M.S. THESIS

An Adaptive Page Replacement Scheme for Scientific Applications

과학계산 응용을 위한 적응형 페이지 교체 기법

2020 년 8 월

서울대학교 대학원

컴퓨터공학부

이 윤 재

Abstract

Memory management is playing an increasingly important role for the application performance, owing to the rapid increase in the amount of data utilized in modern computing workloads and the slow growth in the capacity of the main memory devices. We also observe that recent scientific applications are processing huge data never been seen before. Scientific applications show a tendency that they usually repeat operations on intermediate data using loops, but such access patterns are hard to be appropriately handled by LRU and its approximations, which are generally used as a page replacement policy in operating systems. In this paper, we propose Adaptive Page Replacement (APR) scheme, which deals with looping access patterns in scientific applications properly. APR detects various looping access patterns online and handles them with the consideration of the resulting performance. It can be implemented using limited events or information (e.g., page faults and access bits) that the virtual memory subsystem of operating systems provides. We evaluate APR by trace-driven simulation with traces extracted from workloads in SPLASH-2x benchmark. Throughout the comparison with multiple previous schemes, we demonstrate that APR successfully improves the performance over the previous works by complementing their downsides properly.

Keywords: page replacement, access pattern, scientific applications, trace-driven simulation, memory management

Student Number: 2018-20169

Contents

List of Figures

List of Tables

Chapter 1

Introduction

Recently, data-intensive applications (e.g., big data and machine learning) in various areas have handled a huge volume of datasets, and this tendency is becoming more prevalent [1]. Thus, the amount of main memory directly impacts the overall system performance. On the other hand, the capacity of DRAM devices is increasing slowly [2], and managing data in the main memory effectively is becoming important for the performance of the workloads. In modern operating systems (OSes), paging is the general memory management scheme, and LRU and its derivatives are widely used owing to their simplicity and the advantage in performance [3, 4].

Scientific applications in the high-performance computing (HPC) community also show the above tendency in terms of memory usage. In addition, they generally repeat similar operations on intermediate data in whole or in part by using iterative loops. It is well known that such an access pattern is an Achilles' heel of LRU [5], and thus, OSes in HPC systems need to effectively handle loop access patterns as well as existing access patterns suitable for LRU. When we

design an efficient page replacement scheme of OSes for looping access patterns, we take following criteria into consideration.

Detecting looping access patterns. A good page replacement scheme can detect multiple types of looping access patterns. For example, the looping access pattern may be non-sequential in address space (e.g, iterating linked lists). Looping access patterns may also be mixed or interleaved even in a single loop (e.g., updating multiple array elements in a loop), and they may have different access window sizes.

Handling looping access patterns. A page replacement scheme should handle looping access patterns properly. When a process needs an empty page, the page replacement scheme in the OS needs to first identify an appropriate page access pattern for the requesting process. If it detects a loop access pattern, it decides the best victim page corresponding to the access pattern. If the victim page is not properly selected, the performance can be even worse than existing schemes without consideration of looping access patterns.

Anonymous pages. Most of the intermediate data in scientific applications is kept in memory areas made of anonymous pages not backed by any files (e.g., `malloc()`) [6]. Unlike file-backed pages which are accessed using system calls (e.g., `read()`), anonymous pages of scientific applications are accessed in user-level by their virtual addresses, thus it is impossible to track every access (i.e., read accesses) to anonymous pages. This means that a scheme must be built upon only page faults and access bits of pages.

Table 1.1 compares page replacement schemes for looping access patterns with the criteria. EELRU [7] handles looping access patterns using cost/benefit analysis without explicit detection. UBM [8] detects looping access patterns in files and evicts pages from the access pattern that is expected to show the lowest performance penalty. However, EELRU and UBM require tracking every access;

Table 1.1 A comparison of page replacement schemes.

	EELRU	**UBM**	**SEQ**	**CLOCK-Pro**	**APR**
Detection	△	△	△	×	◯
Handling	◯	◯	×	△	◯
Anon. pages	×	×	◯	◯	◯

thus, they are not feasible when it comes to managing anonymous pages. SEQ [5] detects page fault sequences in address space and evicts the most recently used page in the sequences first. CLOCK-Pro [9] is an improvement of CLOCK that detects access patterns, including loops, using inter-reference recency and changes its behavior according to the access pattern. SEQ and CLOCK-Pro schemes lack the deep consideration of how the performance is affected by handling the looping access patterns. Unlike our scheme, all previous schemes cannot detect various looping access patterns, including mixed access patterns or access patterns with varying access window sizes.

In this work, we propose APR, an adaptive page replacement scheme that properly detects and handles looping access patterns in scientific applications. Basically, our scheme behaves like CLOCK by default and handles looping access patterns using a fallback policy. To this end, our scheme (1) detects looping access patterns by evaluating the default and fallback policies online, (2) handles the detected looping access patterns to improve overall performance, and (3) exploits only page faults and access bits that makes it possible to manage anonymous pages. The fallback policy is based on the *backward access checking* technique in per-object last-in-first-out (LIFO) page lists; we call this policy LIFO+. We evaluate our scheme using traces from 12 workloads in the SPLASH-2x

benchmark suite [10], and compare it with prior well-known schemes under varying memory sizes. Experimental results show that our scheme outperforms the previous schemes in most of the workloads, and reduces the number of page faults by 22.0%, 22.1%, and 16.4% on average compared to CLOCK, CLOCK-Pro, and SEQ, respectively.

The remainder of this paper is structured as follows: in Chapter 2, we review a number of related works regarding page replacement schemes for looping access patterns. In Chapter 3, the design and the main techniques of APR are described. In Chapter 4, we show the evaluation results of APR as well as the evaluation methodology. The performance and the main techniques of APR are evaluated in detail. Finally, we conclude the paper in Chapter 5.

Chapter 2

Related work

2.1 General replacement schemes

While LRU shows quite high performance for wide range of systems in terms of caching, it is not effective enough when it comes to weak locality accesses. Historically, there have been a number of works that tried to deal with this disadvantage of LRU [11, 12, 13, 14, 15, 16, 9, 17]. They fundamentally utilized frequency information in addition to recency information, to overcome the performance disadvantage of LRU on weak locality accesses. Among them, CAR [16] and CLOCK-Pro [9] are based on CLOCK [3], while the other schemes are based on LRU that track every references. In addition, they are designed to handle general access patterns in workloads, however, CLOCK-Pro includes regular access patterns (e.g., scanning and looping) in its design. It adaptively tunes its behavior according to the access pattern and classifies pages into two groups: cold pages and hot pages. A page is cold if it is fetched into memory, and it becomes hot if it is accessed again in a test period. The behavior is con-

trolled by adjusting a parameter m_c that represents the maximum number of cold pages, from which the victim for a page replacement is chosen. If a fetched page is re-accessed in its test period, the m_c value is increased. Otherwise, if the page passes its test period without re-access, the m_c value is decreased. A large m_c leads to memory dominated by cold pages; thus, CLOCK-Pro behaves as CLOCK. On the contrary, a small m_c leads to the small number of cold pages in memory; thus, cold pages are evicted soon after being fetched, which is an appropriate behavior for regular access patterns (e.g., scanning or looping). According to the tuning rule, high-locality access patterns that access a page multiple times temporally will be handled in the CLOCK manner, and the regular access patterns that passes by each page will be handled in the MRU manner.

However, CLOCK-Pro does not handle many looping access patterns correctly. When a looping access pattern is mixed with other access patterns in the system, which is common in many cases, CLOCK-Pro cannot deal with mixed patterns properly since access patterns are managed globally, not individually. More important, its tuning rule is less successful with anonymous pages. Unlike file-backed pages that are often accessed in page units, anonymous pages are generally accessed in units of a few bytes. This leads to most of the pages being accessed multiple times, even in looping access patterns; CLOCK-Pro will fail to distinguish such patterns from high-locality access patterns.

2.2 Page replacement schemes for looping access patterns

Since a looping access pattern has been considered as one of the weaknesses of LRU, a number of previous works [5, 8, 7] have tried to deal with such access

patterns in different page replacement schemes. SEQ [5] is a page replacement algorithm that exploits sequential access patterns. Its behavior is the same as LRU by default, however, its behavior becomes like MRU when there is a sequential access pattern in virtual address space. Each sequential access pattern is maintained as a *sequence*, which is created when page faults in sequential addresses are detected. SEQ can handle sequential looping patterns; however, non-sequential looping patterns (e.g., looping over linked lists) and looping patterns with various window sizes are not considered. Moreover, rarely referenced pages may remain in the memory for a long time if they are not in the sequence since SEQ tends to evict pages in the sequence first. UBM [8] classifies access patterns into three types (*sequential*, *looping* and *other*) and maintains pages in different buffers according to classified access patterns. The type of access patterns is detected online in a per-file fashion. When UBM needs to free a page, it first chooses a buffer that is expected to have the smallest performance effect by shrinking the buffer, and then it frees a victim page in the chosen buffer based on its access pattern. Victim pages belonging to buffers for sequential and looping types are chosen in the MRU manner, and pages in the other type are evicted in the LRU manner. For example, if UBM chooses a buffer managing looping access patterns, it frees the most recently used page in the buffer. Like SEQ, however, UBM does not consider non-sequential looping access patterns and access window sizes. For the online detection mechanism, UBM observes the reference sequence to file blocks; thus, it is difficult to be applied in the case of anonymous pages in scientific applications. Instead of detecting looping access patterns explicitly, EELRU [7] detects looping access patterns indirectly and uses a fallback algorithm for such patterns. It uses LRU by default, and if recently evicted pages are fetched again, it switches to the fallback algorithm. The fallback algorithm does not evict pages that are least recently used;

it instead evicts pages that are recently used. By this rule, EELRU can handle general looping access patterns including non-sequential loops. However, it does not support mixed looping access patterns. Additionally, EELRU cannot be used for managing anonymous pages since it must track every reference. Our study is in line with these works [5, 8, 7] in terms of addressing looping access patterns, but our scheme supports both sequential and non-sequential access patterns and addresses the management of anonymous pages.

2.3 Online evaluation of multiple static policies

Choosing the best-fit policy by evaluating multiple page replacement policies online is not a new idea. ACME [18] shows a good behavior even with the access patterns changing dynamically by maintaining a pool of static replacement algorithms. The replacement decision is made considering weighted decisions of the algorithms in the pool, and the weights are updated online using machine learning. However, its learning rule utilizes every access even cache hits, which makes it infeasible to manage anonymous pages. DIP [19] chooses the policy expected to incur a fewer misses online between LRU and Bimodal Insertion Policy (BIP), which adapts to changes in the working set while avoiding thrashing problem unlike LRU. A fixed portion of the cache is dedicated to each policy, and DIP tracks the miss counts in the dedicated cache area of each policy. The policy with smaller miss counts is chosen and applied to the other portion of the cache. Since the old misses have the same weight on the decision, however, access patterns changing dynamically might not be followed correctly. Moreover, DIP chooses the insertion policy rather than replacement policy, thus its data structures and the application of the policy cannot be adopted when it comes to replacement policies. LeCaR [20] maintains two orthogonal policies, LRU

and LFU, and picks a proper one dynamically that fits more to access pattern. The choice is based on the weight (probability), which is tuned online according to a learning rule. The learning rule utilizes the eviction history, thus LeCaR maintains additional ghost buffers per policy to record the order in which pages are evicted. However, when there is a large gap between the memory and the working set size, a large portion of the memory must be allocated to ghost buffers in order to learn the access pattern effectively. Only small portion of the memory is used for the caching data, and this might result in the poor performance compared to even LRU. Our scheme, APR, evaluates multiple static policies online like these works [18, 19, 20]; however, APR requires only limited events and information (e.g, page faults and access bits) for the evaluation, and utilizes additional metadata which size is proportional to the physical memory rather than virtual memory size.

Chapter 3

APR: Adaptive Page Replacement

The APR scheme improves the performance of scientific applications by handling looping access patterns effectively. It utilizes two page replacement policies; CLOCK used as the default, and a fallback policy used for handling looping access patterns. The fallback policy is based on last-in-first-out (LIFO) and utilizes a *backward access checking* technique, which helps the fallback policy handle looping access patterns with various access window sizes.

Since it is common for looping access patterns to be interleaved with other access patterns, APR applies an individual replacement policy to each access pattern. We observe that objects in applications can be strong candidates for the source of the access patterns; thus, APR maintains an additional page list per object. The page list is called the *local page list* and it is managed by an individual page replacement policy.

PROCEDURE 1 Algorithm for APR.

Global Variables

global_page_list; ▷ Global list of used pages

freelist; ▷ Global list of free pages

obj_list; ▷ List of objects

```
1: procedure FREE_PAGE()
2:     victim_CLOCK = choose_victim_clock(global_page_list);
3:     object = find_object_by_page(obj_list, victim_CLOCK);
4:     victim_LIFO+ = choose_victim_lifop(object.page_list);
5:     victim = actual_victim(victim_CLOCK, victim_LIFO+, object.policy);
6:     evaluate(object, victim_CLOCK, victim_LIFO+);
7:     free_victim(victim, freelist);
8: end procedure

9: procedure HANDLE_PAGE_FAULT(addr)
10:    if freelist is empty then
11:        FREE_PAGE();
12:    end if
13:    page = get_page(freelist);
14:    object = find_object_by_addr(obj_list, addr);
15:    add_page(page, global_page_list, object.page_list);
16:    setup_page(page, addr);
17:    return page;
18: end procedure
```

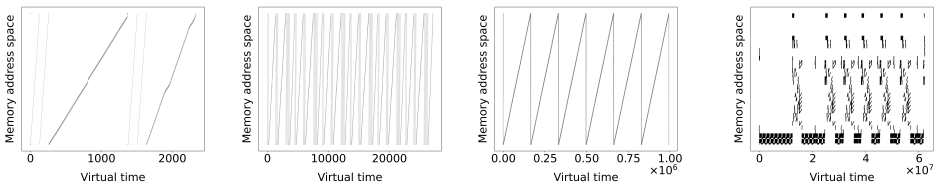
APR detects looping access patterns indirectly; it evaluates both the default

and the fallback policies in each object and uses the policy that fits to the object. This simplifies the procedure for explicitly distinguishing various kinds of looping access patterns from general access patterns. The algorithm for the APR scheme is described in Procedure 1 briefly. When a page fault occurs, a free page is allocated. However, if all pages are in use, one of them must be freed. To this end, APR first chooses a victim page (p_1) from the global page list using the default policy, which belongs to an object (o_1). Then, APR chooses another victim page (p_2) from the object (o_1) using the fallback policy. The two policies are evaluated by their victims for better choice, and finally the victim page of the active policy is freed.

In the following subchapters, we describe the APR scheme in more detail. Chapter 3.1 discusses the reason for maintaining local page lists and how they are utilized along with the global page list, Chapter 3.2 explains the fallback policy in detail, and Chapter 3.3 describes how APR evaluates the default and fallback policies online and chooses the proper policy for each object.

3.1 Local page list

APR targets handling looping access patterns mixed with other access patterns. Thus, two questions arise: first, how can we extract looping access patterns from a given global access sequence? Second, how can a looping access pattern be handled properly without affecting other access patterns? We adopt *object* information to address these questions. APR separates the global access sequence by objects, where an object is a memory area allocated to a process larger than a specific size. Figure 3.1 shows the memory access patterns of ocean_cp. In the figure, access patterns of three objects in the workload are displayed along with the global access pattern. The virtual time indicates the relative time of ac-



(a) local (b) global

Figure 3.1 Memory access patterns of ocean_cp. The first three access patterns are local access patterns in memory objects, and the last access pattern is the global access pattern.

cesses in each object. While the the workload exhibits a complex access pattern globally, each object has a simple access pattern; the looping access patterns can be easily extracted by separating the access pattern by memory object. We further find that this tendency is generally observable in other workloads.

In order to mitigate the second question, we isolate handling each looping access pattern; the fallback policy compensates the decision of the default policy. More specifically, APR first chooses a victim page using CLOCK in the global page list. When the page is from an object exhibiting a looping access pattern, the fallback policy re-selects the victim page in the object. To do this, APR maintains a local page list for each object as well as the global page list. In addition, since a page is managed by two page lists, the local page lists and the global page list have shelters for the access bit of each page; access information of a page in the global page list can be kept even if the access bit of the page is cleared in a local page list, and vice versa.

To identify objects, we use memory allocation functions (e.g., `malloc()`). If the size of memory allocation request is greater than a specific threshold, our scheme regards the allocated memory space as an object and creates a local

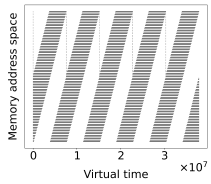


Figure 3.2 A memory access pattern of an object in water_nsquared.

```
for (mol = 0; mol < NMOL; mol++) {
    for (i = mol + 1; i <= mol + NMOL / 2; i++) {
        comp = i % NMOL;
        CSHIFTS(VAR[mol], VAR[comp]);
    }
}
```

Listing 3.1 A simplified code generating the looping access pattern in water_nsquared.

page list to manage it. In this study, the size threshold of an object is 40 KB, which is equivalent to 10 pages [1]. Additionally, APR maintains a *default* object, which manages pages that do not belong to any object.

3.2 Fallback policy: LIFO+

The fallback policy for looping access patterns, namely LIFO+, is fundamentally based on MRU, which is the optimal policy when pages are accessed repeatedly through loops [5]. However, it is not possible to implement MRU using only page faults and access bits. Thus, the fallback policy is practically based on LIFO, which can be an approximation of MRU when it comes to handling looping access patterns. This means that a local page list is in fact a LIFO stack; pages are pushed to and popped from the same side of the stack, which we call the *top* of the stack. The opposite side of the stack is called the *bottom*.

[1] We have tested a number of threshold values, and we have found that the 10-page threshold is the best in terms of the performance.

PROCEDURE 2 Algorithm for LIFO+.

```
 1: procedure CHOOSE_VICTIM_LIFOP(stack)
 2:     victim = NULL;
 3:     while victim == NULL do
 4:         page = page_entry(stack→rhead);
 5:         if !test_and_clear(&page→access_bit) then
 6:             victim = page;
 7:         end if
 8:         stack→rhead = stack→rhead.next;
 9:     end while
10:     return victim;
11: end procedure
```

In addition, we propose a *backward access checking* technique that prevents pages in the access window of a looping access pattern from being evicted. The need and the key intuition for the technique can be easily explained by the following example. Figure 3.2 shows the access pattern of an object in water_nsquared and Code 3.1 exhibits the corresponding code for the access pattern in the workload. The pages in the object *VAR* is accessed iteratively by loops, however, once a page is accessed, it is accessed repeatedly for a short period. performance because the recently fetched page will be used again in the period. Thus, pages that are being accessed repeatedly must be kept and the eviction should be performed below the pages in the stack. LIFO+ utilizes the access bit in order to prevent the pages from being evicted; it evicts the page nearest to the top that has the access bit cleared.

The algorithm of LIFO+ is shown in Procedure 2. A pointer `rhead` is maintained per stack and is used for finding the victim page in the stack. Its role is similar to that of the hand in CLOCK; it tests and clears the access bits of pages by moving through the list (stack) and it stops at the page with the access bit cleared. The page is chosen as a victim. When it reaches the bottom of the stack without finding a victim, it moves to the top of the stack.

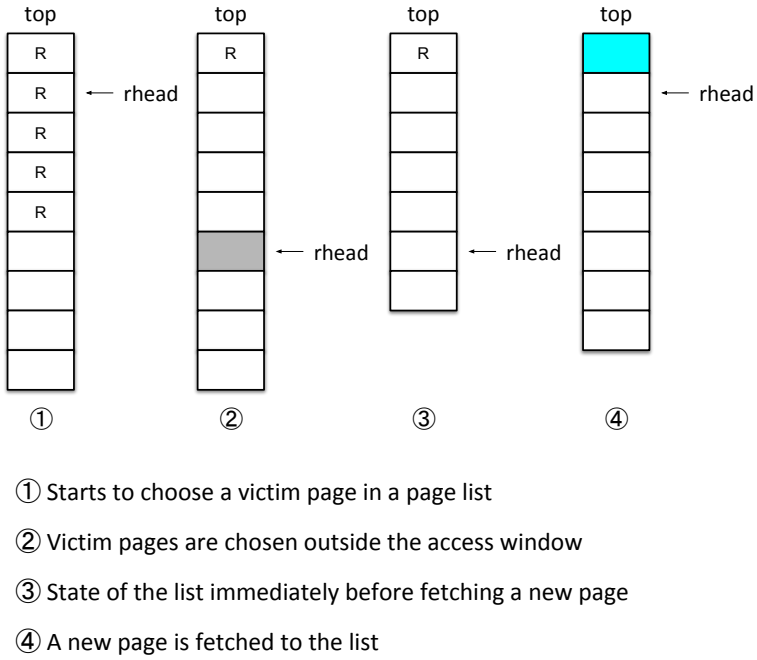


Figure 3.3 Example operation of LIFO+. The pages of an object are maintained in the local page list. A page with dark gray and a page with light blue represent a victim page and a newly fetched page, respectively. The R mark indicates that the access bit of a page is set.

When a new page is fetched to the stack, `rhead` is reset to pointing to the *second* page at the top of the stack. Since the most recently fetched page is likely to be re-accessed due to the large gap between the access size (a few bytes) and the page size (4KB), LIFO+ resets `rhead` to point to the second page rather than the first page at the top. Additionally, LIFO+ clears the access bit of the page that was previously at the top of the stack, in order to check if the page is actually in the period that it is repeatedly accessed.

Figure 3.3 describes the example operation of the LIFO+. `rhead` moves toward the bottom of the stack and stops at the page with the access bit cleared. This page is chosen as victim, and access bits of traversed pages are

cleared. When a page is fetched to the page list, the page previously at the top of the stack becomes the second page at the top. The access bit of the page is cleared, and the `rhead` is reset to pointing to the page.

3.3 Online evaluation

APR evaluates the default and fallback policies online in order to choose the best victim. Every time APR evicts a page, the default policy chooses a victim in the global page list. Then the fallback policy chooses another victim in the object that includes the victim of the default policy. The two policies are evaluated by their victim choices in each object; if the default policy is evaluated better than the fallback policy in performance, APR evicts the victim chosen by the default policy in the global page list. Otherwise, it evicts the victim chosen by the fallback policy in the local page list. The key idea for the evaluation is "better policy will choose idle pages earlier". If a page is chosen as victim by the fallback policy and later by the default policy without being accessed in between, APR considers praises the fallback policy. Otherwise, if the page chosen by the fallback policy is re-accessed before being chosen by the default policy, APR considers punishes the fallback policy; the policy have chosen an active page, not an idle page. This rule also holds when the two policies are interchanged in the situations above.

This idea is partially based on the *refault distance*, which is the distance in the access sequence between the eviction of a page and the soonest access to the page after the eviction. It can be a good metric to evaluate a policy, since the optimal policy is to evict a page with the largest refault distance [21]. If a policy chooses an idle page earlier than the other policy, it is reasonable to consider the decision as appropriate; it will result in the larger refault distance comparing to the decision of the other policy.

Replacement number	Victim		Evicted victim	Event	Score	
	Default	Fallback			Default	Fallback
...	...	...	...	...	...	...
12	A	B	A		0.6	0.4
13	**B**	C	B		0.3	0.2
14	D	E	E	Re-access to A observed	0.15	**0.6**
15	**C**	F	F		**- 0.175**	0.3
16	G	A	A		- 0.0875	**0.4**
...	...	...	...	...	...	...

Figure 3.4 An example of online evaluation. The figure shows the victims of the default and fallback policies at each page replacement in an object, the evicted victim among the two victims, additional events occurred, and scores of the two policies at the start of the replacement. A $\sim$ G represents the pages in the object and $d = 0.5$ in this example.

In order to evict a page among the victims of the two policies according to the online evaluation results, APR utilizes evaluation scores of the default and the fallback policy, and the victim of the policy with a higher score is evicted. The score is updated every time the decision of a policy is evaluated. If it was an appropriate decision, the score is incremented. Otherwise, the score is decremented. Additionally, in order to adapt to the change in the access pattern of an object, APR penalizes old evaluation results. More specifically, the evaluation result of the decision at $(t - \tau)$th page replacement has an influence of d^{τ} on tth page replacement in the object. The decay factor d controls how quickly the old results are forgotten; the scores of policies are decayed by d with each page replacement in an object.

Figure 3.4 displays an example of online evaluation. It shows the victims and scores of the policies, the victim that is evicted among the two victims, and the event happened in each page replacement in an object. The scores have the

values at the start of the page replacement before applying any policies. At the 12th page replacement, the scores of the default and fallback policies are 0.6 and 0.4, respectively. The scores are decayed by d (0.5 in the example), thus they become 0.3 and 0.2 at the start of the 13th page replacement. The default policy chooses the page B as its victim, however, it is previously chosen by the fallback policy in the 12th page replacement; Thus, the score of the fallback policy is increased by 0.5 (which is $d^{(13-12)}$) after decaying, and this makes the fallback policy have a higher score. At the 14th page replacement, access to page A, which is the victim of the default policy at the 12th page replacement, is observed. The page is accessed before being chosen by the fallback policy; thus, the score of the default policy is decreased by 0.25 (which is $d^{(14-12)}$) after decaying as well. At the 15th page replacement, the victim of the fallback policy at the 13th page replacement is chosen by the default policy. The score of the fallback policy is increased by 0.25 (which is $d^{(15-13)}$), and it becomes 0.4 at the start of the 16th page replacement. The page A, which is the victim of the default policy at the 12th page replacement, is chosen by the fallback policy in the 16th page replacement. However, it is re-accessed before being chosen again; thus, the score of the default policy is not changed except decaying.

APR needs to record three kinds of information for the evaluation: whether a page is already chosen, the policy that chose the page, and the page replacement number at the choice (e.g., records 10 if it was made at 10th page replacement in the object). The information is saved in the page descriptor, and is cleared after being used for the evaluation. It is also utilized in each policy to prevent choosing a page already chosen by itself; the page must be shown as evicted from memory to the policy as it expects. Unlike the victims of the policy with lower score, however, the victims of the policy with higher score are evicted from memory. In order to evaluate the former policy as well, we leave ghost pages in

the lists that contain only the information for the evaluation. If a ghost page is re-accessed or chosen by the other policy, it is evaluated and removed from the lists. However, maintaining all ghost pages until they are evaluated may result in an unexpectedly high space overhead, since the number of ghost pages in the worst case is proportional to the virtual memory size, not the physical memory size. Hence, APR maintains up to p ghost pages, where p is the number of page frames. It keeps the maximum number of ghost pages by dropping the oldest pages if necessary.

Evaluating the default and the fallback policies dynamically, APR applies the fallback policy selectively to objects properly without classifying the access pattern of each object explicitly. This removes the complexity of the page replacement scheme induced by the classification logic for various looping access patterns, and makes it possible to follow the changes in the access patterns.

Chapter 4

Evaluation

In this section, we evaluate the performance of APR. From the evaluation, we answer the following questions:

- How much performance is improved by APR?
- Does APR make a good decision for page replacement?
- Is the performance of APR sensitive to the decay factor?

4.1 Evaluation setup

We conducted trace-driven simulations to evaluate APR using the traces extracted from 12 workloads in the SPLASH-2x benchmark suite [10]. The details of the workloads are described in Table 4.1. The traces are extracted using a dynamic binary instrumentation tool [22]. In each trace, the total number of instructions and the memory allocation information (e.g., `malloc()`) are recorded in addition to referenced virtual addresses.

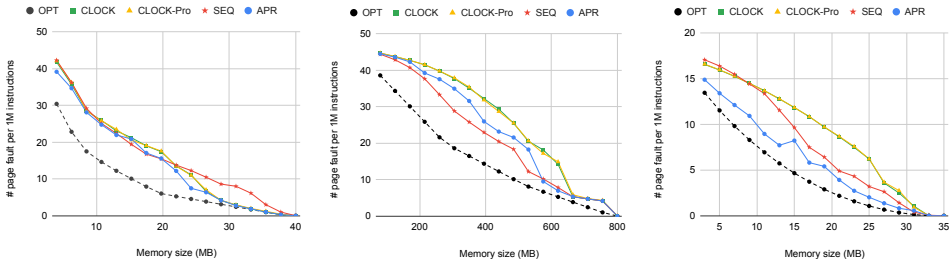
Table 4.1 Characteristics of workloads used in the evaluation.

Workload	**# insts. (millions)**	**# mem refs. (millions)**	**mem size (MB)**	**# objects**
cholesky	785	269	39	31
fft	15,916	5,033	787	5
lu_cb	22,881	7,915	33	2
lu_ncb	22,938	8,308	33	1
ocean_cp	15,540	7,324	908	27
ocean_ncp	17,792	7,834	1,151	12
radiosity	7,380	2,972	65	4
radix	11,896	1,796	1,049	2
raytrace	9,905	4,099	5.3	2
volrend	15,012	4,417	6.6	305
water_nsquared	16,496	6,626	3.4	2
water_spatial	29,122	11,574	23	0

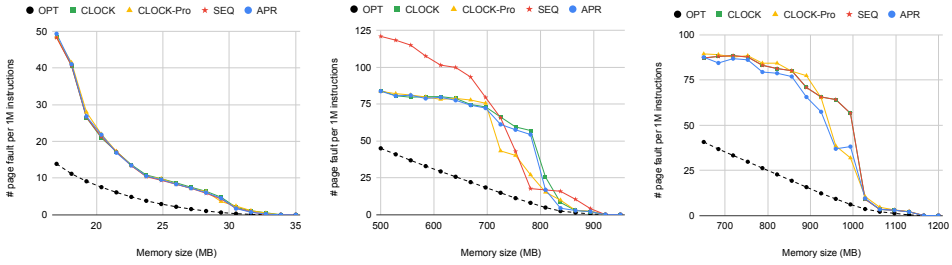
The performance of APR is compared with those of OPT [21], CLOCK, CLOCK-Pro, and SEQ. We add a modification to SEQ that replaces LRU with CLOCK for the default policy to utilize only events and information provided by virtual memory subsystems. OPT is an optimal page replacement algorithm that makes optimal decisions using future reference information. It is an offline algorithm; however, its role is to determine the upper bound for the performance of page replacement algorithms. In APR, the decay factor d is set to 0.7.

4.2 Performance of schemes

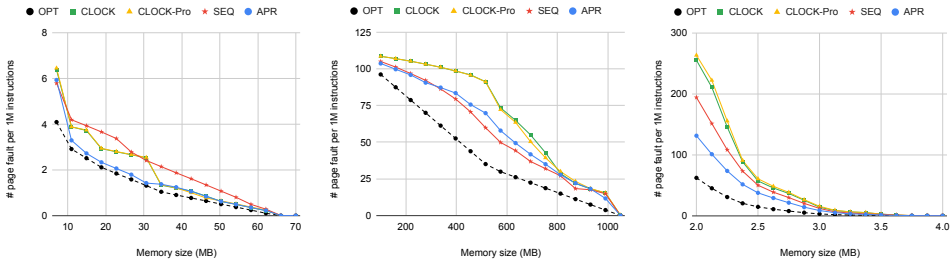
To compare the performance of APR with other page replacement schemes, we measured the number of page faults per million instructions with various memory sizes. Figure 4.1 shows the measured results with each workload. The memory size is increased until the performance saturates; in most cases, the saturation points are close to the virtual memory sizes of workloads, however,



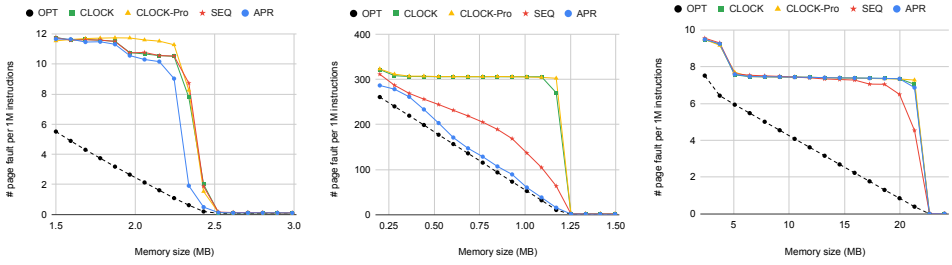
(a) cholesky (b) fft (c) lu_cb



(d) lu_ncb (e) ocean_cp (f) ocean_ncp



(g) radiosity (h) radix (i) raytrace



(j) volrend (k) water_nsquared (l) water_spatial

Figure 4.1 The performance of OPT, CLOCK, CLOCK-Pro, SEQ and APR.

the performances of some workloads saturate at memory sizes quite smaller than their virtual memory sizes (raytrace, volrend, and water_nsquared). In addition, we excluded the cold page faults that occurs when the memory is not yet full.

Since OPT is an optimal page replacement algorithm, it always shows the fewest page faults. The performance of CLOCK-Pro is almost the same as CLOCK in most cases except ocean_cp, ocean_ncp, and volrend. In ocean_cp and ocean_ncp, it outperforms CLOCK in most of the memory sizes; however, the amounts of the improvements are not large. Moreover, it shows performance even worse than CLOCK in volrend. SEQ, which explicitly detects and utilizes looping access patterns, shows improvement higher than CLOCK-Pro. Its performance is even higher than APR in two cases: fft and radix. This is due to the detection algorithm of SEQ; since it observes the page fault sequence in address space, it can detect sequential loops early. Contrarily, APR requires the online evaluation results of the two policies in order to identify looping access patterns, thus APR detects looping access patterns after a period of time. Nevertheless, due to the limited range of looping access patterns detected by SEQ, its performance improvement is smaller than APR in water_nsquared, which exploits looping access patterns with the access window. Furthermore, SEQ lacks proper handling of looping access patterns in terms of performance; thus, the performance is even worse than that of CLOCK in cholesky, ocean_cp and radiosity.

Compared to the other three online schemes (CLOCK, CLOCK-Pro and SEQ), APR shows much greater performance. More specifically, it outperforms the three schemes consistently in 5 workloads (lu_cb, radiosity, raytrace, volrend, and water_nsquared), and shows remarkable improvement over the three schemes in 2 workloads (cholesky and ocean_ncp). Additionally, unlike CLOCK-

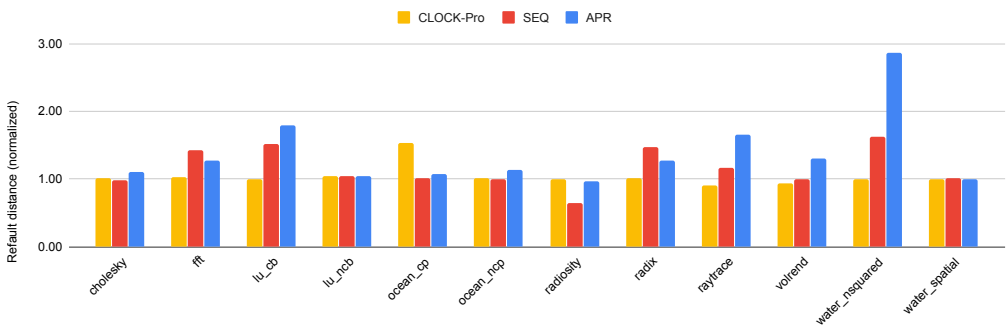
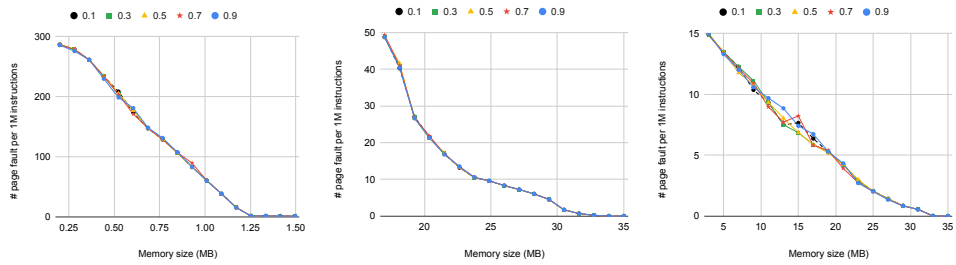


Figure 4.2 The average refault distance relative to CLOCK. The memory size of each workload is equal to the mean of the memory sizes in Figure 4.1.

Pro and SEQ, it shows only negligible performance degradation compared to CLOCK, even in the worst case. In radiosity and water_nsquared, APR shows performance close to OPT, meaning that it successfully detects and handles looping access patterns crucial for optimal performance.

The key idea of the online evaluation algorithm of APR is based on refault distances; it praises the policy that evicts an idle page earlier, which results in the larger refault distance. Since the refault distance is highly related to the performance of a page replacement scheme, we measure the average refault distances of CLOCK, CLOCK-Pro, SEQ and APR. The schemes are simulated using workloads used in Figure 4.1 with the average memory sizes. To calculate the average refault distance, we record the eviction and re-access time of each page replacement victims. Figure 4.2 shows the average refault distances of CLOCK-Pro, SEQ, and APR relative to CLOCK. CLOCK-Pro shows refault distances equal to or less than CLOCK except for ocean_cp, which is its beneficiary also observed in Figure 4.1. Also, SEQ shows larger refault distances than CLOCK in workloads that benefits from SEQ in Figure 4.1. This tendency also holds for APR, and it shows large refault distances in overall. Unlike the two



(a) water_nsquared (b) lu_ncb (c) lu_cb

Figure 4.3 The effect of the decay factor value on the performance. The performance of APR with various decay factors (0.1, 0.3, 0.5, 0.7 and 0.9) are measured using three workloads used in Figure 4.1.

previous schemes, moreover, its refault distance is almost the same as that of CLOCK in the worst case.

In summary, APR successfully improves performance compared to other schemes and maintains performance at a level comparable to CLOCK in the worst case, unlike previous schemes. The performance of 9 out of 12 workloads is markedly increased by APR. APR improves the performance by 22.0%, 22.1%, and 16.4% on average compared to CLOCK, CLOCK-Pro, and SEQ, respectively. In addition, refault distances of APR tell that it also properly decides the page for replacement. Comparing the performance and refault distances of the schemes, we are able to validate the online evaluation rule of APR.

4.3 Sensitivity to the decay factor

APR utilizes the decay factor d in the online evaluation algorithm to follow the change in access patterns properly. If the value is too small, APR will try to adapt to every small variance in access patterns. If the value is too large, APR will fail to adapt to the change quickly. In order to see the performance sensitivity to the parameter, we measure the performance of APR with various parameter values: 0.1, 0.3, 0.5, 0.7 and 0.9. Figure 4.3 shows the performance of

three workloads: a workload that largely benefit from APR (water_nsquared), a workload that APR shows performance same as CLOCK (lu_ncb), and a workload that shows the highest sensitivity to the parameter in 12 workloads (lu_cb). The effect of the parameter value on the performance is negligible in the first two workloads. The performance of the most sensitive workload is affected by the parameter in memory sizes around 15 MB, however, the effect is quite small in other memory sizes. Also, comparing the performance of APR at the point to those of the other schemes, the variance is small enough that the performance order of the schemes are not changed. We further observe that the performance of the other 9 workloads is not sensitive to the parameter as well.

Chapter 5

Conclusion

The rate of increase in the amount of data utilized in modern computing workloads, has outpaced the growth rate of capacity of main memory devices. Thus, memory management is becoming an important factor for the performance of workloads than before. The tendency is also observed in scientific workloads, which have a common characteristic that they repeat operations on intermediate data iteratively using loops. However, it is known that such looping access patterns are handled poorly by LRU and CLOCK, which are generally used as a page replacement policy in modern operating systems. Moreover, since the intermediate data is usually not maintained in files, previous works on dealing with such access patterns are hard to adopted in this case.

In this paper, we have proposed APR, a page replacement scheme that handles looping access patterns in scientific applications efficiently. APR can detect a wide range of looping access patterns, and deal with them carefully to improve the resulting performance. Moreover, APR is built on limited information and events (e.g., page faults and access bits); thus, it can be used to manage a wide

range of pages, including pages not backed by files. We evaluated APR using traces from 12 workloads in the SPLASH-2x benchmark suite and compared its performance to previous works. The evaluation results show that APR successfully reduces the number of page faults in most workloads by handling looping access patterns in the workloads effectively.

Bibliography

[1] C. P. Chen and C.-Y. Zhang, "Data-intensive applications, challenges, techniques and technologies: A survey on big data," *Information sciences*, vol. 275, pp. 314–347, 2014.

[2] E. Jonas, J. Schleier-Smith, V. Sreekanti, C.-C. Tsai, A. Khandelwal, Q. Pu, V. Shankar, J. Carreira, K. Krauth, N. Yadwadkar, *et al.*, "Cloud programming simplified: A berkeley view on serverless computing," *arXiv preprint arXiv:1902.03383*, 2019.

[3] F. J. Corbato, "A paging experiment with the multics system," *MIT Project MAC Report MAC-M-384*, May 1968.

[4] S. Jiang and X. Zhang, "Token-ordered lru: an effective page replacement policy and its implementation in linux systems," *Performance Evaluation*, vol. 60, no. 1-4, pp. 5–29, 2005.

[5] G. Glass and P. Cao, "Adaptive page replacement based on memory reference behavior," *ACM SIGMETRICS Performance Evaluation Review*, vol. 25, no. 1, pp. 115–126, 1997.

[6] "malloc(3) - linux man page." `https://www.man7.org/linux/man-pages/man3/malloc.3.html`. Accessed: 2020-05-27.

[7] Y. Smaragdakis, S. Kaplan, and P. Wilson, "Eelru: simple and effective adaptive page replacement," *ACM SIGMETRICS Performance Evaluation Review*, vol. 27, no. 1, pp. 122–133, 1999.

[8] J. M. Kim, J. Choi, J. Kim, S. H. Noh, S. L. Min, Y. Cho, and C. S. Kim, "A low-overhead high-performance unified buffer management scheme that exploits sequential and looping references," in *Proceedings of the 4th conference on Symposium on Operating System Design & Implementation-Volume 4*, p. 9, USENIX Association, 2000.

[9] S. Jiang, F. Chen, and X. Zhang, "Clock-pro: An effective improvement of the clock replacement.," in *USENIX Annual Technical Conference, General Track*, pp. 323–336, 2005.

[10] X. Zhan, Y. Bao, C. Bienia, and K. Li, "Parsec3. 0: A multicore benchmark suite with network stacks and splash-2x," *ACM SIGARCH Computer Architecture News*, vol. 44, no. 5, pp. 1–16, 2017.

[11] E. J. O'neil, P. E. O'neil, and G. Weikum, "The lru-k page replacement algorithm for database disk buffering," *Acm Sigmod Record*, vol. 22, no. 2, pp. 297–306, 1993.

[12] T. Johnson, D. Shasha, *et al.*, "2q: a low overhead high performance bu er management replacement algorithm," in *Proceedings of the 20th International Conference on Very Large Data Bases*, pp. 439–450, Citeseer, 1994.

[13] D. Lee, J. Choi, J.-H. Kim, S. H. Noh, S. L. Min, Y. Cho, and C. S. Kim, "Lrfu: A spectrum of policies that subsumes the least recently used and least frequently used policies," *IEEE transactions on Computers*, no. 12, pp. 1352–1361, 2001.

[14] S. Jiang and X. Zhang, "Lirs: an efficient low inter-reference recency set replacement policy to improve buffer cache performance," *ACM SIGMETRICS Performance Evaluation Review*, vol. 30, no. 1, pp. 31–42, 2002.

[15] N. Megiddo and D. S. Modha, "Arc: A self-tuning, low overhead replacement cache.," in *FAST*, vol. 3, pp. 115–130, 2003.

[16] S. Bansal and D. S. Modha, "Car: Clock with adaptive replacement.," in *FAST*, vol. 4, pp. 187–200, 2004.

[17] A. Jaleel, K. B. Theobald, S. C. Steely Jr, and J. Emer, "High performance cache replacement using re-reference interval prediction (rrip)," *ACM SIGARCH Computer Architecture News*, vol. 38, no. 3, pp. 60–71, 2010.

[18] I. Ari, A. Amer, R. B. Gramacy, E. L. Miller, S. A. Brandt, and D. D. Long, "Acme: Adaptive caching using multiple experts.," in *WDAS*, pp. 143–158, 2002.

[19] M. K. Qureshi, A. Jaleel, Y. N. Patt, S. C. Steely, and J. Emer, "Adaptive insertion policies for high performance caching," *ACM SIGARCH Computer Architecture News*, vol. 35, no. 2, pp. 381–391, 2007.

[20] G. Vietri, L. V. Rodriguez, W. A. Martinez, S. Lyons, J. Liu, R. Rangaswami, M. Zhao, and G. Narasimhan, "Driving cache replacement with ml-based lecar," in *10th USENIX Workshop on Hot Topics in Storage and File Systems (HotStorage 18)*, 2018.

[21] L. A. Belady, "A study of replacement algorithms for a virtual-storage computer," *IBM Systems journal*, vol. 5, no. 2, pp. 78–101, 1966.

[22] C.-K. Luk, R. Cohn, R. Muth, H. Patil, A. Klauser, G. Lowney, S. Wallace, V. J. Reddi, and K. Hazelwood, "Pin: building customized program analysis tools with dynamic instrumentation," *Acm sigplan notices*, vol. 40, no. 6, pp. 190–200, 2005.

초록

현대 컴퓨팅 워크로드에서의 데이터 사용량의 비약적인 증가와 메인 메모리 기기 용량의 느린 발전 속도로 인해, 응용 성능에 있어서 효율적인 메모리 관리의 중요성이 증가하고 있다. 과학 계산 응용 또한 처리하는 데이터의 양이 이전과 비교하여 크게 증가하였고, 해당 응용들은 주로 반복문을 통해 중간 데이터에 반복적으로 연산을 적용다는 특징을 갖는다. 그러나, 운영체제에서 일반적으로 페이지 교체 기법으로 사용되는 LRU는 해당 접근 패턴에 대해 좋은 성능을 보이지 않는다. 본 논문에서 우리는 과학 계산 응용에서의 루프 접근 패턴을 효율적으로 처리해주는 적응형 페이지 교체 (APR) 기법을 제안한다. APR은 다양한 루프 접근 패턴을 실시간으로 파악하고, 결과적으로 성능을 높일 수 있도록 처리해준다. 제안하는 기법은 운영체제의 가상 메모리 서브 시스템이 제공하는 제한된 이벤트 및 정보 (예를 들어, 페이지 폴트 및 접근 비트) 만을 사용하여 구현 가능하다. 우리는 SPLASH-2x의 워크로드로부터 트레이스를 추출하여, 트레이스 기반 시뮬레이션을 통해 APR을 평가하였다. 기존 기법들과의 비교를 통해, 우리는 APR이 기존 기법들의 단점을 적절히 개선함으로써 성능을 성공적으로 개선함을 보였다.

주요어: page replacement, access pattern, scientific applications, trace-driven simulation, memory management

학번: 2018-20169