sciences **50** (1995) 244–258
4. Cao, P., Felten, E., Karlin, A., Li, K.: A study of integrated prefetching and caching strategies. Performance Evaluation Review **23** (1995) 188–197
5. Cao, P., Felten, E., Li, K.: Application-controlled file cashing policies. Proc. Usenix Summer Technical Conference (1994)
6. Cao, P., Felten, E., Li, K.: Implementation and performance of application-controlled file caching. Proc. 1st Usenix Symposium on Operating Systems Design and Implementation (1994) 165–178
7. Denning, P.: Working sets past and present. IEEE Transactions on Software Engineering **6** (1980) 64-84
8. Fiat, A., Rosen, Z.:. Experimental studies of access graph based heuristics: beating the LRU standard? Proc. 8th ACM-SIAM Symposium on Discrete Algorithms (1997) 63–72

9. Glass, G., Cao, P.: Adaptive page replacement based on memory reference behavior. Performance Evaluation Review **25** (1997) 115–126
10. Harty, K., Cheriton, D.: Application-controlled physical memory using external page-cache management. Proc. 5th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS-V) (1992) 12-15
11. Karlin, A., Phillips, S., Raghavan, P.: Markov paging. Proc. 33rd Symposium on Foundations of Computer Science (1992) 208–217
12. Irani, S., Karlin, A., Phillips, S.: Strongly competitive algorithms for paging with locality of reference. SIAM Journal on Computing **25** (1996) 477–497
13. Liberatore, V.: Empirical investigation of the Markov reference model. Proc. 10th ACM-SIAM Symposium on Discrete Algorithms (1999) 653–662
14. Lund, C., Phillips, S., Reingold, N.: IP over connection-oriented networks and distributional paging. Proc. 35th Symposium on Foundations of Computer Science (1994) 424–435
15. McNamee, D., Armstrong, K.: Extending the Mach external pager interface to accommodate user-level page replacement policies. Proc. USENIX Workshop (1990) 4-5
16. Shedler, G., Tung, C.: Locality on Page Reference Strings. SIAM Journal on Computing **1** (1972) 218-241
17. Spirn, J.: Program Behavior: Models and Measurements. Elsevier, Amsterdam (1977)
18. Young, M.: Exporting a User Interface to Memory Management from a Communication-Oriented Operating System. PhD thesis, Carnegie-Mellon University (1989)

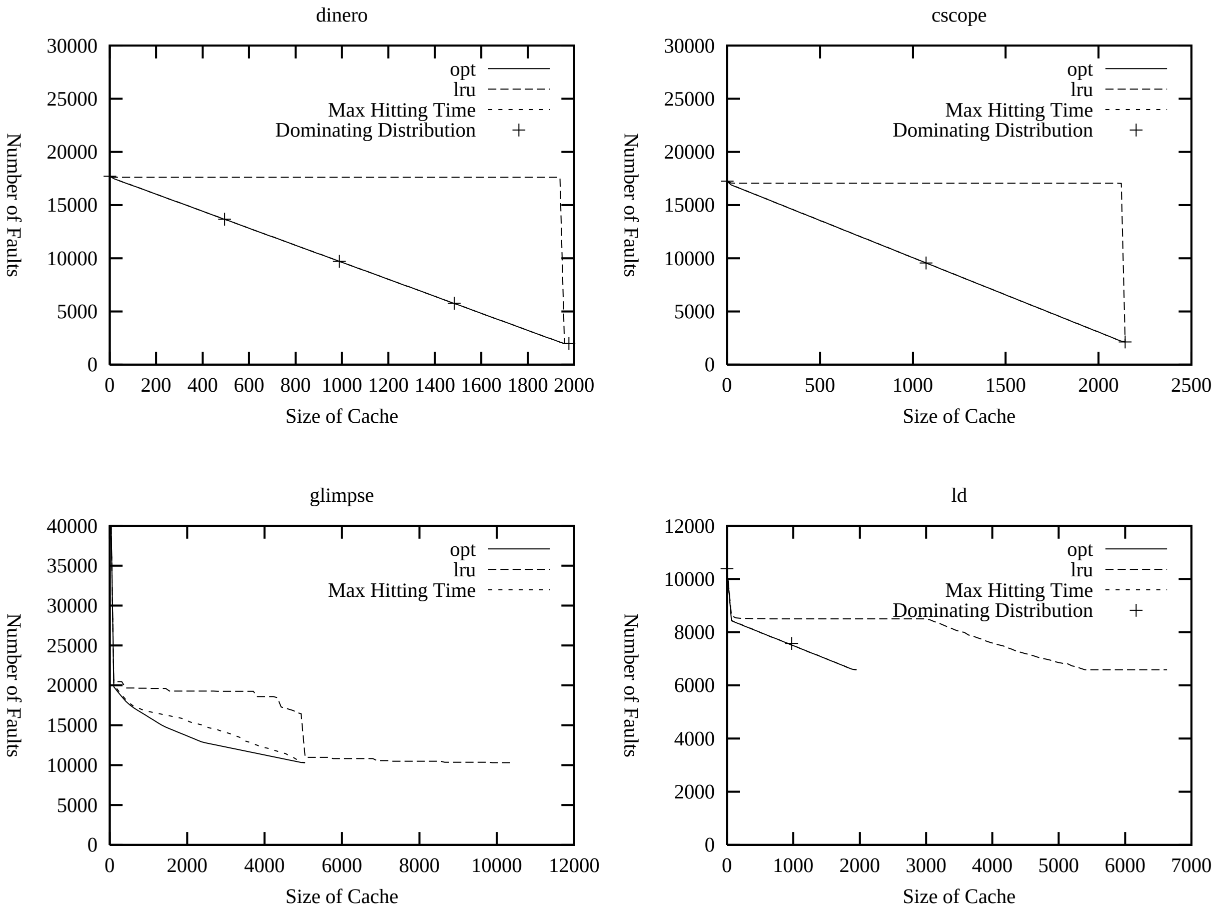

Fig. 1. Comparison of Markov-based algorithms with LRU and the optimal offline algorithm. The page size is 4 kB. In the dinero, cscope and ld traces, OPT and Maximum Hitting Time practically coincide. See Sections 2.1 and 3 for more details on these experiments.

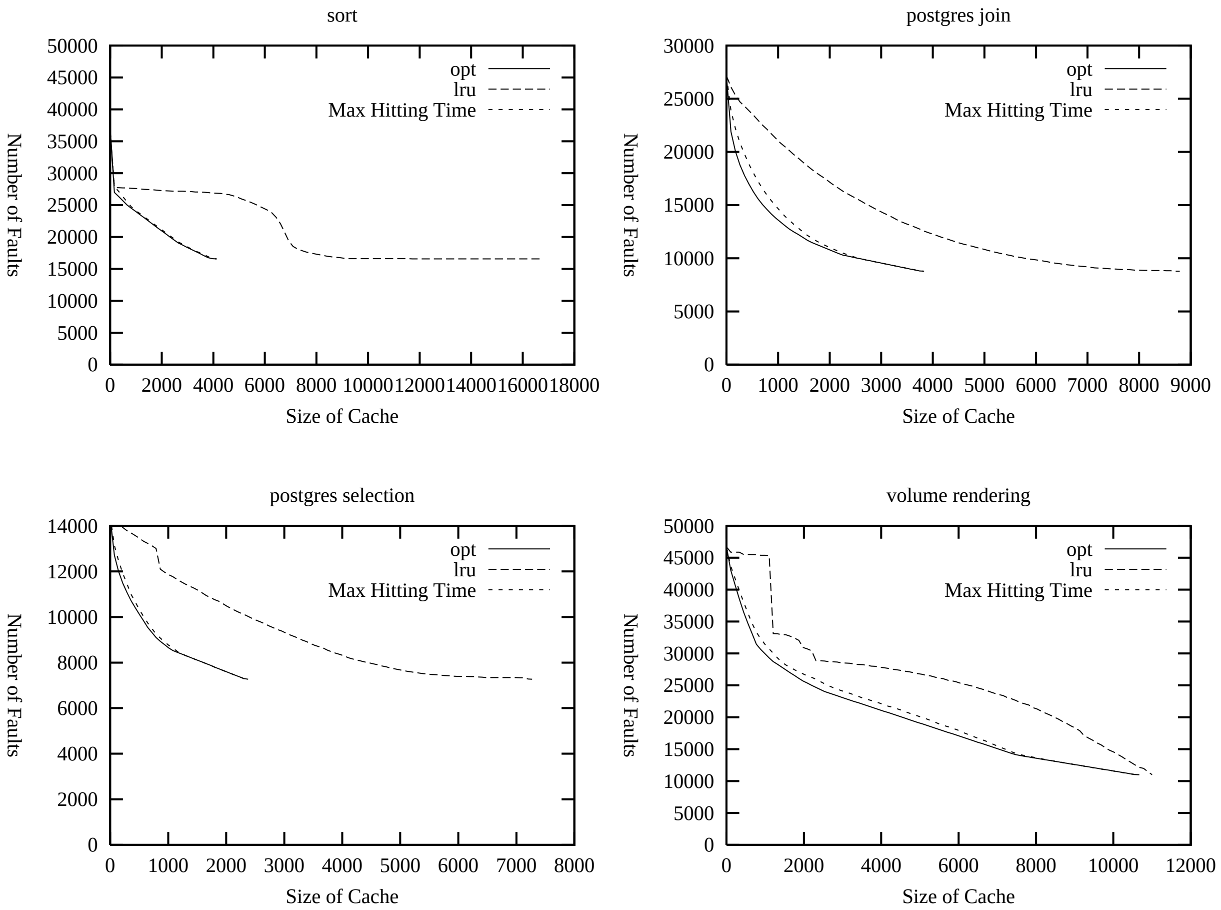

Fig. 2. Comparison of Markov-based algorithms with LRU and the optimal offline algorithm. The page size is 4 kB. See Sections 2.1 and 3 for more details on these experiments.

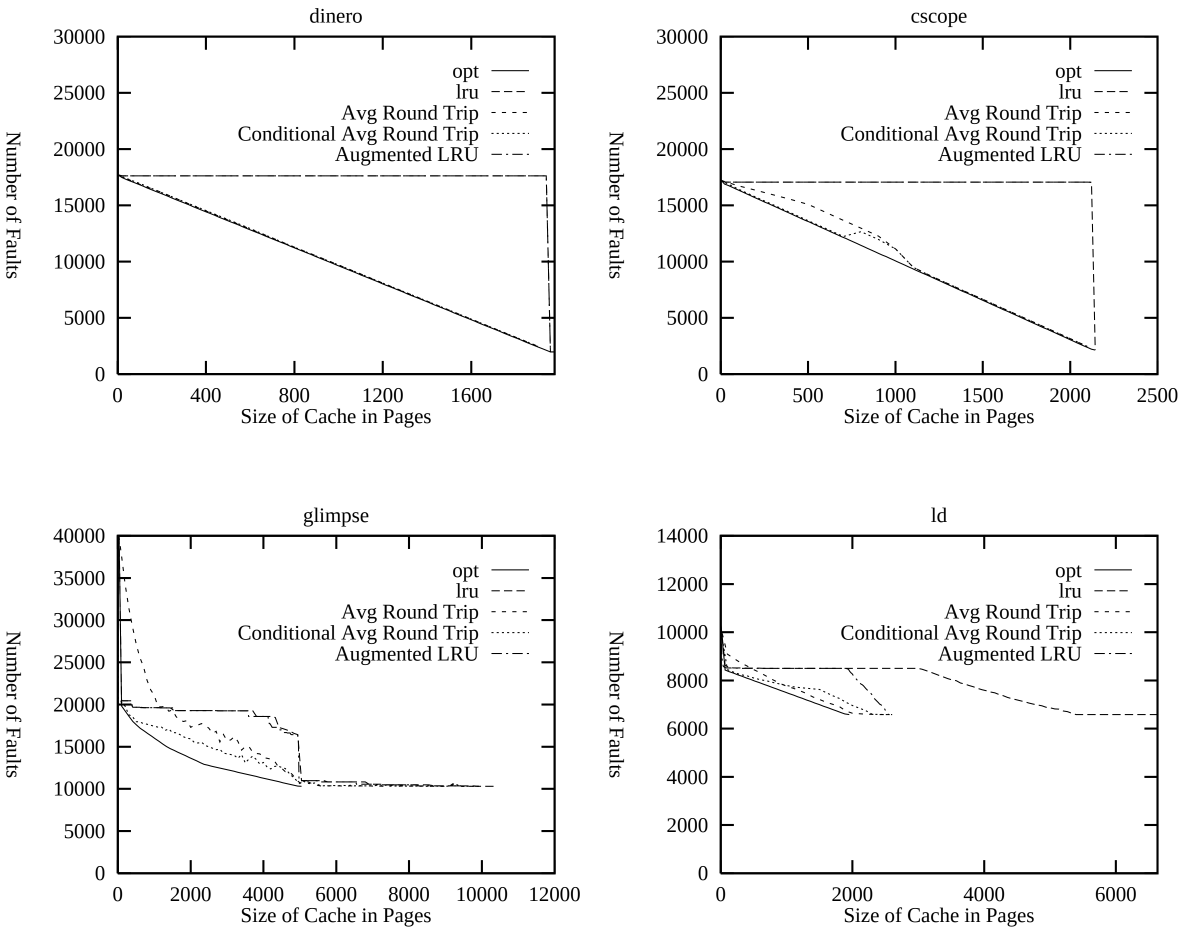

Fig. 3. *Comparison of algorithms which use some partial offline information. The page size is 4 kB. LRU and Augmented LRU practically coincide in all the traces. In the dinero trace, OPT, Avg Round Trip, and Conditional Avg Round Trip practically coincide. See Sections 2.1 and 4 for more details on these experiments.*

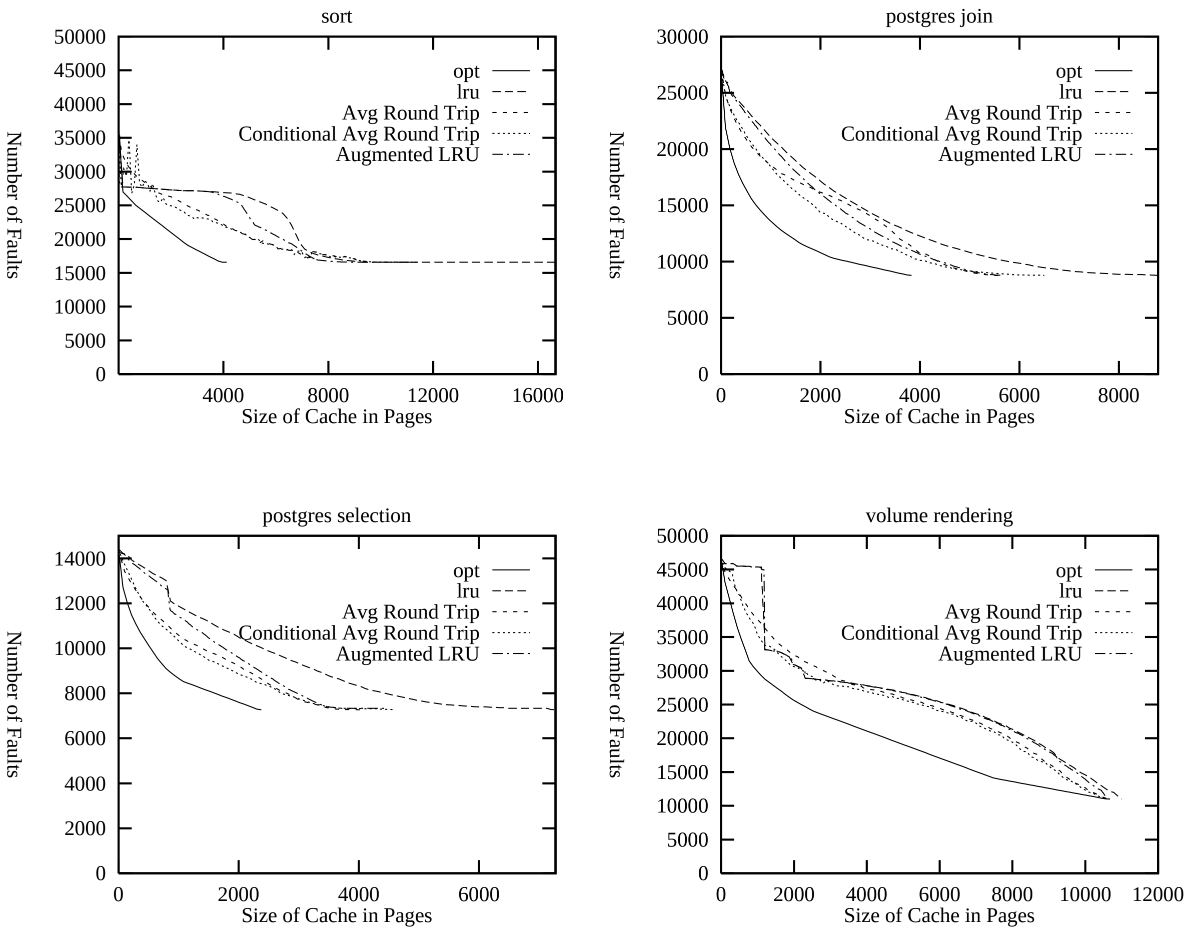

Fig. 4. *Comparison of algorithms which use some partial offline information. The page size is 4 kB. See Sections 2.1 and 4 for more details on these experiments.*

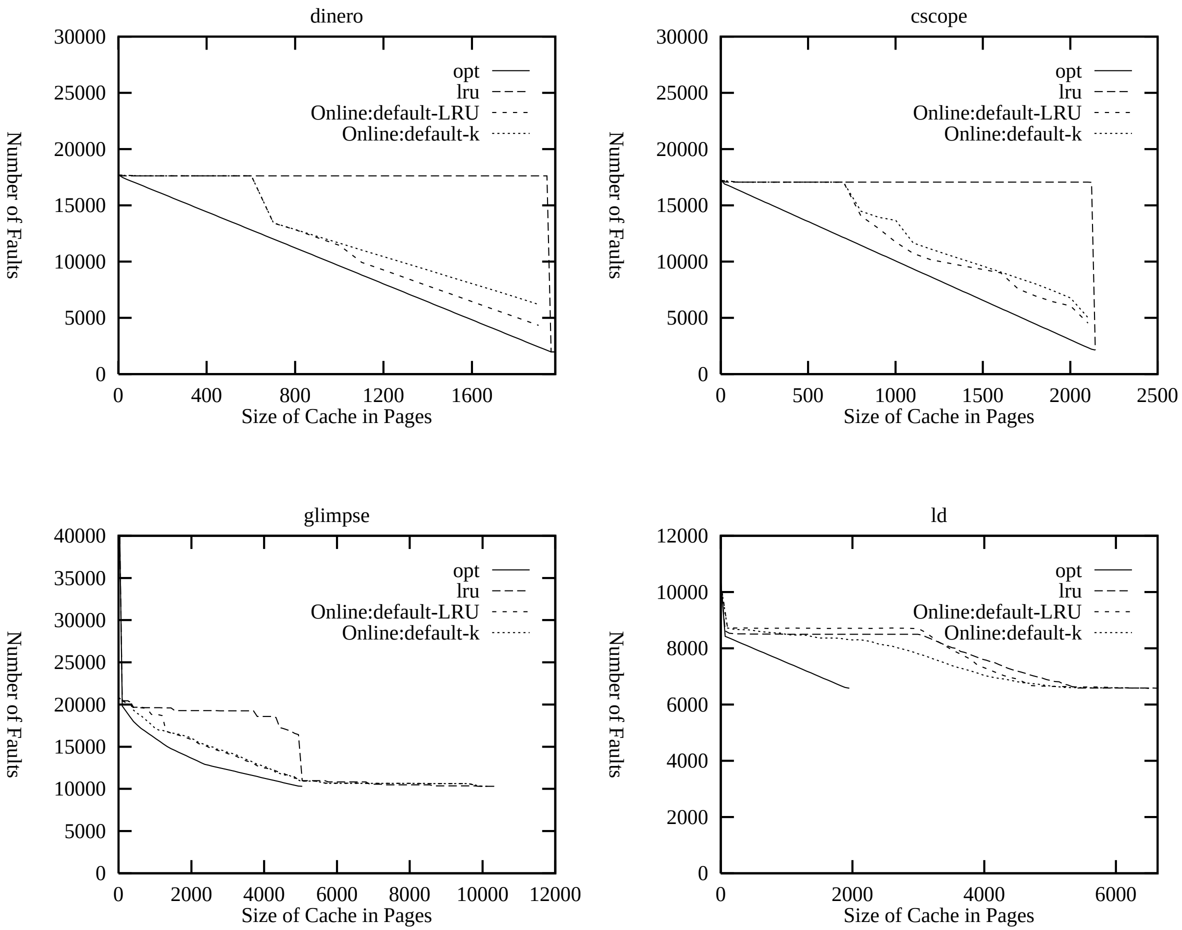

Fig. 5. *Comparison of online algorithms with LRU and the optimal offline algorithm. The page size is 4 kB. See Sections 2.1 and 5 for more details on these experiments.*

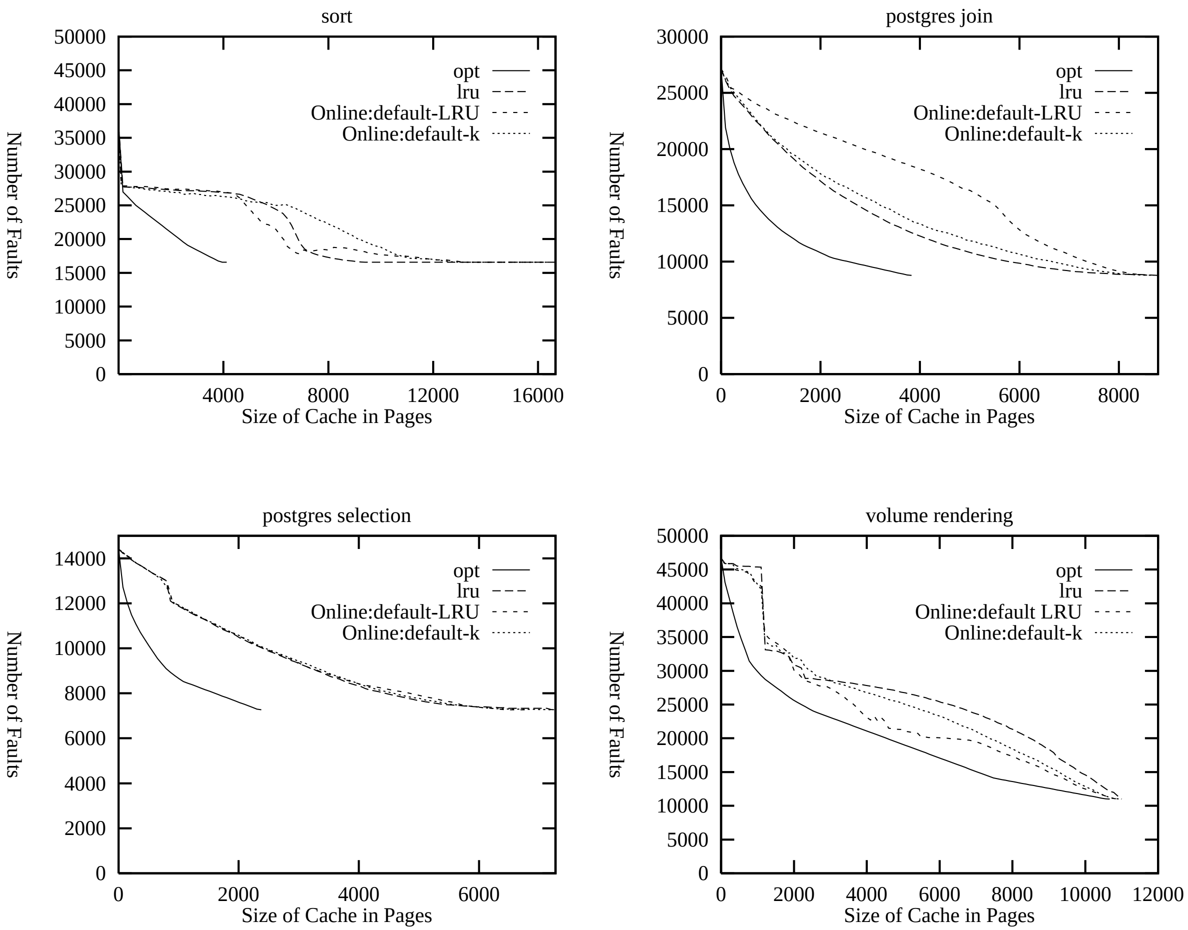

Fig. 6. *Comparison of online algorithms with LRU and the optimal offline algorithm. The page size is 4 kB. See Sections 2.1 and 5 for more details on these experiments.*