CLOCK-Pro: An Effective Improvement of the CLOCK Replacement

Song Jiang

Performance and Architecture Laboratory (PAL)
Los Alamos National Laboratory, CCS Division
Los Alamos, NM 87545, USA
sjiang@lanl.gov

Feng Chen and Xiaodong Zhang

Computer Science Department
College of William and Mary
Williamsburg, VA 23187, USA
{fchen, zhang}@cs.wm.edu

Abstract

With the ever-growing performance gap between memory systems and disks, and rapidly improving CPU performance, virtual memory (VM) management becomes increasingly important for overall system performance. However, one of its critical components, the page replacement policy, is still dominated by CLOCK, a replacement policy developed almost 40 years ago. While pure LRU has an unaffordable cost in VM, CLOCK simulates the LRU replacement algorithm with a low cost acceptable in VM management. Over the last three decades, the inability of LRU as well as CLOCK to handle weak locality accesses has become increasingly serious, and an effective fix becomes increasingly desirable.

Inspired by our I/O buffer cache replacement algorithm, LIRS [13], we propose an improved CLOCK replacement policy, called CLOCK-Pro. By additionally keeping track of a limited number of replaced pages, CLOCK-Pro works in a similar fashion as CLOCK with a VM-affordable cost. Furthermore, it brings all the much-needed performance advantages from LIRS into CLOCK. Measurements from an implementation of CLOCK-Pro in Linux Kernel 2.4.21 show that the execution times of some commonly used programs can be reduced by up to 47%.

1 Introduction

1.1 Motivation

Memory management has been actively studied for decades. On one hand, to use installed memory effectively, much work has been done on memory allocation, recycling, and memory management in various programming languages. Many solutions and significant improvements have been seen in both theory and practice. On the other hand, aiming at reducing the cost of paging between memory and disks, researchers and practitioners in both academia and industry are working hard to improve the performance of page replacement, especially to avoid the worst performance cases. A significant advance in this regard becomes increasingly demanding with the continuously growing gap between memory and disk access times, as well as rapidly improving CPU performance. Although increasing memory size can always reduce I/O pagings by giving a larger memory space to hold the working set, one cannot cache all the previously accessed data including file data in memory. Meanwhile, VM system designers should attempt to maximize the achievable performance under different application demands and system configurations. An effective replacement policy is critical in achieving the goal. Unfortunately, an approximation of LRU, the CLOCK replacement policy [5], which was developed almost 40 years ago, is still dominating nearly all the major operating systems including MVS, Unix, Linux and Windows [7] [1], even though it has apparent performance disadvantages inherited from LRU with certain commonly observed memory access behaviors.

We believe that there are two reasons responsible for the lack of significant improvements on VM page replacements. First, there is a very stringent cost requirement on the policy demanded by the VM management, which requires that the cost be associated with the number of page faults or a moderate constant. As we know, a page fault incurs a penalty worth of hundreds of thousands of CPU cycles. This allows a replacement policy to do its job without intrusively interfering with application executions. However, a policy with its cost proportional to the number of memory references would be prohibitively expensive, such as doing some bookkeeping on every memory access. This can cause the user program to generate a trap into the operating system on every memory instruction, and the CPU would consume much more cycles on page replacement than on user programs, even when there are no paging requests. From the cost perspective, even LRU, a well-recognized low-cost and simple replacement algorithm, is unaffordable, because it has to maintain the LRU ordering of pages for each page access. The second reason is that most proposed replacement algorithms attempting to improve LRU

[1] This generally covers many CLOCK variants, including Mach-style active/inactive list, FIFO list facilitated with hardware reference bits. These CLOCK variants share similar performance problems plaguing LRU.

performance turn out to be too complicated to produce their approximations with their costs meeting the requirements of VM. This is because the weak cases for LRU mostly come from its minimal use of history access information, which motivates other researchers to take a different approach by adding more bookkeeping and access statistic analysis work to make their algorithms more intelligent in dealing with some access patterns unfriendly to LRU.

1.2 The Contributions of this Paper

The objective of our work is to provide a VM page replacement algorithm to take the place of CLOCK, which meets both the performance demand from application users and the low overhead requirement from system designers.

Inspired by the I/O buffer cache replacement algorithm, LIRS [13], we design an improved CLOCK replacement, called CLOCK-Pro. LIRS, originally invented to serve I/O buffer cache, has a cost unacceptable to VM management, even though it holds apparent performance advantages relative to LRU. We integrate the principle of LIRS and the way in which CLOCK works into CLOCK-Pro. By proposing CLOCK-Pro, we make several contributions: (1) CLOCK-Pro works in a similar fashion as CLOCK and its cost is easily affordable in VM management. (2) CLOCK-Pro brings all the much-needed performance advantages from LIRS into CLOCK. (3) Without any pre-determined parameters, CLOCK-Pro adapts to the changing access patterns to serve a broad spectrum of workloads. (4) Through extensive simulations on real-life I/O and VM traces, we have shown the significant page fault reductions of CLOCK-Pro over CLOCK as well as other representative VM replacement algorithms. (5) Measurement results from an implementation of CLOCK-Pro in a Linux kernel show that the execution times of some commonly used programs can be reduced by up to 47%.

2 Background

2.1 Limitations of LRU/CLOCK

LRU is designed on an assumption that a page would be reaccessed soon after it was accessed. It manages a data structure conventionally called LRU stack, in which the Most Recently Used (MRU) page is at the stack top and the Least Recently Used (LRU) page is at the stack bottom. The ordering of other in-between pages in the stack strictly follows their last access times. To maintain the stack, the LRU algorithm has to move an accessed page from its current position in the stack (if it is in the stack) to the stack top. The LRU page at the stack bottom is the one to be replaced if there is a page fault and no free spaces are available. In CLOCK, the memory spaces holding the pages can be regarded as a circular buffer and the replacement algorithm cycles through the pages in the circular buffer, like the hand of a clock. Each page is associated with a bit, called reference bit, which is set by hardware whenever the page is accessed. When it is necessary to replace a page to service a page fault, the page pointed to by the hand is checked. If its reference bit is unset, the page is replaced. Otherwise, the algorithm resets its reference bit and keeps moving the hand to the next page. Research and experience have shown that CLOCK is a close approximation of LRU, and its performance characteristics are very similar to those of LRU. So all the performance disadvantages discussed below about LRU are also applied to CLOCK.

The LRU assumption is valid for a significant portion of workloads, and LRU works well for these workloads, which are called LRU-friendly workloads. The distance of a page in the LRU stack from the stack top to its current position is called **recency**, which is the number of other distinct pages accessed after the last reference to the page. Assuming an unlimitedly long LRU stack, the distance of a page in the stack away from the top when it is accessed is called its **reuse distance**, which is equivalent to the number of other distinct pages accessed between its last access and its current access. LRU-friendly workloads have two distinct characteristics: (1) There are much more references with small reuse distances than those with large reuse distances; (2) Most references have reuse distances smaller than the available memory size in terms of the number of pages. The locality exhibited in this type of workloads is regarded as strong, which ensures a high hit ratio and a steady increase of hit ratio with the increase of memory size.

However, there are indeed cases in which this assumption does not hold, where LRU performance could be unacceptably degraded. One example access pattern is memory scan, which consists of a sequence of one-time page accesses. These pages actually have infinitely large reuse distance and cause no hits. More seriously, in LRU, the scan could flush all the previously active pages out of memory.

As an example, in Linux the memory management for process-mapped program memory and file I/O buffer cache is unified, so the memory can be flexibly allocated between them according to their respective needs. The allocation balancing between program memory and buffer cache poses a big problem because of the unification. This problem is discussed in [22]. We know that there are a large amount of data in file systems, and the total number of accesses to the file cache could also be very large. However, the access frequency to each individual page of file data is usually low. In a burst of file accesses, most of the memory could serve as a file cache. Meanwhile, the process pages are evicted to make space for the actually infrequently accessed file pages, even though they are frequently accessed. An example scenario on this is that right after one extracts a large tarball, he/she could sense that the computer becomes slower because the previous active working set is replaced and has to be faulted in. To address this problem in a simple way, current Linux versions have to in-

troduce some "magic parameters" to enforce the buffer cache allocation to be in the range of 1% to 15% of memory size by default [22]. However, this approach does not fundamentally solve the problem, because the major reason causing the allocation unbalancing between process memory and buffer cache is the ineffectiveness of the replacement policy in dealing with infrequently accessed pages in buffer caches.

Another representative access pattern defeating LRU is loop, where a set of pages are accessed cyclically. Loop and loop-like access patterns dominate the memory access behaviors of many programs, particularly in scientific computation applications. If the pages involved in a loop cannot completely fit in the memory, there are repeated page faults and no hits at all. The most cited example for the loop problem is that even if one has a memory of 100 pages to hold 101 pages of data, the hit ratio would be ZERO for a looping over this data set [9, 24]!

2.2 LIRS and its Performance Advantages

A recent breakthrough in replacement algorithm designs, called LIRS (Low Inter-reference Recency Set) replacement [13], removes all the aforementioned LRU performance limitations while still maintaining a low cost close to LRU. It can not only fix the scan and loop problems, but also can accurately differentiate the pages based on their locality strengths quantified by reuse distance.

A key and unique approach in handling history access information in LIRS is that it uses reuse distance rather than recency in LRU for its replacement decision. In LIRS, a page with a large reuse distance will be replaced even if it has a small recency. For instance, when a one-time-used page is recently accessed in a memory scan, LIRS will replace it quickly because its reuse distance is infinite, even though its recency is very small. In contrast, LRU lacks the insights of LIRS: all accessed pages are indiscriminately cached until either of two cases happens to them: (1) they are re-accessed when they are in the stack, and (2) they are replaced at the bottom of the stack. LRU does not take account of which of the two cases has a higher probability. For infrequently accessed pages, which are highly possible to be replaced at the stack bottom without being re-accessed in the stack, holding them in memory (as well as in stack) certainly results in a waste of the memory resources. This explains the LRU misbehavior with the access patterns of weak locality.

3 Related Work

There have been a large number of new replacement algorithms proposed over the decades, especially in the last fifteen years. Almost all of them are proposed to target the performance problems of LRU. In general, there are three approaches taken in these algorithms. (1) Requiring applications to explicitly provide future access hints, such as application-controlled file caching [3], and application-informed prefetching and caching [20]; (2) Explicitly detecting the access patterns failing LRU and adaptively switching to other effective replacements, such as SEQ [9], EELRU [24], and UBM [14]; (3) Tracing and utilizing deeper history access information such as FBR [21], LRFU [15], LRU-2 [18], 2Q [12], MQ [29], LIRS [13], and ARC [16]. More elaborate description and analysis on the algorithms can be found in [13]. The algorithms taking the first two approaches usually place too many constraints on the applications to be applicable in the VM management of a general-purpose OS. For example, SEQ is designed to work in VM management, and it only does its job when there is a page fault. However, its performance depends on an effective detection of long sequential address reference patterns, where LRU behaves poorly. Thus, SEQ loses generality because of the mechanism it uses. For instance, it is hard for SEQ to detect loop accesses over linked lists. Among the algorithms taking the third approach, FBR, LRU-2, LRFU and MQ are expensive compared with LRU. The performance of 2Q has been shown to be very sensitive to its parameters and could be much worse than LRU [13]. LIRS uses reuse distance, which has been used to characterize and to improve data access locality in programs (see e.g. [6]). LIRS and ARC are the two most promising candidate algorithms that have a potential leading to low-cost replacement policies applicable in VM, because they use data structure and operations similar to LRU and their cost is close to LRU.

ARC maintains two variably-sized lists holding history access information of referenced pages. Their combined size is two times of the number of pages in the memory. So ARC not only records the information of cached pages, but also keeps track of the same number of replaced pages. The first list contains pages that have been touched only once recently (*cold pages*) and the second list contains pages that have been touched at least twice recently (*hot pages*). The cache spaces allocated to the pages in these two lists are adaptively changed, depending on in which list the recent misses happen. More cache spaces will serve cold pages if there are more misses in the first list. Similarly, more cache spaces will serve hot pages if there are more misses in the second list. However, though ARC allocates memory to hot/cold pages adaptively according to the ratio of cold/hot page accesses and excludes tunable parameters, the locality of pages in the two lists, which are supposed to hold cold and hot pages respectively, can not directly and consistently be compared. So the hot pages in the second list could have a weaker locality in terms of reuse distance than the cold pages in the first list. For example, a page that is regularly accessed with a reuse distance a little bit more than the memory size can have no hits at all in ARC, while a page in the second list can stay in memory without any accesses, since it has been accepted into the list. This does not happen in LIRS, because any pages supposed to be hot or cold are placed in the same list and compared in a consistent fash-

ion. There is one pre-determined parameter in the LIRS algorithm on the amount of memory allocation for cold pages. In CLOCK-Pro, the parameter is removed and the allocation becomes fully adaptive to the current access patterns.

Compared with the research on the general replacement algorithms targeting LRU, the work specific to the VM replacements and targeting CLOCK is much less and is inadequate. While Second Chance (SC) [28], being the simplest variant of CLOCK algorithm, utilizes only one reference bit to indicate recency, other CLOCK variants introduce a finer distinction between page access history. In a generalized CLOCK version called GCLOCK [25, 17], a counter is associated with each page rather than a single bit. Its counter will be incremented if a page is hit. The cycling clock hand sweeps over the pages decrementing their counters until a page whose counter is zero is found for replacement. In Linux and FreeBSD, a similar mechanism called page aging is used. The counter is called *age* in Linux or *act_count* in FreeBSD. When scanning through memory for pages to replace, the page age is increased by a constant if its reference bit is set. Otherwise its age is decreased by a constant. One problem for this kind of design is that they cannot consistently improve LRU performance. The parameters for setting the maximum value of counters or adjusting ages are mostly empirically decided. Another problem is that they consume too many CPU cycles and adjust to changes of access patterns slowly, as evidenced in Linux kernel 2.0. Recently, an approximation version of ARC, called CAR [2], has been proposed, which has a cost close to CLOCK. Their simulation tests on the I/O traces indicate that CAR has a performance similar to ARC. The results of our experiments on I/O and VM traces show that CLOCK-Pro has a better performance than CAR.

In the design of VM replacements it is difficult to obtain much improvement in LRU due to its stringent cost constraint, yet this problem remains a demanding challenge in the OS development.

4 Description of CLOCK-Pro

4.1 Main Idea

CLOCK-Pro takes the same principle as that of LIRS – it uses the reuse distance (called IRR in LIRS) rather than recency in its replacement decision. When a page is accessed, the reuse distance is the period of time in terms of the number of other distinct pages accessed since its last access. Although there is a reuse distance between any two consecutive references to a page, only the most current distance is relevant in the replacement decision. We use the reuse distance of a page at the time of its access to categorize it either as a cold page if it has a large reuse distance, or as a hot page if it has a small reuse distance. Then we mark its status as being cold or hot. We place all the accessed pages, either hot or cold, into one single list [2] in the order of their accesses [3]. In the list, the pages with small recencies are at the list head, and the pages with large recencies are at the list tail.

To give the cold pages a chance to compete with the hot pages and to ensure their cold/hot statuses accurately reflect their current access behavior, we grant a cold page a test period once it is accepted into the list. Then, if it is re-accessed during its test period, the cold page turns into a hot page. If the cold page passes the test period without a re-access, it will leave the list. Note that the cold page in its test period can be replaced out of memory, however, its page metadata remains in the list for the test purpose until the end of the test period or being re-accessed. When it is necessary to generate a free space, we replace a resident cold page.

The key question here is how to set the time of the test period. When a cold page is in the list and there is still at least one hot page after it (i.e., with a larger recency), it should turn into a hot page if it is accessed, because it has a new reuse distance smaller than the hot page(s) after it. Accordingly, the hot page with the largest recency should turn into a cold page. So the test period should be set as the largest recency of the hot pages. If we make sure that the hot page with the largest recency is always at the list tail, and all the cold pages that pass this hot page terminate their test periods, then the test period of a cold page is equal to the time before it passes the tail of the list. So all the non-resident cold pages can be removed from the list right after they reach the tail of the list. In practice, we could shorten the test period and limit the number of cold pages in the test period to reduce space cost. By implementing this testing mechanism, we make sure that "cold/hot" are defined based on relativity and by constant comparison in one clock, not on a fixed threshold that are used to separate the pages into two lists. This makes CLOCK-Pro distinctive from prior work including 2Q and CAR, which attempt to use a constant threshold to distinguish the two types of pages, and to treat them differently in their respective lists (2Q has two queues, and CAR has two clocks), which unfortunately causes these algorithms to share some of LRU's performance weakness.

4.2 Data Structure

Let us first assume that the memory allocations for the hot and cold pages, m_h and m_c, respectively, are fixed, where $m_h + m_c$ is the total memory size m ($m = m_h + m_c$). The number of the hot pages is also m_h, so all the hot pages are always cached. If a hot page is going to be replaced, it must first change into a cold page. Apart from the hot pages, all the other accessed pages are categorized as cold pages. Among the cold pages, m_c pages are cached, another at most m non-resident

[2] Actually it is the metadata of a page that is placed in the list.

[3] Actually we can only maintain an approximate access order, because we cannot update the list with a hit access in a VM replacement algorithm, thus losing the exact access orderings between page faults.

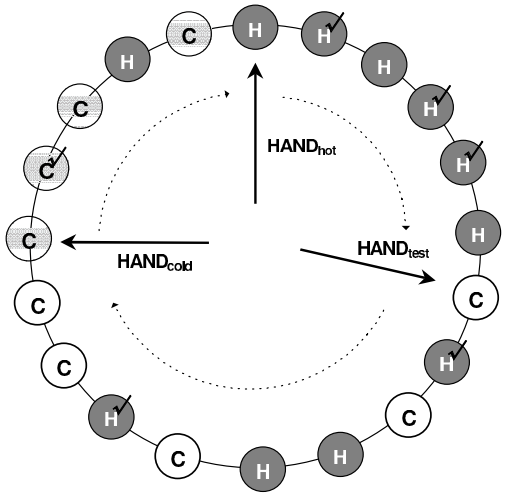


Figure 1: There are three types of pages in CLOCK-Pro, hot pages marked with "H", cold pages marked with "C" (shadowed circles for resident cold pages, non-shadowed circles for non-resident cold pages). Around the clock, there are three hands: **HAND**$_{hot}$ pointing to the list tail (i.e. the last hot page) and used to search for a hot page to turn into a cold page, **HAND**$_{cold}$ pointing to the last resident cold page and used to search for a cold page to replace, and **HAND**$_{test}$ pointing to the last cold page in the test period, terminating test periods of cold pages, and removing non-resident cold pages passing the test period out of the list. The "$\surd$" marks represent the reference bits of 1.

cold pages only have their history access information cached. So totally there are at most $2m$ metadata entries for keeping track of page access history in the list. As in CLOCK, all the page entries are organized as a circular linked list, shown in Figure 1. For each page, there is a cold/hot status associated with it. For each cold page, there is a flag indicating if the page is in the test period.

In CLOCK-Pro, there are three hands. The **HAND**$_{hot}$ points to the hot page with the largest recency. The position of this hand actually serves as a threshold of being a hot page. Any hot pages swept by the hand turn into cold ones. For the convenience of the presentation, we call the page pointed to by **HAND**$_{hot}$ as the tail of the list, and the page immediately after the tail page in the clockwise direction as the head of the list. **HAND**$_{cold}$ points to the last resident cold page (i.e., the furthest one to the list head). Because we always select this cold page for replacement, this is the position where we start to look for a victim page, equivalent to the hand in CLOCK. **HAND**$_{test}$ points to the last cold page in the test period. This hand is used to terminate the test period of cold pages. The non-resident cold pages swept over by this hand will leave the circular list. All the hands move in the clockwise direction.

4.3 Operations on Searching Victim Pages

Just as in CLOCK, there are no operations in CLOCK-Pro for page hits, only the reference bits of the accessed pages are set by hardware. Before we see how a victim page is generated, let us examine how the three hands move around the clock, because the victim page is searched by coordinating the movements of the hands.

HAND$_{cold}$ is used to search for a resident cold page for replacement. If the reference bit of the cold page currently pointed to by **HAND**$_{cold}$ is unset, we replace the cold page for a free space. The replaced cold page will remain in the list as a non-resident cold page until it runs out of its test period, if it is in its test period. If not, we move it out of the clock. However, if its bit is set and it is in its test period, we turn the cold page into a hot page, and ask **HAND**$_{hot}$ for its actions, because an access during the test period indicates a competitively small reuse distance. If its bit is set but it is not in its test period, there are no status change as well as **HAND**$_{hot}$ actions. In both of the cases, its reference bit is reset, and we move it to the list head. The hand will keep moving until it encounters a cold page eligible for replacement, and stops at the next resident cold page.

As mentioned above, what triggers the movement of **HAND**$_{hot}$ is that a cold page is found to have been accessed in its test period and thus turns into a hot page, which maybe accordingly turns the hot page with the largest recency into a cold page. If the reference bit of the hot page pointed to by **HAND**$_{hot}$ is unset, we can simply change its status and then move the hand forward. However, if the bit is set, which indicates the page has been re-accessed, we spare this page, reset its reference bit and keep it as a hot page. This is because the actual access time of the hot page could be earlier than the cold page. Then we move the hand forward and do the same on the hot pages with their bits set until the hand encounters a hot page with a reference bit of zero. Then the hot page turns into a cold page. Note that moving **HAND**$_{hot}$ forward is equivalent to leaving the page it moves by at the list head. Whenever the hand encounters a cold page, it will terminate the page's test period. The hand will also remove the cold page from the clock if it is non-resident (the most probable case). It actually does the work on the cold page on behalf of hand **HAND**$_{test}$. Finally the hand stops at a hot page.

We keep track of the number of non-resident cold pages. Once the number exceeds m, the memory size in the number of pages, we terminate the test period of the cold page pointed to by **HAND**$_{test}$. We also remove it from the clock if it is a non-resident page. Because the cold page has used up its test period without a re-access and has no chance to turn into a hot page with its next access. **HAND**$_{test}$ then moves forward and stops at the next cold page.

Now let us summarize how these hands coordinate their operations on the clock to resolve a page fault. When there is a page fault, the faulted page must be a cold page. We first run **HAND**$_{cold}$ for a free space. If the faulted cold page is not in the list, its reuse distance is highly likely to be larger than the recency of hot pages [4]. So the page is still categorized as a

[4] We cannot guarantee that it is a larger one because there are no opera-

cold page and is placed at the list head. The page also initiates its test period. If the number of cold pages is larger than the threshold ($m_c + m$), we run **HAND**$_{test}$. If the cold page is in the list [5], the faulted page turns into a hot page and is placed at the head of the list. We run **HAND**$_{hot}$ to turn a hot page with a large recency into a cold page.

4.4 Making CLOCK-Pro Adaptive

Until now, we have assumed that the memory allocations for the hot and cold pages are fixed. In LIRS, there is a predetermined parameter, denoted as L_{hirs}, to measure the percentage of memory that are used by cold pages. As it is shown in [13], L_{hirs} actually affects how LIRS behaves differently from LRU. When L_{hirs} approaches 100%, LIRS's replacement behavior, as well as its hit ratios, are close to those of LRU. Although the evaluation of LIRS algorithm indicates that its performance is not sensitive to L_{hirs} variations within a large range between 1% and 30%, it also shows that the hit ratios of LIRS could be moderately lower than LRU for LRU-friendly workloads (i.e. with strong locality) and increasing L_{hirs} can eliminate the performance gap.

In CLOCK-Pro, resident cold pages are actually managed in the same way as in CLOCK. **HAND**$_{cold}$ behaves the same as what the clock hand in CLOCK does: sweeping across the pages while sparing the page with a reference bit of 1 and replacing the page with a reference bit of 0. So increasing m_c, the size of the allocation for cold pages, makes CLOCK-Pro behave more like CLOCK.

Let us see the performance implication of changing memory allocation in CLOCK-Pro. To overcome the CLOCK performance disadvantages with weak access patterns such as scan and loop, a small m_c value means a quick eviction of cold pages just faulted in and the strong protection of hot pages from the interference of cold pages. However, for a strong locality access stream, almost all the accessed pages have relatively small reuse distance. But, some of the pages have to be categorized as cold pages. With a small m_c, a cold page would have to be replaced out of memory soon after its being loaded in. Due to its small reuse distance, the page is probably faulted in the memory again soon after its eviction and treated as a hot page because it is in its test period this time. This actually generates unnecessary misses for the pages with small reuse distances. Increasing m_c would allow these pages to be cached for a longer period of time and make it more possible for them to be re-accessed and to turn into hot pages without being replaced. Thus, they can save additional page faults.

For a given reuse distance of an accessed cold page, m_c decides the probability of a page being re-accessed before its being replaced from the memory. For a cold page with its reuse distance larger than its test period, retaining the page in memory with a large m_c is a waste of buffer spaces. On the other hand, for a page with a small reuse distance, retaining the page in memory for a longer period of time with a large m_c would save an additional page fault. In the adaptive CLOCK-Pro, we allow m_c to dynamically adjust to the current reuse distance distribution. If a cold page is accessed during its test period, we increment m_c by 1. If a cold page passes its test period without a re-access, we decrement m_c by 1. Note the aforementioned cold pages include resident and non-resident cold pages. Once the m_c value is changed, the clock hands of CLOCK-Pro will realize the memory allocation by temporally adjusting the moving speeds of **HAND**$_{hot}$ and **HAND**$_{cold}$.

With this adaptation, CLOCK-Pro could take both LRU advantages with strong locality and LIRS advantages with weak locality.

5 Performance Evaluation

We use both trace-driven simulations and prototype implementation to evaluate our CLOCK-Pro and to demonstrate its performance advantages. To allow us to extensively compare CLOCK-Pro with other algorithms aiming at improving LRU, including CLOCK, LIRS, CAR, and OPT, we built simulators running on the various types of representative workloads previously adopted for replacement algorithm studies. OPT is an optimal, but offline, unimplementable replacement algorithm [1]. We also implemented a CLOCK-Pro prototype in a Linux kernel to evaluate its performance as well as its overhead in a real system.

5.1 Trace-Driven Simulation Evaluation

Our simulation experiments are conducted in three steps with different kinds of workload traces. Because LIRS is originally proposed as an I/O buffer cache replacement algorithm, in the first step, we test the replacement algorithms on the I/O traces to see how well CLOCK-Pro can retain the LIRS performance merits, as well as its performance with typical I/O access patterns. In the second step, we test the algorithms on the VM traces of application program executions. Integrated VM management on file cache and program memory, as is implemented in Linux, is always desired. Because of the concern for mistreatment of file data and process pages as mentioned in Section 2.1, we test the algorithms on the aggregated VM and file I/O traces to see how these algorithms respond to the integration in the third step. We do not include the results of LRU in the presentation, because they are almost the same as those of CLOCK.

tions on hits in CLOCK-Pro and we limit the number of cold pages in the list. But our experiment results show this approximation minimally affects the performance of CLOCK-Pro.

[5] The cold page must be in its test period. Otherwise, it must have been removed from the list.

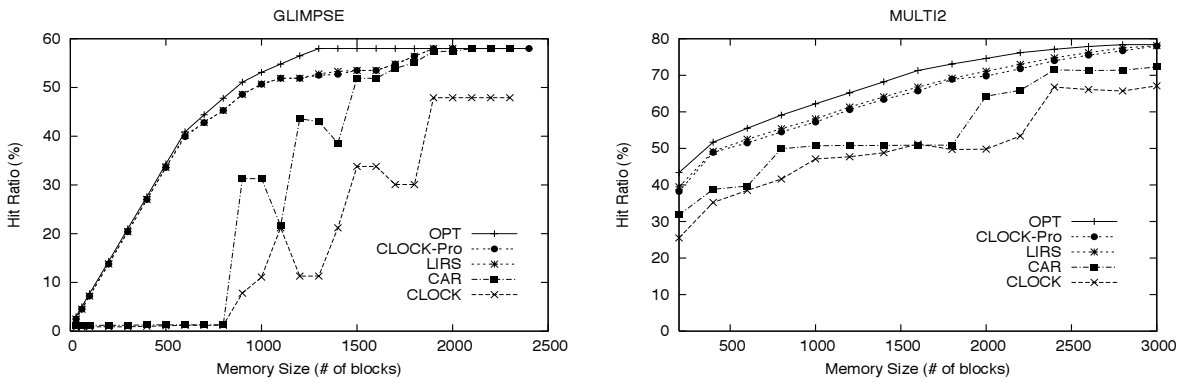


Figure 2: Hit ratios of the replacement algorithms OPT, CLOCK-Pro, LIRS, CAR, and CLOCK on workloads $glimpse$ and $multi2$.

5.1.1 Step 1: Simulation on I/O Buffer Caches

The file I/O traces used in this section are from [13] used for the LIRS evaluation. In their performance evaluation, the traces are categorized into four groups based on their access patterns, namely, loop, probabilistic, temporally-clustered and mixed patterns. Here we select one representative trace from each of the groups for the replacement evaluation, and briefly describe them here.

1. **glimpse** is a text information retrieval utility trace. The total size of text files used as input is roughly 50 MB. The trace is a member of the loop pattern group.

2. **cpp** is a GNU C compiler pre-processor trace. The total size of C source programs used as input is roughly 11 MB. The trace is a member of the probabilistic pattern group.

3. **sprite** is from the Sprite network file system, which contains requests to a file server from client workstations for a two-day period. The trace is a member of the temporally-clustered pattern group.

4. **multi2** is obtained by executing three workloads, cs, cpp, and postgres, together. The trace is a member of the mixed pattern group.

These are small-scale traces with clear access patterns. We use them to investigate the implications of various access patterns on the algorithms. The hit ratios of $glimpse$ and $multi2$ are shown in Figure 2. To help readers clearly see the hit ratio difference for *cpp* and $sprite$, we list their hit ratios in Tables 1 and 2, respectively. For LIRS, the memory allocation to HIR pages (L_{hirs}) is set as 1% of the memory size, the same value as it is used in [13]. There are several observations we can make on the results.

First, even though CLOCK-Pro does not responsively deal with hit accesses in order to meet the cost requirement of VM management, the hit ratios of CLOCK-Pro and LIRS are very close, which shows that CLOCK-Pro effectively retains the performance advantages of LIRS. For workloads $glimpse$ and $multi2$, which contain many loop accesses, LIRS with a small L_{hirs} is most effective. The hit ratios of CLOCK-pro are a little lower than LIRS. However, for the LRU-friendly workload, $sprite$, which consists of strong locality accesses, the performance of LIRS could be lower than CLOCK (see Table 2). With its memory allocation adaptation, CLOCK-Pro improves the LIRS performance.

Figure 3 shows the percentage of the memory allocated to cold pages during the execution courses of $multi2$ and $sprite$ for a memory size of 600 pages. We can see that for $sprite$, the allocations for cold pages are much larger than 1% of the memory used in LIRS, and the allocation fluctuates over the

blocks	OPT	CLOCK-Pro	LIRS	CAR	CLOCK
20	26.4	23.9	24.2	17.6	0.6
35	46.5	41.2	42.4	26.1	4.2
50	62.8	53.1	55.0	37.5	18.6
80	79.1	71.4	72.8	70.1	60.4
100	82.5	76.2	77.6	77.0	72.6
300	86.5	85.1	85.0	85.6	83.5
500	86.5	85.9	85.9	85.8	84.7
700	86.5	86.3	86.3	86.3	85.4
900	86.5	86.4	86.4	86.4	85.7

Table 1: Hit ratios (%) of the replacement algorithms OPT, CLOCK-Pro, LIRS, CAR, and CLOCK on workload *cpp*.

blocks	OPT	CLOCK-Pro	LIRS	CAR	CLOCK
100	50.8	24.8	25.1	26.1	22.8
200	68.9	45.2	44.7	43.0	43.5
400	84.6	70.1	69.5	70.5	70.9
600	89.9	82.4	80.9	82.1	83.3
800	92.2	87.6	85.6	87.3	88.1
1000	93.2	89.7	87.6	89.6	90.4

Table 2: Hit ratios (%) of the replacement algorithms OPT, CLOCK-Pro, LIRS, CAR, and CLOCK on workload $sprite$.

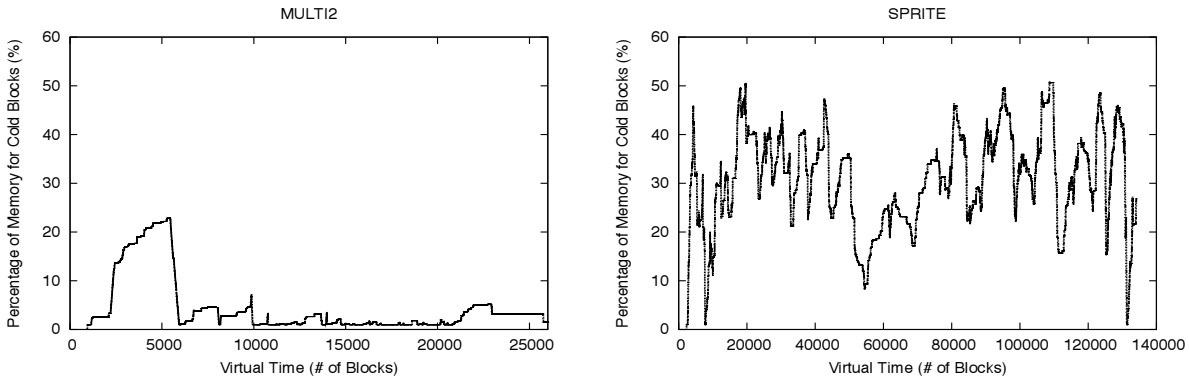


Figure 3: Adaptively changing the percentage of memory allocated to cold blocks in workloads $multi2$ and $sprite$.

time adaptively to the changing access patterns. It sounds paradoxical that we need to increase the cold page allocation when there are many hot page accesses in the strong locality workload. Actually only the real cold pages with large reuse distances should be managed in a small cold allocation for their quick replacements. The so-called "cold" pages could actually be hot pages in strong locality workloads because the number of so-called "hot" pages are limited by their allocation. So quickly replacing these pseudo-cold pages should be avoided by increasing the cold page allocation. We can see that the cold page allocations for $multi2$ are lower than $sprite$, which is consistent with the fact that $multi2$ access patterns consist of many long loop accesses of weak locality.

Second, regarding the performance difference of the algorithms, CLOCK-Pro and LIRS have much higher hit ratios than CAR and CLOCK for $glimpse$ and $multi2$, and are close to optimal. For strong locality accesses like $sprite$, there is little improvement either for CLOCK-Pro or CAR. This is why CLOCK is popular, considering its extremely simple implementation and low cost.

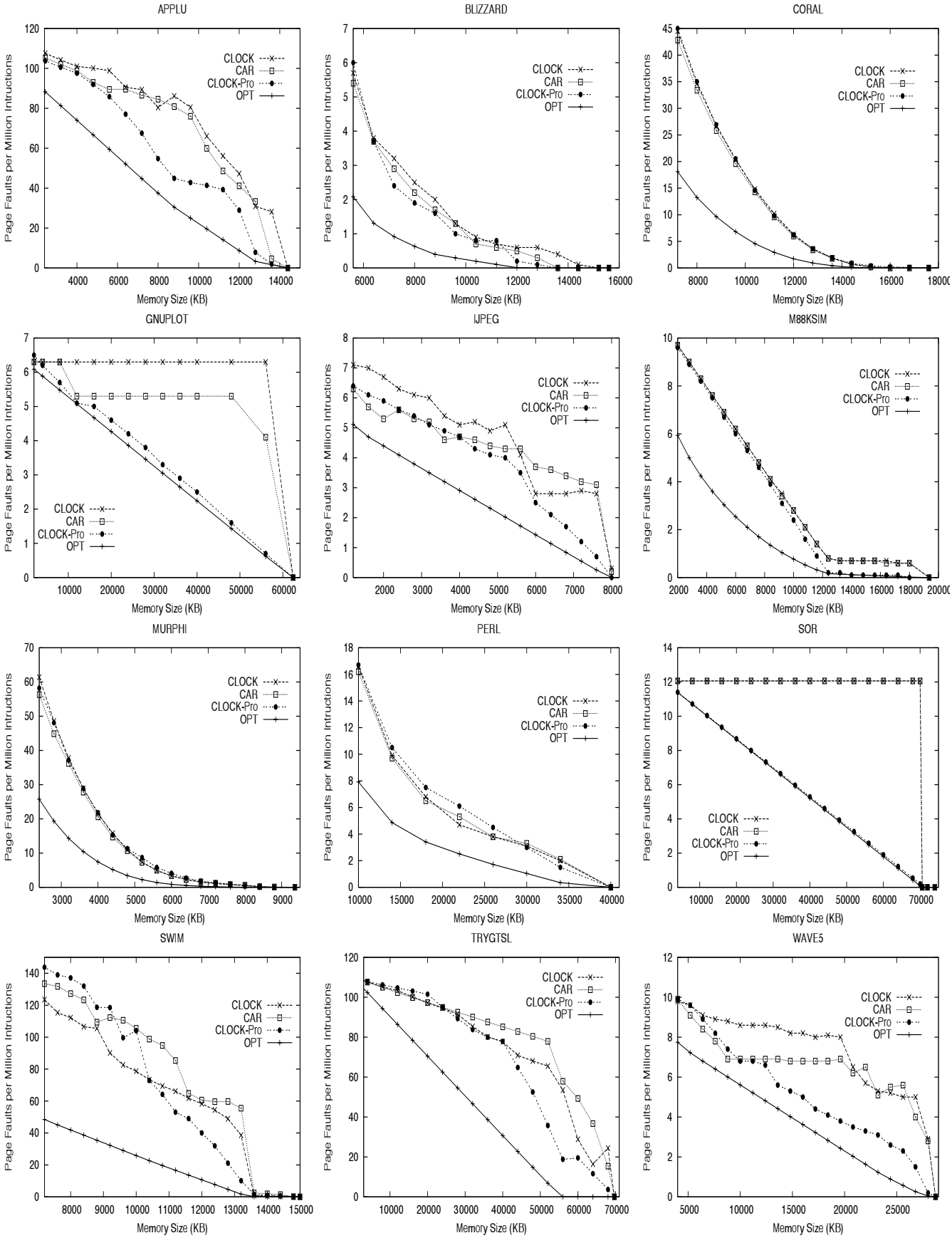
Third, even with a built-in memory allocation adaption mechanism, CAR cannot provide consistent improvements over CLOCK, especially for weak locality accesses, on which a fix is most needed for CLOCK. As we have analyzed, this is because CAR, as well as ARC, lacks a consistent locality strength comparison mechanism.

5.1.2 Step 2: Simulation on Memory for Program Executions

In this section, we use the traces of memory accesses of program executions to evaluate the performance of the algorithms. All the traces used here are also used in [10] and many of them are also used in [9, 24]. However, we do not include the performance results of SEQ and EELRU in this paper because of their generality or cost concerns for VM management. In some situations, EELRU needs to update its statistics for every single memory reference, having the same overhead problem as LRU [24]. Interested readers are referred to the respective papers for detailed performance descriptions of SEQ and EELRU. By comparing the hit ratio curves presented in those papers with the curves provided in this paper about CLOCK-Pro (these results are comparable), readers will reach the conclusion that CLOCK-Pro provides better or equally good performance compared to SEQ and EELRU. Also because of overhead concern, we do not include the LRU and LIRS performance. Actually LRU has its hit ratio curves almost the same as those of CLOCK in our experiments.

Table 3 summarizes all the program traces used in this paper. The detailed program descriptions, space-time memory access graphs, and trace collection methodology, are described in [10, 9]. These traces cover a large range of various access patterns. After observing their memory access graphs drawn from the collected traces, the authors of paper [9] categorized programs $coral$, $m88ksim$, and $murphi$ as having "no clearly visible patterns" with all accesses temporarily clustered together, categorized programs $blizzard$, $perl$, and $swim$ as having "patterns at a small scale", and categorized the rest of programs as having "clearly-exploitable, large-scale reference patterns". If we examine the program access behaviors in terms of reuse distance, the programs in the first category are the strong locality workloads. Those in the second category are moderate locality workloads. And the remaining programs in the third category are weak locality workloads. Figure 4 shows the number of page faults per million of instructions executed for each of the programs, denoted as page fault ratio, as the memory increases up to the its maximum memory demand. We exclude cold page faults which occur on their first time accesses. The algorithms considered here are CLOCK, CLOCK-Pro, CAR and OPT.

The simulation results clearly show that CLOCK-Pro significantly outperforms CLOCK for the programs with weak locality, including programs $applu$, $gunplot$, $ijpeg$, sor, $trygtsl$, and $wave5$. For $gunplot$ and sor, which have very large loop accesses, the page fault ratios of CLOCK-Pro are almost equal to those of OPT. The improvements of CAR over


APPLU
BLIZZARD
CORAL
GNUPLOT
IJPEG
M88KSIM
MURPHI
PERL
SOR
SWIM
TRYGTSL
WAVE5
CLOCK
CAR
CLOCK-Pro
OPT
Page Faults per Million Intructions
Memory Size (KB)

Program	Description	Size (in Millions of Instructions)	Maximum Memory Demand (KB)
applu	Solve 5 coupled parabolic/elliptic PDE	1,068	14,524
blizzard	Binary rewriting tool for software DSM	2,122	15,632
coral	Deductive database evaluating query	4,327	20,284
gnuplot	PostScript graph generation	4,940	62,516
ijpeg	Image conversion into IJPEG format	42,951	8,260
m88ksim	Microprocessor cycle-level simulator	10,020	19,352
murphi	Protocol verifi er	1,019	9,380
perl	Interpreted scripting language	18,980	39,344
sor	Successive over-relaxation on a matrix	5,838	70,930
swim	Shallow water simulation	438	15,016
trygtsl	Tridiagonal matrix calculation	377	69,688
wave5	plasma simulation	3,774	28,700

Table 3: A brief description of the programs used in Section 5.1.2.

CLOCK are far from being consistent and significant. In many cases, it performs worse than CLOCK. The poorest performance of CAR appears on traces $gunplot$ and sor – it cannot correct the LRU problems with loop accesses and its page fault ratios are almost as high as those of CLOCK.

For programs with strong locality accesses, including $coral$, $m88ksim$ and $murphi$, there is little room for other replacement algorithms to do a better job than CLOCK/LRU. Both CLOCK-Pro and ARC retain the LRU performance advantages for this type of programs, and CLOCK-Pro even does a little bit better than CLOCK.

For the programs with moderate locality accesses, including $blizzard$, $perl$ and $swim$, the results are mixed. Though we see the improvements of CLOCK-Pro and CAR over CLOCK in the most cases, there does exist a case in $swim$ with small memory sizes where CLOCK performs better than CLOCK-Pro and CAR. Though in most cases CLOCK-Pro performs better than CAR, for $perl$ and $swim$ with small memory sizes, CAR performs moderately better. After examining the traces, we found that the CLOCK-Pro performance variations are due to the working set shifting in the workloads. If a workload frequently shifts its working set, CLOCK-Pro has to actively adjust the composition of the hot page set to reflect current access patterns. When the memory size is small, the set of cold resident pages is small, which causes a cold/hot status exchange to be more possibly associated with an additional page fault. However, the existence of locality itself confines the extent of working set changes. Otherwise, no caching policy would fulfill its work. So we observed moderate performance degradations for CLOCK-Pro only with small memory sizes.

To summarize, we found that CLOCK-Pro can effectively remove the performance disadvantages of CLOCK in case of weak locality accesses, and CLOCK-Pro retains its performance advantages in case of strong locality accesses. It exhibits apparently more impressive performance than CAR, which was proposed with the same objectives as CLOCK-Pro.

5.1.3 Step 3: Simulation on Program Executions with Interference of File I/O

In an unified memory management system, file buffer cache and process memory are managed with a common replacement policy. As we have stated in Section 2.1, memory competition from a large number of file data accesses in a shared memory space could interfere with program execution. Because file data is far less frequently accessed than process VM, a process should be more competitive in preventing its memory from being taken away to be used as file cache buffer. However, recency-based replacement algorithms like CLOCK allow these file pages to replace process memory even if they are not frequently used, and to pollute the memory. To provide a preliminary study on the effect, we select an I/O trace (WebSearch1) from a popular search engine [26] and use its first 900 second accesses as a sample I/O accesses to co-occur with the process memory accesses in a shared memory space. This segment of I/O trace contains extremely weak locality – among the total 1.12 millions page accesses, there are 1.00 million unique pages accessed. We first scale the I/O trace onto the execution time of a program and then aggregate the I/O trace with the program VM trace in the order of their access times. We select a program with strong locality accesses, $m88ksim$, and a program with weak locality accesses, sor, for the study.

Tables 4 and 5 show the number of page faults per million of instructions (only the instructions for $m88ksim$ or sor are counted) for $m88ksim$ and sor, respectively, with various memory sizes. We are not interested in the performance of the I/O accesses. There would be few page hits even for a very large dedicated memory because there is little locality in their accesses.

From the simulation results shown in the tables, we observed that: (1) For the strong locality program, $m88ksim$, both CLOCK-Pro and CAR can effectively protect program execution from I/O access interference, while CLOCK is not able to reduce its page faults with increasingly large memory sizes. (2) For the weak locality program, sor, only CLOCK-

Memory(KB)	CLOCK-Pro	CLOCK-Pro w/IO	CAR	CAR w/IO	CLOCK	CLOCK w/IO
2000	9.6	9.94	9.7	10.1	9.7	11.23
3600	8.2	8.83	8.3	9.0	8.3	11.12
5200	6.7	7.63	6.9	7.8	6.9	11.02
6800	5.3	6.47	5.5	6.8	5.5	10.91
8400	3.9	5.22	4.1	5.8	4.1	10.81
10000	2.4	3.92	2.8	4.9	2.8	10.71
11600	0.9	2.37	1.4	4.2	1.4	10.61
13200	0.2	0.75	0.7	3.9	0.7	10.51
14800	0.1	0.52	0.7	3.6	0.7	10.41
16400	0.1	0.32	0.6	3.3	0.7	10.31
18000	0.0	0.22	0.6	3.1	0.6	10.22
19360	0.0	0.19	0.0	2.9	0.0	10.14

Table 4: The performance (number of page faults in one million of instructions) of algorithms CLOCK-Pro, CAR and CLOCK on program $m88ksim$ with and without the interference of I/O file data accesses.

Memory(KB)	CLOCK-Pro	CLOCK-Pro w/IO	CAR	CAR w/IO	CLOCK	CLOCK w/IO
4000	11.4	11.9	12.1	12.2	12.1	12.2
12000	10.0	10.7	12.1	12.2	12.1	12.2
20000	8.7	9.6	12.1	12.2	12.1	12.2
28000	7.3	8.6	12.1	12.2	12.1	12.2
36000	5.9	7.5	12.1	12.2	12.1	12.2
44000	4.6	6.5	12.1	12.2	12.1	12.2
52000	3.2	5.4	12.1	12.2	12.1	12.2
60000	1.9	4.4	12.1	12.2	12.1	12.2
68000	0.5	3.4	12.1	12.2	12.1	12.2
70600	0.0	3.0	0.0	12.2	0.0	12.2
74000	0.0	2.6	0.0	12.2	0.0	12.2

Table 5: The performance (number of page faults in one million of instructions) of algorithms CLOCK-Pro, CAR and CLOCK on program sor with and without the interference of I/O file data accesses.

Pro can protect program execution from interference, though its page faults are moderately increased compared with its dedicated execution on the same size of memory. However, CAR and CLOCK cannot reduce their faults even when the memory size exceeds the program memory demand, and the number of faults on the dedicated executions has been zero.

We did not see a devastating influence on the program executions with the co-existence of the intensive file data accesses. This is because even the weak accesses of $m88ksim$ are strong enough to stave off memory competition from file accesses with their page re-accesses, and actually there are almost no page reuses in the file accesses. However, if there are quiet periods during program active executions, such as waiting for user interactions, the program working set would be flushed by file accesses under recency-based replacement algorithms. Reuse distance based algorithms such as CLOCK-Pro will not have the problem, because file accesses have to generate small reuse distances to qualify the file data for a long-term memory stay, and to replace the program memory.

5.2 CLOCK-Pro Implementation and its Evaluation

The ultimate goal of a replacement algorithm is to reduce application execution times in a real system. In the process of translating the merits of an algorithm design to its practical performance advantages, many system elements could affect execution times, such as disk access scheduling, the gap between CPU and disk speeds, and the overhead of paging system itself. To evaluate the performance of CLOCK-Pro in a real system, we have implemented CLOCK-Pro in Linux kernel 2.4.21, which is a well documented recent version [11, 23].

5.2.1 Implementation and Evaluation Environment

We use a Gateway PC, which has its CPU of Intel P4 1.7GHz, its Western Digital WD400BB hard disk of 7200 RPM, and its memory of 256M. It is installed with RedHat 9. We are able to adjust the memory size available to the system and user programs by preventing certain portion of memory from being allocated.

In Kernel 2.4, process memory and file buffer are under an unified management. Memory pages are placed either in an

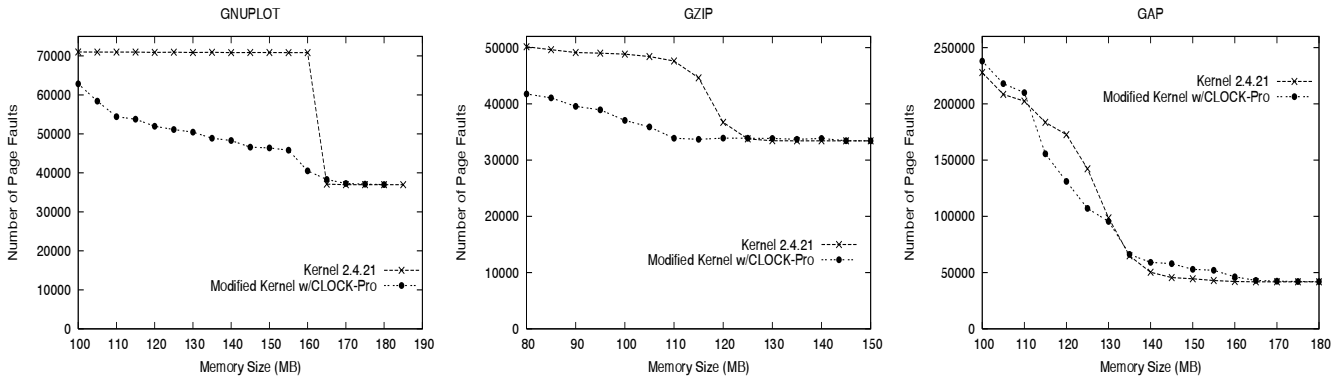


Figure 5: Measurements of the page faults of programs *gnuplot*, *gzip*, and *gap* on the original system and the system adopting CLOCK-Pro.

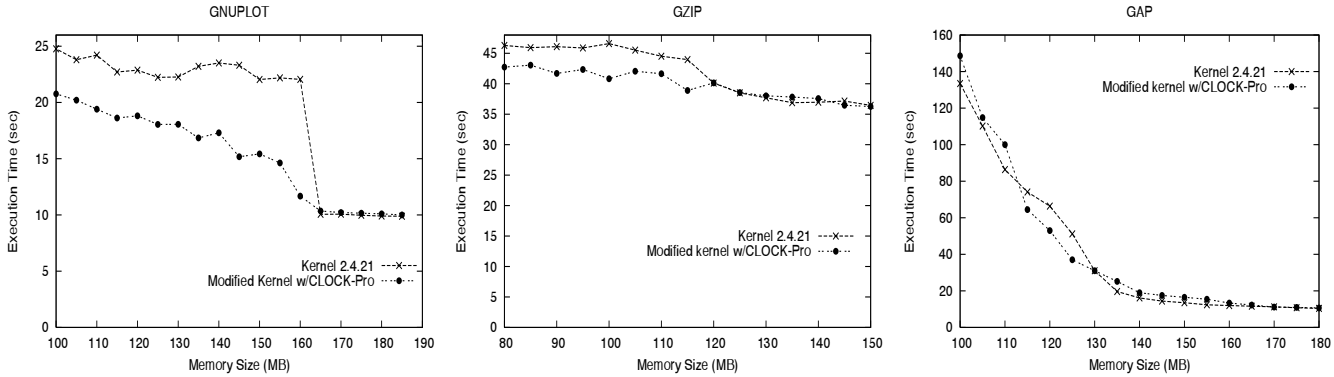


Figure 6: Measurements of the execution times of programs *gnuplot*, *gzip*, and *gap* on the original system and the system adopting CLOCK-Pro.

active list or in an inactive list. Each page is associated with a reference bit. When a page in the inactive list is detected being referenced, the page is promoted into the active list. Periodically the pages in the active list that are not recently accessed are removed to refill the inactive list. The kernel attempts to keep the ratio of the sizes of the active list and inactive list as 2:1. Each list is organized as a separate clock, where its pages are scanned for replacement or for movement between the lists. We notice that this kernel has adopted an idea similar to the 2Q replacement [12], by separating the pages into two lists to protect hot pages from being flushed by cold pages. However, a critical question still remains unanswered: how are the hot pages correctly identified from the cold pages?

This issue has been addressed in CLOCK-Pro, where we place all the pages in one single clock list, so that we can compare their hotness in a consistent way. To facilitate an efficient clock hand movement, each group of pages (with their statuses of hot, cold, and/or on test) are linked separately according to their orders in the clock list. The ratio of cold pages and hot pages is adaptively adjusted. CLOCK-Pro needs to keep track of a certain number of pages that have already been replaced from memory. We use their positions in the respective backup files to identify those pages, and maintain a hash table to efficiently retrieve their metadata when they are faulted in.

We ran SPEC CPU2000 programs and some commonly used tools to test the performance of CLOCK-Pro as well as the original system. We observed consistent performance trends while running programs with weak, moderate, or strong locality on the original and modified systems. Here we present the representative results for three programs, each from one of the locality groups. Apart from *gnuplot*, a widely used interactive plotting program with its input data file of 16 MB, which we have used in our simulation experiments, the other two are from SPEC CPU2000 benchmark suite [27], namely, *gzip* and *gap*. *gzip* is a popular data compression program, showing a moderate locality. *gap* is a program implementing a language and library designed mostly for computing in groups, showing a strong locality. Both take the inputs from their respective training data sets.

5.2.2 Experimental Measurements

Figures 5 and 6 show the number of page faults and the execution times of programs *gnuplot*, *gzip*, and *gap* on the original system and the modified system adopting CLOCK-Pro. In the simulation-based evaluation, only page faults can be obtained. Here we also show the program execution times, which include page fault penalties and system paging overhead. It is noted that we include cold page faults in the statistics, because they contribute to the execution times. We see that the variations of the execution times with memory size generally

keep the same trends as those of page fault numbers, which shows that page fault is the major factor to affect system performance.

The measurements are consistent with the simulation results on the program traces shown in Section 5.1.2. For the weak locality program *gnuplot*, CLOCK-Pro significantly improves its performance by reducing both its page fault numbers and its execution times. The largest performance improvement comes at around 160MB, the available memory size approaching the memory demand, where the time for CLOCK-Pro (11.7 sec) is reduced by 47% when compared with the time for the original system (22.1 sec). There are some fluctuations in the execution time curves. This is caused by the block layout on the disk. A page faulted in from a disk position sequential to the previous access position has a much smaller access time than that retrieved from a random position. So the penalty varies from one page fault to another. For programs *gzip* and *gap* with a moderate or strong locality, CLOCK-Pro provides a performance as good as the original system.

Currently this is only a prototype implementation of CLOCK-Pro, in which we have attempted to minimize the changes in the existing data structures and functions, and make the most of the existing infrastructure. Sometimes this means a compromise in the CLOCK-Pro performance. For example, the hardware MMU automatically sets the reference bits on the *pte* (Page Table Entry) entries of a process page table to indicate the references to the corresponding pages. In kernel 2.4, the paging system works on the active or inactive lists, whose entries are called *page descriptors*. Each *descriptor* is associated with one physical page and one or more (if the page is shared) *pte* entries in the process page tables. Each *descriptor* contains a *reference flag*, whose value is transfered from its associated *pte* when the corresponding process table is scanned. So there is an additional delay for the reference bits (flags) to be seen by the paging system. In kernel 2.4, there is no infrastructure supporting the retrieval of *pte* through the *descriptor*. So we have to accept this delay in the implementation. However, this tolerance is especially detrimental to CLOCK-Pro because it relies on a fine-grained access timing distinction to realize its advantages. We believe that further refinement and tuning of the implementation will exploit more performance potential of CLOCK-Pro.

5.2.3 The Overhead of CLOCK-Pro

Because we almost keep the paging infrastructure of the original system intact except replacing the active/inactive lists with an unified clock list and introducing a hash table, the additional overhead from CLOCK-Pro is limited to the clock list and hash table operations.

We measure the average number of entries the clock hands sweep over per page fault on the lists for the two systems. Table 6 shows a sample of the measurements. The results show that CLOCK-Pro has a number of hand movements comparable to the original system except for large memory sizes, where the original system significantly lowers its movement number while CLOCK-Pro does not. In CLOCK-Pro, for every referenced cold page seen by the moving $\mathbf{HAND}_{cold}$, there is at least one $\mathbf{HAND}_{hot}$ movement to exchange the page statuses. For a specific program with a stable locality, there are fewer cold pages with a smaller memory, as well as less possibility for a cold page to be re-referenced before $\mathbf{HAND}_{cold}$ moves to it. So $\mathbf{HAND}_{cold}$ can take a small number of movements to reach a qualified replacement page, and the number of additional $\mathbf{HAND}_{hot}$ movements per page fault is also small. When the memory size is close to the program memory demand, the original system can take less hand movements during its search on its inactive list, due to the increasing chance of finding an unreferenced page. However, $\mathbf{HAND}_{cold}$ would encounter more referenced cold pages, which causes additional $\mathbf{HAND}_{hot}$ movements. We believe that this is not a performance concern, because one page fault penalty is equivalent to the time of tens of thousands of hand movements. We also measured the bucket size of the hash table, which is only 4-5 on average. So we conclude that the additional overhead is negligible compared with the original replacement implementation.

Memory (MB)	110	140	170
Kernel 2.4.21	12.4	14.2	6.9
CLOCK-Pro	16.2	20.6	18.5

Table 6: Average number of entries the clock hands sweep over per page fault in the original kernel and CLOCK-Pro with different memory sizes for program *gnuplot*.

6 Conclusions

In this paper, we propose a new VM replacement policy, CLOCK-Pro, which is intended to take the place of CLOCK currently dominating various operating systems. We believe it is a promising replacement policy in the modern OS designs and implementations for the following reasons. (1) It has a low cost that can be easily accepted by current systems. Though it could move up to three pointers (hands) during one victim page search, the total number of the hand movements is comparable to that of CLOCK. Keeping track of the replaced pages in CLOCK-Pro doubles the size of the linked list used in CLOCK. However, considering the marginal memory consumption of the list in CLOCK, the additional cost is negligible. (2) CLOCK-pro provides a systematic solution to address the CLOCK problems. It is not just a quick and experience-based fix to CLOCK in a specific situation, but is designed based on a more accurate locality definition – reuse distance and addresses the source of the LRU problem. (3) It is fully adaptive to strong or weak access patterns without any predetermined parameters. (4) Extensive simulation experiments

and a prototype implementation show its significant and consistent performance improvements.

Acknowledgments

We are grateful to our shepherd Yuanyuan Zhou and the anonymous reviewers who helped further improve the quality of this paper. We thank our colleague Bill Bynum for reading the paper and his comments. The research is partially supported by the National Science Foundation under grants CNS-0098055, CCF-0129883, and CNS-0405909.

References

[1] L. A. Belady "A Study of Replacement Algorithms for Virtual Storage", *IBM System Journal*, 1966.

[2] S. Bansal and D. Modha, "CAR: Clock with Adaptive Replacement", *Proceedings of the 3nd USENIX Symposium on File and Storage Technologies*, March, 2004.

[3] P. Cao, E. W. Felten and K. Li, "Application-Controlled File Caching Policies", *Proceedings of the USENIX Summer 1994 Technical Conference*, 1994.

[4] J. Choi, S. Noh, S. Min, Y. Cho, "An Implementation Study of a Detection-Based Adaptive Block Replacement Scheme", *Proceedings of the 1999 USENIX Annual Technical Conference*, 1999, pp. 239-252.

[5] F. J. Corbato, "A Paging Experiment with the Multics System", *MIT Project MAC Report MAC-M-384*, May, 1968.

[6] C. Ding and Y. Zhong, "Predicting Whole-Program Locality through Reuse-Distance Analysis", *Proceedings of ACM SIGPLAN Conference on Programming Language Design and Implementation*, June, 2003.

[7] M. B. Friedman, "Windows NT Page Replacement Policies", *Proceedings of 25th International Computer Measurement Group Conference*, Dec, 1999, pp. 234-244.

[8] W. Effelsberg and T. Haerder, "Principles of Database Buffer Management", *ACM Transaction on Database Systems*, Dec, 1984, pp. 560-595.

[9] G. Glass and P. Cao, "Adaptive Page Replacement Based on Memory Reference Behavior", *Proceedings of 1997 ACM SIGMETRICS Conference*, May 1997, pp. 115-126.

[10] G. Glass, "Adaptive Page Replacement". Master's Thesis, University of Wisconsin, 1997.

[11] M. Gorman, "Understanding the Linux Virtual Memory Manager", *Prentice Hall*, April, 2004.

[12] T. Johnson and D. Shasha, "2Q: A Low Overhead High Performance Buffer Management Replacement Algorithm", *Proceedings of the 20th International Conference on VLDB*, 1994, pp. 439-450.

[13] S. Jiang and X. Zhang, "LIRS: An Efficient Low Interreference Recency Set Replacement Policy to Improve Buffer Cache Performance", *In Proceeding of 2002 ACM SIGMETRICS*, June 2002, pp. 31-42.

[14] J. Kim, J. Choi, J. Kim, S. Noh, S. Min, Y. Cho, and C. Kim "A Low-Overhead, High-Performance Unified Buffer Management Scheme that Exploits Sequential and Looping References", *4th Symposium on Operating System Design & Implementation*, October 2000.

[15] D. Lee, J. Choi, J. Kim, S. Noh, S. Min, Y. Cho and C. Kim, "On the Existence of a Spectrum of Policies that Subsumes the Least Recently Used (LRU) and Least Frequently Used (LFU) Policies", *Proceeding of 1999 ACM SIGMETRICS Conference*, May 1999.

[16] N. Megiddo and D. Modha, "ARC: a Self-tuning, Low Overhead Replacement Cache", *Proceedings of the 2nd USENIX Symposium on File and Storage Technologies*, March, 2003.

[17] V. F. Nicola, A. Dan, and D. M. Dias, "Analysis of the Generalized Clock Buffer Replacement Scheme for Database Transaction Processing", *Proceeding of 1992 ACM SIGMETRICS Conference*, June 1992, pp. 35-46.

[18] E. J. O'Neil, P. E. O'Neil, and G. Weikum, "The LRU-K Page Replacement Algorithm for Database Disk Buffering", *Proceedings of the 1993 ACM SIGMOD Conference*, 1993, pp. 297-306.

[19] V. Phalke and B. Gopinath, "An Inter-Reference gap Model for Temporal Locality in Program Behavior", *Proceeding of 1995 ACM SIGMETRICS Conference*, May 1995.

[20] R. H. Patterson, G. A. Gibson, E. Ginting, D. Stodolsky and J. Zelenka, "Informed Prefetching and Caching", *Proceedings of the 15th Symposium on Operating System Principles*, 1995, pp. 1-16.

[21] J. T. Robinson and N. V. Devarakonda, "Data Cache Management Using Frequency-Based Replacement", *Proceeding of 1990 ACM SIGMETRICS Conference*, 1990.

[22] R. van Riel, "Towards an O(1) VM: Making Linux Virtual Memory Management Scale Towards Large Amounts of Physical Memory", *Proceedings of the Linux Symposium*, July 2003.

[23] R. van Riel, "Page Replacement in Linux 2.4 Memory Management", *Proceedings of the FREENIX Track: 2001 USENIX Annual Technical Conference*, June, 2001.

[24] Y. Smaragdakis, S. Kaplan, and P. Wilson, "The EELRU adaptive replacement algorithm", *Performance Evaluation (Elsevier)*, Vol. 53, No. 2, July 2003.

[25] A. J. Smith, "Sequentiality and Prefetching in Database Systems", *ACM Trans. on Database Systems*, Vol. 3, No. 3, 1978, pp. 223-247.

[26] Storage Performance Council, http://www.storageperformance.org

[27] Standard Performance Evaluation Corporation, SPEC CPU2000 V1.2, http://www.spec.org/cpu2000/

[28] A. S. Tanenbaum and A. S. Woodhull, *Operating Systems, Design and Implementation*, Prentice Hall, 1997.

[29] Y. Zhou, Z. Chen and K. Li. "Second-Level Buffer Cache Management", *IEEE Transactions on Parallel and Distributed Systems*, Vol. 15, No. 7, July, 2004.