Heikki Paajanen

Page replacement in operating system memory management

Master's Thesis
in Information Technology
October 23, 2007

University of Jyväskylä

Department of Mathematical Information Technology

Jyväskylä

Author: Heikki Paajanen
Contact information: heikki.paajanen@iki.fi
Title: Page replacement in operating system memory management
Työn nimi: Sivunkorvaus käyttöjärjestelmän muistinhallinnassa
Project: Master's Thesis in Information Technology
Page count: 109
Abstract: This masters thesis introduces algorithms for page replacement in memory management of general purpose multitasking operating systems using paged virtual memory.
Suomenkielinen tiivistelmä: Tämä pro gradu tutkielma esittelee yleiskäyttöisen käyttöjärjestelmän sivunkorvaus algoritmeja
Keywords: operating systems, memory management, page replacement, algorithms
Avainsanat: käyttöjärjestelmät, muistinhallinta, sivunkorvaus, algoritmit

Preface

I have long been interested in how operating systems work, but lack of free time has kept me from studying them. After I got my courses done and it was time to figure out a topic for my master's thesis, I saw an opportunity. Operating systems are quite large topic, so I needed to focus on some area. Memory management has especially fascinated me and so I chose that. As I started studying memory management, I realized that it is still too large as topic for master's thesis and I further limited the topic into page replacement algorithms. My interest in operating systems has only grown after this experience and I hope I can continue working close to them in the future.

I would like to thank Jarmo Ernvall for taking the effort to act as my instructor and for valuable comments, Erkki Häkkinen for valuable comments, and Matti Katila for code review. Most of all, I would like to thank Kirsi Kiljala for valuable comments and for motivating me to get my studies done.

Glossary

2Q	Two Queue, a replacement algorithm
ARC	Adaptive Replacement Cache, a replacement algorithm
CAR	CLOCK with Adaptive Replacement, a replacement algorithm
CART	CAR with Temporal filtering
Cold page	A page that is considered unimportant by replacement algorithm. See Hot page.
Copy-on-write	Technique to defer copying until first write
Correlated access	Two or more accesses to same page quickly
Demand paging	Paging in pages as they are used
Evict	Move a page from main memory to a secondary memory
Hot page	A page that is considered important be replacement algorithm. See Cold page.
IRM	Independent Reference Model
FIFO	First-In, First-Out
Loop	Sequential, repeated pass over a set of pages
Locality	Tendency of referencing some pages more frequently than others
Locality set	A set of frequently accessed pages
LRU	Least Recently Used, a replacement algorithm
LRUSM	LRU Stack Model
NFU	Not Frequently Used, a replacement algorithm
Non-stationary	Process where probabilities depend on time
NRU	Not Recently Used, a replacement algorithm
Major page fault	A page fault on a page that is not present in main memory

MIN	See OPT
Minor page fault	A page fault on a page that is already in main memory
MMU	Memory Management Unit
OPT	Beladys optimal replacement algorithm
Page	Fixed size memory block
Page fault	Interrupt, when process accesses a page in a invalid way. (e.g Page is mapped, but not present main memory, or write on a read-only page)
Page frame	A place in main memory, where a page can be inserted
Page table	Virtual address to physical address mapping
Phase behaviour	Program behaviour model focusing on handling phases in program execution.
Phase-transition model	Program behaviour model, where execution consists of phases and transitions between them.
Physical address	Address in physical memory
Prepaging	Paging in pages before they are used
Reuse distance	Number of distinct page accessed between current access and previous access of a page
Scan	Sequential one time pass over pages
Stack distance	Position of the referred page in (LRU) stack just before next reference
Swap space	secondary memory used in virtual memory systems
Trashing	Extensive page faulting caused by process
Virtual address	Address in virtual address space
Virtual address space	Linear address space that is private to a process. Needs to be translated into a physical address.
Virtual time	Time that moves forward only when the program is executing, i.e. execution time.
Working set	(Informally) Smallest collection of pages, that are needed in main memory for the program to execute efficiently.

Contents

List of Figures

1 Introduction

Main memory is an important and a very limited resource in a computer. Although common amount of main memory has increased in the past decades, so has its demand [1]. As processor and main memory get faster more rapidly than secondary memories, the impact of swapping increases. The everyday improving hardware has also underlined some bottlenecks in real life applications, and created a need for further research.

In this thesis we will first introduce general concepts of virtual memory and paging, followed by proposed formal models for page replacement algorithms and program behaviour. The theory part ends with presenting well known basic page replacement algorithms, as well as some more advanced ones, and describing their characteristics.

In the empirical part of this thesis, we will present implementation of nine algorithms and compare their performance on generated virtual memory traces. Performance comparison is done by using page fault rate charts and other page fault metrics. The page fault rate chart shows the distribution of page faults.

In the real world examples Unix class of operating systems, mainly Linux, will be used as a reference. The theory is not, however, tied to Unix, and many other operating systems implement memory management by using the same techniques.

It should also be noted, that page replacement problem is not specific for operating system memory management, but is present also in various systems using caches, such as databases and web proxies. This thesis, however, limits its scope to the context of virtual memory management in general purpose operating systems. Some algorithms are designed for databases and later adapted for virtual memory. These are included because they contain important ideas and provide a better view of the evolution of replacement algorithms.

[1] Parkinson's Law states that "work expands so as to fill the time available for its completion" [15]. This can be adapted to computer memory usage as "programs and their data expand to fill the memory available to hold them".

2 Overview of Memory Management

Almost all modern general purpose operating systems use virtual memory to solve the **overlay** problem. In virtual memory the combined size of program code, data and stack may exceed the amount of main memory available in the system. This is made possible by using secondary memory, in addition to main memory. The operating system tries to keep the part of the memory in active use in main memory and the rest in secondary memory. When memory located in secondary memory is needed, it can be retrieved back to main memory [16]. The process of storing data from main memory to secondary memory is called **swapping out**, and retrieving data back to main memory is called **swapping in**. These will be referred as **swapping** except when distinction between the two is necessary. The part of the secondary memory, that is reserved for virtual memory, is called **swap space**, and is often implemented as a **swap partition** or a **swap file**.

There are two granularities in which swapping is commonly done in multitasking operating systems. The simplest one is to swap out a whole program when memory is needed. This simple method can be used as a load balancing technique [11]. The other granularity is to swap out small fixed size memory areas, pages. Paging will be introduced in more detail in chapter 2.1.

In Unix type of operating systems, virtual memory is usually divided in to two segments, the kernel segment and the process segment. The kernel segment contains all memory visible directly only to kernel itself. The kernel segment is always kept in main memory for various reasons, including performance and security. In Linux, on x86 architecture, the kernel segment starts at physical address 0x01000000. The first 16 MB is reserved for DMA as some devices are not able to address any higher.

Virtual memory provides processes a **virtual address space**. Programs use **virtual addresses** to refer to their own **virtual address space**. When virtual address space is used, each program sees a flat continuous memory dedicated for it alone. All memory, however, is not available for a running program. The kernel usually maps its own address to constant area of each program's address space. In Linux the kernels space is normally mapped at the end of the processes address space. As an example, on x86 architecture the last 1 GB of the 4 GB address space is reserved for

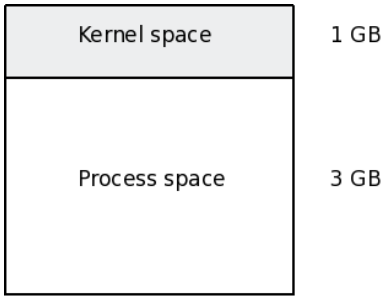

Figure 2.1: Kernel space mapping to processes address space in Linux on x86 architecture.

the kernel (See figure 2.1). This leaves 3 GB for the user process [8]. Virtual address space simplifies compilers and applications as the memory used by the operating system, and other running programs, are not directly visible to a running program.

2.1 Paging

The operating system divides virtual address space into units called **pages**. Main memory is also divided to fixed size units called **page frames** [16]. Each used page can be either in secondary memory or in a page frame in main memory. Naturally neither of these memories is needed for the pages, in the virtual address spaces of processes, that are not used.

A **paging algorithm** is needed to manage paging. A paging algorithm consists of three algorithms: **placement algorithm**, **fetch algorithm** and **replacement algorithm**. The placement algorithm is used to decide on which free page frame a page is placed. The fetch algorithm decides on which page or pages are to be put in main memory. Finally, the page replacement algorithm decides on which page is swapped out. Further, paging algorithms can be **demand paging** or **prepaging**. A demand paging algorithm places a page to main memory only when it is needed, while a prepaging algorithm attempts to guess which pages are needed next by placing them to main memory before they are needed. In general cases, it is very difficult to make accurate guesses of page usage and demand paging is generally accepted as a better choice. It can also be proved, that for certain constraints, optimal paging algorithm is a demand paging algorithm. Exact constraints and proof is given in [1].

A virtual address must be translated to corresponding physical address before the memory can be used. As this address translation is done with every memory ref-

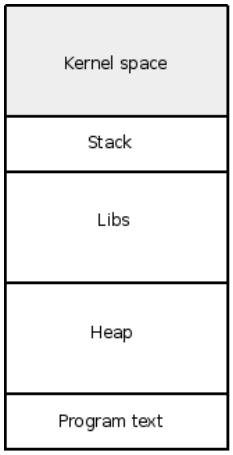

Figure 2.2: Process virtual memory layout

erence, it is important that it is very fast. Usually special hardware, called **Memory Management Unit (MMU)**, is used to make this translation. MMU uses virtual-to-physical address mapping information, located in operating systems **page table**, to make the translation. Each process has its own virtual address space and therefore page tables are per process. If the given virtual address is not mapped to main memory, the MMU traps the operating system. This trap, called **page fault**, gives the operating system an opportunity to bring the desired page from secondary memory to main memory, and update to page table accordingly [16]. Chapter 2.2 will explain page fault handling in more detail.

Because each process has its own virtual address space (See Figure 2.2), the operating system must keep track of all pages used by processes, location of each page, and some per page information. When a page in main memory is referenced or written to, it is marked accordingly. If all page frames are in use when a page fault occurs, the operating system moves, or evicts, some page to secondary memory to make room for the referenced page. As pointed earlier, the operating system must also update the page table, as the MMU knows nothing about the pages that are not in the main memory [16].

A page can be anonymous or file backed. Anonymous pages are used by program code as a work area, namely heap, stack and global variables. File backed pages are, as the name indicates, data read from a file in secondary memory. File backed pages are not meant to be modified during execution (e.g. program text). When operating system decides to evict a file backed page, it is not necessary to

spend time writing it to secondary memory. The page already has a valid copy of the data available, if it is needed again later. The page frame used can just be marked as free, and used for another page. In case the operating system decides to evict an anonymous page, it will have to check a few things. If the page has never been in secondary memory, or it has been modified while in main memory, it will have to be written to secondary memory before the page frame can be reused [16]. Such modified page is called **dirty**.

2.2 Page fault handling

As described earlier, a page fault typically occurs when a process references to a page that is not marked 'present' in main memory. Page faults can be classified in to **major page faults** and **minor page faults**. In a minor page fault, the referenced page is in main memory, but not yet marked 'present', which means that there is no disk IO required to handle to fault. Page faults are also caused by 1) reference to address 0, 2) reference to an unmapped page, 3) user process referencing the kernel area (remember that kernel areas are mapped to process address space!), 4) attempts to write to an **read-only** page and 5) attempts to execute code from **non-execute** page. The page fault in case 1 is also called an **invalid page fault**. Page faults in cases 2–5 are called **protection faults** and are, together with the actual mappings, used to implement memory access rights. While access rights are per page properties, they are usually maintained in larger memory areas, segments. Read-only pages are useful in implementing **copy-on-write**. Automatically growing stack is also implemented by using page faults in Linux. When a process tries to access an address right next to the current stack limit, the Linux kernel allocates a new page and maps it for stack usage.

3 Page replacement algorithm theory

Page replacement algorithms have been studied and some formal models are proposed to be used as basis of theoretical analysis. The following conventions are used.

Set of pages of a n-page program is defined as

$$N = \{p_1, ..., p_n\}$$

and

$$M = \{pf_1, ..., pf_m\}$$

is a set of page frames of main memory with space for m pages. Function $f : N \rightarrow M$ gives current page map and can be defined as

$$f(p_i) = \begin{cases} pf_j, & \text{if page } p_i \text{ is in page frame } pf_j, \\ \text{undefined}, & \text{otherwise.} \end{cases}$$

If $f(p_i)$ is undefined, a page fault must occur, if the page p_i is referenced. S is used to present the current state of main memory. For every page $x \in S$, $f(x)$ is defined and for all other pages $y \notin S$, $f(y)$ is undefined. Finally, $M_m = \{S | S \subseteq N \text{ and } |S| \leq m\}$ is defined as a set of possible memory states in main memory with m-page frames.

Programs memory usage can be described as a **reference string** $\omega = r_1 r_2 ... r_T$, where $r_t = x$ means that program referenced page x at time t.

3.1 A formal model for paging algorithm

Aho, Denning and Ullman presented a formal model for page replacement algorithms in [1]. This formulation allows mathematical analysis of properties of an algorithm.

A page replacement algorithm A can be formally defined as

$$A = (Q, q_0, g),$$

where Q is a set of algorithm specific **control states**, $q_0 \in Q$ is the initial control state and $g : M_m \times Q \times N \rightarrow M_m \times Q$ is the **allocation map**.

From this formulation we can see, that page a replacement algorithm can be thought as a state machine with $n \in N$ as input, $M_m \times Q$ as state and g as **transition function**. It is required, that $x \in S'$ when $g(S, q, x) = (S', q')$. A pair $(S, q) \in M_m \times Q$ is called a **configuration**.

A page replacement algorithm A produces a sequence of configurations

$$\{(S_t, q_t)\}_{t=0}^{T},$$

where

$$(S_t, q_t) = g(S_{t-1}, q_{t-1}, r_t),$$

$1 \leq t \leq T$. Examples of algorithms presented using this formal model are given when actual algorithms are described later.

The model uses one reference as the unit of time, which is not ideal as it models only the **virtual time** of the program (i.e. time when the program is actually executing) . Some algorithms, which are presented later, use the elapsed real-time in the decision making. To include a situation, where no process is running, we need to model time passing without any references. This can be done by using a special empty reference, $\emptyset$. Effectively, an empty reference means that the processor is in idle for that time.

3.2 The cost

The theoretical cost of a demand replacement algorithm can be measured as a sum of pages swapped out and swapped back in. For demand paging, the transition function g, as defined previously, can be presented as follows:

$$g(S, q, x) = \begin{cases} (S, q') & \text{if page } x \in S, \\ (S \cup \{x\}, q') & \text{if } x \notin S, |S| < m, \\ (S \cup \{x\} \setminus \{y\}, q') & \text{if} x \notin S, |S| = m, y \in S. \end{cases} \tag{3.2.1}$$

While there are free page frames, demand algorithms fill them before considering replacement of existing pages. Page replacement is always completed one page at a time.

The cost of processing reference string $\omega = r_1 r_2 ... r_T$, using a page replacement algorithm A, can be calculated as a sum of pages swapped out and swapped in. This means the pages x and y in the third case of equation (3.2.1). The page y can be ignored in analysis as the number of pages removed is normally some fixed fraction

of pages x in the long run [1]. The cost of a demand page replacement algorithm, for theoretical purposes, is the number of page faults it generates, when processing a given reference string.

3.3 Program behaviour

For useful formal analysis of a page replacement algorithm, a formal model for program behaviour is also required. Program behaviour models how reference strings are generated by a program.

Much of the work in the field of memory management has included the search for a good model for program behaviour. The nature of the work has shifted from complex mathematical models, like in [5], to more straightforward method of trying to handle real world problem workloads better, as in [7] (ARC, Chapter 4.11), [2] (CAR and CART, Chapters 4.12 and 4.13), [9] (Token-ordered LRU, Chapter 4.14) and [11] (CLOCK-Pro, Chapter 4.15).

3.3.1 Working set

The application memory usage, or demand, can also be modeled with **working sets** [4]. Informally, a working set consists of the smallest collection of pages that are needed in main memory for the program to execute efficiently. Denning defines the working set as a set of pages referenced during last θ units of time. Later Denning, together with Donald R. Slutz, developed the concept into a **generalized working set**, which consist of pages whose retention cost is not more than ζ. The retention cost of a page is any function, that increases monotonically between references and starts always from 0 after a reference [5].

The working set may vary greatly during the life time of a program. For example, some compilers first generate intermediate presentations of the program source code compiled, and then produce the actual target machine code from that. This is usually implemented as multiple phases, which all use different compiler code and different data, resulting in different working sets.

In [5] Denning makes an argument for local working sets policy, over global LRU or CLOCK policies (CLOCK and LRU will be introduced in Chapters 4.6 and 4.5). Time has, however, shown that the presented arguments were not valid or have become invalid for contemporary software and hardware, as variants of CLOCK

policy have long been dominant in general purpose operating systems [9].

Although formal policies based on working set have not survived, the informal working set concept has. It is still actively used to describe desired and undesired situations in page replacement, but its exact meaning varies slightly. This is partly due to the changed balance in cost caused by contemporary hardware and change in the nature of computer usage, to more interactive and versatile.

3.3.2 Locality

The tendency of referencing to some pages more frequently than others is called **locality** [6]. Locality is an important principle in virtual memory systems as it is found practically in all real world programs. Locality in programs is one of the reasons that LRU works reasonably well. The working set described earlier captures some properties of locality in a program and can be used to find its **locality set**. Another measure of locality is **reuse distance** of pages. Reuse distance is the number of distinct page accessed between current access and previous access of a page. Most LRU-friendly workloads have strong locality [9].

3.3.3 l-order non-stationary Markov process

A model, in which a program generates a reference string by *l*-**order non-stationary Markov process**, is presented in [1].

A program is defined as

$$P = (N, U, u_0, f, p),$$

where N is a set of pages, used by the program, U the set of states of the program and $f : N \times U \rightarrow U$ is the state transition function. $p(x, u, t)$ is the probability that a page x is referenced at time t, when the program is in state u. A program is said to be l-order, when $|U| = l + 1$, and non-stationary, when the probability p depends on time t.

For a page replacement algorithm to handle l-order program optimally, it needs to model all $l + 1$ states of the program in its own control states [1]. So generally, an optimal page replacement solution for an l-order program is not practical, even when handling page replacement for a single program.

3.3.4 Stochastic models

Independent Reference Model (IRM) and **LRU Stack Model (LRUSM)** are examples of **stochastic models** [5]. In IRM, the references are considered independent random variables with a common stationary distribution:

$$P_r[r(t) = i] = a_i,$$

where $r(t)$ is the reference at time t. So the probability of each distinct page is fixed. This leads to a geometric inter-reference distribution,

$$h_i(k) = (1 - a_k)^{k-1} a_i,$$

where $h_i(k)$ is the probability that two references to the page i are k references apart.

LRUSM is based on the behaviour of LRU algorithm (See Chapter 4.5). After each reference, LRU stack is defined as a list of referenced pages ordered by most recent reference. Stack distance of a reference, $d(t)$, is the position of the referred page in the LRU stack before the reference. LRUSM assumes that distances are independent random variables with a common stationary distribution:

$$Pr[d(t) = i] = b_i \text{ for all } t,$$

LRU algorithm is optimal, if

$$b_1 \geq b_2 \geq ... \geq b_i \geq ...$$

LRUSM is slightly better in practice than IRM. However, neither of these models match observations of real programs.

3.3.5 Phase behaviour

A program that has slow-drifting locality follows the **phase behaviour** model [5]. The basis of this model is the idea that the locality of the program changes slowly over the course of execution. As a consequence, the model considers relatively stable phases more important than rapid changes in locality in program execution. As Denning notes in [5], this assumption is wrong.

3.3.6 Phase-transition model

A good model for real world program behaviour is **Phase-transition model** [5]. In phase-transition model, program execution consists of distinct phases and short

transitions between them. During a phase the working set is relatively stable and other models, like LRUSM, can be used independently inside the phases. The phase-transition model can be simulated by generating locality-set/holding-time pairs (S, T), where the locality-set S is the references in phase and is of length T [5]. Transition is the small period before and after changing locality-set. Page faults are categorized as either phase faults or transition faults.

Measurements have shown that most (ca. 98%) of the programs virtual time is spent in phases. However, 40–50% of the page faults happen during short transition periods [5]. It is therefore very important for a page replacement algorithm to handle these transitions as well as possible. There is also a difference in the distribution of these page faults. Phase faults show strong serial correlation, but the distribution of the transition faults is closer to a geometric distribution [5].

3.4 Typical memory usage patterns

The changes in the working set make up a memory usage pattern. There are several identified memory usage patterns. Page replacement algorithms are usually analyzed by using a set of programs that contains a representative from all major memory usage patterns.

3.4.1 Correlated access

Many programs refer to pages two or more times quickly and then do not use that page for a long time. This is known as **correlated access** [2]. In many traditional replacement algorithms, as CLOCK (See Chapter 4.6), correlated accesses cause pages to be falsely identified as **hot**, i.e. worth to keep in main memory for longer. As the correlated access pattern is very common in programs, the more recent algorithms contain some mechanism for filtering them out. CART in Chapter 4.13 and CLOCK-Pro in Chapter 4.15 are good examples.

3.4.2 Scan

The **scan** pattern consists of a sequential one time pass over pages. In scan, a page has many correlated accesses in a short period of time and then it is not accessed for a long time. Scan is performed on a large number of pages in relatively short time. Typical processes, that causes scanning, are time triggered maintenance jobs,

like daily backup, and user initiated one time actions, like searching a set of files for a matching string. Scanning may cause pushing of frequently used pages out of main memory, if the scanned pages are misclassified as hot or active pages. This is generally considered undesirable and the page replacement algorithm behaviour is usually tested for scanning. The page replacement algorithm, that does not allow scanning to push frequently used pages out of main memory, is said to be scan resistant. Scan is a typical LRU-unfriendly pattern [9]

3.4.3 Loop

The **loop** pattern is, like name the suggests, sequential, repeated pass over a set of pages. The loop pattern is also an LRU -unfriendly pattern [9]. In the worst case scenario, where loop accesses more pages than fit in memory, LRU will have zero hit rate and page fault happens every time the next page in loop is accessed.

3.5 Page replacement policies: global and local

A page replacement algorithm can be either global or local. Local replacement means that replacement candidates are searched only from pages, that belong to the page faulted process. This effectively means that, if the working set of the process does not fit to the memory reserved for it, the process will page fault constantly. This generally leads to poor usage of memory resources. In global page replacement all processes compete for the main memory and the replacement algorithm automatically adapts to changing working set sizes of running processes.

3.6 Goals

As in many algorithms, performance is the main goal in developing page replacement algorithms. There are many factors that affect the performance of page replacement. As computer is implemented in physical hardware, the features of the hardware change greatly the balance of different trade-offs. In fact, it is the evolution of hardware that has made page replacement algorithm research needed again [11]. The amount of main memory has increased far more than the speed of reading and writing disks (i.e. secondary memory). So if an application has been swapped to a disk, the time to return it to main memory has increased significantly because the

memory usage of applications has increased more than the speed of disks. As a consequence, many operations users normally do, like starting the computer, have become slower. This leads to an important goal of minimizing disk IO and IO latency. Minimizing disk IO and keeping IO latency low also frees disk capacity for actual application work, and keeps the memory subsystem from disturbing latency sensitive applications, like media players, too much.

4 Page replacement algorithms

4.1 Optimal replacement

The Optimal page replacement algorithm is easy to describe. When memory is full, you always evict a page that will be unreferenced for the longest time [3]. This scheme, of course, is possible to implement only in the second identical run, by recording page usage on the first run. But generally the operating system does not know which pages will be used, especially in applications receiving external input. The content and the exact time of the input may greatly change the order and timing in which the pages are accessed. But nevertheless it gives us a reference point for comparing practical page replacement algorithms. This algorithm is often called **OPT** or **MIN**.

Using the formal model specified earlier, the optimal page replacement algorithm can be defined as

$$g(S, t, r_{t+1}) = \begin{cases} (S, t), & \text{if } r_{t+1} \in S \cup \{\emptyset\}, \\ (S \cup \{r_{t+1}\} \setminus \{y\}, t+1), & \text{if } r_{t+1} \notin S, \end{cases}$$

where y has the longest time to the next reference for all pages in S. The page replacement decision depends only on the time of reference and the control state is fully described as t. In this algorithm and following algorithms, the size of S is m pages.

4.2 Random replacement

Probably the simplest page replacement algorithm is the replacement of a random page. If a frequently used page is evicted, the performance may suffer. For example, some page, that contains program initialization code which may never be needed again during the program execution, could be evicted instead. So there are performance benefits available with choosing the right page [3].

Using the formal model specified earlier, the random page replacement algo-

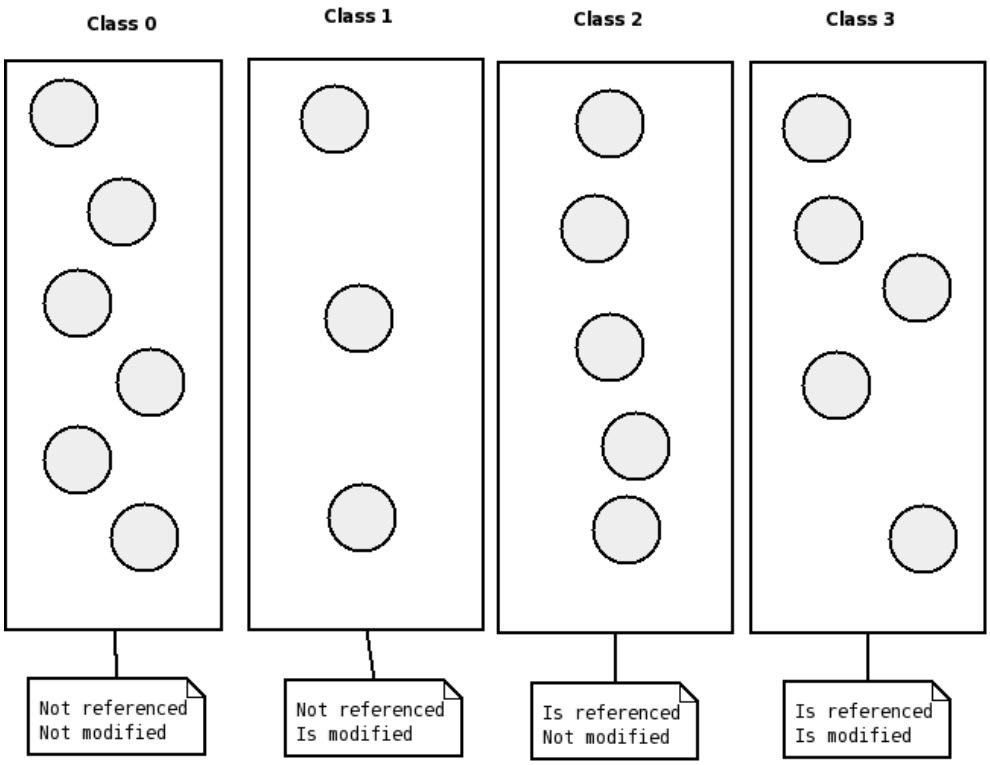

Figure 4.1: In NRU, the page to be evicted is selected from lowest class that contains pages.

rithm can be defined as

$$g(S, q_0, r_{t+1}) = \begin{cases} (S, q_0), & \text{if } r_{t+1} \in S \cup \{\emptyset\}, \\ (S \cup \{r_{t+1}\} \setminus \{y\}, q_0), & \text{if } r_{t+1} \notin S, \end{cases}$$

where y is selected randomly from all pages in S. The algorithm has only one control state as the replacement decision is done identically every time.

4.3 Not Recently Used (NRU)

In the NRU algorithm [16], pages in main memory are classified based on usage during the last clock tick (See figure 4.1). Class 0 contains pages that are not referenced nor modified, Class 1 pages that are not referenced but modified, Class 2 pages that are referenced but not modified, and Class 3 contains pages that are both referenced and modified. When a page must be evicted, NRU evicts a random page from the lowest class that contains pages.

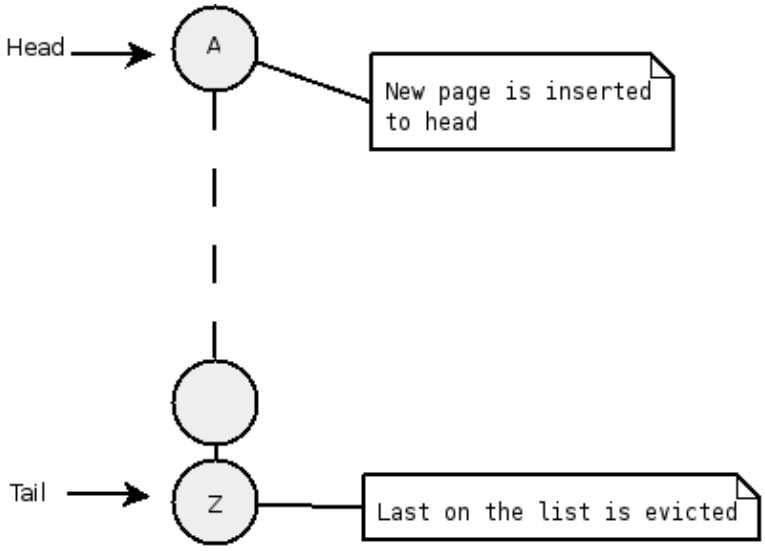

Figure 4.2: FIFO

Using the formal model specified earlier, NRU page replacement can be defined as

$$g(S, q_t, r_{t+1}) = \begin{cases} (S, q_{t+1}), & \text{if } r_{t+1} \in S \cup \{\emptyset\}, \\ (S \cup \{r_{t+1}\} \setminus \{y\}, q_{t+1}), & \text{if } r_{t+1} \notin S, \end{cases}$$

where y is a random page from the lowest class that has pages. The control state is defined as collection of classes $q_t = \{C0_t, C1_t, C2_t, C3_t\}$. So if $C0_t = \emptyset$ and $C1_t \neq \emptyset$ page $y \in C1_t$ and $q_{t+1} = \{C0_{t+1}, C1_{t+1}, C2_t, C3_t\}$, where $C0_{t+1} = \{r_{t+1}\}$ and $C1_{t+1} = C1_t \cup r_{t+1} \setminus \{y\}$.

NRU is relatively simple to understand and implement. Implementation has a relatively low overhead, although the reference bit needs to be cleared after every clock tick. Performance is significantly better compared to pure random selection in general usage.

4.4 First-In, First-Out (FIFO)

The simple First-In, First-Out (FIFO) algorithm [16] is also applicable to page replacement. All pages in main memory are kept in a list where the newest page is in head and the oldest in tail. When a page needs to be evicted, the oldest page is selected (page Z in figure 4.2), and the new page is inserted to head of the list (page A in figure 4.2).

Using the formal model specified earlier, FIFO page replacement can be defined as

$$g(S, q_t, r_{t+1}) = \begin{cases} (S, q_t), & \text{if } r_{t+1} \in S \cup \{\emptyset\}, \\ (S \cup \{r_{t+1}\} \setminus \{y_m\}, q_{t+1}), & \text{if } r_{t+1} \notin S. \end{cases}$$

The control state is defined as $q_t = (y_1, y_2, ..., y_m)$.

Implementation of FIFO is very simple and it has a low overhead, but it is not very efficient. FIFO does not take advantage of page access patterns or frequency. Most applications do not use memory, and subsequently the pages that hold it, uniformly, causing heavily used pages to be swapped out more often than necessary.

4.5 Least Recently Used (LRU)

The Least Recently Used (LRU) [16] algorithm is based on generally noted memory usage patterns of many programs. A page that is just used will probably be used again very soon, and a page that has not been used for a long time, will probably remain unused. LRU can be implemented by keeping a sorted list of all pages in memory. The list is sorted by time when the page was last used. This list is also called LRU stack [13]. In practice this means that on every clock tick the position of the pages, used during that tick, must be updated. As a consequence, the implementation is very expensive, and not practical in its pure form. Updating on every clock tick is also an approximation, as it does not differentiate between two pages that were referenced to during the same clock tick [16].

Using the formal model specified earlier, LRU page replacement can be defined as

$$g(S, q_t, r_{t+1}) = \begin{cases} (S, q_{t+1}), & \text{if } r_{t+1} \in S \cup \{\emptyset\}, \\ (S \cup \{r_{t+1}\} \setminus \{y_m\}, q_{t+1}), & \text{if } r_{t+1} \notin S, \end{cases}$$

where y_m is least recently used page in S. Control state is defined as

$$q_t = (y_1, y_2, y_3..., y_m),$$

where resident pages are ordered by their most recent reference, y_1 being the most recently referenced and y_m the least recently referenced.

LRU is relatively simple to implement and it has constant space and time overhead for given memory. LRU uses only page access recency as the base of the decision. As discussed earlier, many common memory access patterns follow the principle of locality and LRU does work well with these. Typically, workloads with strong

locality have the most page references with reuse distance small enough to allow these pages to fit in the main memory [11]. LRU is good also for stack distances that have common stationary distribution [5]. LRU is even optimal when LRU stack depth distribution is assumed in page referencing [2]. LRU adapts fast to changes in the working set [13] and works well with workloads with no clear large-scale access patterns [7].

LRU is not without problems. The LRU list, or stack, is a shared structure and must be protected from concurrent access. Especially in virtual memory, where it is updated on every reference, it will severely limit the performance [2]. LRU also does not take page use frequency into account. If a frequently used page has slightly larger inter reference interval (or reuse distance), than can fit to main memory, the page will always be evicted. This also makes LRU extremely vulnerable to scan access pattern [7], [9], [2]. In general, LRU does not work well with workloads that have clear large-scale access patterns until the pages belonging to the pattern fit to the main memory [7]. While stack distances with common stationary distribution work well with LRU, most real world programs have strong correlation among stack distances [5]. Non-uniform page referencing in general is also not handled well by LRU [13] and it also does not support programs following the phase-transition model [5]. When used as a global policy, program execution interaction may cause programs pages to fall further in the LRU stack as a result of the program itself page faulting, which is again the principle of LRU [9]. In other words, pages are falling undesirably low in LRU stack because the program's virtual time has stopped for the duration of the page fault.

In LRU-k [14], the LRU algorithm is improved by using the time of the kth most recent reference, instead of the most recent reference to page. This gives LRU a basic scan resistance as a page must have been used twice recently in order to be high in the LRU stack.

4.6 Second Change and CLOCK

The second change [16] algorithm makes slight modification to FIFO algorithm. Instead of swapping out the last page, the referenced bit is checked. If the bit is set, the page is then moved to the head of the list as if it had just arrived and the search continues (See figure 4.3). If all pages are referenced then the oldest page is evicted like in FIFO.

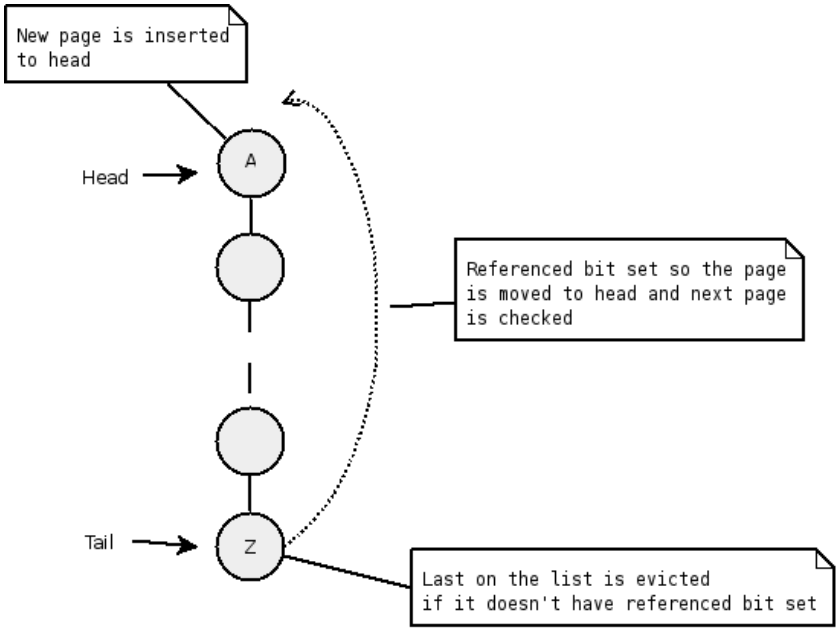

Figure 4.3: Second change

The CLOCK algorithm deserves a note as an implementation detail. CLOCK implements the second change so that pages are kept in a circular list with a pointer to the oldest page. When evicting, only the pointer needs to be updated and there is no need to move pages around (See figure 4.4). As mentioned earlier, CLOCK and its variants have been dominant in general purpose operating systems for a long time [9].

Using the formal model specified earlier, the second change algorithm can be defined as

$$g(S, q, r_{t+1}) = \begin{cases} (S, q_{t+1}), & \text{if } r_{t+1} \in S \cup \{\emptyset\}, \\ (S \cup \{r_{t+1}\} \setminus \{y_j\}, q_{t+1}), & \text{if } r_{t+1} \notin S, \end{cases}$$

where y_j is the first of the maintained circular FIFO list, which does not have the referenced bit set. The control state is defined as a circular FIFO list $q_t = (y_1, y_2, ..., y_i ..., y_m)$, where y_i is ith most recently referenced page and pages processed before the evicted page y_j are placed to the head of the list.

Second change algorithm tackles the basic problem of FIFO by literally giving the page a second change before swapping it out. It can also be though as a one-bit approximation of LRU. The second change removes the problem of keeping LRU list updated, but it also shares the rest of the problems of LRU [2], [9].

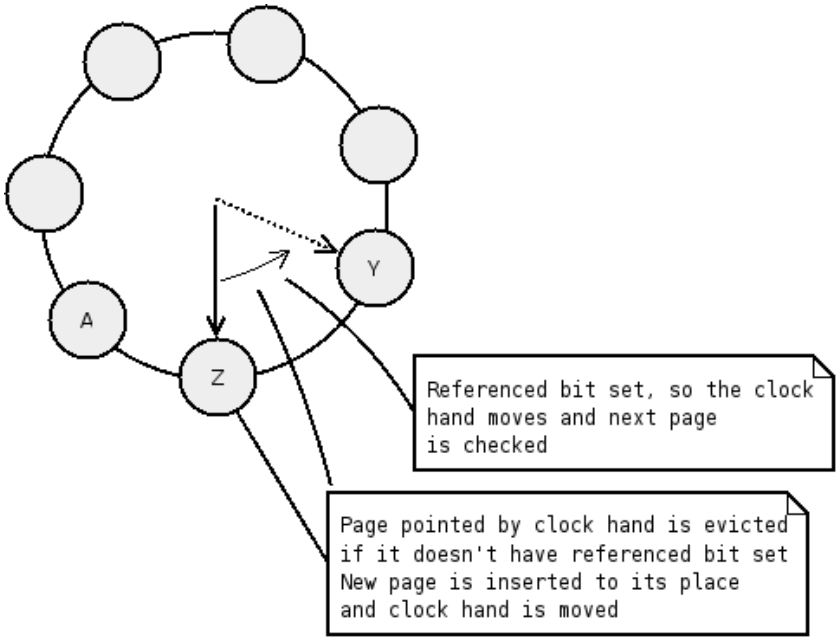

Figure 4.4: CLOCK in operation

4.7 Not Frequently Used (NFU)

Not Frequently Used (NFU) [16] is another approximation of LRU. In NFU, every page has an associated usage counter which is incremented on every clock tick the page is used. When a page needs to be evicted, the page with the lowest counter value is selected. The downside of this approach is that once some process uses some pages heavily, they tend to stay there for a while, even if they are not actively used anymore. This program model of doing computation in distinct phases is very common. Also programs, that have just been started, do not get much space in the main memory as the counters start from zero.

Using the formal model specified earlier, NFU algorithm can be defined as

$$g(S, q_t, r_{t+1}) = \begin{cases} (S, q_{t+1}), & \text{if } r_{t+1} \in S \cup \{\emptyset\}, \\ (S \cup \{r_{t+1}\} \setminus \{y_m\}, q_{t+1}), & \text{if } r_{t+1} \notin S, \end{cases}$$

where y_m is the least frequently used page in S. Control state is defined as

$$q_t = ((y_1, c_1), (y_2, c_2), (y_3, c_3) ..., (y_m, c_m)),$$

where y_i are resident pages ordered by the usage counters, c_i, and y_1 being the most frequently referenced page and y_m the least frequently referenced page.

4.8 Aging

With few modifications to NFU we get an **aging** algorithm that is a much better approximation of LRU. Instead of incrementing an integer counter, a bit presentation of unsigned integer can be used. On every clock tick the counter value of each page is bit shifted to right, and the referenced bit of the page is inserted at the left. While the integer value of the counter is still used to make the selection of the page to be evicted, the counter value behaviour favours pages that are referenced recently, and pages that were heavily used few seconds ago, but not anymore, will get evicted sooner. The downside in this is that as the counter decrements to zero quickly, and we have no way of knowing when two pages, with zero as counter value, have been used. The other might have just been decremented to zero, while the other may have been unused for a long time. In this case a random selection, with its performance implications, is performed [16].

4.9 Two Queue (2Q)

The two Queue, or 2Q, algorithm [12] tries to improve the detection of real hot pages and remove cold pages faster from the main memory. 2Q works by maintaining two separate lists. One is maintained as an LRU list, *Hot*, and the other as FIFO, *F*. The list *F* is further partitioned in to two parts *Fin* and *Fout*. The *Fin* list contains pages in main memory, while the *Fout* list contains only information of pages, not the actual contents. When page is first accessed, it is placed on the head of the *Fin* list. The position of the page, in the *Fin* list, is left untouched while it remains there. As new pages are used, *Fin* list will become full. When this happens the last page in *Fin* list is reclaimed next, but the information of the page is inserted to the head of the *Fout* list (page X in figure 4.5). If the page, now on the *Fout* list, is used, space is reclaimed for it and it is inserted to the head of the *Hot* list (page Y in figure 4.5). When the *Fin* list is not full, reclaiming is done from the tail of the *Hot* list. The page reclaimed from the *Hot* list is not inserted to any list, as it has not been used for a while (page Z in figure 4.5). Remember, that the *Hot* list is maintained as an LRU list.

The 2Q algorithm has two parameters, *Kin* and *Kout*. *Kin* is the maximum size of *Fin* and *Kout* is the maximum size of *Fout*. Authors note that setting these parameters is potentially a tuning target, but recommend reasonable values 25 % and

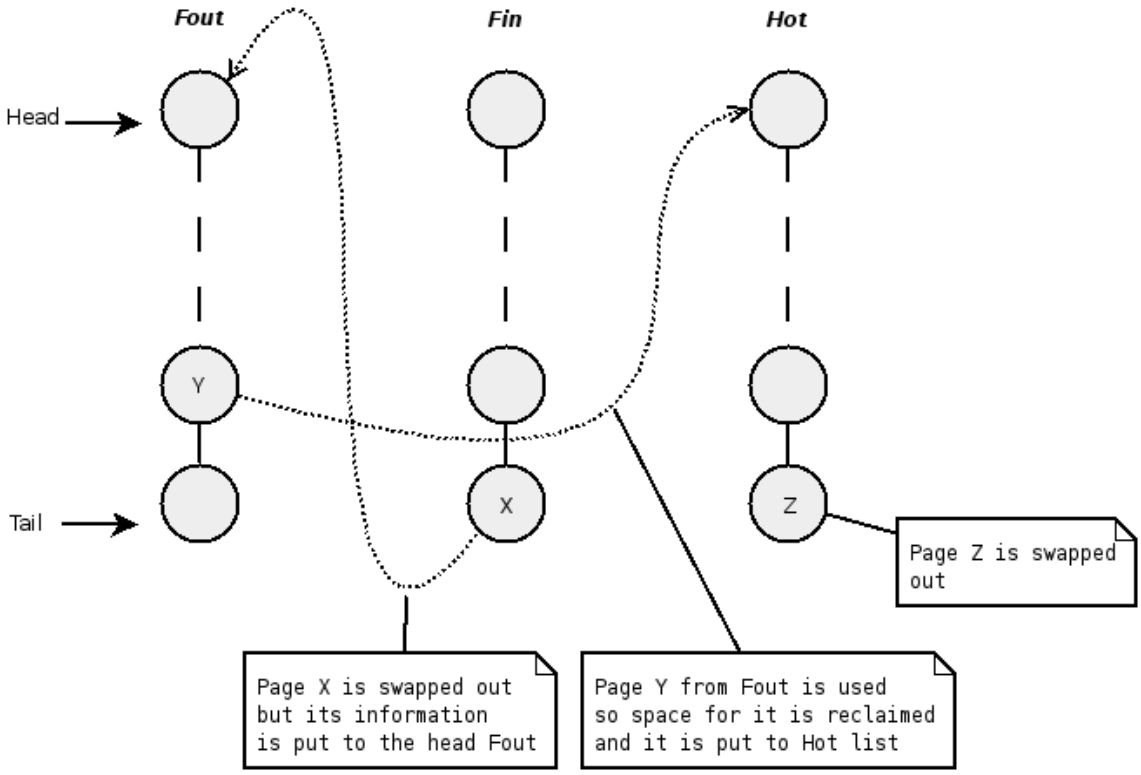

Figure 4.5: Operation of 2Q

50 %, of page frame count, for *Kin* and *Kout*, respectively.

Using the formal model specified earlier, 2Q algorithm can be defined as

$$g(S, q_t, r_{t+1}) = \begin{cases} (S, q_{t+1}), & \text{if } r_{t+1} \in S \cup \{\emptyset\}, \\ (S \cup \{r_{t+1}\} \setminus \{y\}, q_{t+1}), & \text{if } r_{t+1} \notin S, \end{cases}$$

where y is the last page on either the *Hot* list or the *Fin* list. The control state is defined as $q_t = \{Hot, Fin, Fout\}$, and each list is maintained like described earlier.

2Q algorithm targets to fix a typical memory access pattern by being scan resistant. Simple LRU list is easily trashed by scanned pages, that have many correlated accesses within small time, but are not needed after that. This is achieved by moving only truly hot pages to the main LRU list and allowing scanned pages to exit quickly without effect on the hottest working set. 2Q is an low overhead approximation of LRU-2 presented in [14] and the *Kin* parameter is essentially the same as CIP (Correlated Information Period) parameter in LRU-2 [13] .

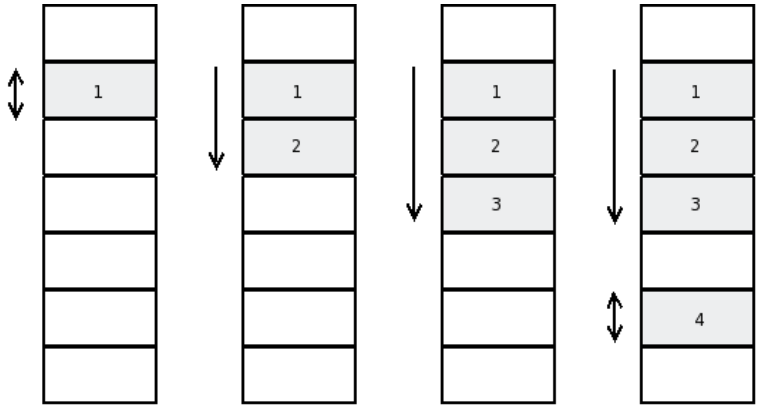

Figure 4.6: Sequence detection in SEQ

4.10 SEQ

The SEQ algorithm [7] by Gideon Glass and Pei Cao attacks rather directly against sequential memory access, an LRU unfriendly memory access pattern. SEQ works by detecting sequences of page faults within single processes address space and performs pseudo Most Recently Used (MRU) replacement. MRU tries to approximate optimal replacement algorithm. When no appropriate sequences are detected, SEQ falls back to LRU replacement.

A sequence is defined by four values: *PID*, *low*, *high* and *dir*. High and low are pages with highest and lowest virtual addresses, respectively. PID identifies the process and dir identifies which direction the sequence is. When a page fault occurs, SEQ checks if the faulted page is adjacent to a sequence, with appropriate direction, belonging to the process. If so, the page is catenated to that sequence. If the extended sequence overlaps an existing sequence, the overlapped sequence is deleted. If the faulted page is in the middle of a sequence, the sequence is broken into two sequences. One sequence includes pages from the low of the sequence to page before faulted page, if direction is up, and from the page next to the faulted page, when the direction is down. The other sequence consists only of the faulted page without direction. Finally, if none of the above apply, the faulted page forms a directionless sequence by itself.

A sequence detection is shown in figure 4.6. The first three faulted pages are detected as a sequence, where 1 is low, 3 is high and the direction is up. The fourth page faulted is a directionless sequence.

SEQ uses the detected sequences to find pages suitable for eviction. First, SEQ

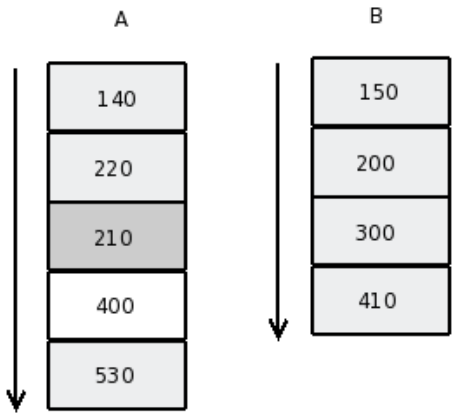

Figure 4.7: SEQ: Selecting a page to evict

chooses a sequence, among sequences longer than L pages, that has most recent time of the Nth most recent fault in the sequence. For example, if L is 3, N is 2 and there are two sequences presented in figure 4.7, sequence A is selected, because the second page of the sequence has timestamp 220 and the second page of sequence A has 200. From the selected sequence, SEQ evicts the first resident page, that is at least M pages from the head of the sequence. The head of a sequence is the direction of the sequence, usually the most recent page faulted to the sequence. If M is 2, the third page from the head of sequence A is evicted (the slightly darker page in figure 4.7), in previous example, because the second page is already swapped out. If no suitable sequences are found, SEQ performs LRU replacement. Authors of the algorithm suggest that appropriate values for L, N and M are 20, 5 and 20 ,respectively [7].

Using the formal model specified earlier, SEQ algorithm can be defined as

$$g(S, q_t, r_{t+1}) = \begin{cases} (S, q_{t+1}), & \text{if } r_{t+1} \in S \cup \{\emptyset\}, \\ (S \cup \{r_{t+1}\} \setminus \{y\}, q_{t+1}), & \text{if } r_{t+1} \notin S, \end{cases}$$

where the state q can be defined as collection of sequences and an LRU list for fall handling, $\{lru, seq_1, seq_2, ...\}$. The page y is selected as described above.

4.11 Adaptive Replacement Cache (ARC)

The Adaptive Replacement Cache (ARC) [13] algorithm, designed by Nimrod Megiddo and Dharmendra S. Modha, provides an improvement over LRU based algorithms

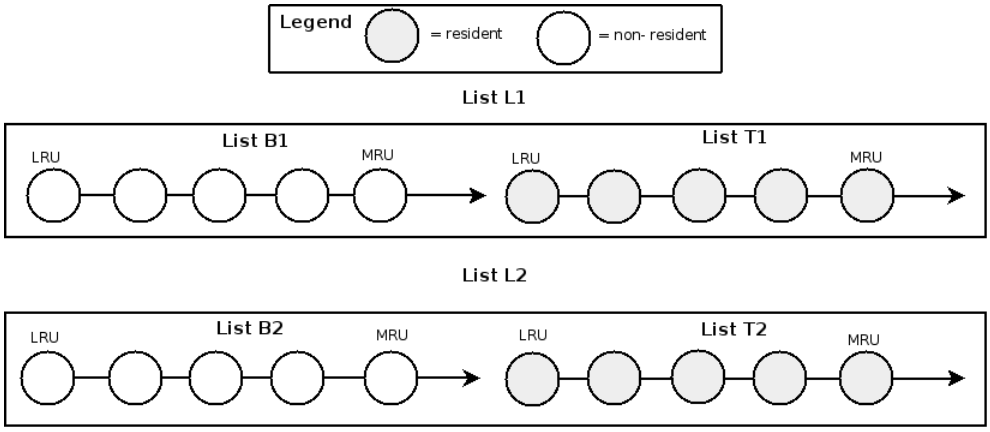

Figure 4.8: ARC: Lists (i.e. the cache directory)

by taking both recency and frequency into account. This is accomplished by maintaining two lists, L_1 and L_2 and remembering the history of the pages. The two lists together are called the cache directory. The list L_1 is used to capture the recency and the list L_2 to capture the frequency. Both L_1 and L_2 are kept at roughly the size of the number of page frames in the main memory (=c), so the history of at most c pages, not in the main memory, is remembered. The lists L_1 and L_2 are partitioned into two lists, T_1, B_1, and T_2, B_2, respectively (See figure 4.8). T_1 contains in-cache pages that have been accessed only once, and T_2 pages that have been accessed more than once, while on lists. Consistently, list B_1 stores the history of pages evicted from the list T_1, and B_2 stores the history of pages evicted from the list T_2. The algorithm has one integer parameter, p, which is the target size for T_1. As the size of the main memory is c, the target size of the list T_2 is implicitly defined as $c - p$. This parameter p is the balance between recently used and frequently used pages. It is desirable for the algorithm to perform well under various , changing workloads and therefore ARC includes automatic adaptation of the balance between the recency and the frequency by varying the parameter p.

The eviction policy in ARC is simple. If $|T_1| > p$, the least recently used page in T_1 is evicted, and if $|T_1| < p$, the least recently used page in T_2 is evicted. The eviction of a page, when $|T_1| = p$, is little more complex and, as the authors note, somewhat arbitrary. This situation is divided into three cases and it uses information of the page that caused the eviction. If the page is in B_1 or not found in $B_1 \cup B_2$, the least recently used page in T_2 is evicted. If the page is in B_2, the least recently used page in T_1 is evicted. The placement policy is even more simple. If the page is

found in the lists, it is placed to head of list T_2, otherwise it is placed to the head of the list T_1. When swapping in a new page (i.e. not in the lists), some history page needs to be removed. If $|T_1 \cup B_1| = c$ and B_1 is not empty, the least recently used page in B_1 is removed and if B_1 is empty, the least recently used page in T_1 is swapped out and removed from the list. If $|T_1 \cup B_1| < c$ and the history is full (i.e. $|T_1| + |B_1| + |T_2| + |B_2| = 2c$), the least recently used page in B_2 is removed.

The T_1 target size parameter, p, is continuously adapted to better serve the current workload. The basic idea of the adaption is to favour either recently used pages or frequently used pages. Adaptation is automatic and the direction is based on the cache hits to lists B_1 and B_2, while the amount of adaptation is based on the relative size of lists B_1 and B_2. If the requested page is found in B_1, the parameter p is increased. Likewise, if the requested page is found in B_2, the parameter p is decreased. The amount of increase is 1, if $|B_1| \geq |B_2|$, and $|B_2|/|B_1|$ otherwise. Similarly, the amount of decrease is 1, if $|B_2| \geq |B_1|$ and $|B_1|/|B_2|$ otherwise. Naturally, p is limited to the range $[0 - c]$. Increasing p means that more main memory is reserved for recently used pages, thus favouring them, while decreasing p favours frequently used pages.

Using the formal model specified earlier, ARC algorithm can be defined as

$$g(S, q_t, r_{t+1}) = \begin{cases} (S, q_{t+1}), & \text{if } r_{t+1} \in S \cup \{\emptyset\}, \\ (S \cup \{r_{t+1}\} \setminus \{y\}, q_{t+1}), & \text{if } r_{t+1} \notin S, \end{cases}$$

where y is the least recently used page of the list T_1 or the list T_2. The state q can be defined simply as set of four lists and the target size parameter $\{T_1, B_1, T_2, B_2, p\}$ which are all maintained as described before.

ARC has several good qualities. It has constant complexity per request, it uses both the recency and the frequency of page usage and it balances between them automatically. Scan resistancy is provided by differentiation of frequently used pages from pages that have only been used once recently. The scanned pages enter only the list T_1 and are removed relatively quickly, as they are not requested for a second time. Pages, that need protection from scanning, like program text, are naturally located in the list T_2, which remains unaffected by processing of scanned pages. Pages with correlated accesses may, however, enter T_2 undesirably. Unfortunately, ARC is designed for databases and has operations on every page request, and is therefore unsuitable for virtual memory. ARC has, however, been an important inspiration for two other algorithms, CAR and CART (See chapters 4.12 and 4.13), which are suitable for a high throughput environment, such as virtual memory.

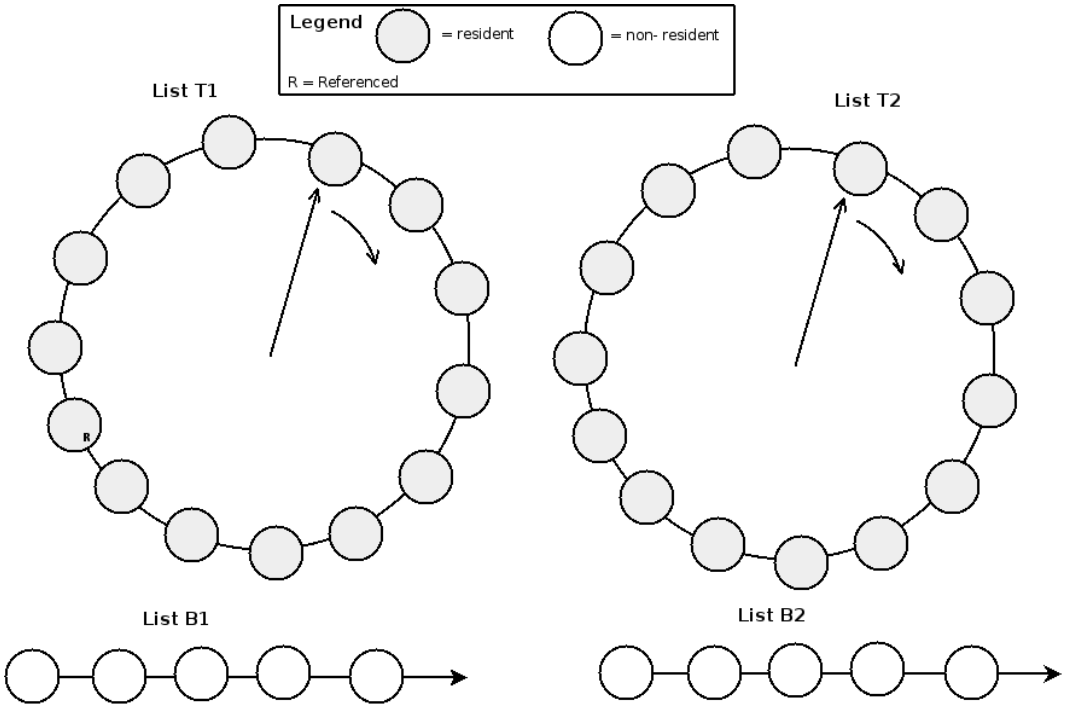

Figure 4.9: CAR style clocks

4.12 CLOCK with Adaptive Replacement (CAR)

CLOCK with Adaptive Replacement (CAR) [2] algorithm, and its variation CAR with Temporal filtering (CART) [2], presented in chapter 4.13, are strongly inspired by ARC (See chapter 4.11).

As in ARC, cache directory of $2c$ pages is kept, when the main memory can hold c pages. The directory is also partitioned to two lists L_1 and L_2, which are further partitioned to T_1 and B_1, and T_2 and B_2, respectively. The lists are maintained much in the same fashion as in ARC, but there is one big difference to ARC. The strict LRU ordering of pages in T_1 and T_2 is changed to a second change (or more precisely CLOCK, see figure 4.9). This gives the advantage of requiring only the referenced bit to be set on a page access, which is already handled by MMU, and thus action is only needed on page fault.

On page fault, the list T_1 is scanned until a page with the referenced bit unset is found. Let T_1' be the pages that the scan passed. Now, the eviction policy of CAR is as follows. If $|T_1 \setminus T_1'| \geq p$, a page from T_1 is evicted. Otherwise, a page from $T_1' \cup T_2$ is evicted. The placement policy is exactly as in ARC; if the page is not found in the history (i.e. $B_1 \cup B_2$) it is placed to T_1, otherwise it is placed to T_2. The history is

managed by removing a page from B_1, if $|T_1 \cup B_1| = c$, and from B_2 otherwise.

CAR is rather straightforward adaption of ARC for a high throughput environment, as it removes the overhead of maintaining strict LRU lists.

The formal model for CAR is very similar to ARC and can be defined as

$$g(S, q_t, r_{t+1}) = \begin{cases} (S, q_{t+1}), & \text{if } r_{t+1} \in S \cup \{\emptyset\}, \\ (S \cup \{r_{t+1}\} \setminus \{y\}, q_{t+1}), & \text{if } r_{t+1} \notin S, \end{cases}$$

where y is the least recently used page of the list T_1 or the list T_2. The state q can be defined simply as a set of four lists and the target size parameter $\{T_1, B_1, T_2, B_2, p\}$ which are all maintained as described above.

4.13 CAR with Temporal filtering (CART)

CAR with Temporal filtering (CART) [2] is a variation of CAR (See chapter 4.12) by same authors. To CAR, CART adds a filter to better handle correlated accesses, which are typical in virtual memory.

In CAR, a page is moved from T_1 to T_2 if it has been used while on T_1. CART changes this by requiring, that either $|T_1| \geq min(p + 1, |B_1|)$ or the page must first enter B_1, before it can be put to T_2. This means, that a frequently used page will stay on the "recently used" list, if it is used often enough. This prevents pages with few correlated accesses to enter T_2, where it would possibly be much longer than necessary (for example, a scan that uses each page more than once). Another major difference is, that a page from T_2 is moved back to T_1, if it has the referenced bit set. These combined mean, that the list T_1 acts as a temporal locality window.

CART implements filtering by marking each page in the cache directory as either S, for short-term utility, or L, for long-term utility (See figure 4.10). Every page in B_1 is marked as S and every page in $T_2 \cup B_2$ is marked as L. Pages in T_1 can be marked as S or L. When a page enters the cache directory for the first time, it is marked as S. The page stays in T_1 as long as it has the referenced bit set when processed. While on T_1, the page is changed from S to L if $|T_1| \geq min(p + 1, |B_1|)$. A page marked as L is moved to T_2 if it has the referenced bit unset when processed. Pages in T_2 are moved to T_1, if they have the referenced bit set. If the page is found in $B_1 \cup B_2$, it is marked as L and placed to T_1. Naturally, every time a page is processed, the referenced bit is unset.

The history page from B_1 or B_2 is removed when $|B_1| + |B_2| = c + 1$. If $|B_1| >$

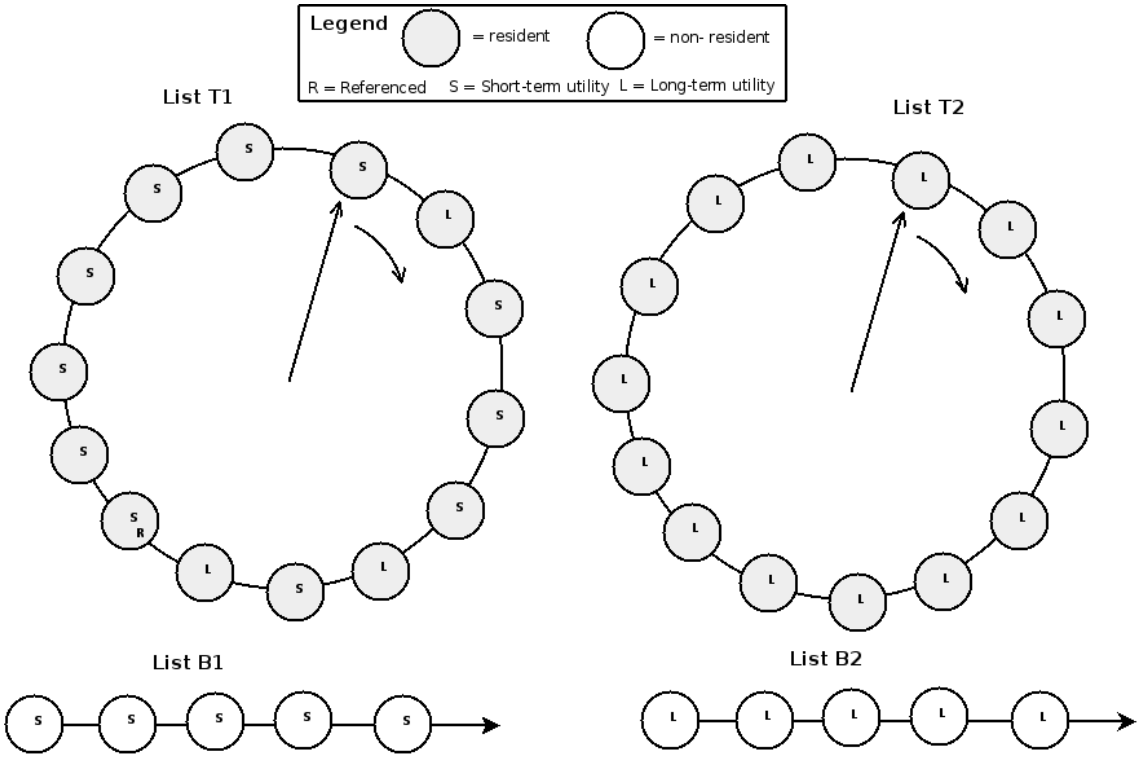

Figure 4.10: CART style clocks

$max(0, q)$ or B_2 is empty, the least recently used page from B_1 is removed. Otherwise, the least recently used page from B_2 is removed.

The adaption in CART is done by maintaining a target size p for T_1, and additionally a target size q for B_1. Also the number of pages marked as S and as L are maintained by n_S and n_L, respectively. Like in CAR and ARC, p is increased when the requested page is found in B_1 and decreased when it is found in B_2. The amount of increase is $n_S/|B_1|$, if $n_S > |B_1|$, and 1 otherwise. Similarly, the amount of decrease is $n_L/|B_2|$, if $n_L > |B_2|$, and 1 otherwise. Again, the value of p is limited to the range $[0 - c]$. Target size q for the list B_1 is maintained as follows. If the requested page is found in B_2 and $|T_2| + |B_2| + |T_1| - n_S \geq c$, the value of q is set as $q = min(q + 1, 2c - |T_1|)$. When moving a page from T_1 to T_2, the value of of q is set as $q = max(q - 1, c - |T_1|)$.

The formal model for CART is very similar to ARC and can be defined as

$$g(S, q_t, r_{t+1}) = \begin{cases} (S, q_{t+1}), & \text{if } r_{t+1} \in S \cup \{\emptyset\}, \\ (S \cup \{r_{t+1}\} \setminus \{y\}, q_{t+1}), & \text{if } r_{t+1} \notin S, \end{cases}$$

where y is the least recently used page of list T_1 or list T_2. The state q can be defined

simply as a set of four lists and the target size parameters $\{T_1, B_1, T_2, B_2, p, q\}$ which are all maintained as described above.

4.14 Token-ordered LRU

Song Jiang and Xiaodong Zhang [11] noticed one significant problem in the global LRU replacement policy. A process may not be using pages belonging to its working set just because it is page faulting. A single page fault may cause one IO operation for reading a page from the secondary memory and another for writing a dirty page to it. This can lead to a significant delay in the execution of the process and to marking real working set pages of the process as candidates for eviction. Situation gets worse, if these pages are then actually evicted, as the process is effectively causing its own memory to be evicted. These working set pages, that are marked as candidates because the process is page faulting, are called **false LRU pages**. Eviction of these false LRU pages can cause serious **trashing** in the system.

Token-ordered LRU [11] uses a system wide token to prevent false LRU pages from being evicted. When no main memory is available and a process tries to allocate more memory, the process grabs a token before pages for eviction are searched. Now, that candidates for eviction are searched, the pages, belonging to the process holding the token, are excluded. The pages of the token holder are strongly protected from eviction, and thus allows it to be executed with working set in main memory. This guarantees that at least one process continues to execute efficiently and prevents trashing in momentarily memory demand peaks, which are typical when, for example, system maintenance operations are performed. The token is always first taken by the process that caused the page fault. As the execution continues, the process holding the token is monitored and other processes can compete for the token. If no pages from any other process can be evicted, pages from the process holding the token are evicted and the process may lose the token. Also, if the process holds the token for too long, it is released. Overall, the target is to give the token to a short lived process or to a process that holds lots of resources in the hope that it will finish execution and release all the resources it holds.

Token-ordered LRU is not actually an algorithm itself, but an addition to LRU based algorithms to prevent trashing caused by program interaction. Implementation of token-ordered LRU was officially adopted in the Linux kernel 2.6.10.

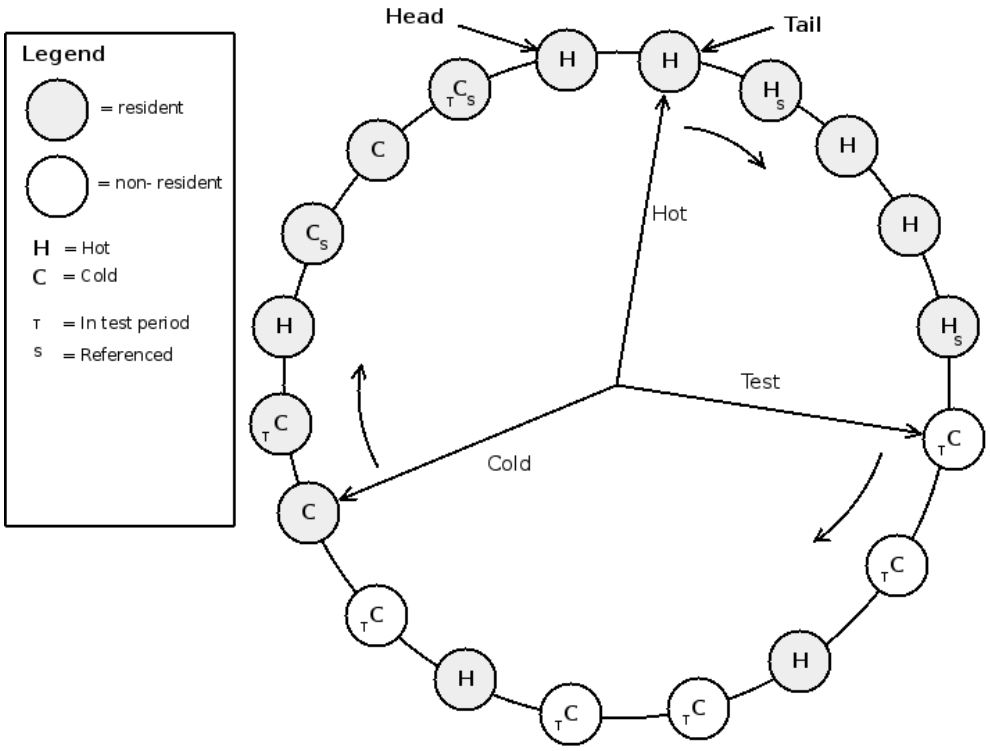

Figure 4.11: CLOCK-Pro style clock (The graphical style used in this thesis is adapted from [9])

4.15 CLOCK-Pro

CLOCK-Pro replacement algorithm [9] attacks weaknesses of LRU by changing the criteria of selecting pages for eviction, while maintaining the simple single circular list approach of CLOCK algorithm. Instead of using recency as the main criteria, as LRU does, CLOCK-Pro uses reuse distance. As discussed earlier, reuse distance is defined as the number of distinct page accesses between current access and previous access of a page. CLOCK-Pro was inspired by LIRS [10] I/O buffer cache replacement algorithm.

Using the same strategy as other algorithms, such as 2Q, ARC, CAR and CART, CLOCK-Pro keeps information of swapped out pages for some time. It uses that information to detect the reuse distance of the swapped out pages. However, instead of maintaining separate lists of resident and non-resident pages, like 2Q does, CLOCK-Pro keeps all pages in the same clock.

CLOCK-Pro keeps track of all pages in the main memory and the same amount of pages, that are swapped out. The resident pages are divided to two types, hot

pages and cold pages. The number of hot pages is m_h and the number of resident cold pages is m_c. The size of the total main memory, in pages, is m, which is equal to $m_h + m_c$. Additionally, information of m non-resident pages is kept for the reuse detection.

Instead of one clock hand, CLOCK-Pro has three hands (figure 4.11): hot, cold and test.

In CLOCK-Pro a page has three basic states. A page can be either hot, cold, and cold page with an additional flag indicating that it is in test period. When a new page is inserted to the main memory it is marked as cold page in test period.

The page fault handling, presented in figure 4.12, is performed as follows. First, the cold hand is used to find a page to be evicted. If the page pointed by the cold hand has its reference bit set and it is not in test period, the reference bit is cleared, test period is initiated and it is moved to the head. If the page is already in test period, the referenced bit is cleared and it is promoted to a hot page. After this, the hot hand is run once (the process of running the hot hand is explained later) and running of the cold hand is then continued. If the page pointed by the cold hand doesn't have the reference bit set, it is evicted. If the evicted page is in test period it is kept on the clock, otherwise it is removed from the clock. The cold hand skips any hot pages it encounters. After a page is evicted, the cold hand is run to the next resident cold page and stopped there. Next, the faulted page is handled.

If the faulted page is not on the list, it is placed to the head and its test period is initiated. If the number of cold pages reaches the threshold ($m_c + m$), the test hand is run next (the running of the test hand is explained later). If the faulted page is in the clock, it is promoted to a hot page, moved to head and the hot hand is run next. If the faulted page is on list, it is promoted to a hot page and placed to the head. After that, the hot hand is run next.

We will first handle running of the test hand. If the page pointed by the test hand is resident and in test period, its test period is terminated. If it is in test period and not resident, it is removed from clock, the test hand is run to the next cold page in test period, and page fault handling is done. The test hand doesn't touch any pages that are not in test period.

Now we will go through the running of the hot hand. The hot hand clears the referenced bit in hot pages that it passes and handles cold pages exactly like the test hand does. If the page pointed by the hot hand does not have referenced bit set, it is demoted to cold page and the hot hand is run to the next hot page, still handling the

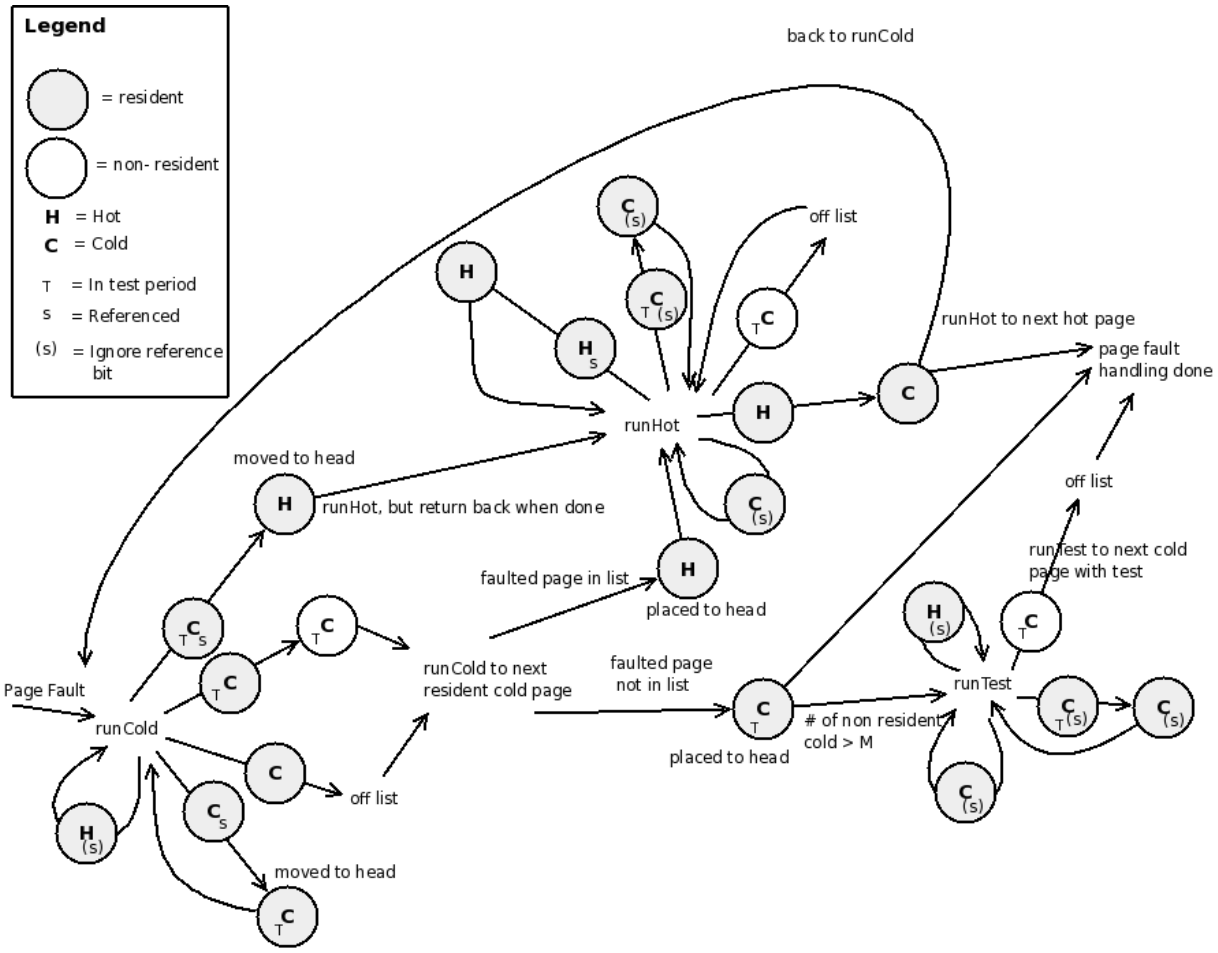

Figure 4.12: CLOCK-Pro page fault handling

cold pages on the way, and stopped there. If the hot hand was run while searching for a page to evict, the running of the cold hand is continued. Otherwise the page fault handling is done.

There are few things worth to note. First, only a cold page can be swapped out. Before a hot page is considered for eviction, it has to be demoted to a cold page. A cold page is granted a test period when it is added to clock and when it is, with reference bit set, passed by the cold hand. The test period of a cold page is terminated when it is promoted to hot page or passed by either the hot hand or the test hand. Second, the cold hand work handles cold pages exactly like in basic CLOCK algorithm and is moved independently of the hot hand and the test hand. The hot hand pushes the test hand as it performs work on behalf of the test hand. The concept of reuse distance is achieved because all pages are kept in the same clock and the ordering of the pages is only changed by the cold hand. This guarantees that the relative reuse distance of all pages from the page pointed by the cold hand to the tail of list are maintained as long as they are in the clock.

Significant parameter in CLOCK-Pro is the percentage of resident cold pages from main memory. As the cold hand is mainly in charge of handling the resident cold pages, setting m_c near 100 percent of main memory size, causes the algorithm to perform very similarly to basic CLOCK. When m_c is small, the new pages do not have much time to be reused before they are evicted.

CLOCK-Pro can be made adaptive by dynamically adjusting the balance of m_c and m_h. Adaption is based on current reuse distance distribution. When a cold page, whether resident or not, is accessed in its test period, the value of m_c is incremented by one. When a test period of a cold page, again whether resident or not, is terminated, the value of m_c is decremented by one.

Using the formal model specified earlier, CLOCK-Pro algorithm can be defined as

$$g(S, q_t, r_{t+1}) = \begin{cases} (S, q_{t+1}), & \text{if } r_{t+1} \in S \cup \{\emptyset\}, \\ (S \cup \{r_{t+1}\} \setminus \{y\}, q_{t+1}), & \text{if } r_{t+1} \notin S, \end{cases}$$

where y is selected as described above . The state q can be defined as the three clock hands and the parameter m_c, $\{Hand_H, Hand_C, Hand_T, m_c\}$ which are all maintained as described before.

In trace testing, authors found that CLOCK-Pro performs significantly better than CLOCK and better than CAR with most workloads.

5 Empirical analysis

We have implemented nine algorithms for empirical analysis, OPT (Ch. 4.1), FIFO (Ch. 4.4), CLOCK (Ch. 4.6), LRU (Ch. 4.5), 2Q (Ch. 4.9), ARC (Ch. 4.11), CAR (Ch. 4.12), CART (Ch. 4.13) and CLOCK-Pro (Ch. 4.15). Implementation of these algorithms is done as offline replacement with demand paging policy. In this analysis, we focus on the offline performance and thus the overhead of the algorithms is not analyzed. It should, however, be noted, that OPT, LRU and ARC are not suitable for real world implementation because of the overhead and because OPT requires the full trace as a parameter. Trace data used (See Chapter 5.2) is very simple and the results based on it can not be used as a real world performance indicator. The source code of the implementation can be found in Appendix B.

5.1 Metrics

Offline performance of the algorithms is measured as page fault count and hit ratio. Hit ratio (hr) is calculated as

$$\text{hr} = 100 - \text{mr}.$$

Miss ratio (mr) is

$$\text{mr} = 100 * ((\#\text{pf} - \#\text{distinct}) / (\#\text{refs} - \#\text{distinct})),$$

where *#pf* is the number of page faults, *#distinct* is the number of distinct pages used in the trace and *#refs* is the number of references in the trace.

Page fault charts give graphical overview of how the algorithm behaves compared to the OPT algorithm.

5.2 Trace Data

Trace data is a reference string and consists of a sequence of page references to virtual address space of a single process. Traces are both generated with scripts (Chapter 5.3) and gathered from a real execution of a program (Chapter 5.4). The generated traces are used to show some key differences in the algorithms. The real

execution trace is included more of as an example of what can be done, than for an algorithm analysis.

5.3 Generated traces

Generated traces, used in the analysis, are short and are created using a simple script (See Appendix C.1). The generation is based on the phase-transition model (See Chapter 3.3.6) and typical memory usage patterns (See Chapter 3.4).

Each generated trace has a set of actively used pages (between *hot_range_start* and *hot_range_end* in the *DEFAULT* section of the trace configuration file), called a hot range. Pages from this range are referenced often throughout the trace. The hot range represents the program text, stack and important data structures, that are needed constantly throughout the execution of the modeled program. Therefore, these are the pages, that are most likely valuable, to be kept in main memory all the time. It is assumed that a good replacement algorithm identifies these pages as hot.

Trace generation script supports four type of phases, *loop*, *scan*, *random* and *correlated*, which each imitate the corresponding memory access pattern. Loop phase generates given amount (*ref_count*) of references looping from *range_start* to *range_end* with one reference per page. Scan phase generates *ref_count* references starting from *range_start* with one reference per page. Random phase generates *ref_count* references with pages uniformly random selected between *range_start* and *range_end*. Correlated phase generates *ref_count* references starting from *range_start* with two references per page. References to pages in hot range are inserted in each type of phases.

In this analysis, all the generated traces are processed with 1000 pages of memory (i.e. page frames).

5.3.1 Scan trace

Scan trace (See Appendix C.2) consists of a hot range and five phases. The first phase is a loop, which overlaps the hot range. This is meant to stabilize the hot range, so that the algorithms should be able to identify it in the second phase. The second phase is a scan over previously unused pages. The third phase is the same loop as in the first phase. The fourth phase is a small random phase with pages from the range that was scanned in the second phase. Finally, the fifth phase is a scan over

unused pages.

Algorithm	Ref count	Page count	Page faults	Hit count	Hit ratio
OPT	16175	7150	8031	8144	90.24 %
FIFO	16175	7150	11539	4636	51.37 %
CLOCK	16175	7150	9985	6190	68.59 %
LRU	16175	7150	10471	5704	63.20 %
2Q	16175	7150	9222	6953	77.04 %
ARC	16175	7150	8841	7334	81.26 %
CAR	16175	7150	8843	7332	81.24 %
CART	16175	7150	9310	6865	76.07 %
CLOCKPro	16175	7150	9535	6640	73.57 %

Table 5.1: Results on scan data

Results for each algorithm are in Table 5.1. Page fault charts are in Figures 5.1, 5.2, 5.3 and 5.4.

The results in Table 5.1 support the need for scan resistance. FIFO, CLOCK and LRU lack scan resistance and perform very similarly, as can be seen in Figures 5.1 and 5.2. All other algorithms show clear advantage over FIFO, CLOCK and LRU, and best performance is achieved with ARC and CAR. Most of the performance differences can be found in the short loop phases in the trace, which can be observed in the charts.

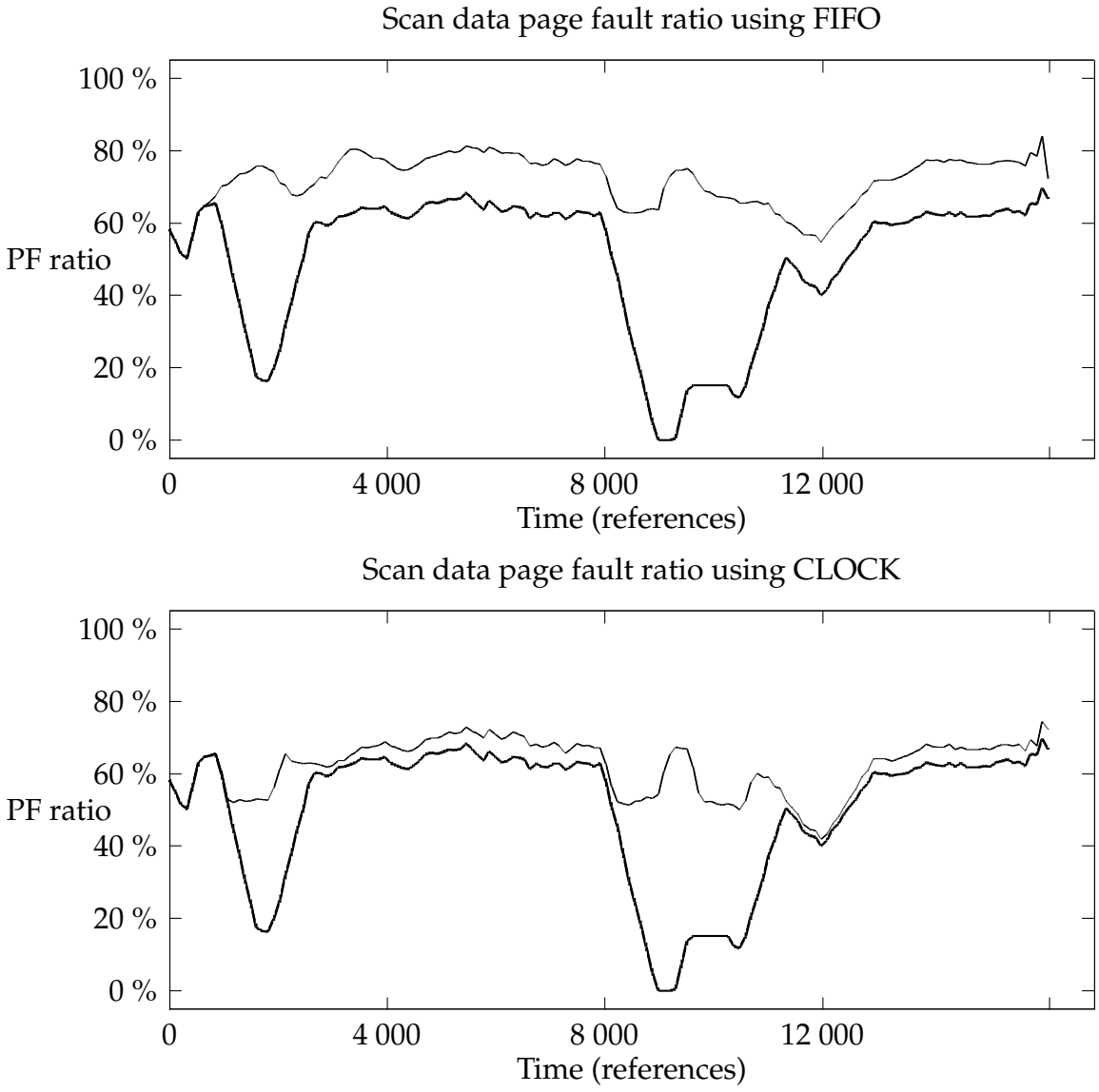

Figure 5.1: Page fault ratio on scan data: FIFO and CLOCK

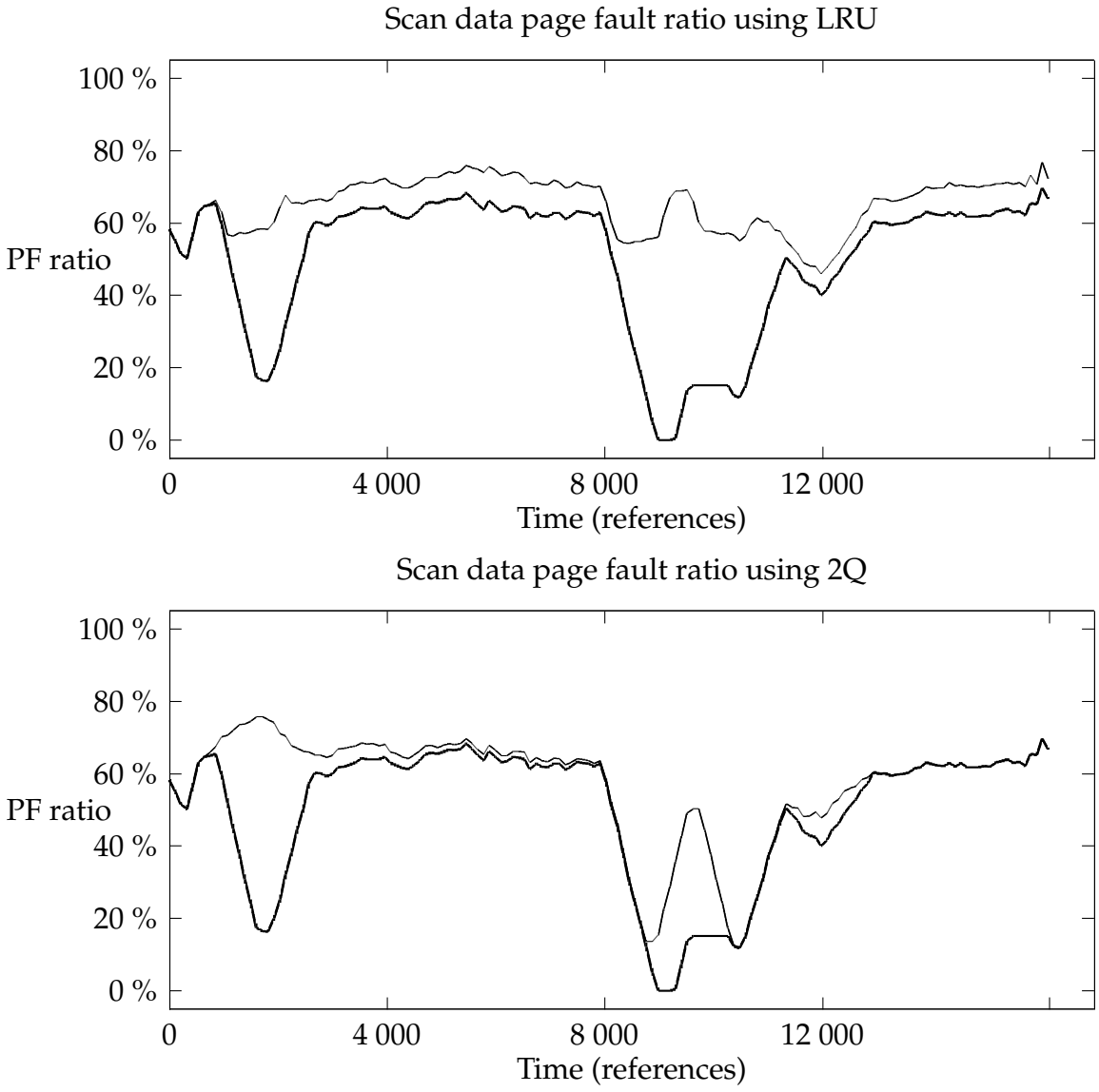

Figure 5.2: Page fault ratio on scan data: LRU and 2Q

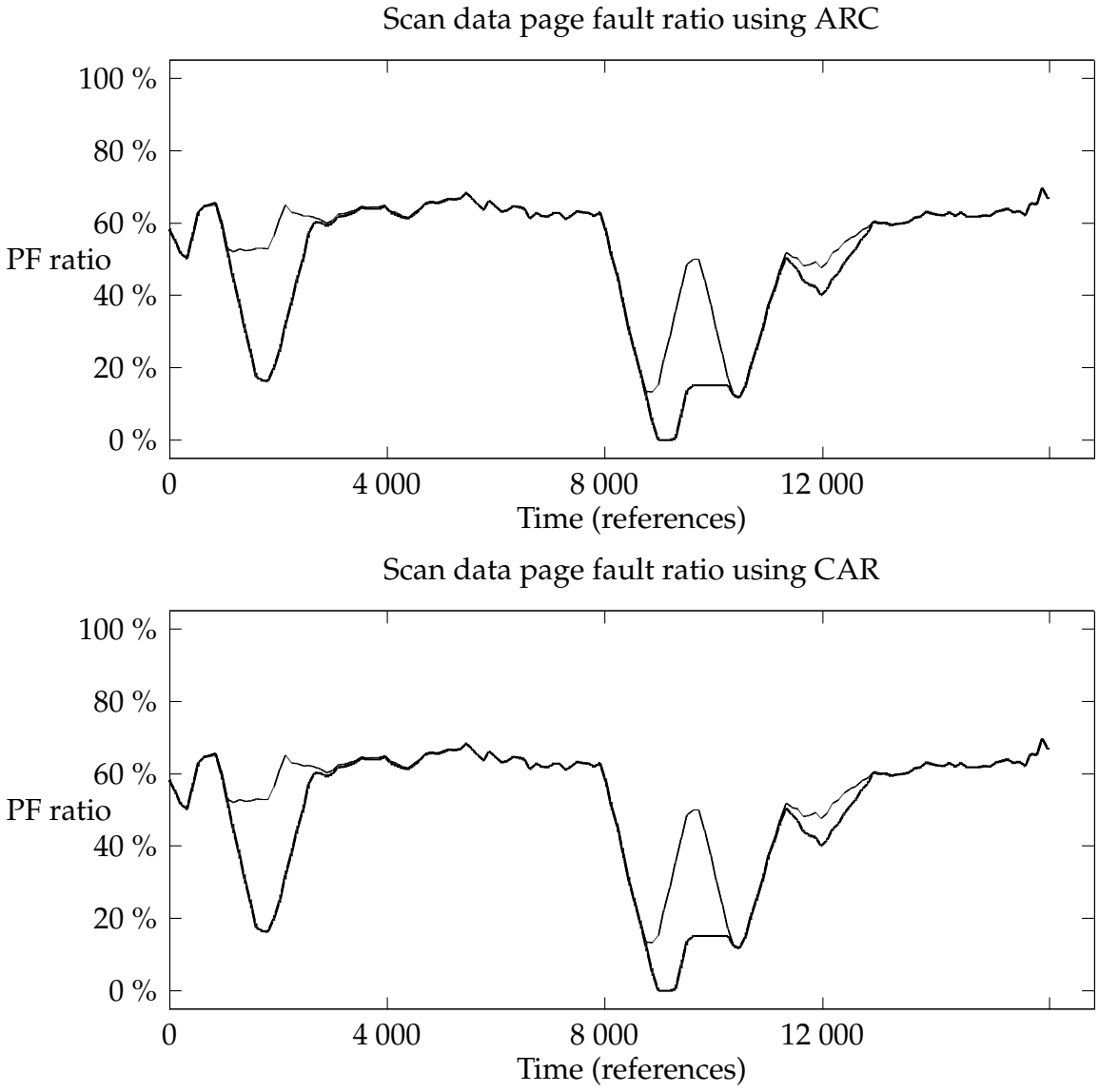

Figure 5.3: Page fault ratio on scan data: ARC and CAR

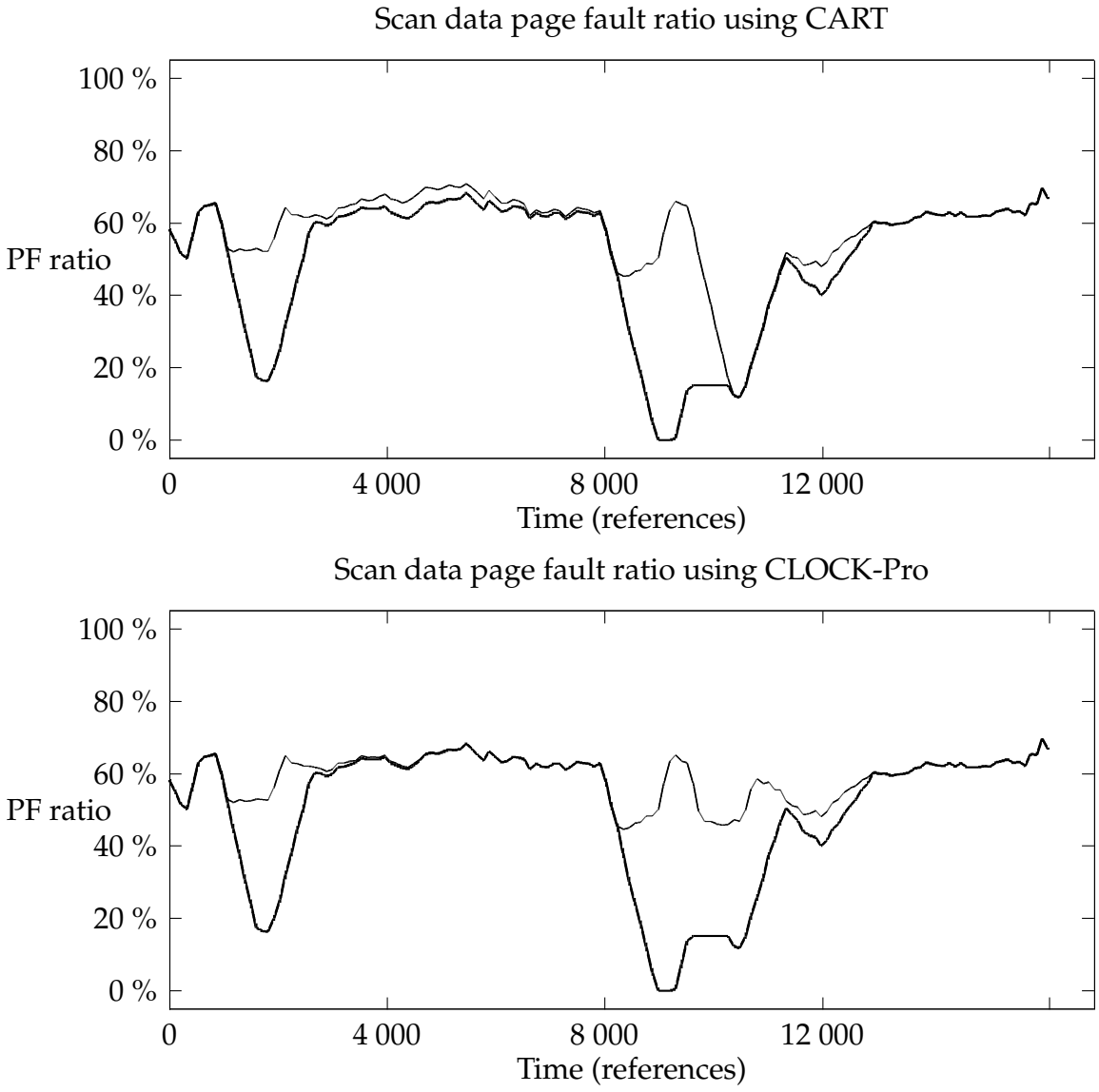

Figure 5.4: Page fault ratio on scan data: CART and CLOCK-Pro

5.3.2 Loop trace

Loop trace (See Appendix C.3) consists of a hot range and five phases. The First, third and fifth phases are loops over the same page range. The second phase is a scan over unused pages. The fourth phase is a random phase with pages from a range that partly overlaps with the loop range.

Algorithm	Ref count	Page count	Page faults	Hit count	Hit ratio
OPT	27795	2678	6711	21084	83.94 %
FIFO	27795	2678	20518	7277	28.97 %
CLOCK	27795	2678	18065	9730	38.74 %
LRU	27795	2678	18863	8932	35.56 %
2Q	27795	2678	16163	11632	46.31 %
ARC	27795	2678	17569	10226	40.71 %
CAR	27795	2678	17056	10739	42.76 %
CART	27795	2678	16265	11530	45.91 %
CLOCKPro	27795	2678	10383	17412	69.32 %

Table 5.2: Results on loop data

Metrics for each algorithm are in Table 5.2. Page fault charts are in Figures 5.5, 5.6, 5.7 and 5.8.

Unlike in scan trace, there is a clear winner in loop trace. CLOCK-Pro has over 30 percent less page faults compared to next best algorithm, 2Q. In the Figures 5.6 and 5.8, we can clearly see, that both 2Q and CLOCK-Pro try to imitate the OPT algorithm. CLOCK-Pro, however, succeeds significantly better, than 2Q, in it. Other algorithms are unable handle the loop pattern efficiently.

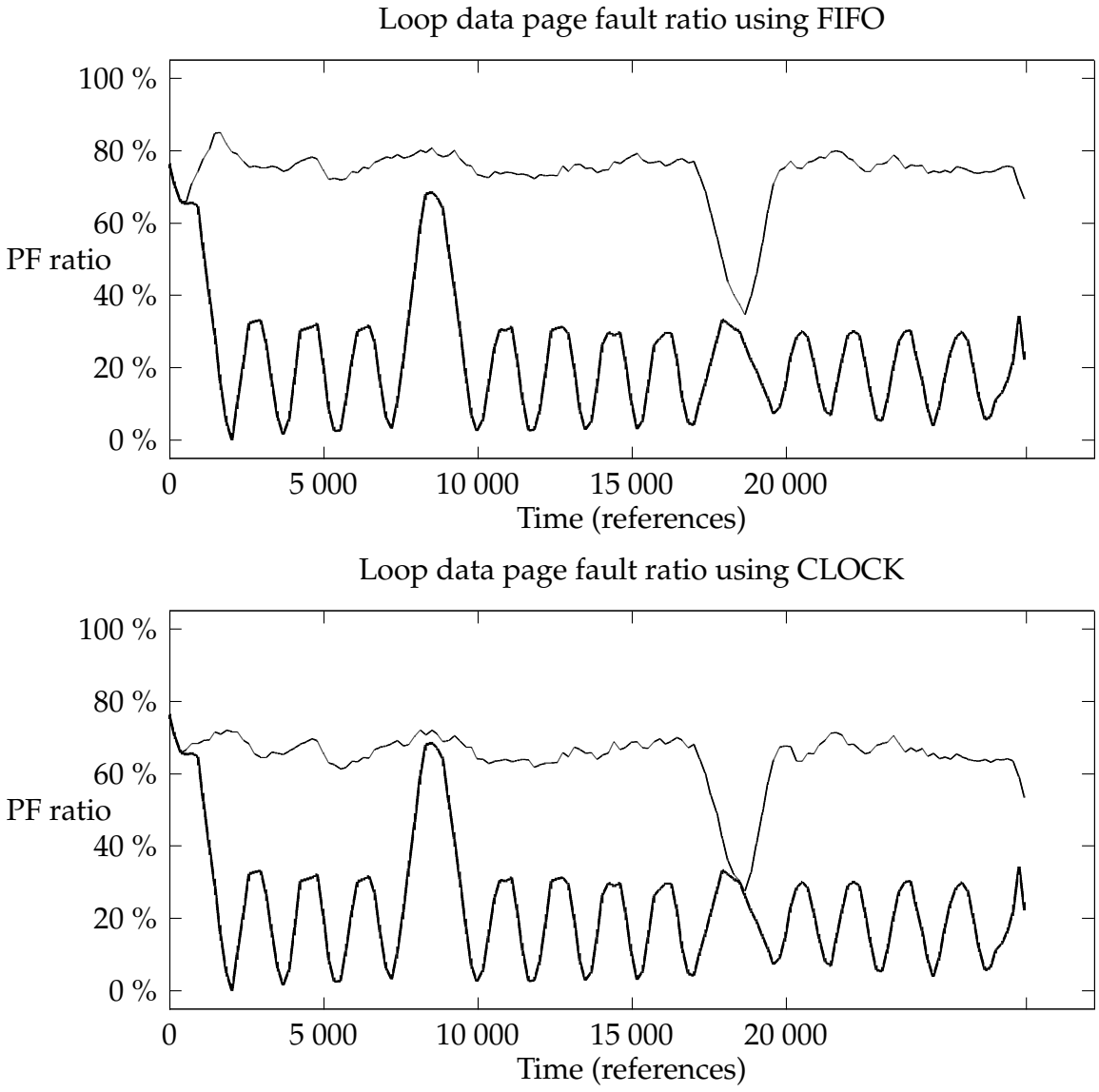

Figure 5.5: Page fault ratio on loop data: FIFO and CLOCK

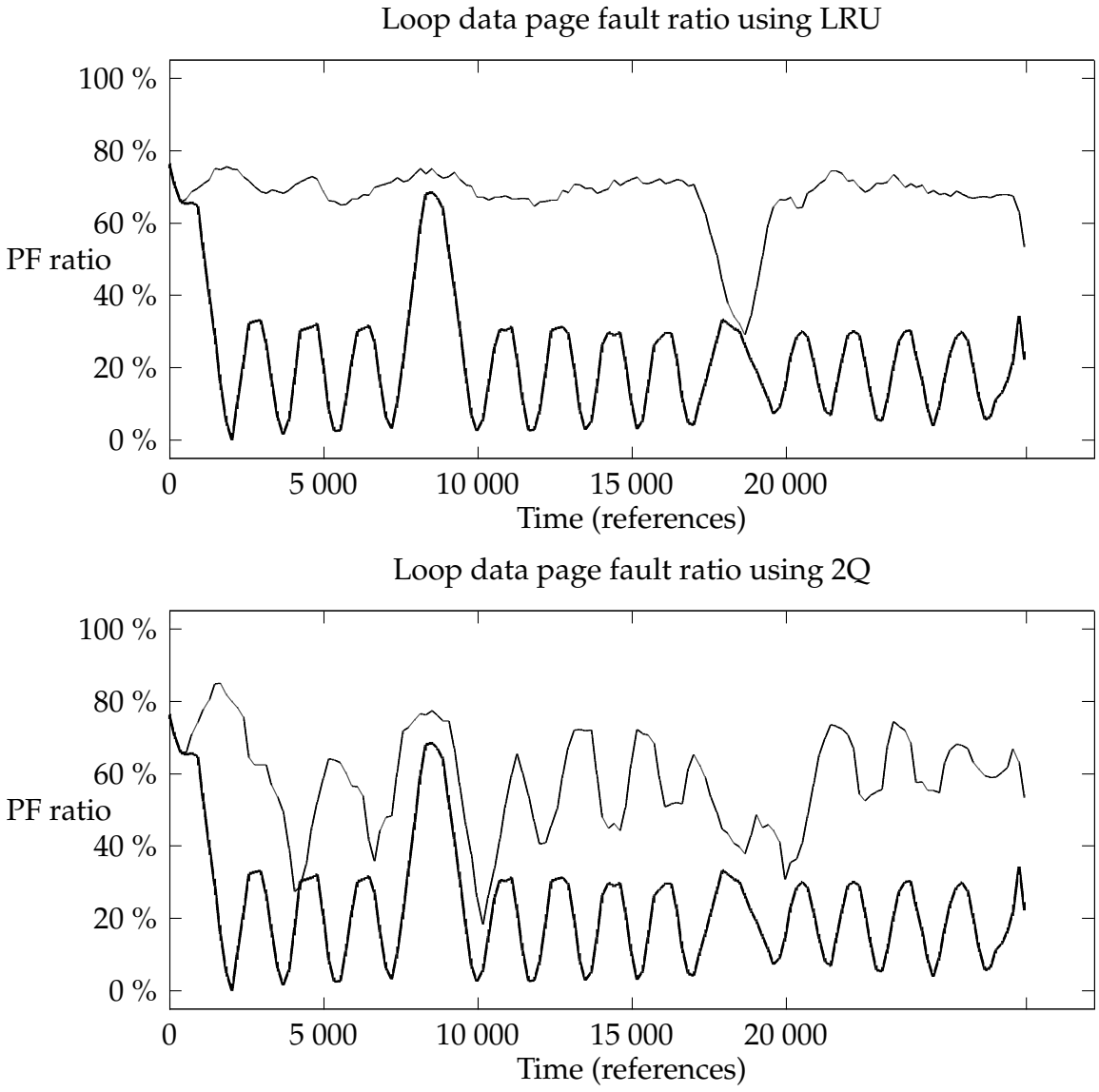

Figure 5.6: Page fault ratio on loop data: LRU and 2Q

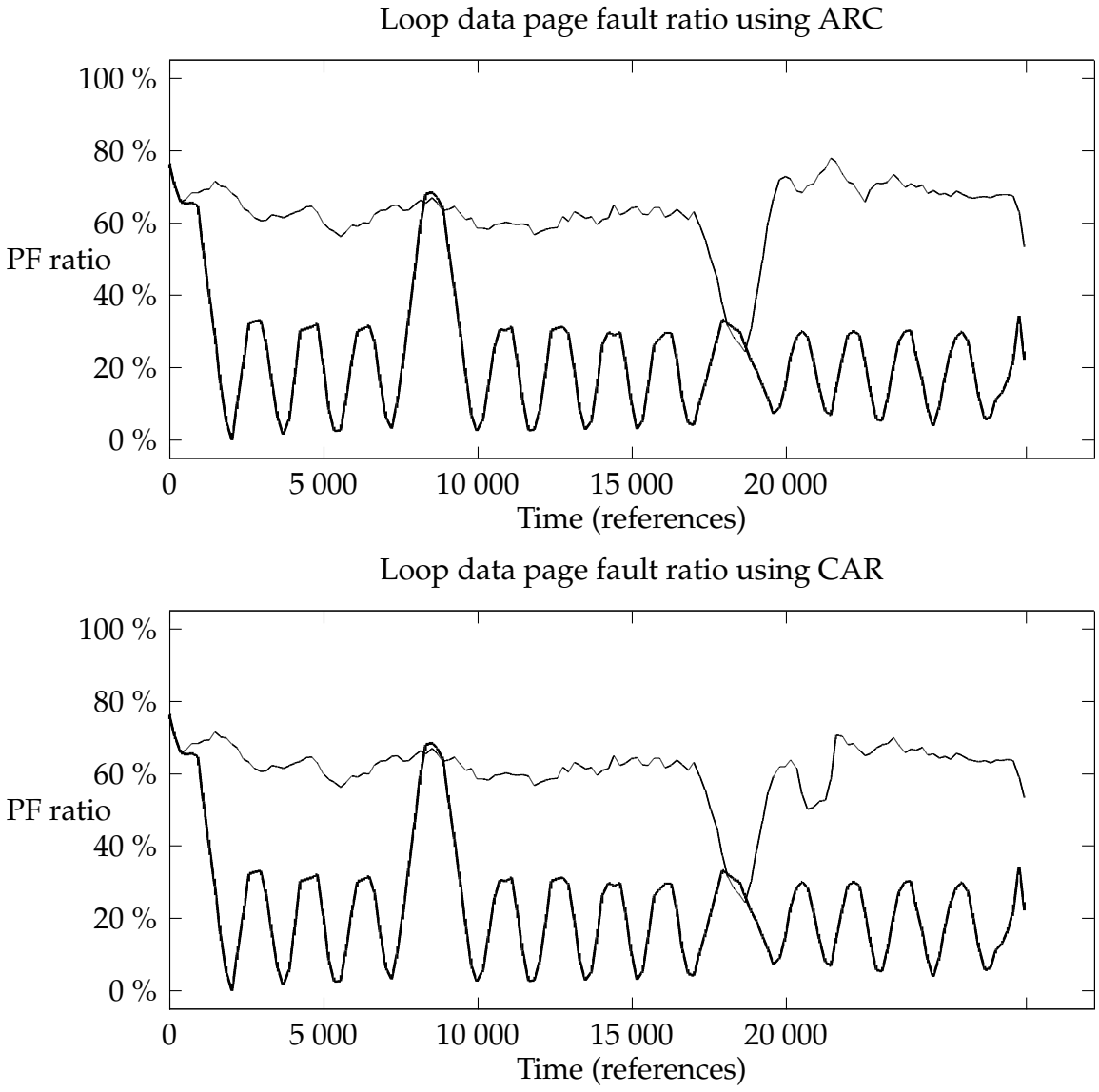

Figure 5.7: Page fault ratio on loop data: ARC and CAR

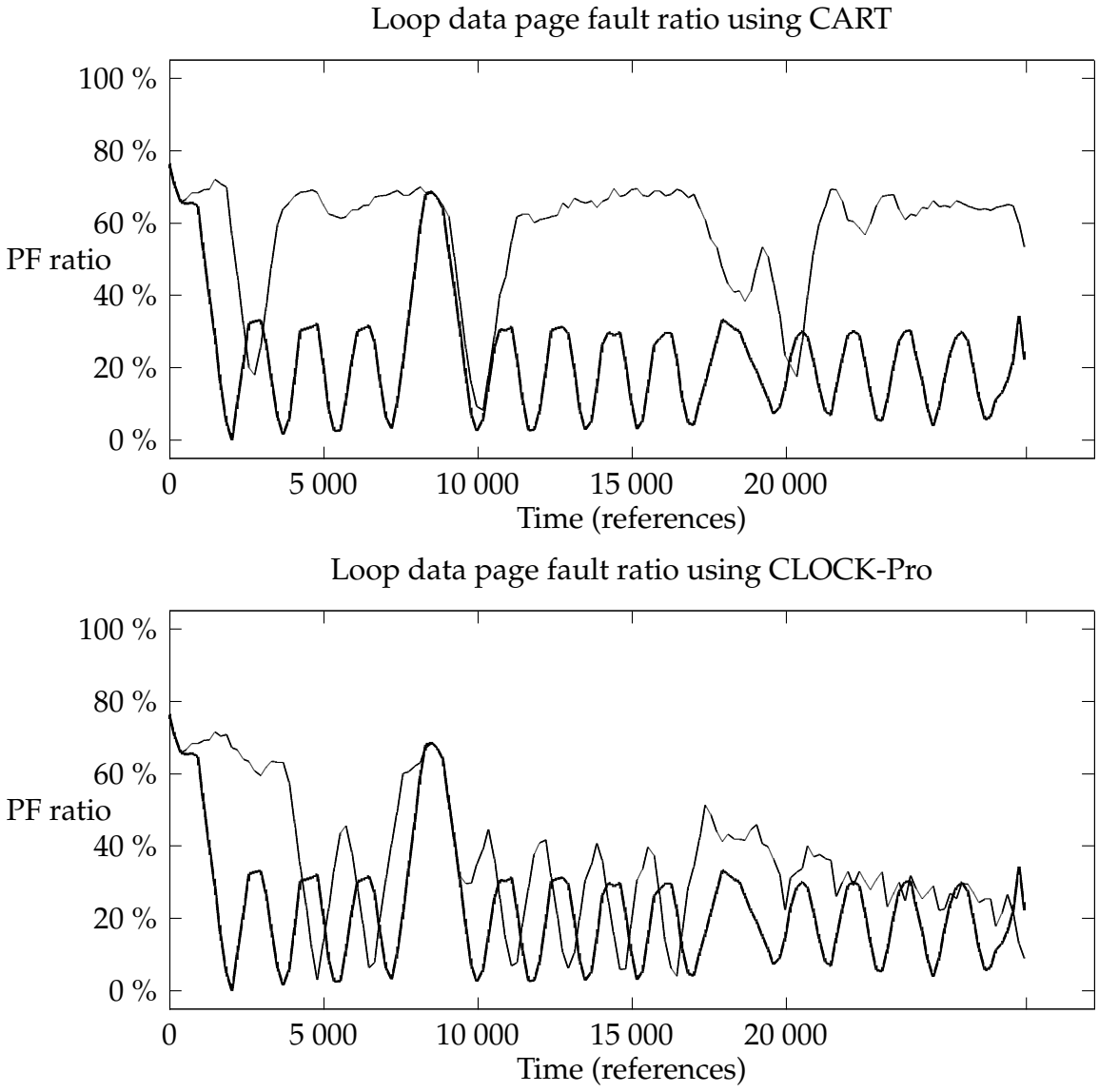

Figure 5.8: Page fault ratio on loop data: CART and CLOCK-Pro

5.3.3 Correlated accesses trace

Correlated accesses trace (See Appendix C.4) consists of page a hot range and five phases. The first, third and fifth phases are scans with correlated accesses over the same page range. The second phase is a plain scan over unused pages. The fourth phase is a random phase with pages from a range that partly overlaps with the pages used in correlated accesses.

Metrics for each algorithm are in Table 5.3. Page fault charts are in Figures 5.9, 5.10, 5.11 and 5.12.

Algorithm	Ref count	Page count	Page faults	Hit count	Hit ratio
OPT	42420	5301	14881	27539	74.19 %
FIFO	42420	5301	20582	21838	58.83 %
CLOCK	42420	5301	20299	22121	59.59 %
LRU	42420	5301	19022	23398	63.04 %
2Q	42420	5301	17505	24915	67.12 %
ARC	42420	5301	18675	23745	63.97 %
CAR	42420	5301	23058	19362	52.16 %
CART	42420	5301	16446	25974	69.97 %
CLOCKPro	42420	5301	19648	22772	61.35 %

Table 5.3: Results on correlated data

Best performers in this trace are 2Q and CART. CLOCK-Pro suffers from adaptation in the correlated access trace phases. The adaptation of CLOCK-Pro (See Chapter 4.15) causes it to behave more like basic CLOCK, which can be confirmed from Figures 5.9 and 5.12. Difference in CAR and CART can be seen in handling of the correlated accesses, in Figures 5.11 and 5.12.

Interesting characteristic is the opposite spikes present in all charts. The spikes are caused by the short scan in second phase. The correlated access scan in the first phase uses the same pages, that are scanned in the second phase. The OPT algorithm is able take this into account and keep them in main memory for the second phase. Because of this, the OPT has a sudden decrease in the page fault ratio.

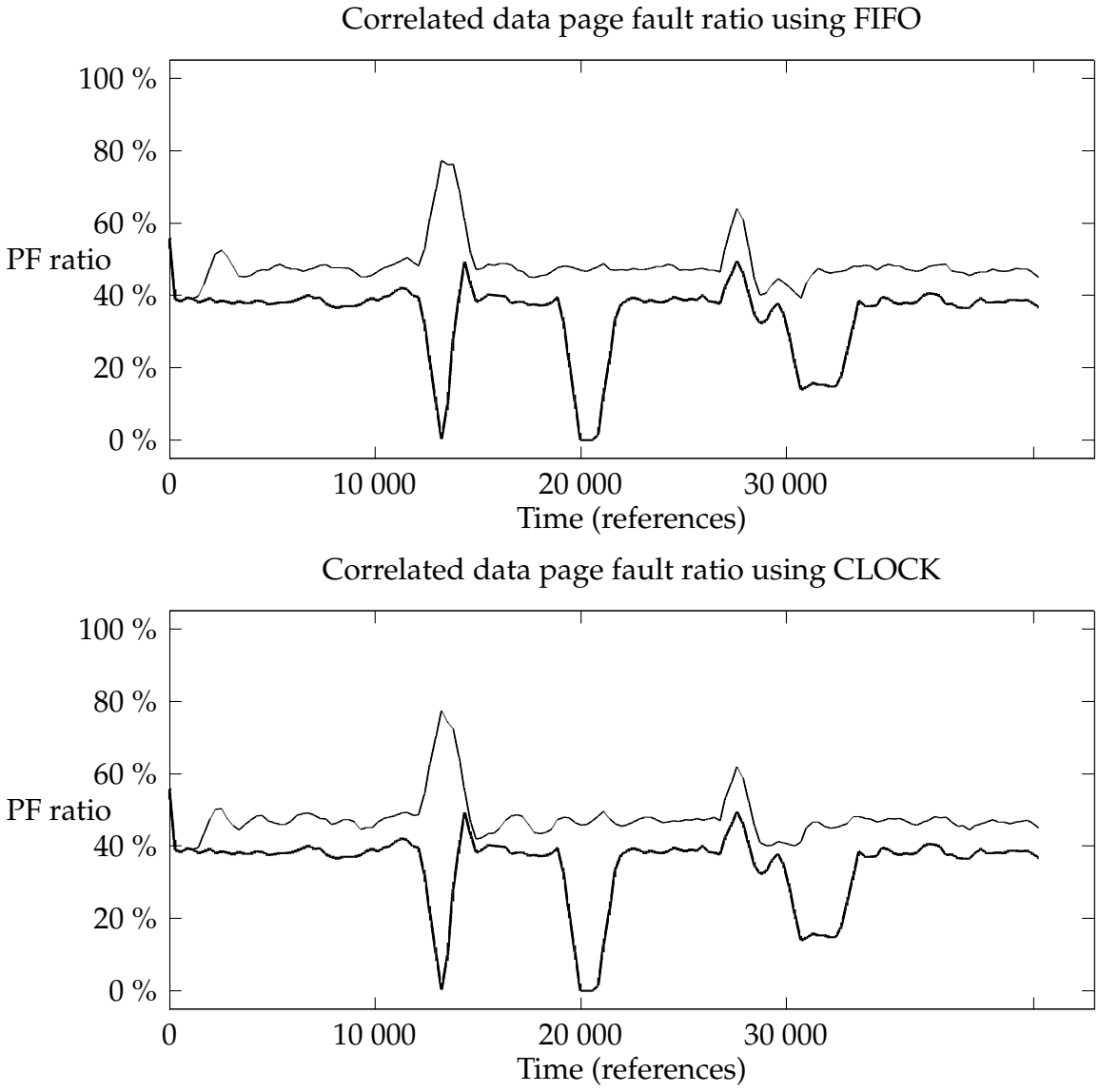

Figure 5.9: Page fault ratio on correlated data: FIFO and CLOCK

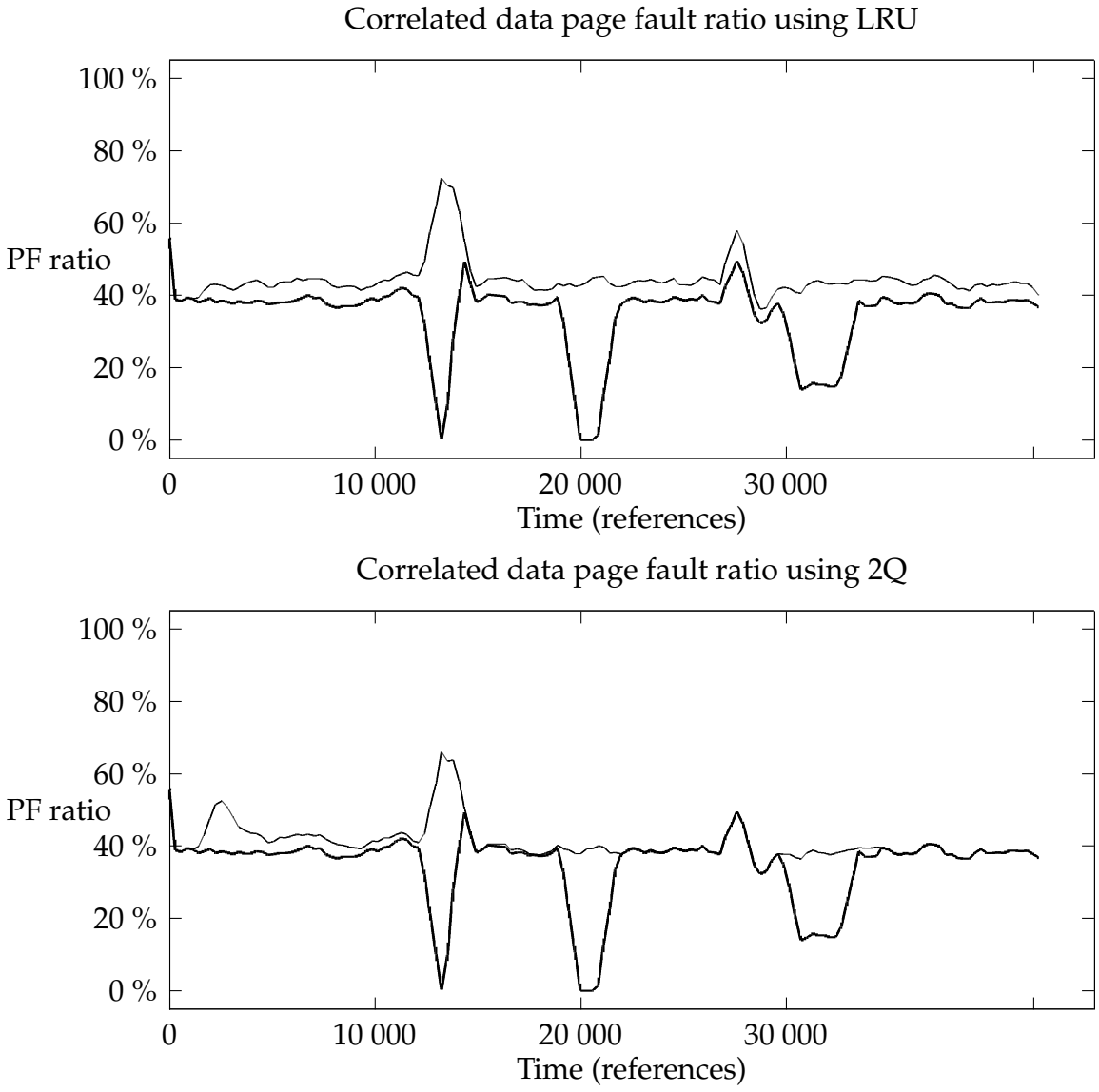

Figure 5.10: Page fault ratio on correlated data: LRU and 2Q

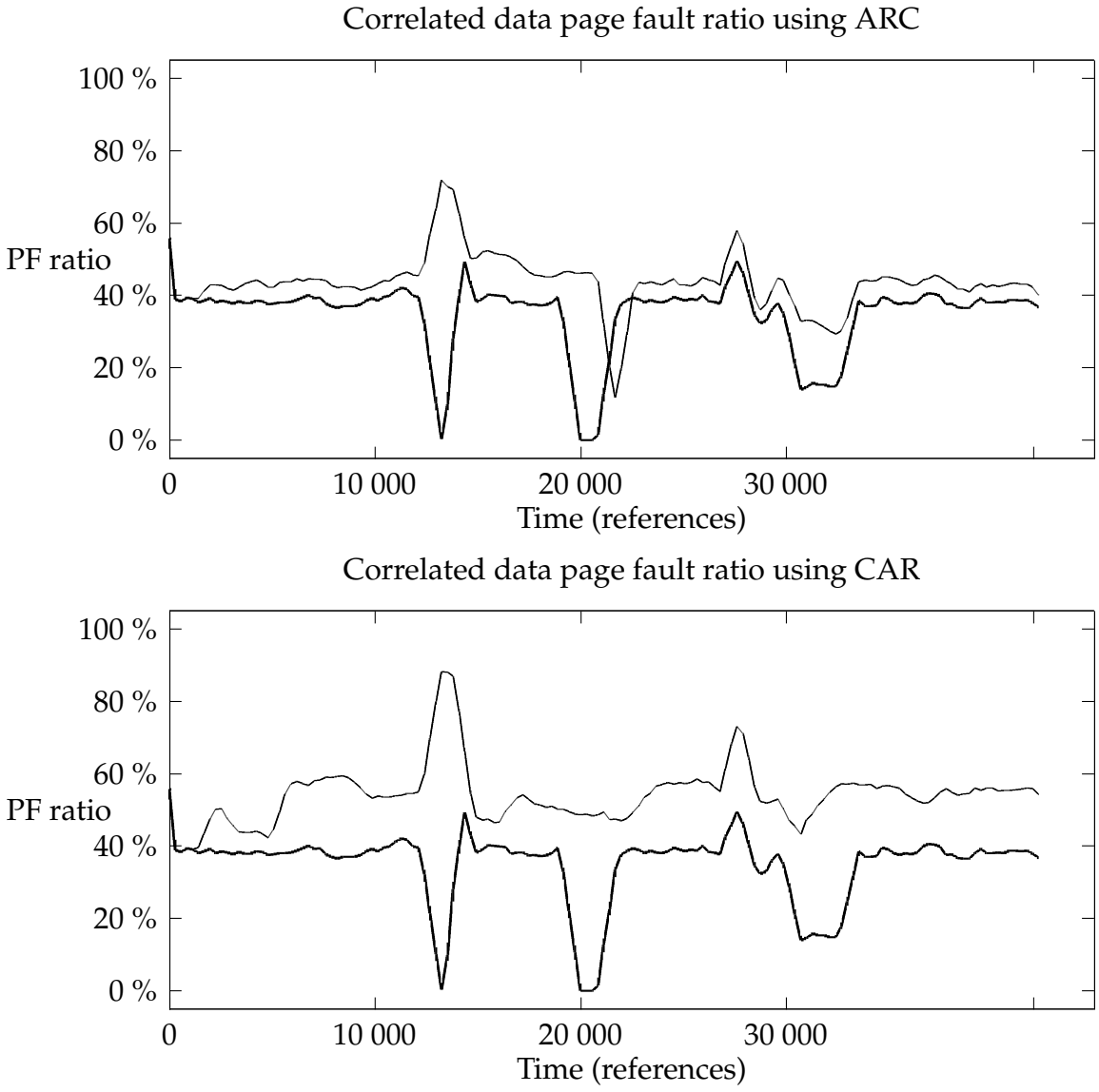

Figure 5.11: Page fault ratio on correlated data: ARC and CAR

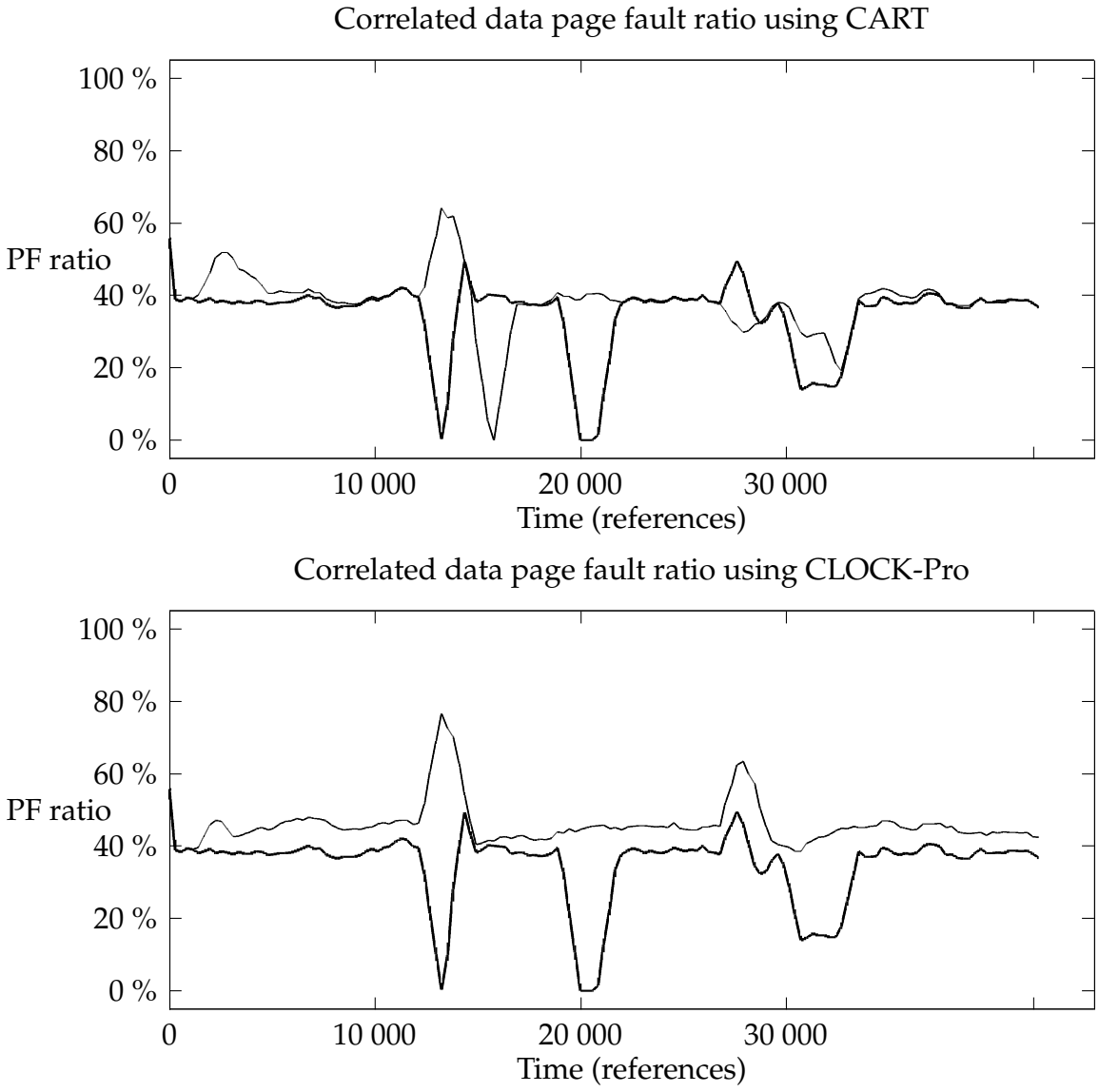

Figure 5.12: Page fault ratio on correlated data: CART and CLOCK-Pro

5.4 Real trace

The real trace presented here is gathered using updated vmtrace patch[1]. The execution was done by running Ubuntu 7.04[2] , with patched Linux kernel 2.6.22.2[3], in Qemu 0.8.2[4].

The vmtrace gathers the reference trace by disabling all page table entries of the process periodically. Disabling is done by clearing the present bit and setting the disabled bit. Clearing the present bit causes a page fault on next reference to the page. Setting of the disabled bit allows the rest of the virtual memory subsystem to handle the disabled page normally.

The executed program was test.py (A.3), which was run using Python 2.5 [5]. The trace was processed with 500 pages of main memory.

Metrics for each algorithm are in Table 5.4. Page fault charts are in Figures 5.13, 5.14, 5.15 and 5.16.

Algorithm	Ref count	Page count	Page faults	Hit count	Hit ratio
OPT	1514886	6229	151062	1363824	90.40 %
FIFO	1514886	6229	352546	1162340	77.04 %
CLOCK	1514886	6229	577159	937727	62.16 %
LRU	1514886	6229	566656	948230	62.85 %
2Q	1514886	6229	369134	1145752	75.95 %
ARC	1514886	6229	555982	958904	63.56 %
CAR	1514886	6229	567009	947877	62.83 %
CART	1514886	6229	569615	945271	62.66 %
CLOCKPro	1514886	6229	266646	1248240	82.74 %

Table 5.4: Results on test.py data

Best performer in this trace is CLOCK-Pro. Surprisingly, FIFO takes the second place, while 2Q is third. CLOCK, LRU, ARC, CAR and CART perform equally well, although the charts are not identical.

[1] http://linux-mm.org/VmTrace
[2] http://ubuntu.com
[3] http://www.kernel.org
[4] http://fabrice.bellard.free.fr/qemu/
[5] http://www.python.org

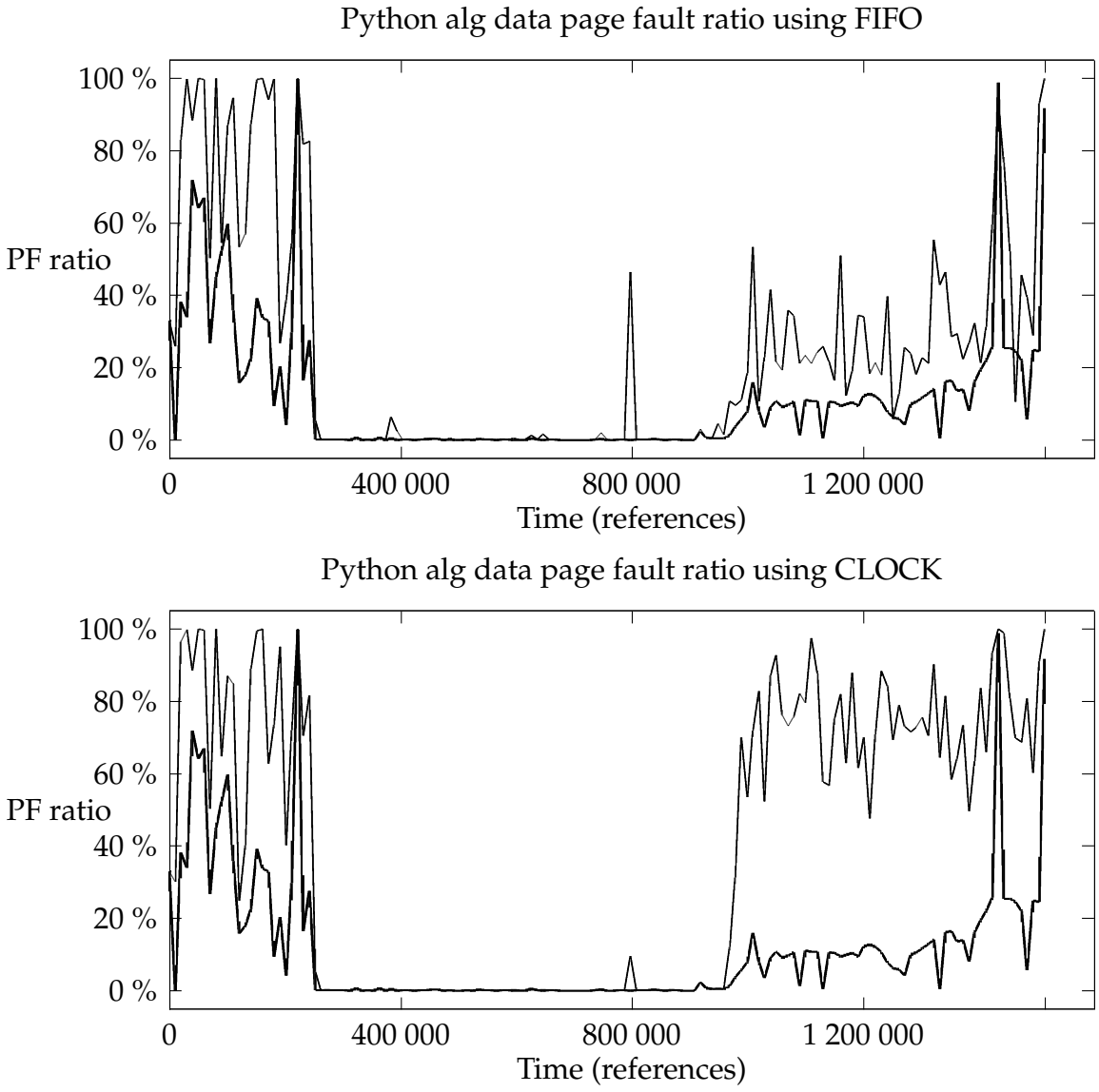

Figure 5.13: Page fault ratio on test.py: FIFO and CLOCK

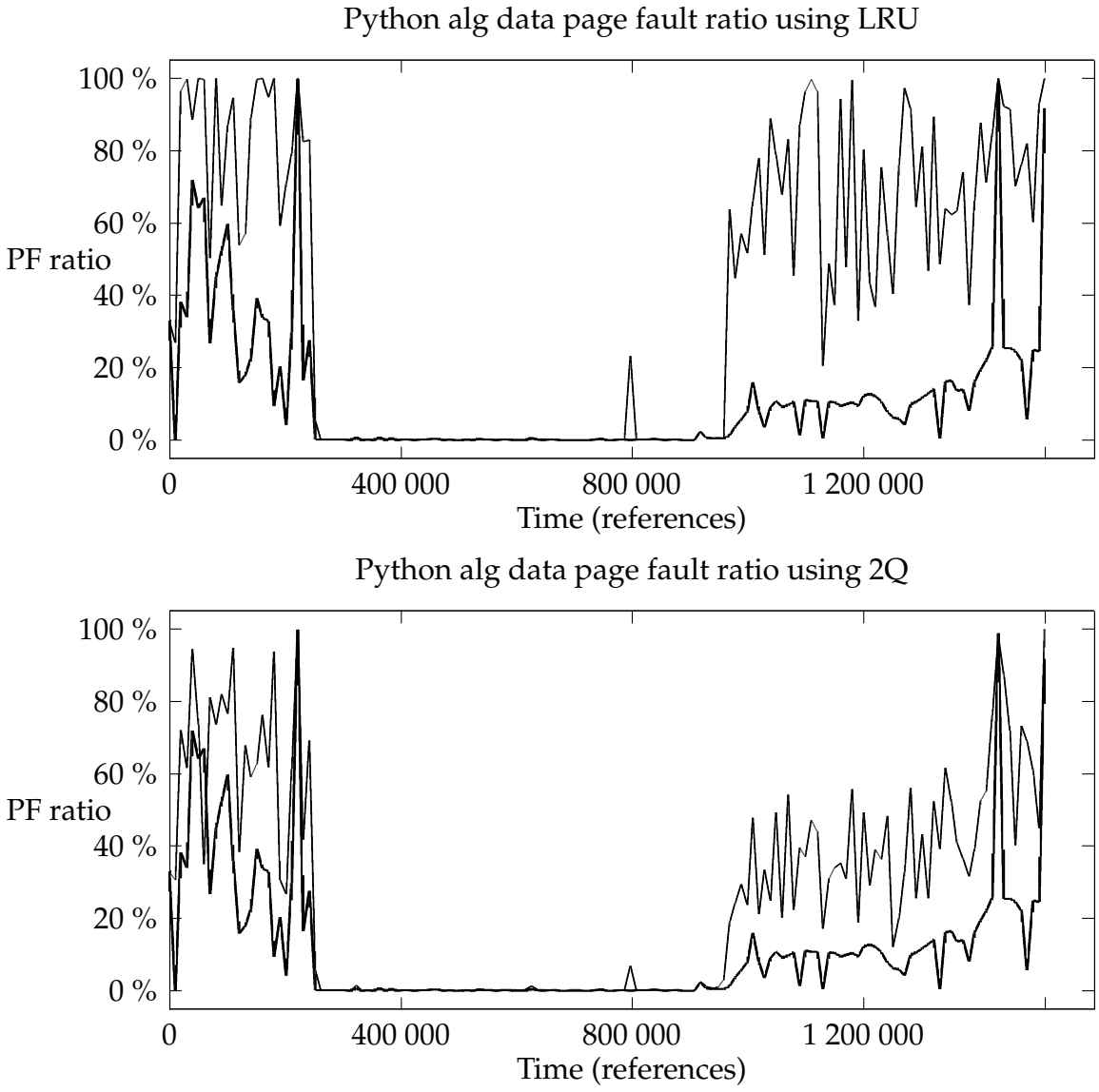

Figure 5.14: Page fault ratio on test.py: LRU and 2Q

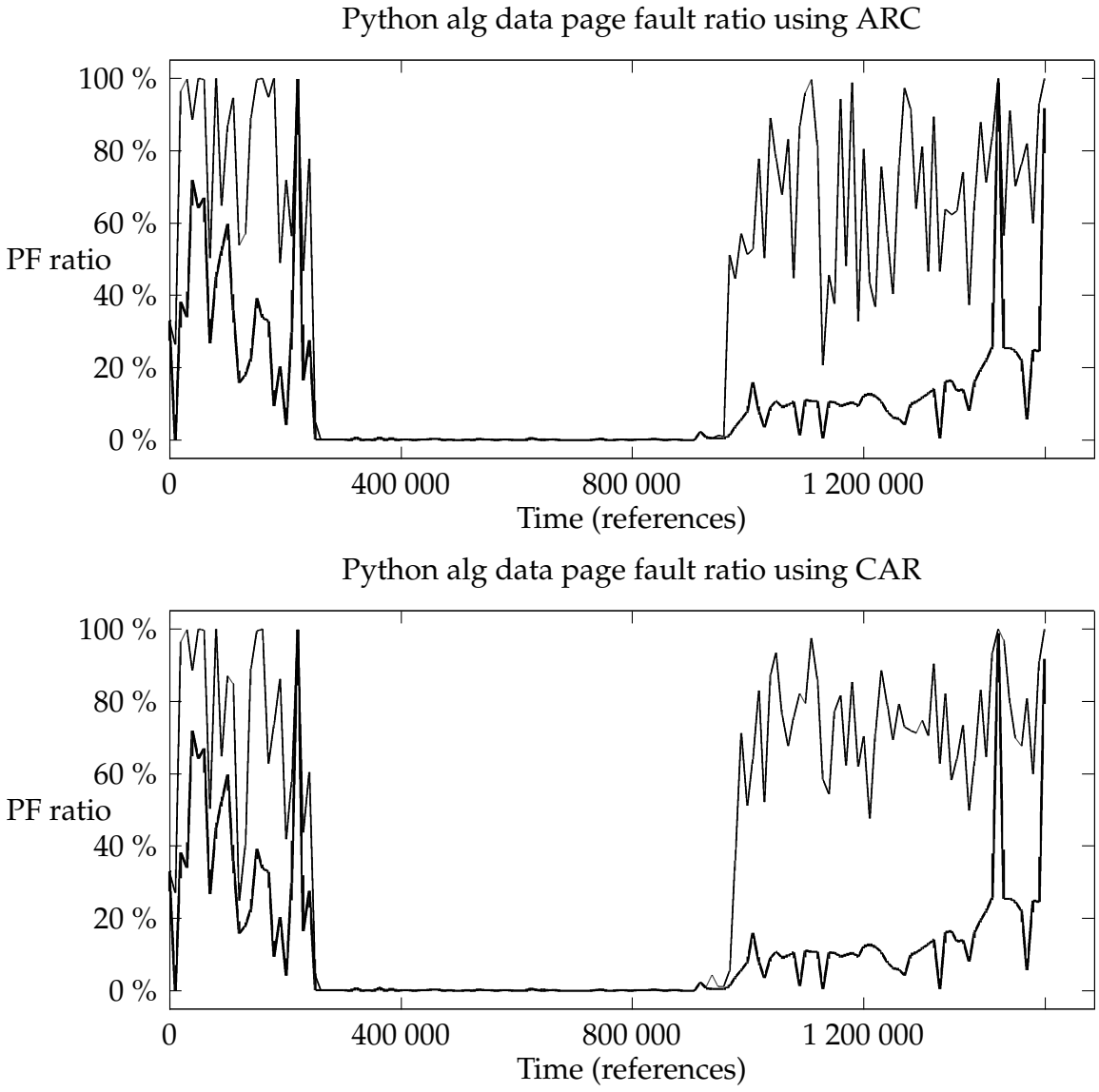

Figure 5.15: Page fault ratio on test.py: ARC and CAR

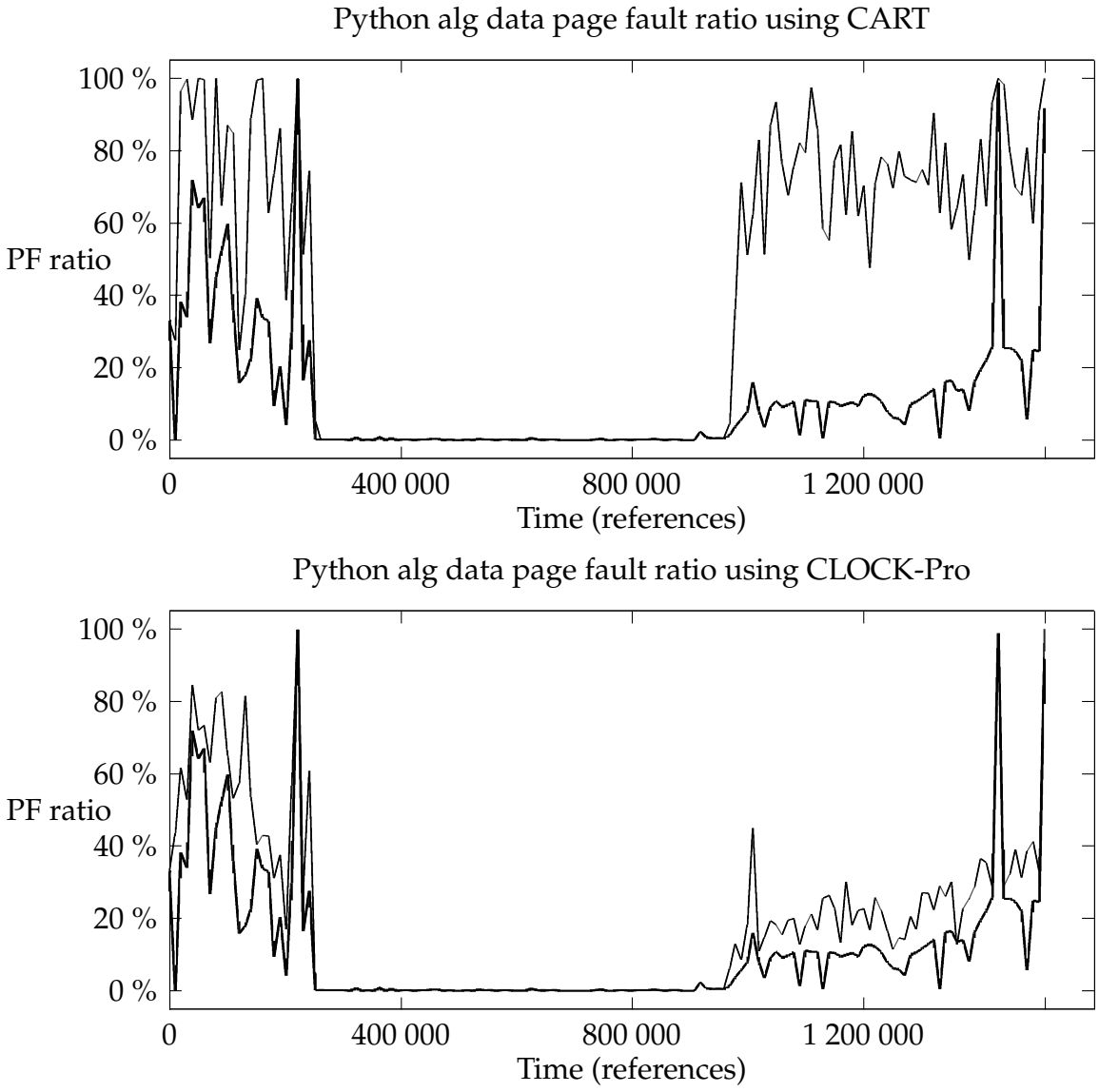

Figure 5.16: Page fault ratio on test.py: CART and CLOCK-Pro

6 Conclusions

The evolution of replacement algorithms shows two clear trends. First, the analyzing and proof of better performance has moved from mathematical analysis to testing against real world program traces, and later even by implementing prototypes on real operating systems. This trend shows how difficult it is to mathematically model the memory behaviour of programs. An important factor is also the large amount and easy availability of important programs. The other clear trend is the realization of the need for workload adaption. As Denning noted in [5], the phase-transition model, a good high level model for program behaviour, shows the importance of the transition phase to the replacement algorithm. This implies that an algorithm that handles the transition phase (i.e. adapts) better will have better performance on real programs.

The simple traces used in this thesis support the conclusions of the authors. CART and CLOCK-Pro seem most promising algorithms and offer significant improvement over basic CLOCK.

Page replacement plays only a small part in overall performance of applications, but studies, like [2] and [9], have shown that the benefits are real. It certainly seems like a worthwhile idea to further evaluate implementations of both CART and CLOCK-Pro in real operating system. Linux development community has picked this up and there are experimental implementations of both CART and CLOCK-Pro available.

7 References

[1] Alfred V. Aho, Peter J. Denning and Jeffrey D. Ullman *Principals of Optimal Page Replacement* Journal of the Association for Computing Machinery, Volume 18, No. 1, January 1971

[2] Sorav Bansal and Dharmendra S. Modha *CAR: Clock with Adaptive Replacement* FAST'04 - 3rd USENIX Conference on File and Storage Technologies, 2004

[3] L. A. Belady, *A study of replacement algorithms for a virtual-storage computer*, IBM Systems Journal, Volume 5, Issue 2, pp. 78–101 (1966).

[4] Peter J. Denning, *The working set model for program behavior*, Communications of the ACM, Volume 11 , Issue 5, pp. 323–333 (May 1968).

[5] Peter J. Denning, *Working Sets Past and Present*, IEEE Transactions on Software Engineering, Volume 6 , Number 1, pp. 64–84 (January 1980).

[6] Peter J. Denning, *The Locality Principle*, In Communication Networks and Computer Systems (J. Barria, Ed.) , Imperial College Press, pp. 43–67 (2006).

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

[8] Mel Gorman, *Understanding the Linux Virtual Memory Manager*, Bruce Perens' Open Source Series, Prentice Hall, 2004.

[9] Song Jiang, Feng Chen and Xiaodong Zhang, *CLOCK-Pro: An Effective Improvement of the CLOCK Replacement*, USENIX Annual Technical Conference, 2005.

[10] Song Jiang and Xiaodong 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.

[11] Song Jiang and Xiaodong Zhang, *Token-ordered LRU: an effective page replacement policy and its implementation in Linux systems*, Performance Evaluation 60 5–29, 2005.

[12] Theodore Johnson and Dennis Shasha. *2q: a low overhead high performance buffer management replacement algorithm* In Proceedings of the Twentieth International Conference on very Large Databases, pp. 439-450, Santiago, Chile, 1994.

[13] Nimrod Megiddo and Dharmendra S. Modha *ARC: A Self-tuning, Low Overhead Replacement Cache* USENIX File and Storage Technologies Conference (FAST), San Francisco, CA, 2003

[14] Elizabeth J. O'Neil, Patrick E. O'Neil and Gerhard Weikum, *The LRU-K Page Replacement Algorithm For Database Disk Buffering* Proceedings of ACM SIGMOD Conference, pp. 297–306, 1993

[15] C. Northcote Parkinson, *Parkinson's Law* The Economist, November 1955.

[16] Andrew S. Tanenbaum and Albert S. Woodhull, *Operating Systems: Design and Implementation*, Third Edition, Prentice Hall, 2006.

A Utilities

A.1 utils.py

```
"""Utility classes
"""

class Algorithm(object):
    """Abstract algorithm base class
    """

    def __init__(self, name, memory):
        self.name = name
        self.memory = memory
        self.debug_on = False

    def debug(self, obj):
        """Print debug if debug is enabled
        """
        if self.debug_on:
            print obj

    def process(self, ref):
        """Process next reference
        """
        raise Exception("Implement me...")

class Reference(object):
    """Reference
    """
    def __init__(self, page_id, timestamp):
        self.page_id = page_id
        self.timestamp = timestamp

    def __repr__(self):
        return '(page: %(page_id)d, ts: %(timestamp)d)' \
            % vars(self)

class Page(object):
```

```
        """Page
        """
        def __init__(self, page_id):
            self.page_id = page_id

            self.referenced = False

        def __repr__(self):
            return '(page: % d, r: %d)' % (self.page_id,
                                           self.referenced)

    class ClockPage(Page):
        """Page in Clock
        """

        def __init__(self, page_id):
            Page.__init__(self, page_id)

            self.next = None
            self.prev = None

    class LRUList(object):
        """LRU list
        """

        def __init__(self, name, debug=False):
            self.page_to_ts = {}
            self.ts_to_page = {}
            self.debug_on = debug
            self.name = name

        def debug(self, debug_string):
            """Print debug if debug is enabled
            """
            if self.debug_on:
                print debug_string

        def size(self):
            """Return current size of LRUList
            """
            return len(self.page_to_ts)
```

```
        def has_page(self, ref):
            """Returns true if page referenced by given ref is in LRUList
80          """
            return self.page_to_ts.has_key(ref.page_id)

        def remove_page(self, ref):
            """Remove page referenced by given ref from LRUList
85          """
            timestamp = self.page_to_ts[ref.page_id]
            self.debug('(%s) Removing page %d with ts %d' % (self.name,
                                                             ref.page_id,
                                                             timestamp))
90          del self.page_to_ts[ref.page_id]
            del self.ts_to_page[timestamp]

        def remove(self):
            """Remove LRU page from LRUList
95          """
            keys = self.ts_to_page.keys()
            keys.sort()
            timestamp = keys[0]
            page_id = self.ts_to_page[timestamp]
100         self.debug('(%s) Removing page %d with ts %d' % (self.name,
                                                             page_id,
                                                             timestamp))
            del self.ts_to_page[timestamp]
            del self.page_to_ts[page_id]
105         return page_id

        def update(self, ref):
            """Update referenced timestamp of page referenced by given ref
            """
110         if self.page_to_ts.has_key(ref.page_id):
                del self.ts_to_page[self.page_to_ts[ref.page_id]]

            self.ts_to_page[ref.timestamp] = ref.page_id
            self.page_to_ts[ref.page_id] = ref.timestamp
115
```

A.2 memory_state.py

```
"""Presentation of current memory state
"""

class MemoryState(object):
    """Presentation of current memory state
    """

    NO_PAGE_FAULT = 0
    PF_NEW = -1
    PF_OLD = 1

    def __init__(self, size):
        self.size = size
        self.page_fault_count = 0
        self.page_faults = []
        self.page_map = {}
        self.all_pages_map = {}
        self.last_ref = None

    def pages_used(self):
        """Return number of page frames used
        """
        return len(self.page_map)

    def has_free_page_frames(self):
        """Returns true, if there are free page frames
        false, otherwise
        """
        return self.size - len(self.page_map) > 0

    def reference(self, ref):
        """Returns
        true, if page is found in memory,
        false, otherwise
        """
        self.last_ref = ref
        status = self.page_map.has_key(ref.page_id)

        if status:
            page = self.page_map[ref.page_id]
            page.referenced = True
```

```
                self.page_faults.append(MemoryState.NO_PAGE_FAULT)
            else:
                self.page_fault_count += 1
                if self.all_pages_map.has_key(ref.page_id):
                    self.page_faults.append(MemoryState.PF_OLD)
                else:
                    self.page_faults.append(MemoryState.PF_NEW)

        return status

    def get_page(self, page_id):
        """Return page with given page_id
        """
        return self.page_map[page_id]

    def is_resident(self, page_id):
        """Return true, if page with given page_id is resident
        """
        return self.page_map.has_key(page_id)

    def insert(self, page):
        """Insert page referenced by given ref to free page
        frame
        """
        if len(self.page_map) == self.size:
            raise Exception("BUG! No free page frames!")
        else:
            self.page_map[page.page_id] = page
            self.all_pages_map[page.page_id] = page

    def evict(self, page_id):
        """Evict page with given page_id from memory
        """
        del self.page_map[page_id]
```

A.3 test.py

```
""" Run tests with given parameters.
"""
import sys
```

```
from memory_state import MemoryState
from utils import Reference

from opt import OPT
from fifo import FIFO
from clock import CLOCK
from lru import LRU
from twoqueue import TwoQueue
from arc import ARC
from car import CAR
from cart import CART
from clockpro import ClockPro

def output_page_fault_ratio(outputfile, memory):
    """Print page fault ratio data of memory to outputfile
    """
    output = open(outputfile,"w")
    pfs = memory.page_faults
    length = memory.size
    pfs_len = len(pfs)
    if pfs_len < 150:
        plen = pfs_len
    else:
        plen = pfs_len / 150
    for ind in xrange(0, pfs_len, plen):
        tmp = pfs[ind:(ind + length)]
        tmp_len = len(tmp)
        count = tmp_len - tmp.count(0)
        percentage = 100*(count / float(tmp_len))
        output.write("%d %f\n" %(ind, percentage))

    output.close()

def check_memory(memory):
    """Check memory status
    """
    if not memory.page_map.has_key(memory.last_ref.page_id):
        raise Exception("BUG! Page referenced is not in memory!")

def print_state(memory):
    """Return string describing memory state
```

```
        """
        state = ""
        distinct_pages = len(memory.all_pages_map)
        state += '\nDistinct pages used: %d' % distinct_pages
        refs = len(memory.page_faults)
        state += '\nReferences: %d' % refs
        pf_no_first = memory.page_fault_count - distinct_pages
        state += '\nPage faults(no first accesses): %d' % (pf_no_first)
        state += '\nHit ratio percentage: %f %%' \
                 % (100 - (100*(float(pf_no_first) / refs)))
        return state

    def print_table_row(alg):
        """Return string describing memory state as latex row to
        outputfile
        """
        memory = alg.memory
        name = alg.name
        distinct_pages = len(memory.all_pages_map)
        pf_no_first = memory.page_fault_count - distinct_pages
        refs = len(memory.page_faults)
        hits = refs - memory.page_fault_count
        refs_no_first = (refs - distinct_pages)
        hit_ratio = (100*(float(pf_no_first) / refs_no_first))
        hit_ratio = round((100 - hit_ratio), 2)
        return "%s & %d & %d & %d & %d & %.2f \\%%\\\\\n" \
               %(name,
                 refs,
                 distinct_pages,
                 memory.page_fault_count,
                 hits, hit_ratio)

    class TestRunner(object):
        """Class for running tests
        """
        def __init__(self, argv):
            self.debug_on = False
            self.datafile = argv[1]
            self.refs = self.read_refs()

            self.memory_size = int(argv[2])
```

```
    def debug(self, obj):
        """Print debug if debug is enabled
        """
        if self.debug_on:
            print obj

    def process_refs(self, alg):
        """Process self.refs with given alg
        """
        for ref in self.refs:
            alg.process(ref)
            check_memory(alg.memory)

    def read_refs(self):
        """Read references from file self.datafile
        """
        data = open(self.datafile)
        refs = []
        timestamp = 0
        for line in data:
            page = line
            ref = Reference(int(page), timestamp)
            refs.append(ref)
            timestamp += 1
        data.close()
        return refs

    def run_algorithm(self, alg):
        """Run alg with refs
        """
        print '----------------------------------------------'
        print '%s:\n' % alg.name
        self.process_refs(alg)
        print print_state(alg.memory)
        import os.path
        outputfile = "%s/%s-%s" % (os.path.dirname(self.datafile)
                                   ,alg.name,
                                     os.path.basename(self.datafile))
        output_page_fault_ratio(outputfile, alg.memory)
        return print_table_row(alg)

    def run_tests(self):
```

```
        """Run tests
        """
        algorithms = [OPT(MemoryState(self.memory_size),
                          self.refs),
                      FIFO(MemoryState(self.memory_size)),
                      CLOCK(MemoryState(self.memory_size)),
                      LRU(MemoryState(self.memory_size)),
                      TwoQueue(MemoryState(self.memory_size)),
                      ARC(MemoryState(self.memory_size)),
                      CAR(MemoryState(self.memory_size)),
                      CART(MemoryState(self.memory_size)),
                      ClockPro(MemoryState(self.memory_size))]

        table = ""
        for alg in algorithms:
            table += self.run_algorithm(alg)
        import os.path
        outputfile = "%s/table-%s.tex" \
                     % (os.path.dirname(self.datafile)
                        ,os.path.basename(self.datafile))
        output = open(outputfile, "w")
        output.write("\\begin{tabular}{|l|r|r|r|r|r|}\n")
        output.write("\\hline\n")
        output.write("Algorithm & Ref count & Page count & " +\
                     "Page faults & Hit count & Hit ratio\\\\\n")
        output.write("\\hline\n")
        output.write(table)
        output.write("\\hline\n")
        output.write("\\end{tabular}\n")
        output.close()

if __name__ == "__main__":
    try:
        import psyco
        psyco.full()
    except ImportError:
        print "Warning! Running without psyco!"
    TestRunner(sys.argv).run_tests()
```

B Algorithms

B.1 opt.py

```
"""Implements OPT (=optimal) algorithm
"""
from utils import Page, Algorithm

class OPT(Algorithm):
    """Implements OPT (=optimal) algorithm
    """
    def __init__(self, memory, refs):
        Algorithm.__init__(self, "OPT", memory)

        self.refs = refs
        self.refs_count = len(refs)
        self.index = 0

        self.next_ref_cache = {}

    def search_next_ref(self, page_id):
        """Search next reference to give page
        """
        for ind in xrange(self.index+1, self.refs_count):
            if page_id == self.refs[ind].page_id:
                return self.refs[ind].timestamp
        return None


    def evict(self, ref):
        """Evict page that is unused for longest
        """
        pages = []
        for page_id in self.memory.page_map.keys():
            next_ref = -1
            if self.next_ref_cache.has_key(page_id):
                next_ref = self.next_ref_cache[page_id]
            if next_ref < ref.timestamp:
                next_ref = self.search_next_ref(page_id)
```

```
                    self.next_ref_cache[page_id] = next_ref
                if next_ref is None:
                    # No references ever again
                    evict_page_id = page_id
                    break
                else:
                    pages.append((next_ref, page_id))

            if next_ref is not None:
                pages.sort()
                pages.reverse()
                (next_ref, evict_page_id) = pages[0]
            del self.next_ref_cache[evict_page_id]
            self.memory.evict(evict_page_id)


        def process(self, ref):
            """Process a reference
            """
            if not self.memory.reference(ref):
                if not self.memory.has_free_page_frames():
                    self.evict(ref)
                page = Page(ref.page_id)
                self.memory.insert( page )

            self.index += 1
```

B.2 fifo.py

```
"""Implements First-In, First-Out algorithm
"""
from utils import Page, Algorithm

class FIFO(Algorithm):
    """Implements First-In, First-Out algorithm
    """
    def __init__(self, memory):
        Algorithm.__init__(self, "FIFO", memory)

        self.fifo = []
```

```
        def process(self, ref):
            """Process a reference
            """
            if not self.memory.reference(ref):
                if not self.memory.has_free_page_frames():
                    self.memory.evict(self.fifo.pop(0).page_id)

                page = Page(ref.page_id)
                self.memory.insert( page )
                self.fifo.append(page)
```

B.3 clock.py

```
"""Implement CLOCK algorithm
"""
from utils import ClockPage, Algorithm

class CLOCK(Algorithm):
    """Implement CLOCK algorithm
    """

    def __init__(self, memory):
        Algorithm.__init__(self, "CLOCK", memory)
        self.clock = None

    def insert_to_head(self, page, tail):
        """Insert given page to head of the clock
        """
        tail.prev.next = page
        page.prev = tail.prev
        tail.prev = page
        page.next = tail

    def remove_from_clock(self, page):
        """Remove given page from clock.
        """
        page.prev.next = page.next
        page.next.prev = page.prev

    def process(self, ref):
        """Process a reference
```

```
        """
        if not self.memory.reference(ref):
            if not self.memory.has_free_page_frames():
                while self.clock.referenced:
                    self.clock.referenced = False
                    self.clock = self.clock.next

                page = self.clock
                self.clock = self.clock.next
                self.remove_from_clock(page)
                self.memory.evict(page.page_id)

            page = ClockPage(ref.page_id)
            self.memory.insert(page)
            if self.clock is None:
                page.next = page
                page.prev = page
                self.clock = page
            else:
                self.insert_to_head(page, self.clock)


```

B.4 lru.py

```
"""Implement LRU algorithm
"""
from utils import LRUList, Page, Algorithm

class LRU(Algorithm):
    """Implement LRU algorithm
    """

    def __init__(self, memory):
        Algorithm.__init__(self, "LRU", memory)
        self.lru_list = LRUList("LRU")

    def process(self, ref):
        """ Process a reference
        """
        if not self.memory.reference(ref):
            if not self.memory.has_free_page_frames():
```

```
                    page_id = self.lru_list.remove()
                    self.memory.evict(page_id)

                page = Page(ref.page_id)
                self.memory.insert(page)


            self.lru_list.update(ref)

            self.debug(self.lru_list.page_to_ts)
            self.debug(self.lru_list.ts_to_page)
```

B.5 twoqueue.py

```
"""Implements 2Q algorithm
"""
from utils import LRUList, Page, Algorithm

class TwoQueue(Algorithm):
    """Implements 2Q algorithm
    """

    def __init__(self, memory):
        Algorithm.__init__(self, "2Q", memory)

        #self.debug_on = True

        self.hot = LRUList("Hot")
        self.fin = []
        self.fout = []
        self.fin_limit = int(memory.size / 4)
        self.fout_limit = int(memory.size / 2)


    def process(self, ref):
        """Process a reference
        """
        if not self.memory.reference(ref):
            if not self.memory.has_free_page_frames():
                self.reclaim()

            if ref.page_id in self.fout:
```

```
                    self.debug('Insert to hot: %d' % ref.page_id)
                    self.hot.update(ref)
                else:
                    self.debug('Insert to fin: %d' % ref.page_id)
                    self.fin.append(ref.page_id)

                page = Page(ref.page_id)
                self.memory.insert(page)
            elif self.hot.has_page(ref):
                self.debug('Update in hot: %d' % ref.page_id)
                self.hot.update(ref)
            return


        def reclaim(self):
            """Reclaim a page
            """
            if len(self.fin) > self.fin_limit:
                page = self.fin.pop(0)
                self.debug('Reclaim from fin: %d' % page)
                if len(self.fout) > self.fout_limit:
                    self.debug('Remove from fout: %d' % page)
                    self.fout.pop(0)
                self.fout.append(page)
            else:
                page = self.hot.remove()
                self.debug('Reclaim from hot: %d' % page)

            self.memory.evict(page)
```

B.6 arc.py

```
"""
Implements ARC replacement algorithm
"""

from utils import LRUList, Page, Algorithm

class ARC(Algorithm):
    """
    Implements ARC replacement algorithm
    """
```

```
    def __init__(self, memory):
        Algorithm.__init__(self, "ARC", memory)

        self.debug_on = False

        self.t_1 = LRUList("T1")
        self.t_2 = LRUList("T2")
        self.b_1 = []
        self.b_2 = []
        self.t1_target_size = 0

    def cache_hit(self, ref):
        """Handle cach hit
        """
        if self.t_1.has_page(ref):
            self.debug("Cache hit T1. Page %s to T2" % ref.page_id)
            self.t_1.remove_page(ref)
            self.t_2.update(ref)
        elif self.t_2.has_page(ref):
            self.debug("Cache hit T2. Update page %s in T2" \
                           % ref.page_id)
            self.t_2.update(ref)
        else:
            raise Exception("BUG!")

    def evict(self, page_in_b2):
        """Evict some page
        """
        t1_size = self.t_1.size()
        if t1_size > self.t1_target_size:
            page_id = self.t_1.remove()
            self.b_1.append(page_id)
            self.debug("Evict: len(T1)>p => Page %d from T1 to B1" \
                           % page_id)
        elif t1_size < self.t1_target_size:
            page_id = self.t_2.remove()
            self.b_2.append(page_id)
            self.debug("Evict: len(T1)<p => Page %d from T2 to B2" \
                           % page_id)
        else:
            if not page_in_b2 or t1_size == 0:
```

```
                page_id = self.t_2.remove()
                self.b_2.append(page_id)
                self.debug("Evict: len(T1)=p => %d from T2 to B2" \
                           % page_id)
            else:
                page_id = self.t_1.remove()
                self.b_1.append(page_id)
                self.debug("Evict: len(T1)=p => %d from T1 to B1" \
                           % page_id)

        self.memory.evict(page_id)

    def cache_miss(self, ref):
        """Handle cach miss
        """
        page_in_b1 = ref.page_id in self.b_1
        page_in_b2 = ref.page_id in self.b_2
        b1_len = len(self.b_1)
        b2_len = len(self.b_2)
        old_p = self.t1_target_size
        if page_in_b1:
            self.t1_target_size = min(self.t1_target_size \
                                      + max(1, (b2_len / b1_len))
                                    ,self.memory.size)
            self.debug("History hit, B1. p: %d => %d" \
                       % (old_p, self.t1_target_size))
        elif page_in_b2:
            self.t1_target_size = max(self.t1_target_size \
                                      - max(1, (b1_len / b2_len))
                                    , 0)
            self.debug("History hit, B2. p: %d => %d" \
                       % (old_p, self.t1_target_size))

        if not self.memory.has_free_page_frames():
            self.evict(page_in_b2)

        # if found in history, place to T2
        # otherwise, place to T1 and clean history if necessary
        if page_in_b1:
            del self.b_1[self.b_1.index(ref.page_id)]
            self.t_2.update(ref)
        elif page_in_b2:
```

```
            del self.b_2[self.b_2.index(ref.page_id)]
            self.t_2.update(ref)
        else:
            self.debug("New page %d" % ref.page_id)
            # clean history
            t1_size = self.t_1.size()
            t2_size = self.t_2.size()
            if b1_len + t1_size >= self.memory.size:
                if t1_size < self.memory.size:
                    page = self.b_1.pop(0)
                    self.debug("|B1|+|T1|>=c. Removed %d from B1" \
                                   % page)
                else:
                    page = self.t_1.remove()
                    self.debug("|T1|>=c. Removed %d from T1" % page)
            elif (t1_size + t2_size + b1_len + b2_len
                  >= 2*self.memory.size):
                page = self.b_2.pop(0)
                self.debug("|T1|+|T2|+|B1|+|B2|>=2c." \
                           + " Removed %d from B2" % page)

            # put requested page to T1
            self.t_1.update(ref)

        # insert to memory
        page = Page(ref.page_id)
        self.memory.insert(page)

    def process(self, ref):
        """Process reference
        """
        if self.memory.reference(ref):
            self.cache_hit(ref)
        else:
            self.cache_miss(ref)

        self.debug("Mem: %d / %d" % (self.memory.pages_used(),
                                     self.memory.size))
        self.debug("T1:")
        self.debug(self.t_1.page_to_ts)
        self.debug(self.t_1.ts_to_page)
        self.debug("B1:")
```

```
            self.debug(self.b_1)
            self.debug("T2:")
            self.debug(self.t_2.page_to_ts)
            self.debug(self.t_2.ts_to_page)
            self.debug("B2:")
            self.debug(self.b_2)
            self.debug("p: %d" % self.t1_target_size)
            self.debug("----------------------------------------------")
```

B.7 car.py

```
"""Implementation of CAR replacement algorithm
"""
from utils import Page, Algorithm

class CAR(Algorithm):
    """Class implementing CAR algorithm
    """

    def __init__(self, memory):
        Algorithm.__init__(self, "CAR", memory)

        #self.debug_on = True

        self.t_1 = []
        self.t_2 = []
        self.b_1 = []
        self.b_2 = []
        self.t1_target = 0

    def evict(self):
        """Evict a page from cache
        """
        evicted_from_t1 = False
        while len(self.t_1) >= max(1, self.t1_target):
            page_id = self.t_1.pop(0)
            page = self.memory.get_page(page_id)
            if page.referenced:
                page.referenced = False
                self.debug("Moved %d from T1 to T2" % page_id)
                self.t_2.append(page_id)
            else:
```

```
                    self.debug("Evicted %d from T1" % page_id)
                    self.b_1.append(page_id)
                    evicted_from_t1 = True
                    break

            if not evicted_from_t1:
                page_id = self.t_2.pop(0)
                page = self.memory.get_page(page_id)
                while page.referenced:
                    page.referenced = False
                    self.t_2.append(page_id)
                    page_id = self.t_2.pop(0)
                    page = self.memory.get_page(page_id)
                self.debug("Evicted %d from T2" % page_id)
                self.b_2.append(page_id)

            # Remove page from memory
            self.memory.evict(page_id)

        def clean_history(self, ref):
            """Removes a history page, if neccessary
            """
            # Clean history
            page_in_b1 = ref.page_id in self.b_1
            page_in_b2 = ref.page_id in self.b_2
            b1_size = len(self.b_1)
            b2_size = len(self.b_2)

            if not page_in_b1 and not page_in_b2:
                t1_size = len(self.t_1)
                t2_size = len(self.t_2)
                if t1_size + b1_size == self.memory.size:
                    page_id = self.b_1.pop(0)
                    self.debug("|T1|+|B1|=c. Removed %d from B1" \
                                    % page_id)
                elif t1_size + t2_size + b1_size + b2_size \
                        == 2*self.memory.size:
                    page_id = self.b_2.pop(0)
                    self.debug("|T1|+|T2|+|B1|+|B2|=2c."
                               + " Removed %d from B2" % page_id)
```

```
    def process(self, ref):
        """Processes a reference
        """
        if not self.memory.reference(ref):
            if len(self.t_1) + len(self.t_2) == self.memory.size:
                self.evict()
                self.clean_history(ref)

            page_in_b1 = ref.page_id in self.b_1
            page_in_b2 = ref.page_id in self.b_2
            b1_size = len(self.b_1)
            b2_size = len(self.b_2)

            old_t1_target = self.t1_target
            if page_in_b1:
                self.t1_target = min(self.t1_target +\
                                     max(1, (b2_size / b1_size))
                                     , self.memory.size)
                self.debug("History hit, B1. p: %d => %d" \
                               % (old_t1_target, self.t1_target))
                del self.b_1[self.b_1.index(ref.page_id)]
                self.t_2.append(ref.page_id)
            elif page_in_b2:
                self.t1_target = max(self.t1_target - \
                                     max(1, (b1_size / b2_size))
                                          , 0)
                self.debug("History hit, B2. p: %d => %d" \
                               % (old_t1_target, self.t1_target))
                del self.b_2[self.b_2.index(ref.page_id)]
                self.t_2.append(ref.page_id)
            else:
                self.debug("New page %d" % ref.page_id)
                self.t_1.append(ref.page_id)

            # insert to memory
            page = Page(ref.page_id)
            self.memory.insert(page)


            self.debug("T1:")
            self.debug(self.t_1)
            self.debug("B1:")
```

```
                self.debug(self.b_1)
                self.debug("T2:")
                self.debug(self.t_2)
                self.debug("B2:")
                self.debug(self.b_2)
                self.debug("p: %d" % self.t1_target)
                self.debug("--------------------------------")
```

B.8 cart.py

```
"""Implementation of CART replacement algorithm
"""
from utils import Page, Algorithm

class CARTPage(Page):
    """Page id + filter bit
    """
    def __init__(self, page_id, filter_bit="S"):
        Page.__init__(self, page_id)
        self.filter_bit = filter_bit

    def __eq__(self, other):
        import types

        if type(self) == type(other):
            return self.page_id == other.page_id
        elif type(other) == types.IntType:
            return self.page_id == other
        else:
            return False

    def __repr__(self):
        return ('(page: %(page_id)d'\
                    +', r: %(referenced)d, f: %(filter_bit)s)')\
                    % vars(self)

class CART(Algorithm):
    """Class implementing CART algorithm
    """

    def __init__(self, memory):
        Algorithm.__init__(self, "CART", memory)
```

```
        #self.debug_on = True

        self.t_1 = []
        self.t_2 = []
        self.b_1 = []
        self.b_2 = []
        self.t1_target = 0
        self.b1_target = 0
        self.n_s = 0
        self.n_l = 0

    def process_list_t2(self):
        """Process T2 before eviction as defined in algorithm
        """
        adapt_b1_target = len(self.t_1) \
            + len(self.t_2) + len(self.b_2) \
            - self.n_s >= self.memory.size

        if len(self.t_2) > 0:
            cart_page = self.t_2.pop(0)
            page = self.memory.get_page(cart_page.page_id)
            while page.referenced:
                page.referenced = False
                self.debug("Moved %d from T2 to T1" \
                               % cart_page.page_id)
                self.t_1.append(cart_page)
                if adapt_b1_target:
                    self.b1_target = min(self.b1_target + 1
                                         , 2*self.memory.size \
                                             - len(self.t_1))
                if len(self.t_2) > 0:
                    cart_page = self.t_2.pop(0)
                    page = self.memory.get_page(cart_page.page_id)
                else:
                    cart_page = None
                    break
            # Insert last page back, if necessary
            if cart_page:
                self.t_2.insert(0, cart_page)

    def process_list_t1(self):
```

```
        """Process T1 before eviction as defined in algorithm
        """
        if len(self.t_1) > 0:
            cart_page = self.t_1.pop(0)
            page = self.memory.get_page(cart_page.page_id)
            while page.referenced or cart_page.filter_bit == "L":
                if page.referenced:
                    page.referenced = False
                    self.t_1.append(cart_page)
                    if len(self.t_1) >= min(self.t1_target + 1 \
                                                , len(self.b_1)):
                        self.debug("Set filter of %d to L" \
                                       % cart_page.page_id)
                        cart_page.filter_bit = "L"
                        self.n_s -= 1
                        self.n_l += 1
                else:
                    page.referenced = False
                    self.debug("Moved %d from T1 to T2" \
                                   % cart_page.page_id)
                    self.t_2.append(cart_page)
                    self.b1_target = max(self.b1_target - 1
                                         , self.memory.size \
                                             - len(self.t_1))
                if len(self.t_1) > 0:
                    cart_page = self.t_1.pop(0)
                    page = self.memory.get_page(cart_page.page_id)
                else:
                    cart_page = None
                    break
            # Insert last page back, if necessary
            if cart_page:
                self.t_1.insert(0, cart_page)

    def evict(self):
        """Evict a page from cache
        """
        self.process_list_t2()
        self.process_list_t1()

        if len(self.t_1) >= max(1, self.t1_target):
            cart_page = self.t_1.pop(0)
```

```
                self.debug("Evicted %d from T1" % cart_page.page_id)
                self.b_1.append(cart_page)
                self.n_s -= 1
            else:
                cart_page = self.t_2.pop(0)
                self.debug("Evicted %d from T2" % cart_page.page_id)
                self.b_2.append(cart_page)
                self.n_l -= 1

            # Remove page from memory
            self.memory.evict(cart_page.page_id)

        def clean_history(self, ref):
            """Removes a history page, if neccessary
            """
            # Clean history
            page_in_b1 = ref.page_id in self.b_1
            page_in_b2 = ref.page_id in self.b_2
            b1_size = len(self.b_1)
            b2_size = len(self.b_2)

            if not page_in_b1 and not page_in_b2 \
                    and b1_size + b2_size == self.memory.size + 1:
                if b2_size == 0 or b1_size > max(0, self.b1_target):
                    page_id = self.b_1.pop(0).page_id
                    self.debug("Removed %d from B1" % page_id)
                else:
                    page_id = self.b_2.pop(0).page_id
                    self.debug("Removed %d from B2" % page_id)


        def process(self, ref):
            """Processes a reference
            """
            if not self.memory.reference(ref):
                if len(self.t_1) + len(self.t_2) == self.memory.size:
                    self.evict()
                    self.clean_history(ref)

                page = None
                page_in_b1 = ref.page_id in self.b_1
                page_in_b2 = ref.page_id in self.b_2
```

```
            b1_size = len(self.b_1)
160         b2_size = len(self.b_2)

            old_t1_target = self.t1_target
            if page_in_b1:
                self.t1_target = min(self.t1_target +\
165                                  max(1, (self.n_s / b1_size))
                                     , self.memory.size)
                self.debug("History hit, B1. p: %d => %d" \
                           % (old_t1_target, self.t1_target))
                ind = self.b_1.index(ref.page_id)
170             page = self.b_1[ind]
                del self.b_1[ind]
                self.debug("Set filter of %d to L" \
                           % page.page_id)
                page.filter_bit = "L"
175             page.referenced = False
                page.modified = False
                self.n_l += 1
                self.t_2.append(page)
            elif page_in_b2:
180             self.t1_target = max(self.t1_target -\
                                     max(1, (self.n_l / b2_size))
                                          , 0)
                self.debug("History hit, B2. p: %d => %d" \
                           % (old_t1_target, self.t1_target))
185             ind = self.b_2.index(ref.page_id)
                page = self.b_2[ind]
                del self.b_2[ind]
                self.t_2.append(page)
                page.referenced = False
190             page.modified = False

                if len(self.t_1) + len(self.t_2) + len(self.b_2) \
                        - self.n_s >= self.memory.size:
                    self.b1_target = min(self.b1_target + 1
195                                      , 2*self.memory.size \
                                          - len(self.t_1))

            else:
                self.debug("New page %d" % ref.page_id)
200             page =  CARTPage(ref.page_id)
```

```
                    self.t_1.append( page )
                    self.n_s += 1

                # insert to memory
                self.memory.insert(page)

                self.print_debug()

        def print_debug(self):
            """Print algorithm state
            """
            self.debug("T1:")
            self.debug(self.t_1)
            self.debug("B1:")
            self.debug(self.b_1)
            self.debug("T2:")
            self.debug(self.t_2)
            self.debug("B2:")
            self.debug(self.b_2)
            self.debug("p: %d" % self.t1_target)
            self.debug("q: %d" % self.b1_target)
            self.debug("n_s: %d" % self.n_s)
            self.debug("n_l: %d" % self.n_l)
            self.debug("-------------------------------------")
```

B.9 clockpro.py

```
"""Implements ClockPro algorithm
"""
from clock import CLOCK
from utils import ClockPage

class ClockProPage(ClockPage):
    """Page in ClockPro Clock
    """

    def __init__(self, page_id):
        ClockPage.__init__(self, page_id)

        self.is_hot = False
        self.in_test = True
        self.is_resident = True
```

```
        def __repr__(self):
            type_s = ""
            if self.is_hot:
                type_s += "H"
            else:
                type_s += "C"

            if self.in_test:
                type_s += "T"

            if self.is_resident:
                type_s += "R"

            return '(page: % d, r: %d, flags: %s)' % (self.page_id,
                                                      self.referenced,
                                                      type_s)

    class ClockPro(CLOCK):
        """Implements ClockPro algorithm
        """

        M_H_MIN = 0
        M_H_MAX = 0

        def __init__(self, memory):
            CLOCK.__init__(self, memory)
            self.name = "CLOCKPro"
            #self.debug_on = True

            self.hot = None
            self.cold = None
            self.test = None

            self.m_h = 1
            self.hot_count = 0
            self.non_resident_count = 0

            self.size = self.memory.size

            ClockPro.M_H_MIN = self.size / 10
            ClockPro.M_H_MAX = self.size - (self.size / 10)
```

```
    def remove_from_clock(self, page):
        """Remove given page from clock.
        Before removing, make sure that no hands point to that
        page.
        """
        # Move hot hand if it points to page
        if page is self.hot:
            self.debug("Hot points to a page-to-be-removed.")
            self.hot = self.hot.next

        # Move test hand if it points to page
        if page is self.test:
            self.debug("Test points to a page-to-be-removed.")
            self.test = self.test.next

        # Move cold hand if it points to page
        if page is self.cold:
            self.debug("Cold points to a page-to-be-removed.")
            self.cold = self.cold.next

        # remove from list
        page.prev.next = page.next
        page.next.prev = page.prev

    def move_to_head(self, page):
        """ Move page to head
        """
        self.remove_from_clock(page)
        self.insert_to_head(page, self.hot)

    def m_h_inc(self):
        """Adapt m_h by increasing value
        """
        self.m_h = min(self.m_h + 1, ClockPro.M_H_MAX)
        self.debug("m_h++")

    def m_h_dec(self):
        """Adapt m_h by decreasing value
        """
        self.m_h = max(self.m_h - 1, ClockPro.M_H_MIN)
        self.debug("m_h--")
```

```
100
        def run_cold(self):
            """Run the cold hand
            """
            #self.debug("Running cold hand")
105         page_evicted = None
            while page_evicted is None:

                if not self.cold.is_resident:
                    self.cold = self.cold.next
110             elif self.cold.is_hot:
                    self.cold = self.cold.next
                elif self.cold.in_test and self.cold.referenced:
                    self.cold.in_test = False
                    self.cold.referenced = False
115                 self.cold.is_hot = True
                    self.hot_count += 1
                    self.move_to_head(self.cold)
                    # If no hot pages before,
                    if not self.hot.is_hot:
120                     # point hot hand to the first hot page
                        self.hot = self.hot.prev
                    self.m_h_dec()
                    if self.hot_count > self.m_h:
                        self.run_hot()
125                     #self.debug("Back to running cold hand")
                elif self.cold.referenced:
                    self.cold.referenced = False
                    self.cold.in_test = True
                    #self.debug("Moved %s to head" % self.cold)
130                 self.move_to_head(self.cold)
                else: # cold, not referenced, in test period
                    self.debug("Evicted %s" % self.cold)
                    page_evicted = self.cold
                    self.cold = self.cold.next
135                 if not page_evicted.in_test:
                        self.remove_from_clock(page_evicted)
                        self.debug("Removed %s from Clock"
                                   % page_evicted)
                    else:
140                     self.non_resident_count += 1
                        self.m_h_inc()
```

```
            page_evicted.referenced = False
            page_evicted.modified = False
            page_evicted.is_resident = False
            self.memory.evict(page_evicted.page_id)

            # run cold hand to next cold resident page
            page = self.cold.prev
            while self.cold.is_hot or not self.cold.is_resident:
                if self.cold is page:
                    break
                self.cold = self.cold.next


        def run_hot(self):
            """Run the hot hand
            """
            #self.debug("Running hot hand")

            page_set_cold = False
            # set some page cold and then run to next hot
            while not page_set_cold or \
                    (page_set_cold and not self.hot.is_hot):
                # push test hand forward, if necessary
                if self.hot == self.test:
                    #self.debug("Pushing test hand forward")
                    self.test = self.test.next
                if not self.hot.is_hot:
                    if not self.hot.is_resident:
                        page = self.hot
                        self.hot = self.hot.next
                        self.remove_from_clock(page)
                        self.debug("Removed %s from Clock" % page)
                        self.non_resident_count -= 1
                        self.m_h_inc()
                    else:
                        if self.hot.in_test:
                            self.hot.in_test = False
                            if self.hot.referenced:
                                self.m_h_dec()
                            else:
                                self.m_h_inc()
```

```
                    self.hot = self.hot.next
            elif self.hot.referenced:
                #self.debug("%s unset referenced" % self.hot)
                self.hot.referenced = False
                self.hot = self.hot.next
            else:
                self.hot_count -= 1
                self.hot.is_hot = False
                #self.debug("%s set cold" % self.hot)
                self.hot = self.hot.next
                page_set_cold = True

    def run_test(self):
        """Run the test hand
        """
        #self.debug("Running test hand")

        non_resident_page_removed = False
        while not non_resident_page_removed:
            if self.test.in_test:
                if self.test.is_resident:
                    self.test.in_test = False
                    if self.test.referenced:
                        self.m_h_dec()
                    else:
                        self.m_h_inc()
                    self.test = self.test.next
                else:
                    self.m_h_inc()
                    page = self.test
                    self.test = self.test.next
                    self.remove_from_clock(page)
                    self.debug("Removed %s from Clock" % page)
                    self.non_resident_count -= 1
                    non_resident_page_removed = True
            else:
                self.test = self.test.next
        # run test hand to next cold page in test period
        while self.test.is_hot or not self.test.in_test:
            self.test = self.test.next
```

```
        def page_in_clock(self, ref):
            """Checks if the faulted page is in clock
            and returns it, if it is
            """
            page = self.cold.prev
            # only pages between test hand
            # and cold hand needs to be checked
            while page is not self.test.prev:
                if page.page_id == ref.page_id:
                    return page
                else:
                    page = page.prev
            return None

        def process(self, ref):
            """Process a reference
            """
            if not self.memory.reference(ref):
                if not self.memory.has_free_page_frames():
                    self.run_cold()

                # No pages yet
                if self.cold is None:
                    page = ClockProPage(ref.page_id)
                    self.debug("%s is the first page" % page)
                    self.cold = page
                    self.cold.next = page
                    self.cold.prev = page
                    self.hot = page
                    self.test = page
                else:
                    page = self.page_in_clock(ref)
                    if page is None:
                        # new cold page
                        page = ClockProPage(ref.page_id)
                        self.debug("%s is a new page" % page)
                        # place to head
                        self.insert_to_head(page, self.hot)
                    else:
                        # move to head and set hot
                        self.non_resident_count -= 1
                        #self.debug("%s is a cache hit" % page)
```

```
                        self.move_to_head(page)
                        self.hot_count += 1
                        page.is_resident = True
                        page.is_hot = True
                        page.in_test = False
                        if not self.hot.is_hot:
                            self.hot = page
                        if self.hot_count > self.m_h:
                            self.run_hot()


                    if self.non_resident_count > self.size:
                        #self.debug("Too many history pages")
                        self.run_test()

                self.memory.insert(page)

                self.debug("")
                self.debug("Clock:")
                self.debug_clock()
                self.debug("m_h: %d" % self.m_h)
                self.debug("non-resident: %d" % self.non_resident_count)
                self.debug("hot: %d" % self.hot_count)
                self.check()
                self.debug("-------------------------------------")

            return

        def debug_clock(self):
            """Print clock
            """
            tmp_page = self.hot.prev
            while tmp_page is not self.hot:
                tmp = ""
                if tmp_page is self.cold:
                    tmp += " C"
                if tmp_page is self.test:
                    tmp += " T"
                self.debug("%s%s" % (tmp_page, tmp))
                tmp_page = tmp_page.prev
            tmp = ""
            if tmp_page is self.cold:
```

```
            tmp += " C"
        if tmp_page is self.test:
            tmp += " T"
        self.debug("%s H%s" % (tmp_page, tmp))

    def check(self):
        """Check some things and warn if something wrong
        """
        if self.hot_count > 0 and not self.hot.is_hot:
            print "There are hot pages but " \
                  + "hot hand does not point to one"
        if self.non_resident_count > 0 \
                and self.number_resident_pages() < self.size:
            print "There are non resident pages but " \
                  + "still more space in memory"


    def number_resident_pages(self):
        """Return number of resident pages
        """
        count = 0
        page = self.hot.next
        while page is not self.hot:
            if page.is_resident:
                count += 1
            page = page.next
        if page.is_resident:
            count += 1
        return count
```

C Trace data scripts

C.1 generate.py

```
import sys
import random
import ConfigParser


class TraceGenerator(object):

    def __init__(self, hot_range_start, hot_range_end):
        self.hot_range_start = hot_range_start
        self.hot_range_end = hot_range_end

        for page in xrange(hot_range_start, hot_range_end):
            print "%d" % (page)

    def print_hot_pages(self):
        if random.random() < 0.2:
            count = random.randint(1,5)
            for i in xrange(0, count):
                print "%d" % random.randint(self.hot_range_start
                                            , self.hot_range_end)

    def generate_loop(self, ref_count, range_start, range_end):
        page_range = range_end - range_start

        for i in xrange(0, ref_count):
            page = range_start + (i % page_range)
            print "%d" % (page)
            self.print_hot_pages()

    def generate_scan(self, ref_count, range_start):
        for i in xrange(0, ref_count):
            page = range_start + i
            print "%d" % (page)
            self.print_hot_pages()

```

```
        def generate_random(self, ref_count, range_start, range_end):
            for i in xrange(0, ref_count):
                page = random.randint(range_start,range_end)
                print "%d" % (page)
                self.print_hot_pages()

        def generate_correlated(self, ref_count, range_start):
            for i in xrange(0, (ref_count / 2)):
                page = range_start + i
                print "%d" % (page)
                self.print_hot_pages()
                print "%d" % (page)

    if __name__ == "__main__":
        conf = ConfigParser.SafeConfigParser()
        conf.read([sys.argv[1]])

        hot_start = conf.getint("DEFAULT","hot_range_start")
        hot_end = conf.getint("DEFAULT","hot_range_end")
        generator = TraceGenerator(hot_start,hot_end)
        ind = 1
        section = "phase%d" % ind
        while(conf.has_section(section)):
            section_type = conf.get(section, "type")
            ref_count = conf.getint(section, "ref_count")
            if section_type == "loop":
                range_start = conf.getint(section, "range_start")
                range_end = conf.getint(section, "range_end")
                generator.generate_loop(ref_count, range_start
                                        , range_end)
            elif section_type == "random":
                range_start = conf.getint(section, "range_start")
                range_end = conf.getint(section, "range_end")
                generator.generate_random(ref_count, range_start
                                          , range_end)
            elif section_type == "correlated":
                range_start = conf.getint(section, "range_start")
                generator.generate_correlated(ref_count, range_start)
            elif section_type == "scan":
                range_start = conf.getint(section, "range_start")
                generator.generate_scan(ref_count, range_start)
            else:
```

```
                sys.stderr.write("Invalid phase type %s in %s \n"
                                 % (section_type, section))
80
            ind += 1
            section = "phase%d" % ind
```

C.2 scan.conf

```
1   [DEFAULT]
    hot_range_start = 100000
    hot_range_end = 100300

5   [phase1]
    type = loop
    ref_count = 1500
    range_start = 100150
    range_end = 101150
10
    [phase2]
    type = scan
    ref_count = 4000
    range_start = 0
15
    [phase3]
    type = loop
    ref_count = 1500
    range_start = 100150
20  range_end = 101150

    [phase4]
    type = random
    ref_count = 1000
25  range_start = 0
    range_end = 1600

    [phase5]
    type = scan
30  ref_count = 2000
    range_start = 10000
```

C.3 loop.conf

```
[DEFAULT]
hot_range_start = 100000
hot_range_end = 100300

[phase1]
type = loop
ref_count = 5000
range_start = 0
range_end = 1050

[phase2]
type = scan
ref_count = 1000
range_start = 2000

[phase3]
type = loop
ref_count = 5000
range_start = 0
range_end = 1050

[phase4]
type = random
ref_count = 1000
range_start = 500
range_end = 1600

[phase5]
type = loop
ref_count = 5000
range_start = 0
range_end = 1050
```

C.4 correlated.conf

```
[DEFAULT]
hot_range_start = 100000
hot_range_end = 100300

[phase1]
```

```
type = correlated
ref_count = 10000
range_start = 0

[phase2]
type = scan
ref_count = 1000
range_start = 2000

[phase3]
type = correlated
ref_count = 10000
range_start = 0

[phase4]
type = random
ref_count = 1000
range_start = 500
range_end = 1600

[phase5]
type = correlated
ref_count = 10000
range_start = 0
```