# The ELRU Page Replacement Algorithm

**Master's Thesis**

**Heikki Suonsivu**

Teknillinen korkeakoulu
Tietotekniikan osasto
Tietojenkäsittelytekniikan laitos

Helsinki University of Technology
Faculty of Information Technology
Department of Computer Science

Otaniemi 1995

HELSINKI UNIVERSITY OF TECHNOLOGY

ABSTRACT OF THE MASTER'S THESIS

**Author:** Heikki Suonsivu

**Name of the thesis:** The ELRU Page Replacement Algorithm

**Date:** May 28, 1995 **Number of pages:** 6+86

**Department:** Faculty of Information Technology **Professorship:** Tik-76

**Supervisor:** Professor Eljas Soisalon-Soininen

**Instructor:**

This work is about cache systems, particularly in the context of databases and operating systems.

A cache is a small area of memory reserved for important and frequently accessed information. Caches improve performance in computer systems.

An important problem in a cache is how to determine what data has to be removed from the cache when new data needs to be stored in it and the cache is full. This process is called *replacement*.

In databases the replacement process is often complicated since there are large queries, which fetch large numbers of pages into the cache, but only use them once. For these problems, a new algorithm, Extended Least Recently Used (ELRU), is proposed. The ELRU algorithm is based on automatic management of multiple priorities. The algorithm is shown to be efficient by simulations.

According to the observations the optimal replacement policy for the whole system is not always the optimal replacement policy for each transaction running. This brings us to the question of whether the hit ratio, the ratio of accesses to the cache and total number of accesses, is the correct criterion for determining the efficiency of a cache.

Keywords: databases, operating systems, cache, page replacement, ELRU

TEKNILLINEN KORKEAKOULU DIPLOMITYÖN TIIVISTELMÄ

| | | |
|---|---|---|
| **Tekijä:** | Heikki Suonsivu | |
| **Työn nimi:** | ELRU-sivunkorvausalgoritmi | |
| **Päivämäärä:** | 28 toukokuuta 1995 | **Sivuja:** 6+86 |
| **Osasto:** | Tietotekniikan osasto | **Professuuri:** Tik-76 |
| **Työn valvoja:** | Professori Eljas Soisalon-Soininen | |
| **Työn ohjaaja:** | | |


Tämä työ käsittelee välimuisteja, erityisesti tietokantojen ja käyttöjärjestelmien näkökulmasta.

Välimuistilla ymmärretään muistia, joka on varattu tärkeän ja toistuvasti tarvittavan tiedon varastoimiseen. Välimuisteilla parannetaan tietokoneiden suorituskykyä.

Välimuisteissa on olennaista miten hyvin voidaan päätellä hyödyttömin tieto, joka voidaan korvata seuraavalla muistiin haettavalla tiedolla. Tätä toimenpidettä kutsutaan *korvaukseksi*.

Yleensä tietokannoissa sivunkorvausalgoritmille aiheuttavat ongelmia pitkät tietokantahaut, jotka hakevat välimuistiin paljon sivuja, mutta käyttävät niitä vain kerran. Näiden ongelmien yhdeksi ratkaisuksi ehdotetaan Extended Least Recently Used (ELRU) algoritmia, joka perustuu useiden prioriteettien automaattiseen hallintaan. Työssä osoitetaan simulaatioin ratkaisu toimivaksi.

Havaintojen perustella voidaan myös väittää, että koko järjestelmän optimaalinen sivunkorvaus ei välttämättä ole optimaalinen yksittäisten hakujen kannalta. Tämä herättää kysymyksen, onko osumasuhde eli välimuistista saatujen ja kaikkien hakujen suhde oikea kriteeri välimuistin hyvyyden mittaamiseen.



Avainsanat: tietokannat, käyttöjärjestelmät, välimuisti, sivunkorvaus, ELRU

# Contents

# List of Figures

# List of Tables

# Acknowledgements

First and foremost, I thank my professor, Eljas Soisalon-Soininen, for his encouragement, support and patience.

I thank Johannes Helander, Tero Kivinen, Per-Åke Larson, Otto Nurmi, Kenneth Oksanen, Heikki Saikkonen, and Tatu Ylönen, who gave valuable comments and ideas on this work. I also thank attendees of the Distributed Operating Systems and Mach seminar during Spring 1992 in Helsinki University of Technology for comments and notifying me that this might be a new idea worth documenting, which I had not really thought it to be.

I thank my parents, the members of the Shadows database system project, and numerous friends for their endless picking and reminding me about getting the work done.

The first, unfinished version of the ELRU page replacement algorithm was written in 1990. The first actual implementation was included in the shadow paging prototype written in Spring 1992 [72] together with Tatu Ylönen, with a few simplifications. The algorithm has been fully implemented in the Shadows database system and in YACBS cache simulator.

I thank Elizabeth J. O'Neil, Patrick E. O'Neil and Gerhard Weikum for providing me with their LRU-K simulator environment and reference strings.

# 1 Introduction

In abstract terms, a cache is a subset of data items, stored in storage which is faster than the storage the whole data set is in. The latter is often called the secondary storage. The subset is usually limited in size, and the contents of it may change. In other words, data items in it may be replaced by other data items from the data set.

An example would be a computer system's disk buffer cache, into which a set of frequently accessed or lately accessed disk pages are stored in order to avoid reloading them if they are needed soon again.

A modern computer usually has several caches; in addition to caching disk data into memory, there are one or more caches between the main memory and the CPU of the computer. The computer's physical memory itself is used to cache the virtual memory.

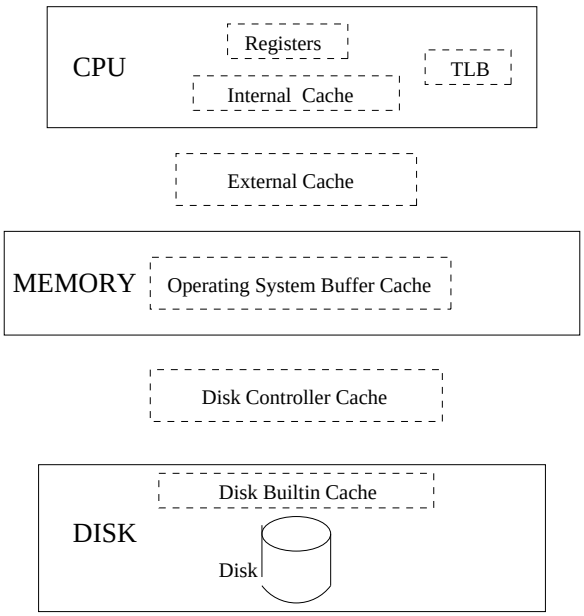



Figure 1: Caches in a modern computer system. CPU, memory and disks form a hierarchy of memories and caches.

The primary use of a cache is to enhance the performance of a computer system by reducing the average time to access data. Caches reduce the overall cost of a computer system by allowing the use of slower and less expensive components. Caches may also serve other purposes in addition to reducing the access time. For example, in portable computers caching of disk blocks can be used to reduce the number of times the hard disk needs to be spinned up and thus reduce energy consumption.

## 1.1 Flow of Data

A cache can be seen as a store into which data items flow in and out. Some data items stay in the cache longer than the other, with a rough division of data into data items accessed once, resulting from sequential access and longer resident data items for "hot spots". This is illustrated in Figure 2.

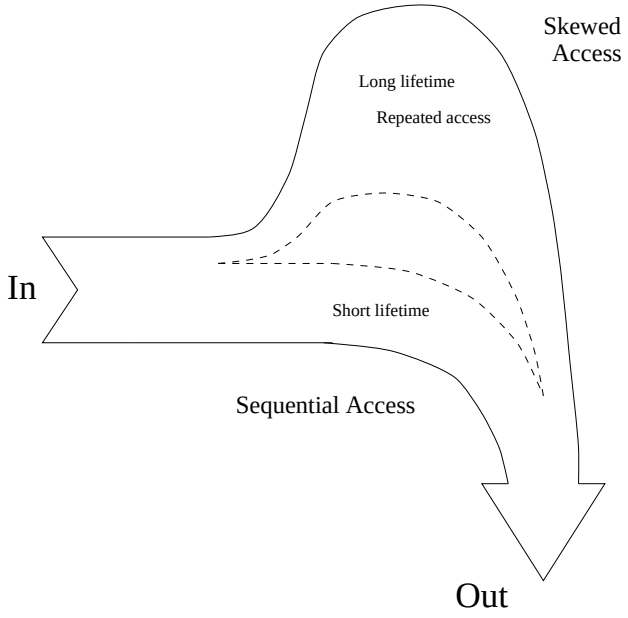



Figure 2: The flow of data through a cache. Some data comes and and drops out right away, other data may stay in the cache for a long time.

## 1.2 Relative Distance of Data

*Not all the memory is alike.*

The original model of a computer with a main memory of constant access time and I/O devices with slower access time is no longer valid.

Even in main memory we have several levels of caches, typically at least an internal processor cache and an external cache. The latest Alpha processor (21164) from Digital has two levels of internal caches. In addition to internal processor caches, there is (at least) TLB, the Translation Lookaside Buffer, a small cache for the mappings from virtual memory addresses to physical memory addresses. For current processors, a memory access may cost just about anything from 5 nanoseconds to 2 microseconds. Accessing an I/O device, like a disk, will add another factor of 1000 to the access time.

To get a picture how far the data really is, it could be compared with access times of a physical object, say, a book (see Figure 3, freely adapted from [26]). Accessing a wide area network (WAN) instead of a CPU register would be roughly comparable to getting a book from the nearest star system instead of your desk, in terms of access time.

Kenneth Oksanen at Helsinki University of Technology ran performance tests on several computer systems, and the results show clearly how different the latencies caused by access times to main memory can be [49]. The tests were run with a program which did random references into memory areas of various sizes. See Table 1.

The steps in the memory access times can be easily explained by caching at different levels:

1. Fastest memory is the processor cache, the size of which typically is somewhere between 1024 bytes and 64k bytes, and whose access time varies from 5 to 10 ns.

2. If the data is not in the processor cache, it is looked up in the external cache, the size of which usually varies from 64k bytes to 1M bytes, and its speed varies between 10 and 30 ns.

3. If the first and second caches fail, the actual memory has to be accessed. Access to commonly used Dynamic RAM (DRAM) costs around 50 to 100ns.

4. Yet another step is caused by a possible TLB miss, ie. the virtual address requested is not mapped by a page table entry in the TLB. The TLB reload is often costly, as it involves a trap and (possibly software) reload of the

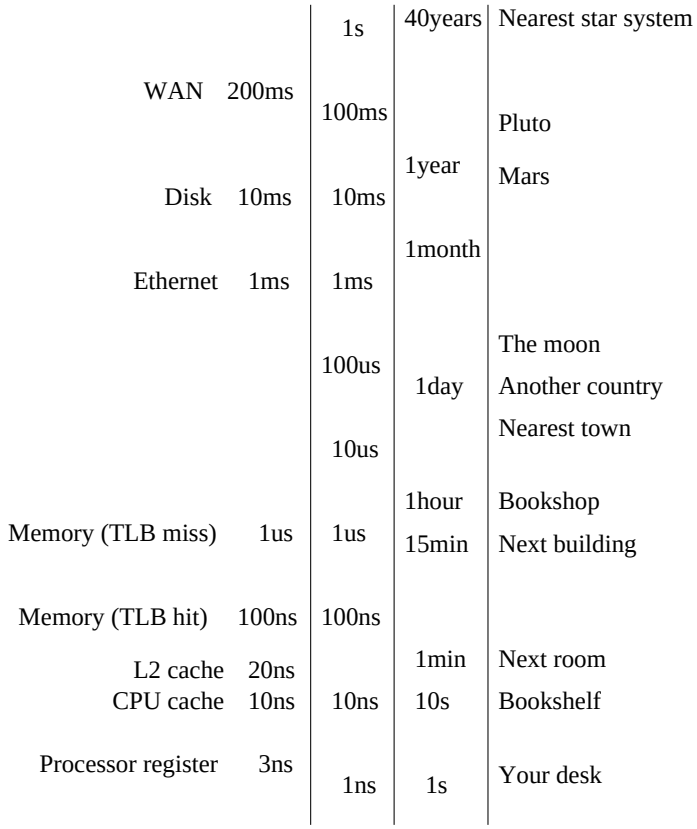



Figure 3: The relative distance of data. The difference in access time to data stored in different parts of a computer system can be huge.

| CPU: | SPARC V8 | SPARC | Alpha 21064 | PowerPC IBM | MIPS R4600 | HP-PA |
|---|---|---|---|---|---|---|
| Computer: | SS-10/2 | SS 1+ | 300 LX | RS6000 | SGI Indigo | HP 705 |
| Working set: | (ns) | (ns) | (ns) | (ns) | (ns) | (ns) |
| 8 MB | 1245 | 634 | 780 | 1237 | 771 | 2174 |
| 4 MB | 1198 | 630 | 742 | 1113 | 711 | 2016 |
| 2 MB | 1107 | 621 | 691 | 857 | 657 | 1815 |
| 1 MB | 979 | 604 | 443 | 455 | 593 | 1511 |
| 512 kB | 712 | 572 | 147 | 421 | 498 | 943 |
| 256 kB | 334 | 505 | 110 | 393 | 345 | 499 |
| 128 kB | 305 | 372 | 107 | 339 | 325 | 343 |
| 64 kB | 288 | 102 | 95 | 234 | 285 | 33 |
| 32 kB | 272 | 92 | 79 | 9 | 206 | 29 |
| 16 kB | 235 | 86 | 50 | 4 | 34 | 28 |
| 8 kB | 164 | 83 | 50 | 3 | 21 | 28 |
| 4 kB | 26 | 83 | 50 | 6 | 21 | 28 |
| 2 kB | 26 | 83 | 49 | 2 | 20 | 28 |
| 1 kB | 26 | 83 | 49 | 3 | 20 | 28 |
| 512 B | 26 | 83 | 41 | 3 | 20 | 28 |
| 256 B | 25 | 83 | 41 | 3 | 20 | 28 |

Table 1: Random reference time in main memory, comparison between several processors and various cache sizes.

TLB entry using several memory accesses. Typical cost of a TLB miss is from 500 to 2000 ns.

It seems that the trend is going to larger and larger portions of CPU chip area used for caches, as clock rates continue to grow further away from chip's external interface speeds.

In addition to cache levels described earlier, there may be need for more levels of caching. An example could be a cache for a database system. Disk access through the operating system involves a considerable overhead, a system call to the operating system kernel, often a couple of context switches, etc. To avoid this most database systems use their own cache and often access the device driver directly, bypassing the operating system cache.

## 1.3 How Disk's Builtin Cache Interacts with Buffer Caching

Not only does caching change the behavior of computer systems in main memory access, but also the intelligent disks with built-in caches are making our assumptions about how to optimally use I/O devices obsolete. Typical disks have caches

now of their own and hide both the geometry and performance characteristics.

This is a mixed blessing, as the operating systems do not yet really know of this, and in some cases, performance is actually lost or lots of it is wasted. Another problem with intelligent disks is the inability to replace unsuitable or buggy replacement or prefetch algorithms.

In particular, current disks have problems with writes. They do perform well in simple tasks like reading large quantities of data in sequence, but fail miserably when trying to write to every other block on the disk. With four 1 gigabyte disks from four major manufacturers (HP, IBM, Micropolis, Seagate), all tested disks ended up requiring one revolution between each write in this test, even when all write cache options were turned on. Current news from last quarter of 1994 are showing that this problem is being addressed by manufacturers. This happens surprisingly late, considering that lots of data is accessed, not in sequence, but in the same area of the disk, usually the same track. In particular, many file systems do a lot of this, trying to optimize the accesses to nearby tracks and/or same cylinder [39].

One source of problems is the SCSI [1] protocol, which was not originally designed for asynchronous operation. Later revisions of the standard have added support for write caching and other features to support multiple concurrent commands and asynchronous operation, but at the cost of added complexity.

## 1.4 The Basic Replacement Policies

The most basic replacement policies are

**LRU** – replace the Least Recently Used page.

**MRU** – replace the Most Recently Used page.

**LFU** – replace the Least Frequently Used page.

**MFU** – replace the Most Frequently Used page.

**FIFO** – replace the page which has been in the cache for longest time (First In First Out).

**RANDOM** – replace a random page.

The MFU policy seems to have no practical use other than approximating the worst case for static access patterns. The other policies all have cases in which they perform reasonably. As most programs generate varying types of access patterns, the best replacement policy is usually somewhere between these policies.

[1] Small Computer System Interface, an interface standard for disks and other peripherals.

## 1.5 Is There Need for Better Caches?

Most operating systems still use CLOCK or derivatives of it for virtual memory page replacement and LRU lists (Figure 34) for disk buffer cache management (CLOCK is a simple but inefficient algorithm to implement LRU replacement, see 2.3.2). Many operating systems clearly suffer from this, performing badly in high load or large sequential file system access (see Figure 4). Clearly a lot of hardware is wasted by using inferior algorithms for replacement.

```
load averages: 29.18, 19.43, 14.97                                    19:10:19
281 processes: 248 sleeping, 27 running, 2 zombie, 4 stopped
Cpu states: 17.7% user,  0.0% nice, 82.3% system,  0.0% idle
Memory:  Free: 84K  Act: 14108K  Inact: 2408K  Wired: 14336K  Swap: 60%

  PID USERNAME PRI NICE   SIZE   RES STATE   TIME   WCPU    CPU COMMAND
    2 root     -18    0     0K   12K sleep 207:07 29.49% 29.49% pagedaemon
 3455 root      22  -20   968K  616K run     0:30  5.71%  5.71% topp
18111 root      87    4  5552K 3652K run    26:01  2.25%  2.25% ircd
 1719 cede2     42    0   212K  112K run     5:41  1.61%  1.61% rlogin
 3528 mina      38    0   528K  636K run     0:00  3.57%  1.51% w
 1891 dancer    36    0   660K  504K run     1:22  1.12%  1.12% irc-2.6
 1721 root      38    0   204K  112K run     3:27  1.07%  1.07% rlogind
14721 root      37    0  2592K  644K run    42:58  1.03%  1.03% emacs
 3523 pasikris  37    0   724K  176K run     0:00  2.14%  1.03% ps
 2125 cede2     35    0   412K  256K run     1:56  0.98%  0.98% term
 3514 root      37    0   276K  416K run     0:00  1.34%  0.98% login
 3406 hsu       36    0  1924K  736K run     0:11  0.93%  0.93% emacs
 3418 cybe      35    0   792K  560K run     0:11  0.93%  0.93% irc-2.6
 3450 kris16     2    0   980K  640K sleep   0:28  0.83%  0.83% irc-2.6
 2590 malinen    2    0   576K  436K sleep   0:23  0.73%  0.73% irc-2.6
```

Figure 4: CLOCK in its work in a loaded BSD Unix system; 30% of CPU time is used for page replacement decisions!

Several other motivations for better cache systems can be seen:

- Better caching is an important issue in special applications where it may be difficult to have large amounts of memory, like in embedded systems.
- As processor speed is increasing faster than the speed of disk devices, the caches are used more intensively. Just distinguishing between "has been used" and "has not been used" is no longer sufficient, as all the cache pages in the cache are used all the time when the load of the system in high. Frequency or other factor of use would be necessary.

- Simple algorithms like LRU may not be efficient with large caches, if there is a sequential access pattern. No matter how large the cache is, it performs badly as long as all sequentially accessed data does not fit into the cache at a time.
- Not just the hit ratio of the whole system is important, but also the access times, the transaction latency and computer response time are probably becoming increasingly important. Different accesses should get different priorities. In the last chapters, it is shown that the best possible page replacement policy is not always the best one for real work. In fact, it may be inferior compared even to a simple realizable replacement policy.

## 1.6 Problems with Cache Systems

A cache miss is as expensive as it would be to fetch any data from the secondary storage. Because of this, the hit ratio is important; the replacement and prefetch algorithms, which control the set of data items in the cache are the primary targets of research on cache algorithms on computers.

The way the data is accessed is another factor which affects the cache performance. The better the replacement and prefetch algorithms can guess what data is going to be accessed next, the better the cache works.

The optimal situation for the system designer is when the access patterns are well-known, do not change, and do not have interdependencies. Unfortunately, these requirements are seldom fullfilled.

The problems start when designing a cache for a real system, as for a database server or for an operating system. Little or no information of the access pattern may be available in advance, and the only information the cache system gets are the data accesses.

## 1.7 Conventions

Due to the nature of this work, the main examples are databases and (page) buffer caches. The presented ideas may still be applied to other applications.

Algorithms, particularly in the survey part, are presented in various formats. I will write them in the style the original author used. This may make comparison difficult, but rewriting the algorithms would have required the tedious process of verifying the rewritten algorithms and possibly checking them with the original authors for correctness.

# 2 Survey on Cache Research

This section briefly describes a number of papers on caches from the late 1960 till 1994. There are a number of algorithms described in these papers in addition to analysis and discussion about other cache related topics.

Several of the papers presented here could be considered classics, which are essential to be familiar with to get a picture how cache problems have been solved before.

## 2.1 Classifications

The algorithms have different properties, but some kind of general classification needs to be done for comparison. The two most important things which need to be distinguished are the realizability and whether the algorithm requires external information to operate or not. Other classifications could be based on technical matters or on the suitability for different applications.

### 2.1.1 Theoretical and Practical Algorithms

The algorithms can be clearly divided into theoretical and practical algorithms. The former are intended for research and comparison purposes, to answer the question "how far we can go?", while practical algorithms try to solve the problem "how we get as near that as we possibly can?". The theoretical algorithms are called MIN, VMIN and WORST, which are designed to give either the best possible results or the worst ones.

### 2.1.2 Automatic and Manual Algorithms

Several algorithms have been developed which use special hints or other information provided by the database system for better replacement. These could be called *manual*, while self-contained algorithms could be called *automatic*. The division is presented in Table 2. *Semi-Automatic* caches require information about context of data, but not actual information about its type. This context could be "which process requested this data" or "which transaction requested this data", and the context is only used for identifying a group, not to decide anything about the importance or priority of the data.

If the algorithm uses external information but does not use it for replacement or prefetch decisions it has been classified as it would not use external information.

The manual algorithms use either accurate information like sizes of tables, or inaccurate information like hints of data type, importance, or estimated access

| Automatic | Semi-Automatic | Manual |
|---|---|---|
| RAND | Working Set | DGCLOCK |
| FIFO | Hot Set | Marginal Gains |
| MRU | DBMIN | EBMI |
| MIN | Priority-DBMIN | Priority-Hints |
| VMIN | Priority-LRU | Starburst BPM |
| CLOCK | | |
| LRU | | |
| PFF | | |
| SIM | | |
| WSCLOCK | | |
| GCLOCK | | |
| LRD | | |
| ELRU | | |
| FIFO with Second Chance | | |
| Fido | | |
| LRU-K | | |

Table 2: Automatic or Manual Algorithms. Automatic algorithms are fully self-contained, semi-automatic algorithms use context information and manual algorithms expect external information like hints or data type.

frequency.

## 2.2 Locality Models of Computer Programs

Each computer program has an access pattern it generates. This may be sequential, random, skewed to one or another parts of storage, or a more complex system. The actual access pattern depends on input data and other concurrent activities. To model programs and their reference patterns for analysis and simulation, various *locality models* have been developed.

Locality models are necessary because analyzing or simulating a real system has too many factors or is just too complex to be completely modeled for useful analysis. Using an abstract model also hides nonidealities in real systems, which would make performance analysis using benchmarks difficult.

### 2.2.1 Independent Reference Model

IRM, Independent Reference Model is defined as a model in which each access to the database is independent of all other previous accesses. This is a simplification, but it makes analysis of cache behavior easier. An example would be

an IRM based model "80% of accesses go into 20% of the data". IRM has been found to be relatively inferior for modeling a real system, as most references have dependencies.

For an example of cache analysis done using IRM, see [47].

### 2.2.2 Query Locality Set Model

Query Locality Set Model, QLSM [13], is a model based on different types of access patterns for different transactions. Normally, there are several different types of transactions, which all have their unique characteristics. Due to their different behavior, they also need different replacement policies to accommodate the different access patterns.

In QLSM, the transactions have been divided to three main groups, and further subdivided to more detailed types:

- **Sequential References**

  **Straight sequential (SS):** each page is accessed exactly once. MRU replacement should be used.

  **Clustered sequential (CS):** small groups of pages are repeatedly accessed, then the access moves forward. These clusters should be fixed into the main memory. If not, MRU replacement should be used.

  **Looping sequential (LS):** the whole file is accessed sequentially and repeatedly. If the file does not fit into the main memory, MRU replacement should be used.

- **Random Reference**

  **Independent random (IR):** Random access without any dependence between accesses. Any replacement algorithm performs equally.

  **Clustered random (CR):** Random access with uneven probability distribution. LRU or a more sophisticated replacement algorithm should be used.

- **Hierarchical References**

  A hierarchical reference is a sequence of page accesses that carries different levels of repeated access to certain data. An example of this behaviour could be a tree traversal path from the root down to the leaves of a tree structured index.

  **Straight hierarchical (SH):** Tree is traversed for only one item. MRU replacement can be used.

**Hierarchical with straight sequential (H/SS):** Tree traversal followed by a sequential scan through the leaves.

**Hierarchical with clustered sequential (H/CS):** As previous, except a CS type scan instead of SS.

**Looping hierarchical (LH):** Tree access through the whole tree. An algorithm which gives higher priority to pages the nearer the root of the tree they are in the tree should be used.

### 2.2.3 Working Set Model

Peter J. Denning's article on working sets in CACM [17] is widely referenced. The paper presents the Working Set Model, and presents some ideas of how this could be utilized in computer systems.

The basic idea of the Working Set Model is simple. The program's working set is the smallest collection of information that must be present in main memory to assure efficient execution of the program.

Formally, the definition of the *working set* of information $W(t, \tau)$ of a process at time $t$ is the collection of information referenced by the process during the process time interval $(t - \tau, t)$.

Thus, the information a process has referenced during the last $\tau$ seconds of its execution constitutes its working set (Figure 5). $\tau$ is called the *working set parameter*.

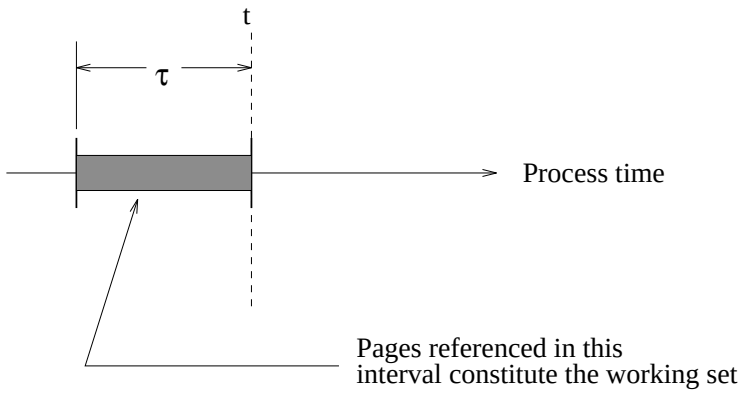



Figure 5: Definition of $W(t, \tau)$, as the set of pages referenced during a specified time interval.

The *working set size* $\omega(t, \tau)$ is defined as the number of pages in $W(t, \tau)$.

The important properties of a working set $W(t, \tau)$ are:

**Size:** The working set has a size, which is monotonically increasing as a function of $\tau$, starting at 0 (Figure 6).

**Prediction:** The working set model is based on assumption that the immediate past page reference behavior of a program constitutes a good prediction of its immediate future page reference behavior. Thus, for short time intervals, the "current" working set is a good approximation of the immediate future working set.

**Reentry Rate:** As $\tau$ is reduced, $\omega(t, \tau)$ decreases, so the probability that useful pages are not in $W(t, \tau)$ increases. Thus, the rate at which pages are recalled to $W(t, \tau)$ increases. Two functions can be further defined: a process-time reentry-rate $\lambda(\tau)$ defined so that the mean process time between the instants at which a page reenters $W(t, \tau)$ is $1/\lambda(\tau)$, and a real-time reentry rate $\varphi(\tau)$ defined so that the mean real-time between the instants at which a given page reenters $W(t, \tau)$ is $1/\varphi(\tau)$.

**Sensitivity ($\tau$):** is defined as the differential of the reentry rate $\lambda(\tau)$, representing the sensitivity of reentry rate to the change of $\tau$.

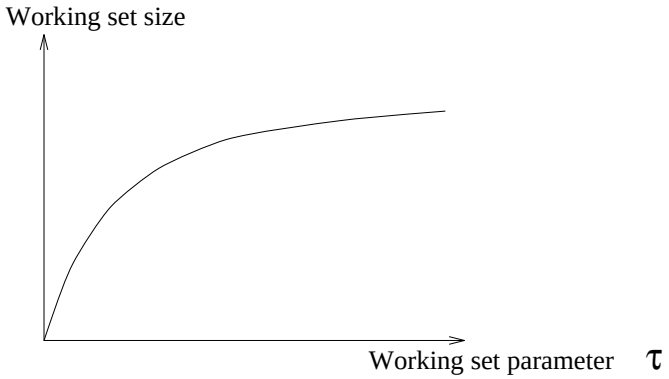



Figure 6: Behavior of $W(t, \tau)$. The working set size increases as the time interval $\tau$ increases, upper limit being all the pages ever accessed.

The other issues discussed in [17] are scheduling and resource allocation. A proposed hardware implementation of memory management system using the properties of the presented Working Set Model is presented.

A later paper [18] summarizes working set related work from the beginning in the 1960's to 1980, spanning 15 years of research. This paper also contains a large bibliography on the subject.

A paper by Marc H. Fogel [24] presents Univac Virtual Memory Operating System (VMOS) implementation of a page replacement algorithm based on the working set model.

### 2.2.4 Simple LRU Stack Model

An LRU stack is a metafora for LRU replacement: picture a stack of books; when a book is read, it is placed on top of the stack. To keep the stack from growing too high, books at the bottom of the stack are moved back to the bookshelf. More abstract versions of this is a table which contains pages. Pages accessed are moved to the "top" of the table and the rest of the table is pushed down.

LRU stack model was analyzed in [62]. The LRU stack model carries similarity to LRU replacement similarly as WS is an offspring of FIFO replacement.

To create the simple LRU Stack model, each position of the stack is assigned a fixed, independent probability. These probabilities are defined $a_1, ..., a_n$, where $n$ is the number of pages in the program (and thus the number of stack positions), $\sum_{i=1}^{n} a_i = 1$ and $a_1 \geq ... \geq a_n$. The $a_i$ are termed *stack distance probabilities* with $i$ being the distance from the top of the stack. At any given time stack position $i$ will be chosen with probability $a_i$; if it is chosen, the page in that position becomes the current reference and is brought to the top of the stack, as in LRU stack replacement.

The useful property here is that if the stack is divided at any point, all pages above the division have got a higher probability than any page below it. Thus, if the stack at time $t$ is $s(t) = (x_1, ..., x_n)$ the locality of size $l$ can be defined to be the pages $[x_1, ..., x_l]$.

### 2.2.5 Intrinsic and Extrinsic Models, LRU Stack Model

In [62] Jeffrey R. Spirn and Peter J. Denning compare simple program locality models and compare their suitability for analysis of locality in programs. The paper also introduces division of locality models to *intrinsic* and *extrinsic* models.

The intrinsic models for locality assume that memory references emit from a program according to some (abstract) structure internal to the program itself. The locality in effect at a given time is a function of the internal state of the program at that time. Examples of these models are page reference distribution functions, the Independent Reference Model (IRM), the locality model, and the LRU stack model.

Extrinsic models do not rely on any assumptions of internal program state. They define locality in terms of observable properties of the memory reference sequence of the program. The Working Set model is a good example of this type of model.

The conclusion of the paper is that the LRU stack model produces the best approximation to the real world behavior. The independent reference model was found to be the poorest, because of its static concept of locality.

## 2.3 Algorithms

### 2.3.1 Belady's MIN

In [3] L. A. Belady described his optimal page replacement algorithm, called "MIN". This algorithm is based on a simple idea, replace the page for which the time to next reference is the longest. One idea for the implementation of this has been presented in the paper. The paper also compares several simple algorithms. The MIN implementation presented in this thesis, different from the implementation Belady presents, is described in section 3.7.4.

In reality, the MIN algorithm cannot determine the optimal replacement for multiprocessing environments, as the change in the replacement also modifies the access string (the order of data accesses). A process which otherwise might get blocked due to a cache miss when LRU or similar algorithm is used now hits and proceeds and makes another cache request instead of blocking and passing the execution to another process. Thus a different access string results.

In principle, the optimal replacement string can be solved for a multiprocessing environment, but it is difficult to compute. In simple cases where the access strings of individual operations, like programs run or transactions executed, do not change because of the different access pattern and time consumed, it can be done with a search of best replacement string(s). There may be more than one solution.

The MIN algorithm assumes fixed cache size, which may be inadequate. It has been shown that with varying cache size, even existing realizeable page replacement algorithms like Working Set may perform better than MIN. This is based on the fact that MIN makes the decisions to remove pages by current cache size. If there actually will be more pages available in future, the replacement decision might have been different.

A new algorithm, VMIN, which is optimal for varying cache sizes, was proposed by Barton G. Prieve and R. S. Fabry in [53].

It is important to realize that MIN computes the best hit ratio for the whole system, for all transactions together. More on this later.

### 2.3.2 CLOCK

CLOCK is probably the simplest algorithm available for doing approximate LRU page replacement. It is also well suited for virtual memory systems, as it can be implemented in most computer architectures. Many current operating systems, such as the BSD 4.3 Unix [39], use this algorithm or a simple derivative of it.

The CLOCK algorithm can be described as follows: the "hand" of the clock

points to a page, and each page has a bit which is set when the page is accessed. Then, the bit under the CLOCK "hand" is checked, and if it is 0, the page can be replaced, otherwise the bit is reset to 0, and the "hand" is advanced to the next page. The idea of the CLOCK algorithm is presented in Figure 7.

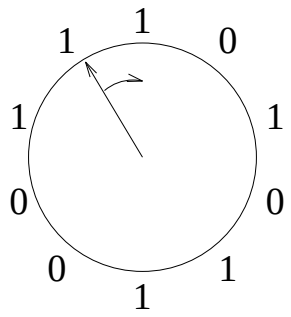



Figure 7: The CLOCK algorithm. A pointer points to a per-page bit, if the bit is 0 replace the page, otherwise reset the bit and increment the pointer.

The clock circle can be implemented as a unidirectional circular list of memory-resident page descriptors, from which pointers are available to memory management unit page tables, or simply scanning through the page tables repeatedly.

The biggest problem with CLOCK is that it has a worst case time complexity of $O(n)$, where $n$ is the number of pages in the cache. CLOCK performance also gets worse as the system becomes more loaded.

Other variants and analysis of CLOCK algorithm can be found in [30, 47, 10].

### 2.3.3 Page Fault Frequency Replacement Algorithm

The Page Fault Frequency Replacement Algorithm was presented in [14] by Wesley W. Chu and Holger Opderbeck. The algorithm is a simple, based on, as the name tells, the page fault frequency of a process. If the page fault frequency of a process is above a value P, called the PFF parameter, measured in number of page faults per millisecond, the number of pages allocated to it is increased. Similarly, if the page fault frequency is below P, the number of pages allocated to the process is decreased. The replacement within the group of pages allocated to a process can be chosen independently of whether the PFF replacement algorithm is used or not. The authors describe a CLOCK algorithm for this purpose. Their results show that PFF performs better than LRU, and is comparable to Working Set algorithm.

The authors develop the PFF algorithm further in [51] by specifying activation and deactivation policy for processes, to improve the system behavior in multiprocessing environment. The paper also presents more simulation results.

### 2.3.4 SIM

A page replacement algorithm called SIM was presented in [32] by John M. Thorington, jr. and J. David Irwin. This algorithm is based on simulating several different page replacement algorithms simultaneously, while one of the algorithms is actually used for the replacement decisions. The idea is based on assumption that the best guess for replacement algorithm is the one used for the previous replacement. During preset time intervals, the performance of each algorithm is compared and the best one is chosen as "current" page replacement algorithm. Thus, this algorithm dynamically varies between different algorithms, and adapts to different operating conditions. The basic algorithms used were LRU, MRU, LFU, MFU, of which MRU was found to be seldom used and thus removing it would have little effect on performance in the presented benchmarks. The paper discusses hardware implementation of this algorithm.

### 2.3.5 WSCLOCK

Richard W. Carr and John L. Hennessy proposed an algorithm called WSCLOCK in [10], which is based on CLOCK, but, in addition, uses Working Set policy (See 2.2.3) to further enchance the replacement criteria. This is accomplished by storing the last reference time for each memory-resident page, and using it as an additional test for replaceablity. The algorithm is presented in Figure 8.

The authors also discuss load balancing issues and present a method called the *loading task/running task (LT/RT)* control. The method is based on the fact that all tasks have two phases, the loading phase, and the running phase. In the loading phase, the task is faulting in the necessary pages for execution, and in the running phase, the working set for the task has been loaded and thus the task executes at its best performance without excessive faulting. To keep the processor utilized, there should be at least one task in running phase at all times. The LT/RT control discriminates the two phases of task processsing and limits the number of concurrent tasks to avoid situations where the cpu running queue would be stack up too many loading tasks to overutilize the available memory, and thus end up with a thrashing system. The primary LT/RT parameter is L, the maximum number of concurrently loading tasks. This value will be determined empirically, but the optimal value is close to the number of paging devices which can be accessed simultaneously. The discrimination of loading tasks and running tasks is by a simple heuristic: a task is loading until it has executed for $r$ units of virtual time or has requested an I/O operation.

[10] presents simulation results, which show that WSCLOCK performs equally to Working Set Policy combined with LT/RT control.

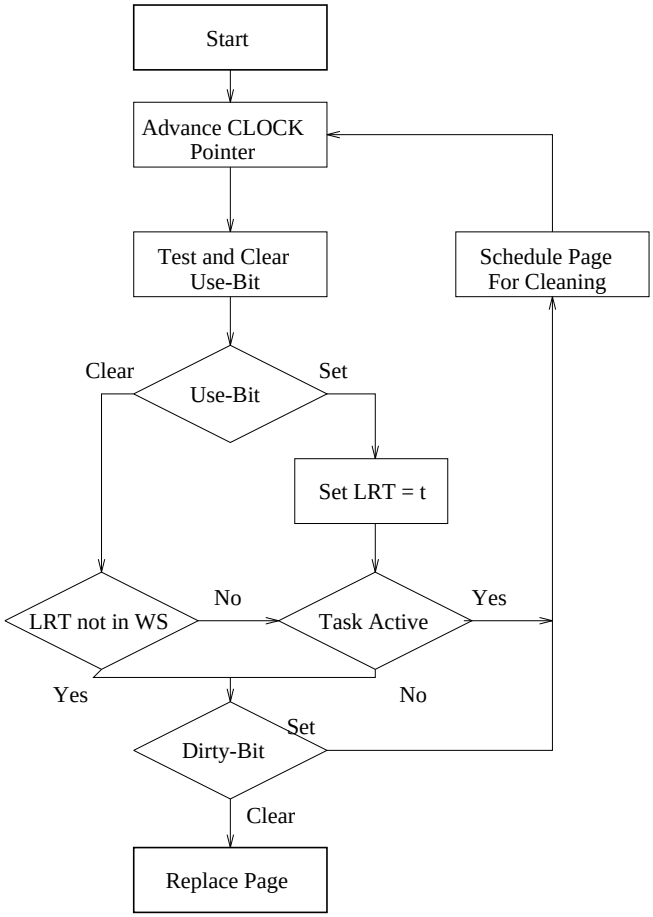



Figure 8: WSCLOCK.

### 2.3.6 The Hot Set Model and Algorithm

The hot set model and a page replacement algorithm based on it was proposed by Giovanni Maria Sacco and Mario Schkolnick in [55, 56]. The hot set algorithm is based on the estimates of the hot sets required by each transaction, calculated by the query optimizer to do the buffer space allocation. The algorithm has been implemented in System R database system.

The data structures for the hot set algorithm consist of LRU lists allocated to each process, associated with two variables, *numref*, the number of pages requested by the process, and *numall*, the actual number of pages in the LRU list. Numref is equal to the hot set size of the process. Numall can be different from numref. If numref is larger than numall, the LRU list is called *deficient*. The algorithm is presented below.

1. **Initialize:** Assign to the free list all the buffer frames.
2. **New process arrives:** Allocate empty LRU list. Set numref to the estimated hot set size, take numref page frames from the free list, and set numall to the number of page frames actually available.
3. **Process requests a page:** Search for a page in the buffer pool.
   (a) Page found. If page is in local LRU stack, update local stack; else do nothing.
   (b) Page not found. Page replacement, two cases to consider:
      i. If the local stack contains unfixed pages then flush the LRU page in it.
      ii. If the local stack contains only fixed pages, then get a frame from the free list, if it is not empty. If the free list is empty, get a frame from another LRU stack containing an unfixed page. Stacks that are deficient are to be preferred. Increase numall by 1.
4. **Process fixes a page:** Increase reference count by one.
5. **Process unfixes a page:** Decrease reference count by one.
6. **Process releases a page:** Do nothing.
7. **Process terminates:** Reallocate the LRU stack among deficient processes, trying to satisfy completely a process before satisfying another one. If no deficient process exists, return frames to the free list.

### 2.3.7 GCLOCK

GCLOCK has been described in Wolfgang Effelsberg's "Principles of Database Buffer Management" [21] in addition to several other basic algorithms. The paper is a tutorial for building cache systems, and describes several basic ideas. The article is frequently referenced but the contents are somewhat outdated. There are several comparisons between different page replacement algorithms, though not one which would include all the algorithms together. The context they use is CODASYL DBMS, but the comparisons have been done with simulation, using a specially made program and two reference strings with somewhat different characteristics of locality of references.

In [21] SECOND CHANCE is mentioned as an alias to CLOCK. It should not be confused with the FIFO with Second Chance used in Mach 3.0 [20], which, while trying to approximate LRU behaviour with somewhat similar methods, is still quite different.

GCLOCK (Generalized CLOCK) is an enhanced CLOCK to get a stopgap between LFU and LRU. LFU is unsuitable for non-static access patterns, while LRU does not perform well, particularly if a sequential process is quickly requesting new pages. The GCLOCK is relatively simple: instead of "Use bit" each page has a reference counter, which is set when the page is brought into the cache and modified at each access, and each clock hand step decrements the counter (See Figure 9).

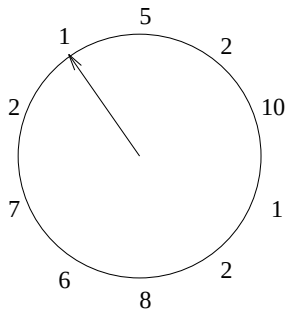



Figure 9: GCLOCK. A pointer points to a per-page counter, if counter is 0, replace the page, otherwise decrement the counter and increment the pointer.

There are several things which can be varied:

- The value which the reference counter gets upon first fetch of a page.
- Increment reference counter at each reference of a page versus set reference counter at each reference to a fixed value.

- Apply page-type or page-related weights instead of incrementing with one or fixed value.

It seems obvious that incrementing the reference counter would provide better results, but still most later references to GCLOCK are only considering the version without increments. The variations here probably should be considered different algorithms, as they behave differently. Further in the text the incrementing version of GCLOCK is referenced as "incrementing GCLOCK".

GCLOCK has similarly to CLOCK O(n) worst case time complexity and its performance deteriorates as the load increases.

Analysis of the weighted GCLOCK version performance has been presented in [47] by Victor F. Nicola et al.

DGCLOCK is a further refined version of weighted GCLOCK. It assigns dynamically calculated, page-related weights $F_j(t)$ and $R_j(t)$.

Another algorithm, LRD, for Least Reference Density, looks at the reference density of a page, and replaces the one with the smallest reference density, in other words, frequency the page is referenced during a certain time window.

### 2.3.8 DBMIN

DBMIN [13] algorithm proposed by Hong-Tai Chou and David J. DeWitt is based on QLSM. In DBMIN, pages are allocated and managed on a per file instance basis, with addition of shared pool for pages common to different active instances. Each instance sets the replacement type in the beginning. Each instance has its own, initially empty, locality set table in addition to the global table. The page can be in one locality set, or it can be both in the global table and one locality set. In addition, there is a global free list.

The interesting part is the access to a page. Here $r$ is the number of pages allocated to the locality set of a file instance of a query, and $l$ is the maximum number of pages allocated to each file instance. When a page is requested by a query, a search is made to the global table, followed by an adjustment to the associated locality set:

1. **The page is found in both the global table and the locality set:** usage statistics are updated.
2. **The page is found in memory but not in the locality set:** If the page already belongs to a locality set, the page is given to the requesting query and no further actions are required. Otherwise, the page is added to the locality set of the file instance, and $r$ is incremented by one. Now if $r > 1$, a page is chosen and released back to the global free list according to

the local replacement policy, and $r$ is set to 1. Usage statistics are updated as required by the local replacement policy.

3. **The page is not in memory:** Disk read is performed, proceed like in case 2.

[13] includes a comparison between FIFO, RAND, LRU, HOTSET, WS and DBMIN.

### 2.3.9 Priority-LRU, Priority-DBMIN and Priority-Hints

In many cases, database and virtual memory systems are intended to support different levels of priorities for transactions or processes. However, if the buffer cache is not honoring any priorities, no actual difference may be seen in the performance of transactions of different priorities. This is due to the fact that even though CPU scheduling allows low-priority processes only a limited amount of CPU time, they get enough of it to deteriorate the system's performance by excessive paging and/or I/O. This problem is clearly visible in the Unix operating system. Under Unix, giving an I/O bound process more priority (lower *nice* value), there is almost no difference in actual performance if there are other I/O bound processes competing of the same buffer cache.

Michael J. Carey et al. proposed in [9] extensions to (Global) LRU and DBMIN, called Priority-LRU and Priority-DBMIN, both of which are intended to solve this problem by introducing priorities to page replacement algorithms.

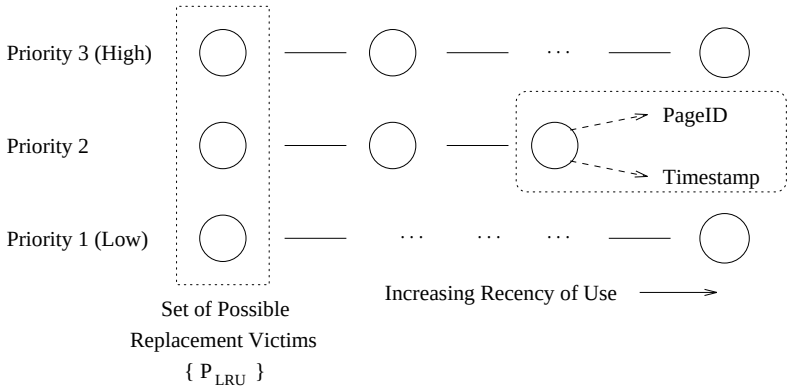



Figure 10: Priority-LRU structure. A list exists for each priority, and rules for replacement prefer lower priority lists and use timestamps to avoid thrashing.

In Priority-LRU there is one LRU queue for each priority (Figure 10). In addition, each page has a timestamp, which is used to keep track of the global recency of usage of pages. A new page, from a transaction of priority $n$, is added to the

LRU queue of priority $n$. The queues are traditional LRU queues, touched page is moved to the high-priority end of the list, but the page replacement is done to all lists together. In addition, no page younger than $W_R$ (for Window of Replacement) should be chosen for replacement regardless of priority. To find the page to replace, the algorithm searches the queues, starting at the lowest priority queue, and checks whether the candidate's timestamp falls inside the protected $W_R$ window. If it does, this is repeated for higher priority lists until either a victim is found or else the search is exhausted. If no page outside $W_R$ could be found, the victim with the lowest priority is chosen.

The Priority-DBMIN, like original DBMIN [13], is based on different properties of transactions, and the ability of the query optimizer to estimate the number of buffers ($OptBufs_{Fi}$) and the optimal replacement algorithm ($OptPol_{Fi}$) to use for each file instance $Fi$ (a table, query result or similar group of data). According to this information combined with the priority of the transaction, the buffer manager allocates required number of buffers for each transaction. In addition, no transaction is allowed to begin running unless it can be guaranteed to get the optimum number of buffers for each of its file instances. In Priority-DBMIN, this admission criteria is extended to say "a transaction is allowed to begin running only if there are sufficient buffers for all currently running transactions of the same or higher priority". In addition, the buffer manager can suspend transactions, starting at the lowest priority, if there is need to allocate buffers for a transaction with higher priority. Otherwise Priority-DBMIN, in respect of allocation and replacement, is similar to the original DBMIN algorithm (Section 2.3.8).

In [31] the same authors present a further developed algorithm, *Priority-Hints*, which adds simple hints to Priority-LRU, trying to be as simple as Priority-LRU with performance equal to Priority-DBMIN, but avoiding the more complex hint mechanism of Priority-DBMIN. The hints provided by the DBMS access methods are simple:

**Favored** pages are likely to be re-referenced by the same transaction.

**Normal** pages include everything else.

Only favored pages are handled specially, normal pages are processed in normal LRU order. Unfixed favored pages are replaced in MRU order, starting with the lowest priority, if no free (normal) pages are available, until it reaches the requesting transaction itself. If no unfixed page suitable for replacement was found, the outstanding request is queued in the Buffer Waiting Queue, and if there are running transactions of lower priority, the transaction with the lowest priority among them is suspended.

With Priority-Hints, an asynchronous write engine is used to flush the pages.

### 2.3.10 FIFO With Second Chance

Another solution to approximation of LRU has been implemented in the Mach microkernel. The algorithm is called *FIFO with Second Chance* [20]. This algorithm is based on using three queues for all pages of memory, called **Active Queue**, **Inactive Queue** and **Free Queue**. The first two queues are FIFOs of pages. A page which is brought in or created, is added to the Active Queue. A kernel thread will then, after a certain time, move it to the Inactive Queue, and unmap it from the memory, thus disabling the access to the page. If the page is accessed, it will generate a fault, and the fault handler will move the page back to the Active Queue. If the page is not faulted in time, it will drop out from Inactive Queue to the Free Queue.

This algorithm can be efficiently implemented without hardware reference bit. The Mach 3.0 also has an implementation which can utilize hardware which requires a software TLB refill, like MIPS R3000 series microprocessors, to emulate reference bits. Normally, the TLB miss handler is invoked when a required TLB entry is not in the TLB buffer, to reload the TLB entry from memory structures. In Mach 3.0, an array of reference bits, one per physical page, is maintained, with non-zero bit indicating that the corresponding physical page has *not* been referenced. This bit is cleared by the TLB miss handler, to mark it referenced [2]. Even though this is an approximation of referenced bit, it works with reasonable precision, as the TLB buffer usually is small and holds only a small number of TLB entries.

### 2.3.11 Marginal Gains

Marginal Gains is a method used in many contexts to estimate the size or best value for some purpose. From [46]: For $s \geq 2$, the *marginal gain* of a reference *Ref* using $s$ buffers is defined as:

$mg(Ref, s) = Ef(Ref, s-1) - Ef(Ref, s)$,

where *ref* can be a random, sequential, or looping (repeated sequential) access.

For any reference $Ref$, the marginal gain value $mg(Ref, s)$ specifies the expected number of extra page hits that would be obtained by increasing the number of allocated buffers from $(s-1)$ to $s$. Typical curves for marginal gain values are presented in Figure 11.

In [46] Raymond Ng, Chrislos Faloutsos and Timos Sellis propose an algorithm based on marginal gains and the division of references to random, sequential, or looping accesses. They also show that the algorithm compares favourably to DBMIN (Section 2.3.8). The algorithm is based on dividing the buffer pool to

[2] Actual implementation used a byte to store the value.

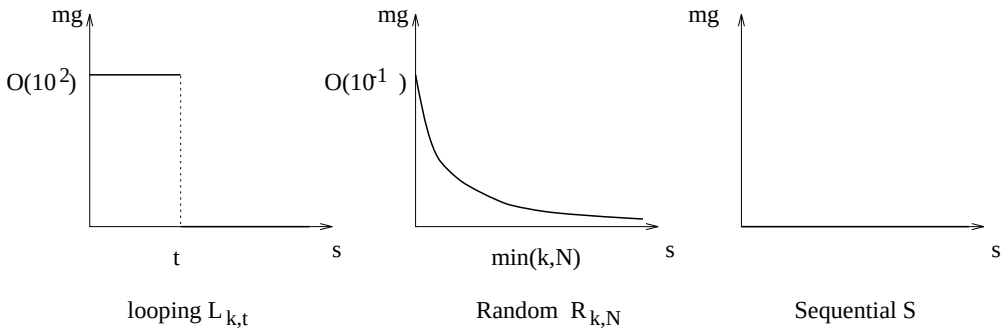



Figure 11: Typical Curves of Marginal Gain Values. The Marginal Gain indicates the delta attainable by an increase of a variable, like cache size in this case.

database accesses, or *references*, using the algorithm presented below.

**Algorithm Allocate:** Let $R$ be the reference at the head of the waiting queue, and $A > 0$ be the number of available buffers. Moreover, let $x$ and $y$ be parameters of MG-x-y to be explained in detail shortly.

**Case 1:** $R$ is a looping reference $L_{k,t}$. If the number $A$ of available buffers exceeds the length $t$ of the loop, allocate $t$ buffers to the reference. Otherwise, if the number of available buffers is too low (i.e. $A < (x\% * t)$), allocate no buffers to this reference. Otherwise (i.e. $A \geq (x\% * t)$), give all $A$ buffers to the reference $R$.

**Case 2:** $R$ is a random reference $R_{k,N}$. As long as the marginal gain values of $R$ are positive, allocate to $R$ as many buffers as possible, but not exceeding the number $A$ of available buffer and $y$ (i.e. allocation $\leq$ minimum $(A, y)$).

**Case 3:** $R$ is a sequential reference $S_{k,N}$. Allocate 1 buffer.

The parameter $x$ is used in determining allocations for looping references. As described in Case 1 above, MG-x-y first checks to see if the number of available buffers exceeds the length of the loop of the looping reference. If there are not enough buffers, MG-x-y checks to determine whether a sub-optimal allocation is beneficial, via the use of parameter $x$. Value $y$ limits the number of buffers allocated to random references.

### 2.3.12 Extensible Buffer Management for Indexes

In [11] Chee Yong Chan et al. present the extensible buffer management for indexes. The paper discusses cases where cache is small compared to even the metadata of the database, and thus, there is little room to generate statistical information about data pages which could be used for determining sensible page replacement policy. The idea is based on hints given by the index management system, and the article concentrates on indexes, though the ideas presented are generally applicable.

### 2.3.13 Algorithms Based on Collecting Statistics

If the amount of memory available for cache is small compared to the data set, it is usually beneficial to store some form of history information to accomplish better replacement and/or prefetch.

If the data pages are large compared to the cache data structures, there is a simple, though somewhat non-elegant solution to this problem. Normally, these data structures for each page consist of a page descriptor, header or similar structure, which has the appropriate pointers and metadata for the page, and a separate area for the page data. Obviously, we can leave this descriptor in memory, free only the page data, and free the descriptor later, or keep it always in the memory if space suffices.

Another way, more memory efficient, would be to store the statistical information about pages access behaviour in a "metacache", a store which has tuples

<pagenumber, statistical data>

These can be used to determine how high priority the page should be assigned to when it is loaded into main memory. This "metacache" can be simple hash table structure, and even random replacement provides reasonable results, though a small metacache could be an easy target for CLOCK or LRU list.

### 2.3.14 LRU-K

Elizabeth J. O'neil et al proposed an algorithm called LRU-K [50]. The algorithm stores $K$ last reference times correlated with the last access time and uses these for replacement decisions. In addition, the LRU-K stores history information for all pages which have been referenced within a specified time, *Retained Information Period*.

According to the simulation results presented in the paper, LRU-K gives almost optimal results with $K$ values from 2 and up, LRU-1 being the normal LRU case.

### 2.3.15 EPFIS

EPFIS (Estimation of Page Fetches in Index Scans) is not a page replacement algorithm, but it is of general interest. EPFIS is an algorithm to estimate the number of page fetches for an index scan when given the number of tuples selected and the number of LRU buffers available. The algorithm is presented by Arun Swami and K. Bernhard Schiefer in [64]. The algorithm is based on running LRU simulations on the index entries once, storing the results to the system catalogs, and subsequently using them for final estimation of the number of page fetches. These results may be useful in query optimizers. The results presented offer much better error ratios than previous algorithms, based on probabilistic models, which EPFIS is compared with.

## 2.4 Cases: Cache Implementations

### 2.4.1 The Starburst Project

A combination of CLOCK and hint mechanism was used in IBM's Starburst project [27], a research project at IBM's Almaden Research Center. This system is a research testbed, which has as its primary goals extensibility and improved performance over the current database system implementations. The Starburst system Buffer Pool Manager (BPM) uses a traditional CLOCK but, in addition, uses hints provided by the (query) optimizer to favor or disfavor pages in replacement stage according to their expected future usage.

In addition to its improved CLOCK algorithm Startburst caches the metadata in the data manager instead of the buffer cache for the data itself. This improves performance considerably, offering a "shortcut" to the very-frequently accessed low-level metadata.

### 2.4.2 The EXODUS Project

Seth J. White and David J. DeWitt describe in [69] their variation of CLOCK implemented on top of *mmap* virtual memory interface. The system is implemented for QuickStore, a memory-mapped storage system for persistent C++ built on top of the EXODUS Storage Manager. As *mmap* does not provide access to access bit in page table entries, the replacement routine uses memory protection to achieve similar effect using Algorithm 12. The idea is to protect all the pages, then let page faults unprotect them, and use the protection status to decide whether a page is a candidate for replacement or not. According to the authors, this works comparably to the traditional CLOCK algorithm. Changing the protections for each page individually would give slightly better replacement, but

using one *mmap* interface to protect all pages turned out to be more efficient due to large overhead of *mmap* system call.

```
QuickStoreCLOCKTick
  if hand.page.protected then
    replace the page at hand;
    hand++;
    return
  endif
  hand++;
  if hand has looped around without a replaceable page found then
    protect all pages in buffer;
  endif
```

Figure 12: The QuickStore simplified CLOCK algorithm. The processor memory management is used through *mmap* Unix system call to protect pages to create an approximation of LRU replacement.

### 2.4.3 The Stealth Distributed Scheduler

The article by Phillip Krueger and Rohit Chawla [36] discusses work on The Stealth Distributed Scheduler in *Workstation-Based Distributed Systems*, WDS'ses. The idea is based on the fact that most workstations are mostly idle, even when someone is using them. To utilize this large "free" CPU potential without disturbing the actual users of the workstations, special methods need to be deviced for scheduling and memory allocation. The system is based on Mach 2.5 microkernel.

The important features in the Stealth Distributed Scheduler are ability to transfer processes and prioritized page replacement and filesystem cache. The page replacement system is relatively simple "always user processes over foreign processes".

The Mach 2.5 file system buffer cache includes three lists of available buffers. The first, the *LRU* list, contains buffers whose contents are likely to be used repeatedly. The second, the *AGE* list, contains buffers that are less likely to be accessed in the near future, such as read-ahead blocks. Finally, the EMPTY list contains buffers which currently have no physical memory associated with them, and which thus are temporarily useless. Whenever a buffer in the AGE or LRU list is accessed, it is returned to the end of the LRU list, making it less likely to be 'recycled'. Recycling is similar to page replacement, occuring when a requested block is not found in the cache. At such a time, a buffer is chosen from the front of the AGE list if the AGE list is not empty, or otherwise from the front of the LRU list. This buffer is then used for the pending block request.

The Stealth Prioritized File System Cache (StealthPFC) duplicates the LRU and AGE lists for the high-priority (user) and low-priority (foreign) pages. The first set of lists contains buffers that are available for recycling by high-priority processes only, while the other set contains buffers that can be recycled by either high or low-priority processes. On a block request, if the block is not found in the cache, a buffer is chosen for recycling as follows: if the request is from a high-priority process, a buffer is chosen from the low-priority AGE or LRU list as described above, if one is available, or from the high-priority AGE or LRU list, if the low-priority lists are empty. If the request is from a low-priority process or is the result of low-priority paging activity by Stealth Prioritized Virtual Memory System (StealthPVM), a buffer is chosen from the low-priority AGE or LRU list, if either of these lists is non-empty. If both lists are empty, a buffer can be 'stolen' from the high-priority AGE or LRU list only if the buffer has not been accessed within *recent-usage-indicator* time. If no such buffer is available, the request must wait until such a buffer is available.

## 2.5 Other Related Issues

### 2.5.1 Non-volatile Memory

In [22] Klaus Elhardt and Rudolf Bayer present a database cache system which uses a combination of a page cache and a non-volatile memory (in this case, a disk). Instead of logging, the cache is made non-volatile by writing the modified (dirty) pages to the non-volatile memory on commits. The key idea is that dirty pages are never actually written to the actual database storage, but they only stay in the cache, and the cache is made crash-proof. This can be done with a non-volatile memory device equal in size to the size of the volatile cache, and the non-volatile cache can be implemented as a disk. This system provides relatively fast transaction times, non-I/O rollback, and fast restart. Page replacement issues are not addressed in the paper.

### 2.5.2 Write-Only Disk Caches

Jon A. Solworth and Cyril U. Orji presented an idea of concentrating on caching write requests instead of the traditional read-oriented approach [61]. In their work, only writes are cached, and they are piggy-backed together with read operations done on the same track. Their research is only about write caching and they do not have a read cache at all, but their methods can be directly coupled with a read cache.

### 2.5.3 Shadow Paging

Shadow paging is not really related to caching, but as there are several references into this technique later in this paper, the idea needs to be explained here.

Shadow paging is a recovery technique which is based on the idea that valid data is never overwritten. The data is always referenced through a page table, which itself is also shadowed (see Figure 13). Thus, the only place which is destructively overwritten is the address of the page table, called the *page table pointer*. Data is referenced by *logical page numbers*, which are then mapped through the page table into *physical page numbers*. The process to do a modification in its simplest form can be described by the following algorithm:

**Modify**

load the page to be modified into memory;
modify it;
write it into a free page on the disk device;
modify the pointers in the page tables to point to new (free) pages;
write the page table pages into their new pages on the disk device;
**if** everything above went ok **then**
  write the pointer to the page table pointer.
  **if** successfull **then**
    free all "old" pages
  **endif**
**endif**

If anything fails during the modification, the old pointer to page table is not modified and all data is still valid. In addition to the above algorithm, it may be desirable to keep a free list on disk to improve the startup time. The free list can also be rebuilt on restart of the system using the page tables, which usually are relatively small compared to the size of the database itself.

Shadow paging is relatively simple to implement and can be made efficient in multiprocessing environments by executing multiple transactions together as a batch [71].

### 2.5.4 Distributed Cache Systems

When a database system and its cache are distributed, things get more complicated. In addition to the page replacement policies, new problems arise because of

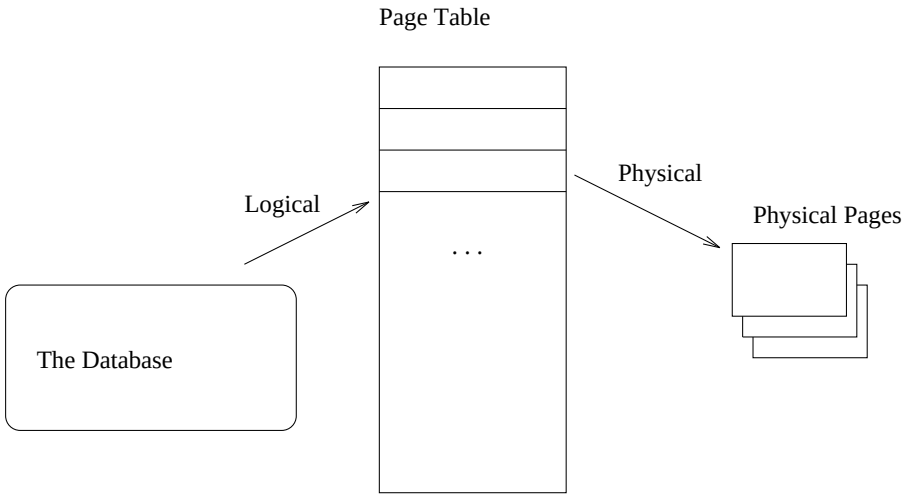



Figure 13: Shadow paging data structure. Each page is accessed using a logical address, which is translated into a physical address using a *page table*. No modifications are made to valid data, but instead modified pages are written to new locations and page table fixed to reflect this. As the page table is also shadowed, only one page, the pointer to the page table, needs to be atomically replaced.

the invalidation of cache data due to updates in other nodes.

In [15] Asit Dan et al. present a model and analysis for distributed cache systems, and the notable result is that both the hit rate and the number of transactions successfully executed decreases, particularly with large buffer sizes.

In [45] C. Mohan and Inderpal Narang address similar problems with a technique called *LP locking*. Similar problems are discussed in [67] by Yongdong Wang and Lawrence A. Rowe.

In [8] Carey et al. compare page and object level caching in distributed systems. In earlier implementations, most systems used either a page level server combined with page level locking, or an object server with object level locking. Their solution is to use adaptive granularity locking combined with a page server. The locking system presented uses page level locking for most pages, but switches to object-level locking when finer-grained sharing is demanded. They also show that it works better than either of the traditional approaches.

With shadow paging, we can note an advantage. A page in the local cache of one of the database servers is only invalidated in the case it is freed, but it is valid as long as it exists. This is due to the fact that in shadow paging, a new page is always allocated for the updated data, and thus there is no need to synchronously invalidate the contents of the modified page in other nodes. The page can be invalidated when it is freed, but this can be done asynchronously by

placing the page in free queue and letting a cleanup process to take care of it. It will be really freed when all nodes have acknowledged its invalidation. At this point it will be reinserted in the pool of available pages.

In addition, with shadow paging and Optimistic Concurrency Control (OCC) [15], a transaction cannot be aborted because of a changed page, as the page cannot be changed but only freed after it becomes valid. On the other hand, a page cannot be freed as long as there are references to it in the system, and thus, it is valid. Thus, OCC may be more useful with shadow paging in this respect than it would be with a traditional non-mapped database concept.

For detailed information on shadow paging, see [71].

### 2.5.5 Caching Query Results

Nabil Kamel and Roger King describe a system to cache the intermediate results of queries in a database in [34].

### 2.5.6 Load Balancing

In [42] HongJun Lu and KianLee Tan propose load balancing in a distributed database system according to the buffer space utilization.

### 2.5.7 Object Caching

Traditional approach of doing separate caching in various parts of the operating system is troublesome, as there is no communication between different caches or it needs to be done separately. These caches should really operate from a common pool of memory, and the replacement policy for all of them should be equal.

The problem is often seen in operating systems, which need to cache several types of data, such as program executable images, program data, file system superblocks, inodes (per-file structures in a Unix filesystem) and other disk blocks (the buffer cache). If all algorithms are implemented separately and they do not communicate with each other, optimal proportions of memory to allocate for each separate cache are difficult to estimate. Sometimes there may be need for lots of inodes being cached, but little data is necessary. An example could be large file name search run on a file system. After a minute, a large number of data blocks would need to be cached, and just a few inodes.

An object cache would allow all the data be allocated as "memory objects", which a common memory manager would manage, using common algorithms for all data. One replacement algorithm is easier to control, and it is easier to tune the system to keep track of the importance of each data object. Mach

microkernel operating system achieves this through the extensive use of virtual memory management [73, 6, 44].

### 2.5.8 Prefetching

In [58] Alan Jay Smith describes an algorithm which uses prefetching and a variant of GCLOCK (Section 2.3.7). Prefetching seems to be the main issue of this article.

[33] describes a prefetching algorithm which prefetches data for sequential accesses. The paper also introduces the term *Memory Pollution*: In some cases, prefetching merely results in reading a page from disk earlier than it would have been read without prefetching; in other cases, pages are prefetched which are not used before they are replaced in the cache, resulting in unnecessary disk accesses. Still worse, an unneeded prefetched pages may replace a page which *is* needed, thus causing at least two extra disk accesses directly attributable to the prefecthing strategy; such prefetches are known as *polluting prefetches* [58, 33].

In [52] Mark Palmer and Stanley B. Zdonik propose an algorithm, called *Fido*, which is based on learning repeated access patterns in the access string. The access patterns which are repeatedly encountered are stored in an associative memory, and used for prefecth. The replacement algorithm used is LRU, but prefetched pages are inserted in the low-priority end of the LRU list in reverse of their expected access order.

The algorithms used in learning could be seen as counterparts of lookup tables in compression algorithms like LZW or GZIP [41]. The goals are similar, to detect similar patterns (access strings) and encode them in the best possible way (replace and prefetch in best possible way).

Tien-Fu Chen and Jean-Loup Baer have proposed implementations of non-blocking and prefetching caches in hardware [12].

### 2.5.9 Locking Data Items into the Cache

In some cases, cache replacement may be prevented by locking certain data to the cache, ie. it will not be replaced in any condition. In an information retrieval system, a user may want to keep the data of his particular interest for certain subjects always handy and never replace it. This has been discussed in [1]. If the data access pattern has well-known characteristics and each reference is independent, it is an easy solution to lock pages which have got the highest probability of access, which results in the same result as LFU replacement policy would, but with no warmup delay.

Many processors permit locking the cache or parts of it to give deterministic

real-time behaviour for timing-critical applications.

### 2.5.10 Effects of Access Pattern Changes

An insight to the effects of changing the access pattern when doing LRU page replacement policy can be found in [5]. In short, the authors show that recovering from a large change in access pattern and frequency is slowly taken into account, and may cause loss of hit ratio for much longer duration than the actual surge causing it takes. When the data access pattern changes, the hit ratio drops to a level of completely random replacement, and even though it improves relatively fast, returning to the original stable state takes a large number of accesses.

### 2.5.11 Relevance of Page and Buffer Faults

In [38, 23] Tomás Lang et al. analyze the relevance of page faults and cache buffer faults, when the buffer cache itself is in the virtual memory, and thus, accessing it may cause page faults. The result is obviously that paging the buffer cache is an absurd idea.

### 2.5.12 Measurements of Cache Performance

Measuring cache access can be tedious and resource-consuming, as the access references result in a large amount of data. Most attempts to reduce the complexity and amount of data are usually based on combining similar accesses to common counters and thus avoiding storing data per each access. Philip Heidelberger and Harold S. Stone discuss this in the context of parallel simulation in [28].

### 2.5.13 Cache on Silicon

There has been some research on hardware implementations of LRU replacement, particularly at IBM [43, 2, 54, 40]. LRU seems to be difficult to efficiently implement in hardware, in scale needed for CPU caches, even though there are several good alternatives for implementation of higher-level page replacement and buffer management problems. More on this subject and several others can be found in Alan Jay Smiths extensive article on cache memories [59].

Most hardware cache implementations are limited by much more complicated implementation on silicon, and thus, most on-chip caches are usually approximations of the LRU policy.

Introductions to CPU caches in general are given by Alan Jay Smith in [59] and by K. R. Kaplan and R. O. Winder in [35].

In [19], Yannick Deville presents SIDE algorithm, as a stopgap between FIFO and LRU. The results are between LRU and FIFO in small cache sizes, but large caches and large associativity drop the hit rate below that of FIFO in some cases.

In [63] Harold S. Stone et al. present an analysis on set-assiciative cache memories and how line size and cache size affect the hit ratio of the cache and seek the optimal partitioning.

Examples of the importance of taking the cache size and locality into account in applications are presented in [37, 48]. The former is about blocking the data to be processed so that processing is done in a small range of memory. The latter paper discusses a fast sorting implementation on a modern RISC processor, and shows the importance of cache considerations, exploiting locality to achieve considerable speed improvements.

### 2.5.14 Prefetching in Hardware

In hardware context, prefetching has not been widely exploited yet, but this issue has been addressed in several papers [58, 33, 12]. Most processors actually already use a sort-of prefetching by doing predictive execution combined with multiple pipelines and instruction scheduling which takes cache hits and misses into account. A good example is the RS6000, which can rearrange the instructions very efficiently (see Table 1) to make external cache miss effect to be neglible.

In shared-memory multiprocessor systems, prefetching and caching are important because of limited bus bandwidth to the processors. The fewer transfers have to be done on the main memory bus, the more processors can be loading it.

Two types of prefetching strategies were proposed in Johnson's paper [33]:

**History-Based Prefetching:** Keep statistics of the previous accesses and try to regenerate the pattern.

**Prescient Prefetching:** The compiler generates information or code which is used to enhance prefetching. Two alternatives are available:

- In *compiled* prefetching strategies the compiler inserts prefetching information or instructions in the code itself.
- In *compiler-directed* prefetching strategies the compiler generates separate information of the working sets

### 2.5.15 Other Papers

One case of formal analysis of LRU, FIFO and $A_0$ algorithms can be found in [68]. The paper defines the algorithms formally and derives expressions for the

expected page fault rate for different distributional assumptions.

[4] describes CPU caches considered for PDP-8 minicomputer, and compares the performance of these.

Rollins Turner and Bill Strecker discuss using LRU stack depth distribution in simulation of paging behavior [66].

Wing Shing Wong and Robert J. T. Morris discuss benchmarking and locality in [70], mostly in the context of CPU cache systems.

[7] by Michael Burrows et al. discusses an implementation of a compressing filesystem developed from a log-structured filesystem (under Sprite operating system). Interesting results in addition to good compression ratio and neglible performance degration, and that even a simple cache of "last track read" can be useful and provide reasonable results.

In [29] Andy Hospodor presents methods for estimating the effects of disk builtin caches on performance. According to the paper, caching would not be useful, if the hit ratio drops under 14% in the disk used as an example, and that there is a lowest limit of useful hit ratio, in general. The odd results are based on the assumption that an I/O operation to the disk is synchronous and multiple commands are not supported, and somewhat pessimistic assumptions on the overhead caused by cache management (1 – 2 milliseconds).

In [65] Dominique Thiebaut et al. present an extension to LRU which uses a partitioning algorithm to increase the hit ratio in a disk buffer cache system. Their two main results are improved hit ratio with small cache sizes and the discovery that a small increase in the hit rate may account for considerable improvement in response time.

Alan Jay Smith has published extensive bibliographies on cache and paging related subjects [57, 59, 60].

# 3 The ELRU Page Replacement Algorithm

This section presents the new algorithm proposed for buffer cache page replacement, the Extended Least Recently Used algorithm, ELRU. The items discussed here are the implementation of it, possible applications and enhancements, and simulation results.

## 3.1 Background

### 3.1.1 Hit Ratio

It seems that the research has too closely concentrated on improving the hit ratio. Relative transaction performance is usually ignored, even though it is what actually is seen as "performance" by the user of the database or an operating system. If it takes less than few seconds, it is fast, anything more is a long time and offers an excuse to go get a cup of coffee or to let concentration wander away from work. The users expect "simple" tasks to happen fast. Simple tasks could be adding a data item into the database, fetching a single record, adding money to a bank account or similar.

From this point of view, it seems that it is not exactly obvious that the hit ratio is the only important factor in usability of a system. Response time and prioritizing the transactions may be much more important.

### 3.1.2 Sequential Accesses

The problems with cache behavior start when a large sequential job starts running; cache is loaded with pages which are only used once. For example, LRU replacement policy behaves badly in this case. The particularly unfortunate effect is that the performance does not get much better as long as the cache is smaller than the area the sequential job is scanning through.

Problems of sequential access have been known for a long time. The first articles which address the sequential access problems are from the end of 1970's.

## 3.2 The Algorithm

ELRU stands for "Extended Least Recently Used". Instead of having one LRU list, ELRU has separate LRU lists for different access frequencies or priorities, and a strategy for moving pages from one list to another to change its priority. The idea is presented in Figure 14.

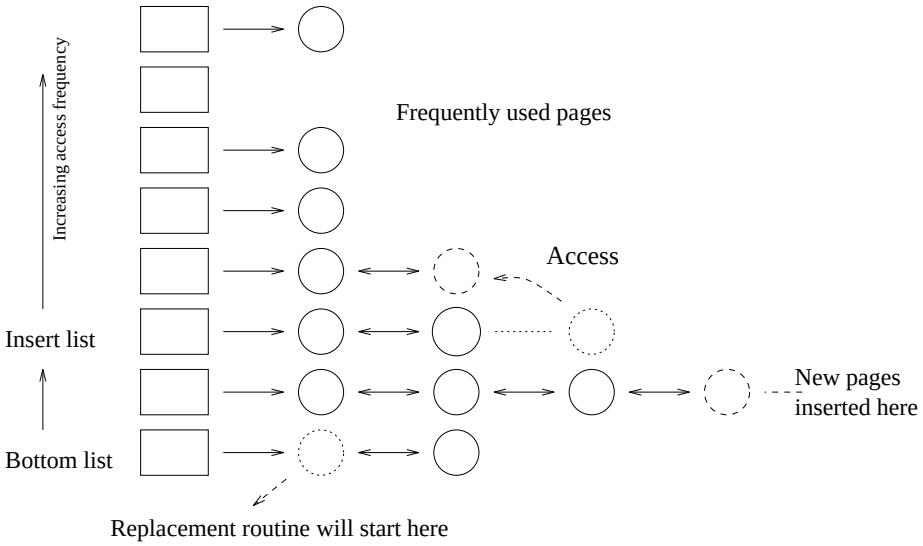



Figure 14: ELRU structure. A number of lists are used for different access frequences. Each access moves a page to a higher list, and new pages are inserted into an *insert list*. Pages are replaced from the *bottom list*.

The ELRU structure is a collection of lists, each of which contains pages for different priorities, and is expressed as

ELRU(number_of_lists,threshold),

where number_of_lists and threshold are variables defined later.

There are special lists called the *bottom list*, the *insert list*, and the *top list*.

A new page loaded into the cache is inserted into the MRU end of the *insert list*. The position of this list is dynamic and depends on the number of pages on the insert list and above it.

On each access to a page resident in the cache, the page is raised up one list by removing it from the list it is on and adding it into the MRU end of the higher list. If the number of lists is fixed, and the page is already on the top list, the page is reinserted back to the top list.

In comparison, in an LRU list, the page would be moved to the tail of the list. In ELRU, the page is moved to the tail of the next priority LRU list. Thus, every access increases the priority of a page and keeps the pages in LRU order for the pages with similar priority.

The replacement candidate is the page on the LRU end of the bottom list. If the bottom list becomes empty and is not the insert list, it is removed and the next higher list which has pages in it becomes the bottom list. The pages on the bottom list thus have the lowest priority.

The number of lists is either fixed or variable. The obvious ways to implement these variations are to use a rotating table of lists or a list of lists.

A list which would become empty by a move of a page could be stripped off. This has to be done if the ELRU structure is implemented as a list of lists, to avoid the number of lists from increasing without limits.

The two parameters which control the operation of ELRU are the *number of lists* and the *ELRU threshold*.

The ELRU threshold is the portion of pages in the cache in the insert list or above it, as seen in Figure 15.

The number of lists is the number of separate lists available for different priorities. The number can be theoretically unlimited, but a practical implementation can and should restrict the number to avoid performance loss and memory waste on large main-memory structures.

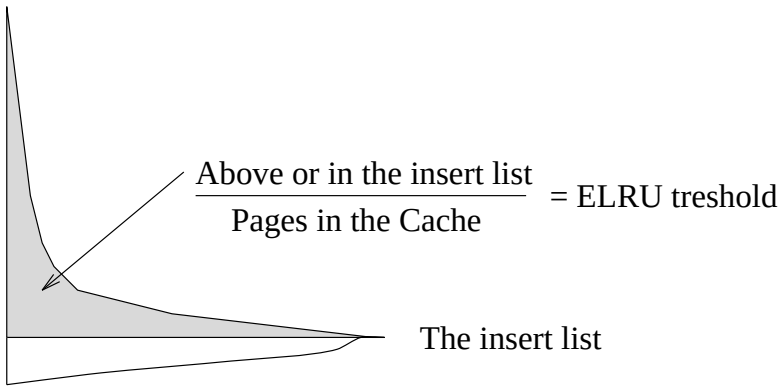



Figure 15: The ELRU threshold is defined as the ratio of pages in or above the insert list and the total number of pages in the cache.

The effects of the parameters are:

**The ELRU threshold** controls how close to LFU the ELRU behaves; if it is 1.0, all the pages in the cache need to be above the insert list, the ELRU does true LFU replacement. If the threshold is 0.0, insert list is always at the top of the structure, in which case ELRU operates somewhere between FIFO and LRU, very strongly favoring last pages loaded into the cache.

**The number of lists** gets values larger or equal to 1 and effects on the resolution of replacement; if it is one, ELRU becomes LRU as only one priority can be distinguished. Larger values improve the resolution of the algorithm; the larger the number, the more the distinguishable priorities may be detected.

A pseudocode for an implementation of the ELRU cache algorithm is listed in appendix B. This implementation uses a table for list headers and thus restricts the number of lists.

## 3.3 Priorities

Priorities can be added relatively simply into an ELRU cache. Each page is given a priority when it is requested from the cache. Instead of moving a page up one list, it is moved one or more lists according to the page priority. When new pages are loaded into the cache, the insertion list is selected from several possibilities according to the page priority.

## 3.4 Virtual Memory Management Using ELRU

When implementing a virtual memory system, there is no possibility of doing list operations or possibility to increment counters. Most memory management schemes used by today's microprocessors only have a "referenced" bit available. This is quite sufficient for CLOCK algorithm, but implementing ELRU would seem to be impossible. A simple solution is available, however, mixing the ELRU algorithm with CLOCK hand concept. In CLOCK, the "Tick" operation, an operation to do one replacement is the following:

**CLOCKTick**

```
while hand.referenced do
    hand.referenced = 0;
    hand = hand.next;
done
replace this page;
hand = hand.next;
```

To transform CLOCK into Extended CLOCK, ECLOCK, the plain CLOCK hand operation is modified into following:

**ECLOCKTick**

```
for count in 1 .. ticks_per_replacement do
    if hand.referenced then
        move to higher priority list;
        hand.referenced = 0;
```

**endif**

hand = hand.next;

**done**

do normal ELRU page replacement from the bottom list;

Insertion to the memory structure is done by inserting the page to the ELRU structure and to the circular list behind the clock hand. Inserting to a random place in the circular list would not necessarily make things worse, but would make the replacement unfair.

The *ticks_per_replacement* value has to be at least 2, but probably should be higher, depending on the ELRU threshold.

The referenced bit is ignored in the actual ELRU replacement. If the page is on the bottom list and it has been passed by several ECLOCK rounds, it probably is not a page with high use probability and can be replaced. This will do "bad replacements" sometimes, but the probability of this is low.

In addition to being more sensible in determining which page to replace compared to CLOCK, the ECLOCK also has O(1) worst case time.

This method could also be used to implement a version of GCLOCK, but unless space overhead is a serious consideration, list based ELRU approach should provide equal replacement performance with less CPU overhead.

ECLOCK needs additional data structures when compared to CLOCK: at least 1 pointer pair for ELRU lists. In addition, if the CLOCK circle is implemented as a list, ECLOCK requires this list to be bidirectional as the replacement can happen anywhere on it. It does not seem to be necessary, though, as the CLOCK circle is usually one iteration through the page table, not a list.

If there is no referenced bit available, this can be implemented in ways similar to Mach 3 [20], effectively modifying *reactivation* to do ELRU list move instead of setting approproate bits.

## 3.5 Comparing ELRU with LRU, CLOCK and GCLOCK

ELRU is comparable to GCLOCK similarly as LRU implemented as a list is comparable to CLOCK. Both algorithms have similar replacement characteristics. To clarify this, see Figure 16.

More detailed side-by-side comparison between GCLOCK and ELRU is presented in Table 3. CLOCK and LRU are similarly comparable and analog in behaviour. Probably the most important characteristic in ELRU is its ability to replace in constant time, similarly to LRU list versus CLOCK.

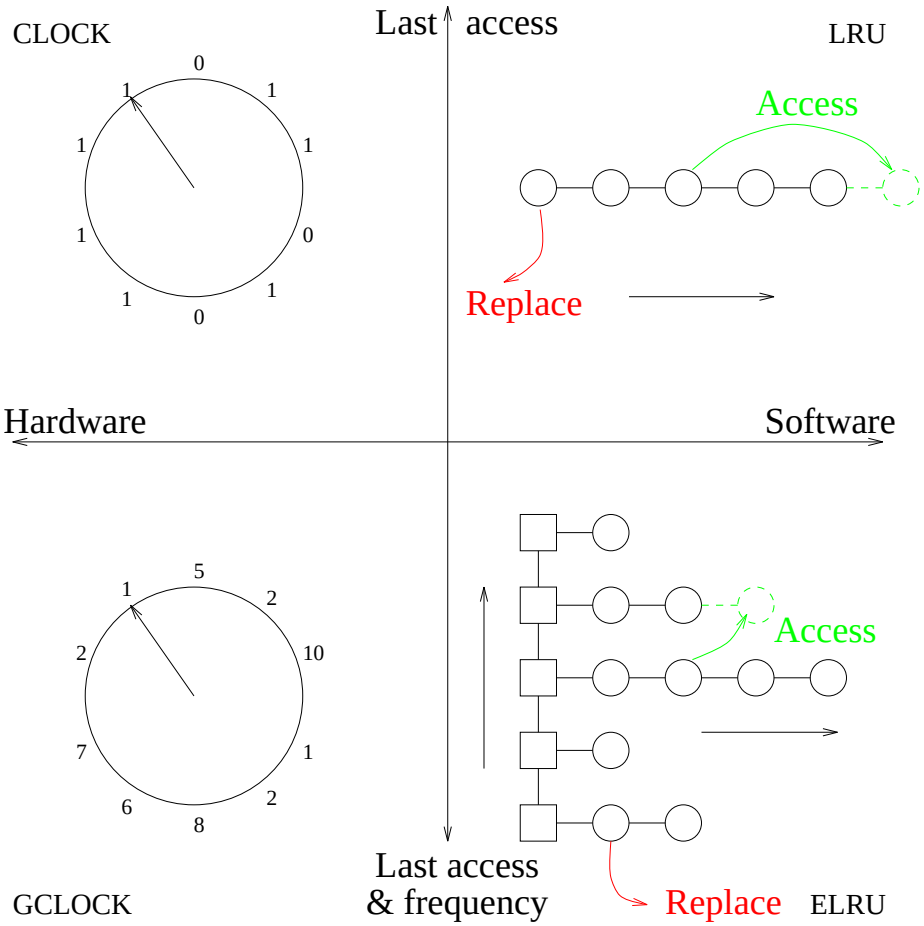



Figure 16: Similarities and suitability for hardware and software applications, CLOCK, LRU, GCLOCK and ELRU.

| Operation | GCLOCK | ELRU |
|---|---|---|
| Access a page: | Increment counter. | Move to a higher list. |
| Search page to replace: | Decrement counter, if 0 replace, else repeat. | Replace the first page from the bottom list. |
| Insert: | Set counter. | Insert to the insertion list. |
| Time complexity: | Min = $O(1)$, Max = $O(n)$. | Min = Max = $O(1)$. |

Table 3: Comparison of GCLOCK and ELRU. The interesting property of the ELRU algorithm is $O(1)$ time complexity.

## 3.6 Simulations

To examine various page replacement algorithms, a simulator was opted instead of running benchmarks in a real database system. This was necessary to easily support simulation of various algorithms and to make MIN calculation possible, using the same environment as the benchmarks are run. In addition, using simulation allowed experimenting with different configurations without large overhead of a real database system.

The cache simulator used was unimaginatively named YACBS, for Yet Another Cache Behavior Simulator.

### 3.6.1 The Simulation Benchmark

In a normal database system, the actual access pattern varies, and consists of a mix large sequential accesses, and a large number of short transactions. The TPCB benchmark [25] alone is unsuitable for this work, as it only demonstrates short identical transactions, and the access pattern is static. The TPCB also is complicated, as it needs multiple tables and has a large number of distinguishable priorities. This would make it difficult to determine the causes of different aspects of detected behavior. A simpler benchmark was necessary.

The benchmark used is based on one table, which is accessed either randomly alone or with one sequential access running at the same time. The random accesses start at random times with a preset average delay between each. The table is a tree structure and thus the random accesses generate a skewed access pattern; the root of the tree is the most frequently accessed page, and the access probability drops towards the leaves. This is a simple benchmark but provides a reasonable source of information on effects of sequential access and ELRU ability to do replacement.

### 3.6.2 Variables

The benchmarks were run for the following variables:

**Replacement algorithm:** Either LRU, ELRU with various parameter values, static allocation, or MIN.

**Sequential:** The number of sequential threads is either 0 or 1.

**Cache size:** The cache size varies from 1 page to 100% of database size, 1 page increments for less than 100 pages and $2^n/4$ steps thereafter.

**Static or varying access pattern:** The access pattern generated by the random access is either static or accessed an area of 1/8th of the database in

size for an average of 256 times and then selected randomly a new area to access.

The results were collected for both sequential and random transaction types separately, in addition to the total values.

### 3.6.3 Table Used in Simulation Runs

The table used in simulation tests is a table of 131072 items, each 128 bytes. Page size is 512 bytes and the number of pages for each tree level is described in Table 4. Only the leaves contain data. Each tree node contains 16-byte key-pointer pairs, filling up 2/3 of the node on the average. Leaves contain links to next/previous pages.

| Tree level | Pages |
|---|---|
| Root | 1 |
| 1 | 21 |
| 2 | 441 |
| 3 | 1560 |
| Leaf | 32769 |

Table 4: Simulation table structure. The data is stored in leaves and tree nodes contain key-pointer pairs only, 2/3 ratio being filled up on average.

## 3.7 YACBS – Database Cache Simulator

YACBS offers configuration of various variables in the system, like cache size, number and types of transactions and intensity of accesses, the replacement algorithm and algorithm-specific parameters in case of the ELRU.

Run times for simulations, depending on the algorithm, are usually around 20 seconds for a simulation of 1 million accesses on a Sun SS10-402. The MIN calculation is somewhat heavier than simulating a true algorithm, $O(n \log c)$ instead of $O(n)$, $c$ is cache size, $n$ number of accesses. All the data structures needed are kept in memory during the simulation.

### 3.7.1 Components of YACBS

YACBS was written in C++, and builds on a group of classes which simulate the parts of a database system and its operating environment. The simulation model is simple, but it implements all the necessary components, process scheduling,

tables, several caches, two types of transactions, I/O queue and devices. The simulation assumed a read-only system.

**PageDescriptor** Each page in the system is represented by a PageDescriptor, which contains statistics data and data necessary for each cache algorithm and list pointers. An array of these is allocated to accomodate all pages in the database, indexed using page numbers.

**Device** The device is a container for PageDescriptors.

**Cache** Various cache types are derived from Cache:

- **LRUCache** Simple LRU strategy, implemented as a LRU list.
- **ELRUCache** ELRU strategy, implemented as circular table of lists.
- **ForcedCache** The Cache is statically allocated, starting from the root towards the leaves.
- **MINCache** Calculated (semi)optimal page replacement.

**Table** Implements a simulation of a table which has access characteristics of a tree with data and links in the leaves. This means that sequential accesses do not touch the tree nodes, but only the leaves.

**Database** Database is a container for Tables. Only one Table was used in the simulations.

**Transaction** YACBS implements two types of access types, random and sequential.

- **RandomTransaction** A random access type generates transactions which make randomly distributed requests to the given table. Optionally, the requests are done into an area of 1/8th of the table, and the area accessed changes every 256 accesses. After completing an access to the table the access is repeated after a random time.
- **SequentialTransaction** A sequential access starts at a random point in the table and starts reading it sequentially.

**TransactionProfile** The description of transactions being executed and the scheduler.

**IO** The I/O system simulates an I/O queue for device reads. This system is simplified, and assumes that each I/O operation takes an equal amount of time.

### 3.7.2 Time

The time concept in YACBS is simple. Time is measured in virtual CPU operations, which represent an operation which is done in main memory and may involve one cache access. Each I/O operation is assumed to take the amount of time consumed by 100 CPU operations. This is roughly consistent with the work versus I/O ratio we observed on a SS10-402, but the value is processor- and environment-dependent.

### 3.7.3 Transactions and Threads

Each YACBS thread runs one type of transactions, each transaction is run by one thread and one thread is running exactly one transaction. This is to simplify the simulator structure, but most real-world implementations use a similar scheme.

### 3.7.4 The Implementation of MIN in YACBS

The implementation of MIN differs from the one described in Belady's paper (see 2.3.1), which is inteded to do the calculation in limited amount of main memory from a sequential device. Neither restriction was bothered by the YACBS, so a straigthforward algorithm was used instead.

The MIN algorithm in YACBS does a collecting pass, which collects all page references into an array. After the collecting pass has been completed, actual MIN calculation takes place. First the array of accesses is processed backwards to determine when each page is used next, then the array is processed in forward direction, doing page replacement by replacing the page which has the longest time to its next use. The algorithm is shown in Figure 17.

The most critical part of calculation time is to quickly find out the page which has got the farthest away next access. This was accomplished by storing all cache-resident pages into a structure sorted by the time of next access. The structure used is AVLMap from GNU libg++ package.

### 3.7.5 Static Cache Allocation

When the access pattern of a database is well known, a good caching algorithm is to cache those pages for which the probability of access is the largest. An example of this would be a tree structure, which is accessed randomly or with given probabilities of access for each of the database records.

To make this easier, assume that each leaf page of the tree has the same probability of access. In this case, the cache is allocated statically to consist of pages from the root towards the leaves (see Figure 18).

```
reference(page)
  access_string[pages_accessed].page = page
  pages_accessed++

calculate
  for i in 1 ... pages do
    page[i].next_access = no_next_access
  done

  for i in pages_accessed ... 1 do
    access_string[i].next_access = page[access_string[i].page].next_access
    page[access_string[i].page].next_access = i
  done

  for i in 1 ... pages_accessed do
    page[access_string[i].page].next_access = access_string[i].next_access
    replace the page which has the farthest away next_access
  done
```

Figure 17: The MIN implementation in YACBS. A simple backward loop over accessed pages and a "next reference table", followed by a forward assignment loop.

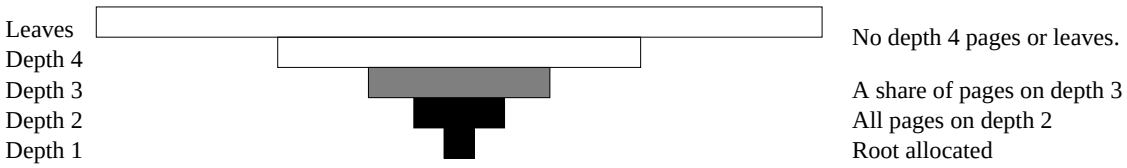



Figure 18: An example of static cache allocation for a tree structure. Keep all the pages near the root in the cache.

If the tree structure has a depth $m$, including the leaves, all the pages which form the tree structure from the root of the tree up to depth $n$ are loaded into the cache, and the rest of always loaded for each access. Thus the cache contains the pages with the highest hit probability. The total hit probability can be calculated from

$P_{total} = \overline{P_{root}, P_1, P_2, ..., P_{n-1}, P_n, P_{n+1}, ...P_m}$

Where $P_i$, $i = 1, 2, ..., n-1$, is 1, $P_n$ is $\frac{N_n}{N_c - N_1 - N_2 - ... - N_{n-1}}$ where $N_n$ is the number of pages of tree at this tree level. $P_{n+1}, ..., P_m$ are all 0.

Static cache allocation suffers one page of cache because it needs one buffer to load the missed accesses into. This has been taken into account in the simulations.

The results calculated using this estimate are identical with a simulation run with static cache allocation, including the case with sequential access. This algorithm works reasonably well even with variable access pattern, and is similarly immune to the effects of sequential access. See Figure 20.

The hit ratio can still be improved if the tree has data in the leaves only, in other words, the access will always proceed up to the leaf level of the tree. When a node in a tree is accessed, the hit probability of all pages which are pointed at from the node get higher hit probability than the they have on average while hit probability drops to 0 for pages unreachable from the node. This can be further used in the decision for replacement.

## 3.8 The Simulation Results

Most figures in this subsection are for random replacement hit ratio for both with and without sequential access, while the random replacement hit ratio is equal to buffer hit ratio for no sequential transactions case. The effect of sequential access for the whole buffer hit ratio does not really make sense, as the results would be difficult to compare (see Figure 21).

### 3.8.1 Effect of Sequential Access on Performance

The effect of sequential access is considerable. LRU suffers severely (Figure 21).

ELRU performs well in comparison; while it still looses some performance, the decrease is less severe (Figure 22). For variable access pattern case, ELRU shows less clear win, particularly for large cache sizes, as hysteresis in the replacement is not an advantage in this case. It still is a noticeable improvement over LRU.

### 3.8.2 MIN Results

The proposal that looking at the system hit ratio is not always the right measure of the performance of the replacement algorithm shows to be valid in MIN replacement results. The MIN replacement, even though producing the best hit ratio for the whole system, degrades the performance of a random access to the table used in the simulations when a sequential access is running (Figure 23). The effect is particularly bad when the access pattern is static, but shows similarly with varying access pattern (Figure 24).

This is caused by the fact that MIN, with a capability to predict future, knows that replacing pages on the path of the sequential access is not sensible, as they will soon be accessed. Any page which is loaded into the cache and is in this path and will fit in the cache will not be replaced until the sequential access has passed by (see Figure 19). The problem is not visible with small cache sizes, but when the cache sizes grow to fit all the tree nodes in the cache, the hit ratio of random access starts to drop. This happens because the larger the cache, the larger the probability of a leaf page getting loaded in and not replaced until the sequential access has passed it. This reduces the space available for tree nodes and randomly accessed leaf pages, for which the miss ratio grows.

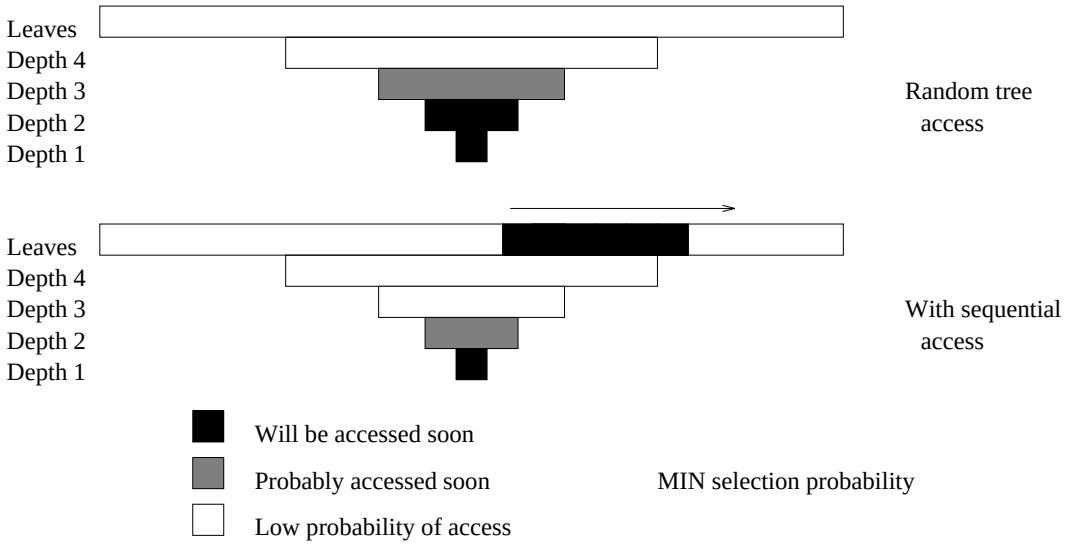



Figure 19: MIN replacement probabilities with a sequential access pattern. A page which is loaded into the cache and is going to be used by the sequential access soon will not be replaced until the sequential access has passed it.

### 3.8.3 Effect of ELRU parameters

For static access pattern case, the optimum value for ELRU threshold is 1.0, as obviously optimum replacement policy for static access pattern is the LFU replacement, which ELRU does using the threshold value of 1.0. See Figure 25.

With varying access patterns, the best ELRU threshold varies according to the cache size. For small cache sizes, less than 5 percent of the database size, the best value starts around 0.5-0.7 and grows towards 1.0. The value of 1.0 is reached when all the tree nodes fit in the cache, and the effect of the ELRU parameters becomes similar to static access pattern case.

The variation of the threshold value suggests that it should be dynamic and change according to the other conditions. The effect of the threshold on the hit ratio of the random access is considerable, up to a factor of 5 for small cache sizes.

The number of ELRU lists should be at least the number of distinguishable levels of reference density in the system. After this number, the performance improvement seems neglible. Using too many lists does not reduce the performance as long as the number of lists stays reasonable to avoid ill effects with processor and TLB caches. The effect of the number of ELRU lists can be seen in Figure 26.

The combined effect of the ELRU parameters for three (small) cache sizes (16 pages, 64 pages and 2048 pages) can be seen in 3D graphs 27, 28 and 29.

### 3.8.4 Comparison of All Algorithms Simulated

The full comparisons of all algorithms simulated are presented in Figures 30, 31, 32 and 33. Relative performance improvement turns out to be at its maximum with small cache sizes, while showing improvement through the whole range of cache sizes for almost all cases.

# 4 Conclusions

This work presents a simple solution to improve cache performance when the cache is small.

The simulation results are promising; the algorithm can detect different levels of access frequency and use them to enhance the replacement behavior to improve the hit ratio and tolerance of sequentially accessed data.

The replacement time complexity of the ELRU algorithm is O(1). Thus it is suitable for applications that require fast page replacement decisions. This is particularly important for real-time systems, but it is beneficial for any general-purpose computer or a database system.

ELRU may have some advantage in parallelism over traditional LRU, when used in shared-memory multiprocessing environments, as the hot spots at the higher lists generate less locking conflicts than an LRU list structure would produce. However, as both the ELRU and LRU list access operations are very short, the effect may be neglible.

It seems that trying to simply improve hit ratio is not always justified; in some cases the best hit ratio may lead to an increase of average transaction latency.

The ELRU algorithm also is relatively simple to implement. Even though it contains couple of hundred lines of code, the code is straightforward. The data structures are simple.

The interlocks in a multithreaded program can be implemented both with individual locks in each object or with a master lock. The latter is probably more efficient in most cases, unless the objects being cached are large and difficult to handle.

# 5 Future Work

More simulations could be run to get better knowledge of how the ELRU parameters effect the hit ratio. In particular, the threshold needs to be more closely examined.

The implementation of ELRU virtual memory replacement, ECLOCK (in 3.4) in place of CLOCK seems possible and useful idea, but the algorithm needs to be verified.

An abstract model for ELRU replacement could be developed. This probably would be in similar lines with GCLOCK analysis.

The algorithms presented here, the ELRU, in particular, have not been applied patents for. This is intentional, not based on assumption that this would not be patentable (*everything* seems to be these days). However, someone else may have co-invented similar algorithms and patented them.

# A Simulation Results

Most of the simulation results are presented here. In all figures, the $x$ axis is the cache size in logarithmic scale. The critical points of the system, ie. the cache sizes in which given level of tree nodes fully fits in the cache have been labeled. The y axis is either the buffer hit ratio or, usually, the random access hit ratio, defined as the hits and misses by the sequential access ignored in the calculation. In comparison between various algorithms, the relative hit ratios are also presented.

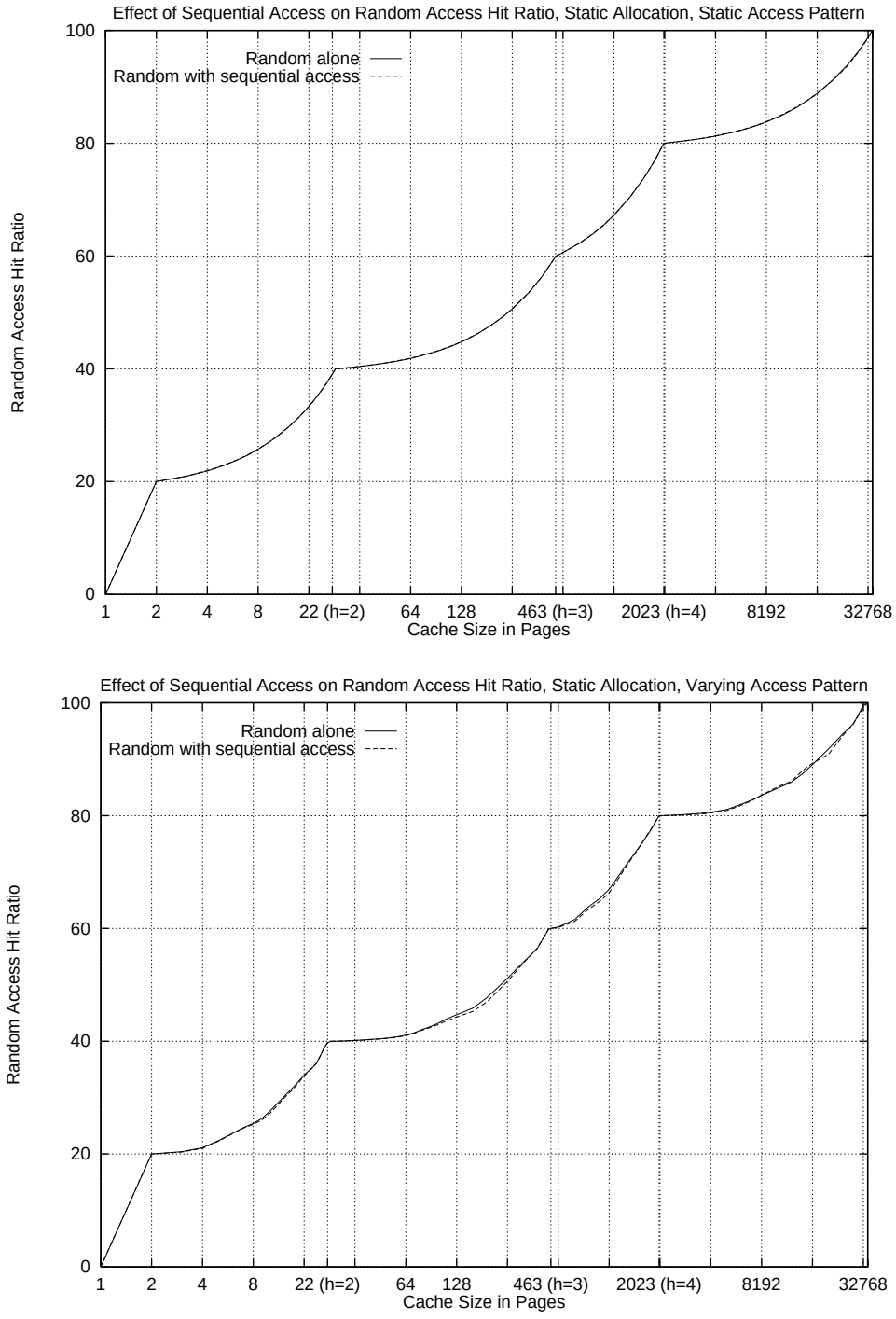



Figure 20: Performance of static cache allocation. The random access hit ratio does not decrease when a sequential access pattern is present in the system, thus the curves join in both static and variable access pattern cases.

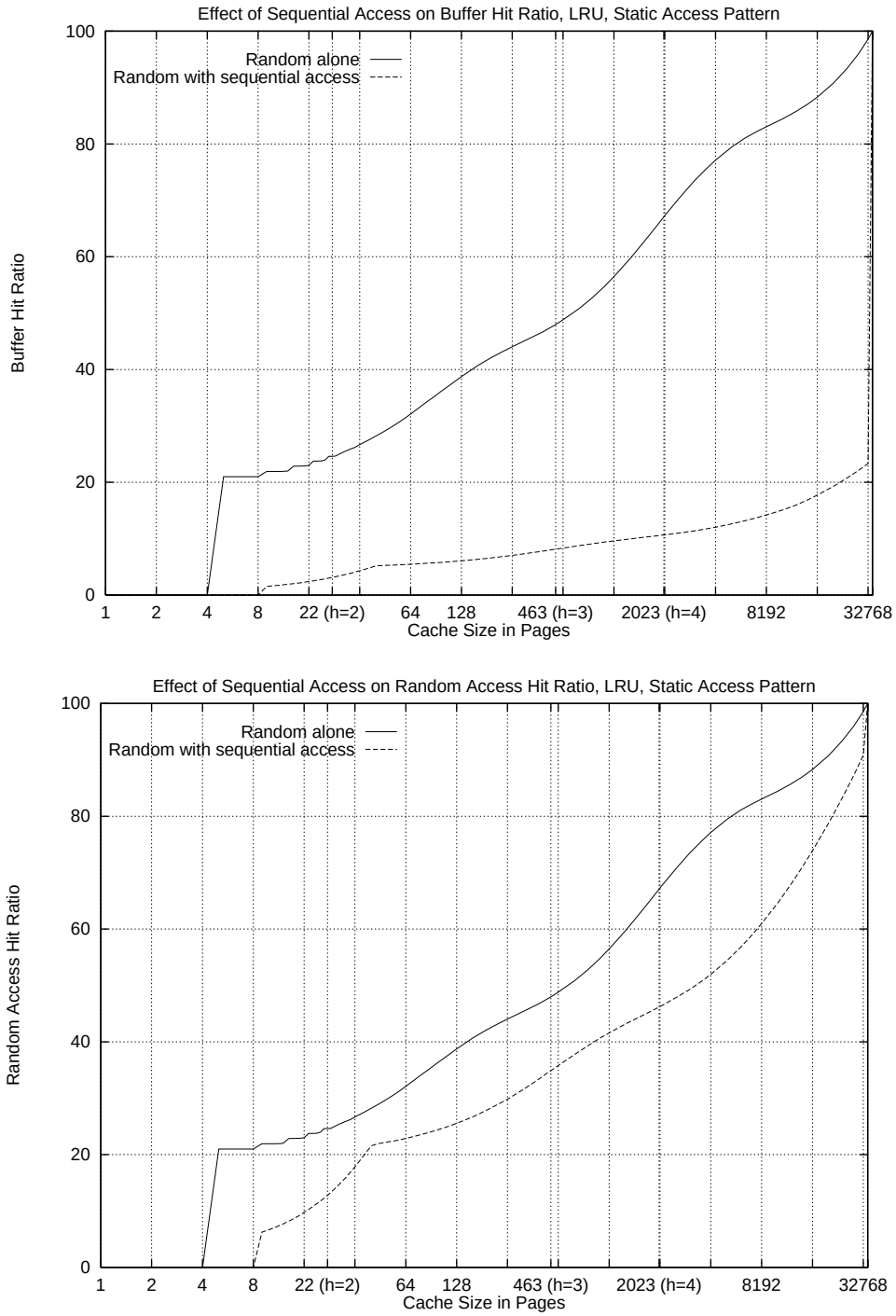



Figure 21: Effect of sequential access to the hit ratio with LRU. The cache hit ratio is difficult to compare to the non-sequential case, while ignoring the misses by the sequential access and only counting the hit ratio of the random access we can easily compare the performance of the replacement.

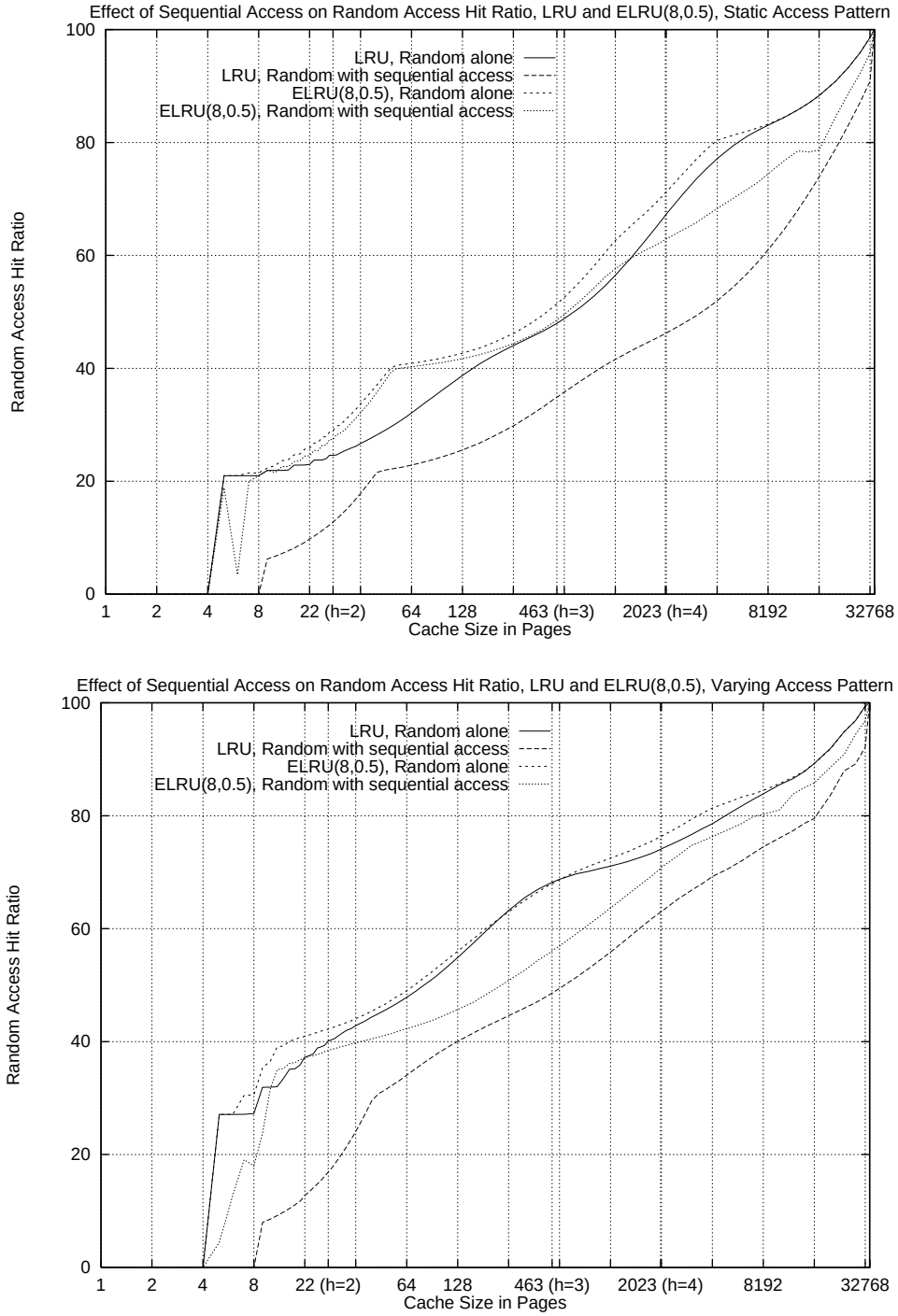



Figure 22: Effect of sequential access to LRU and ELRU(8,0.5) performance. ELRU is less affected by sequential access compared to LRU, particularly when the access pattern is static or cache size is relatively small. On tiny caches ELRU becomes unstable.

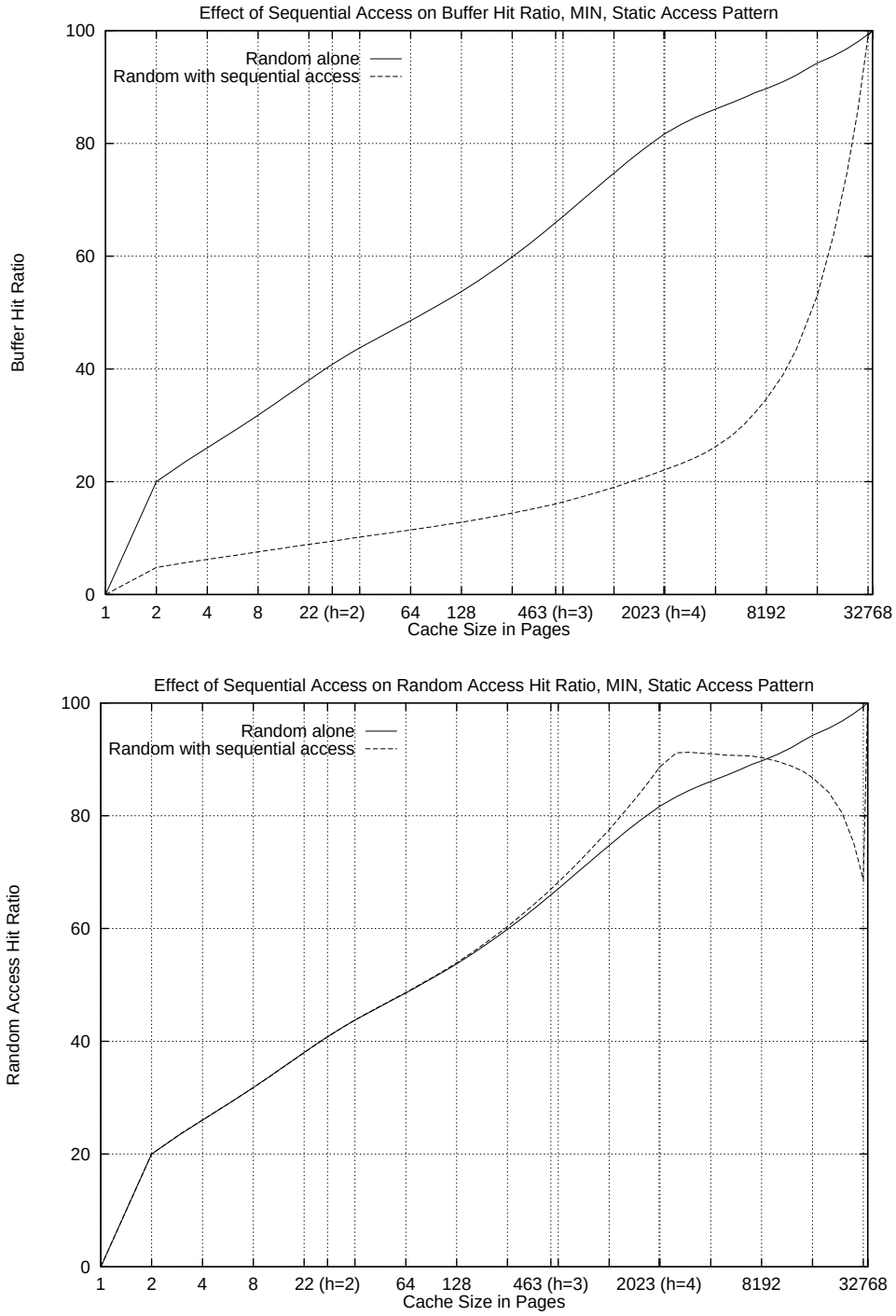



Figure 23: Effect of sequential access to the hit ratio, static access pattern, MIN. Even though the cache hit ratio monotonically increases, the hit ratio of the random tree access does not.

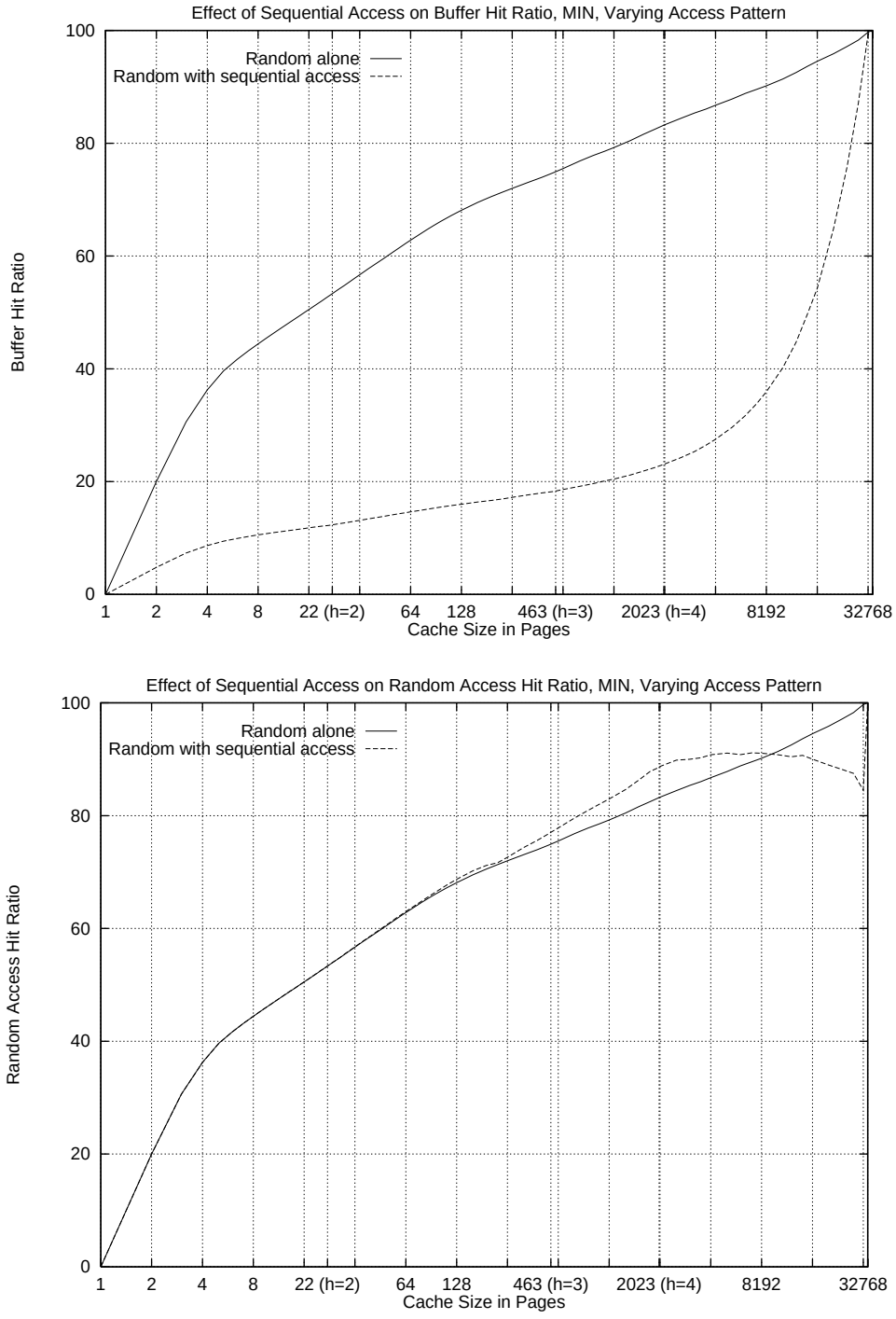



Figure 24: Effect of sequential access to the hit ratio, varying access pattern, MIN. Similarly to the static access pattern case, the hit ratio of the random tree access is unstable with large cache sizes.

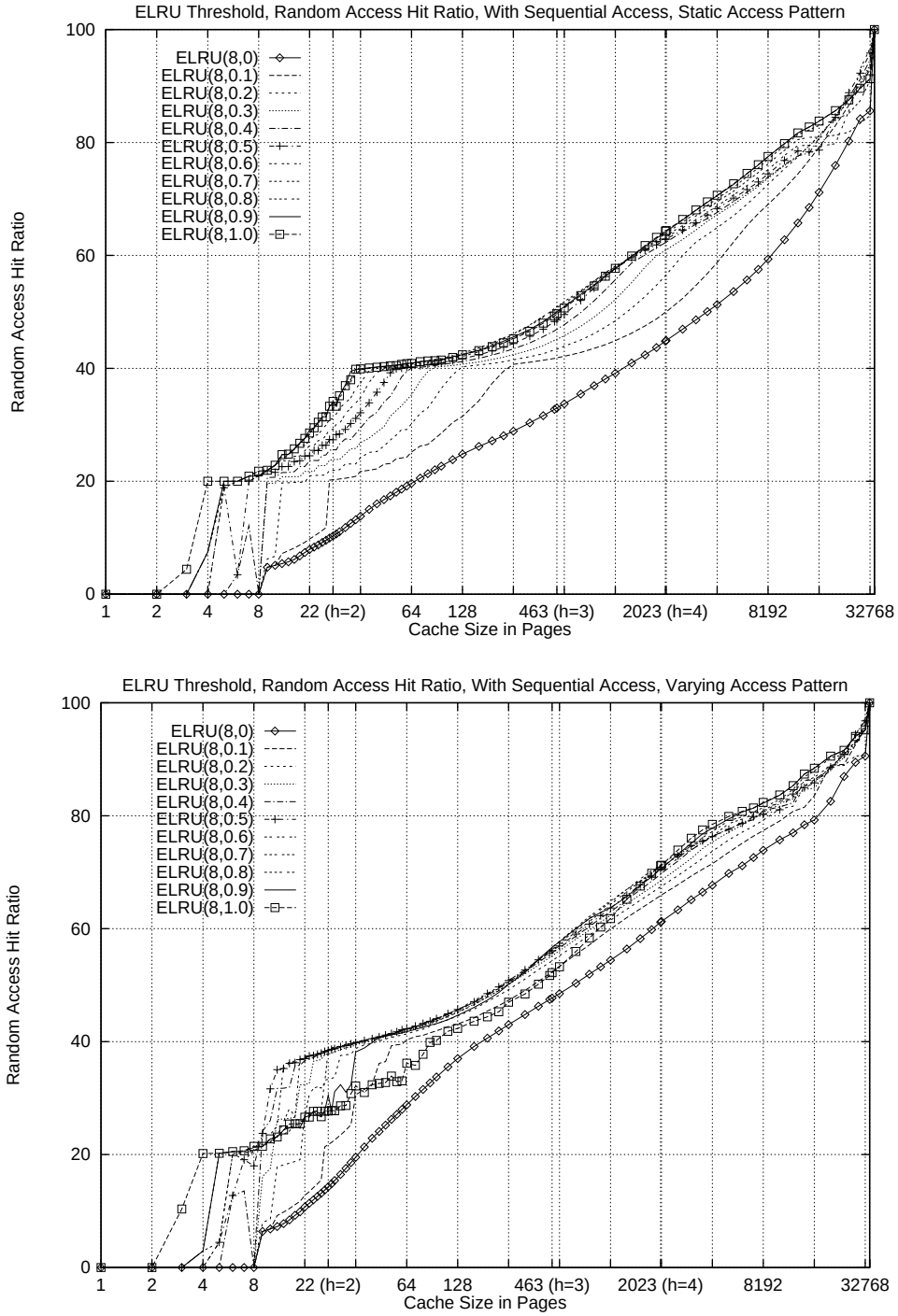



Figure 25: The effect of the ELRU threshold to the hit ratio. With static access patterns, the threshold of 1.0, the LFU case, is obviously the optimal value. With given varying access pattern, a threshold of 0.5 would perform the best at small cache sizes.

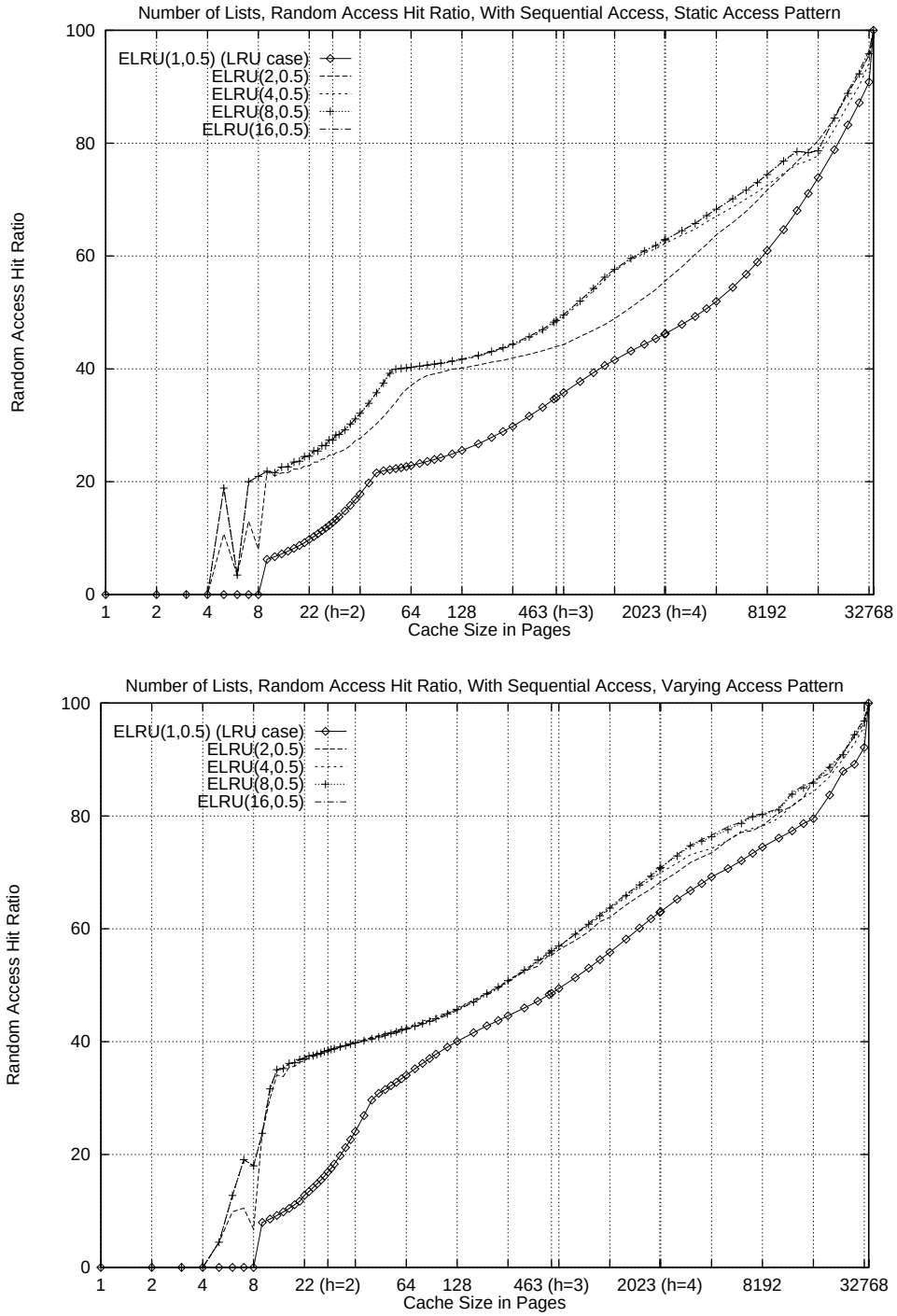
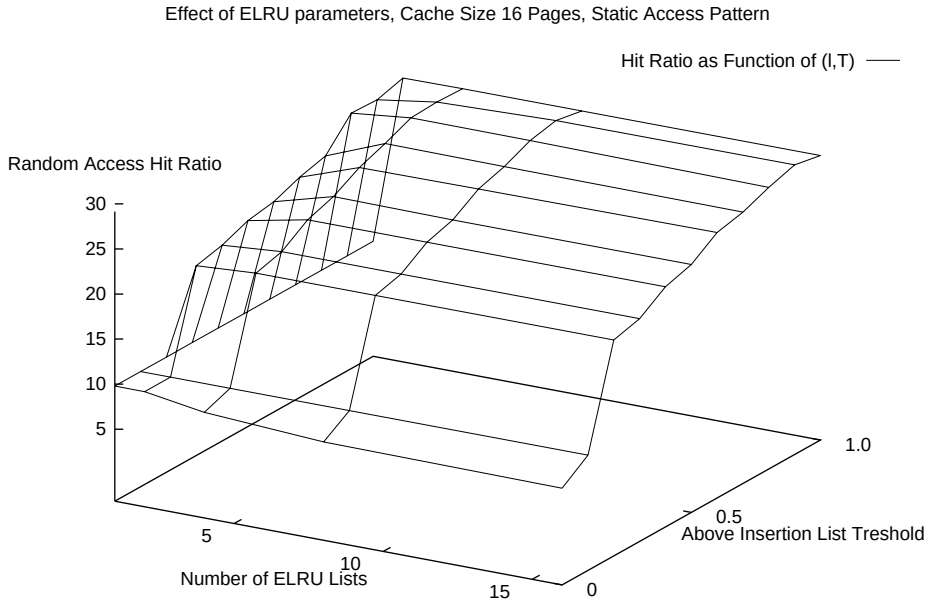



Figure 26: The effect of the number of ELRU lists to the hit ratio. The number of lists stabilizes when it reaches the number of distiguishable access frequences in the system, in this case, the number of tree levels in the table used in the benchmark (5 levels including root and leaves).



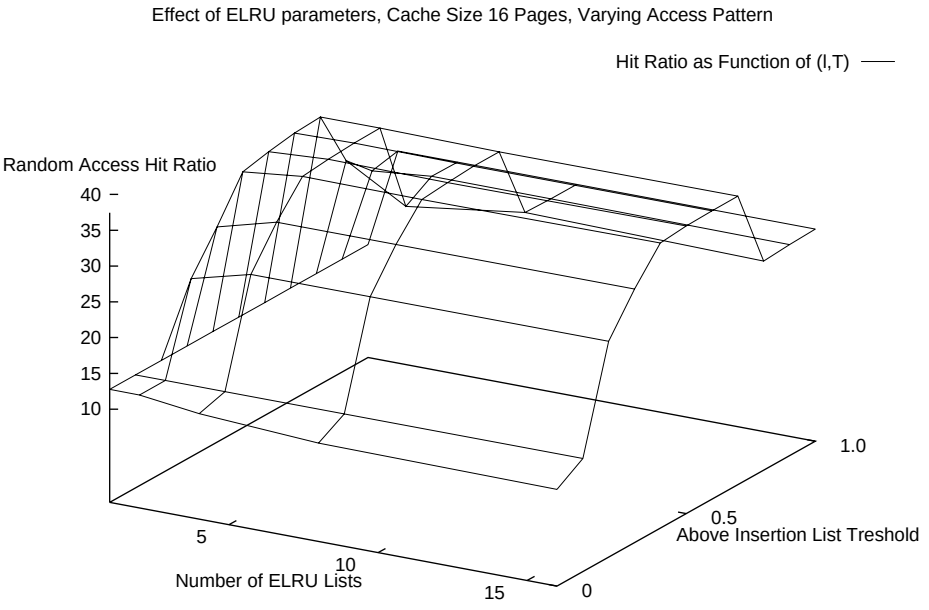




Figure 27: The effect of ELRU parameters to the hit ratio, cache size 16 pages.

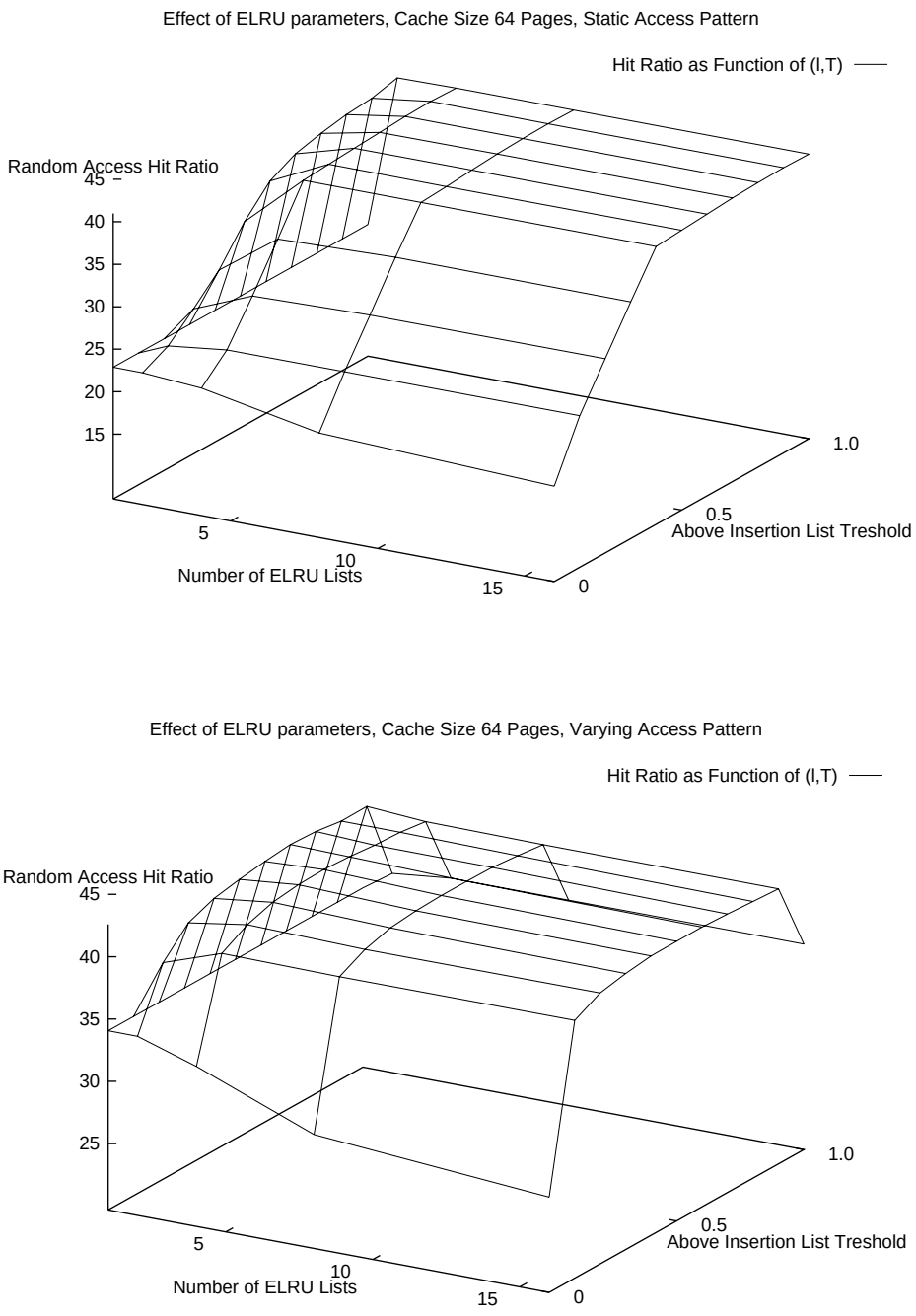



Figure 28: The effect of ELRU parameters to the hit ratio, cache size 64 pages.

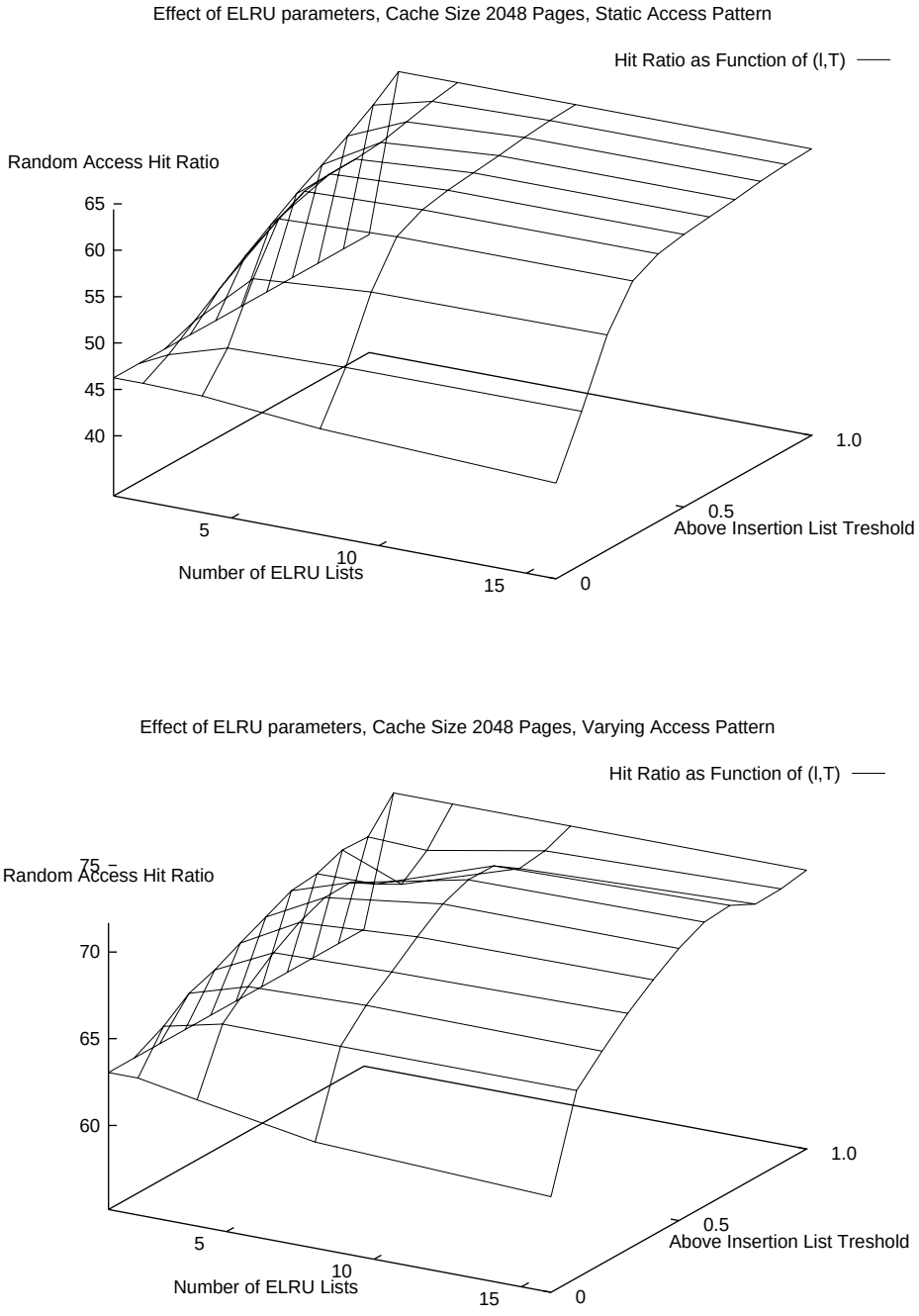
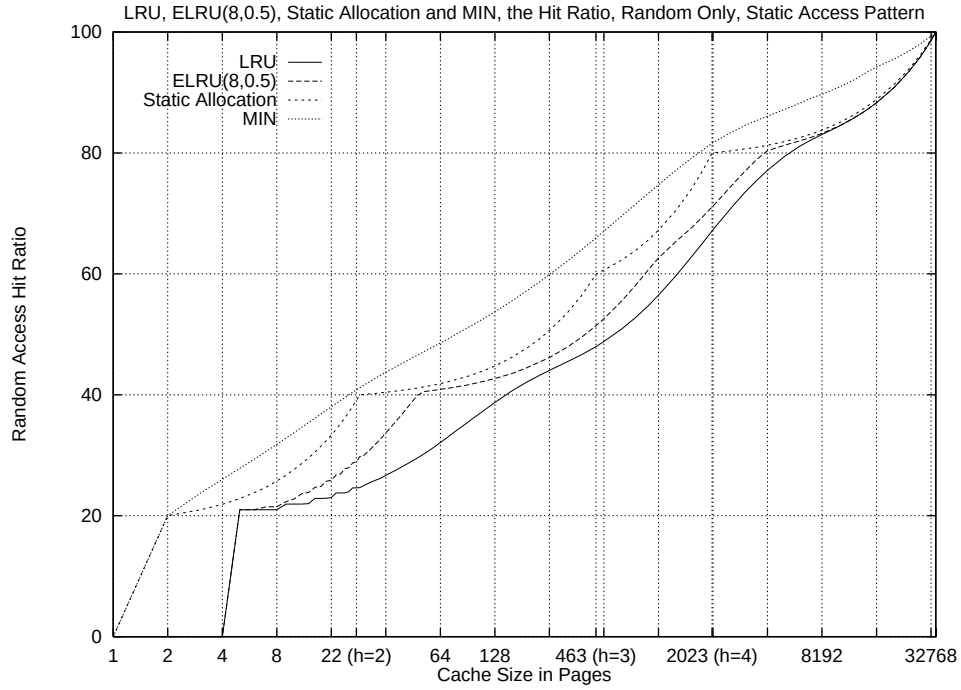



Figure 29: The effect of ELRU parameters to the hit ratio, cache size 2048 pages.



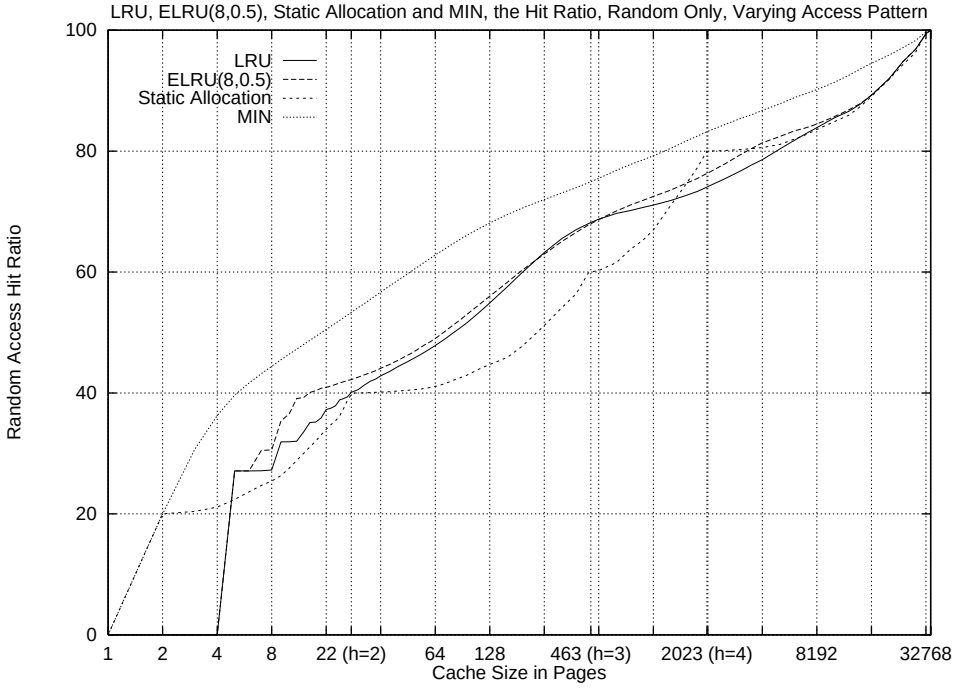
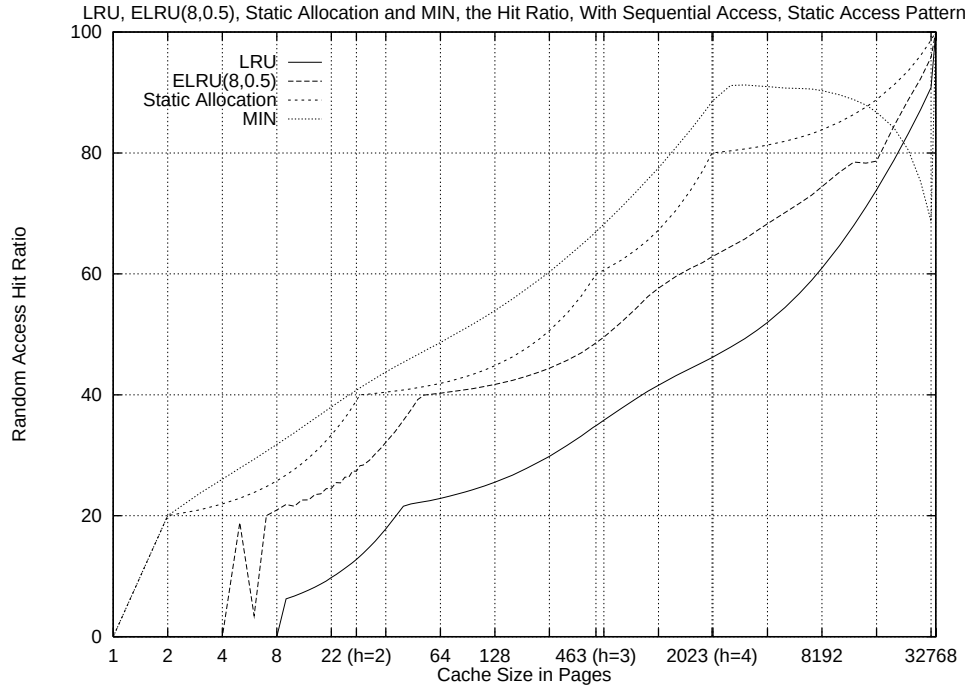




Figure 30: Comparison of all simulated algorithms, random access only.



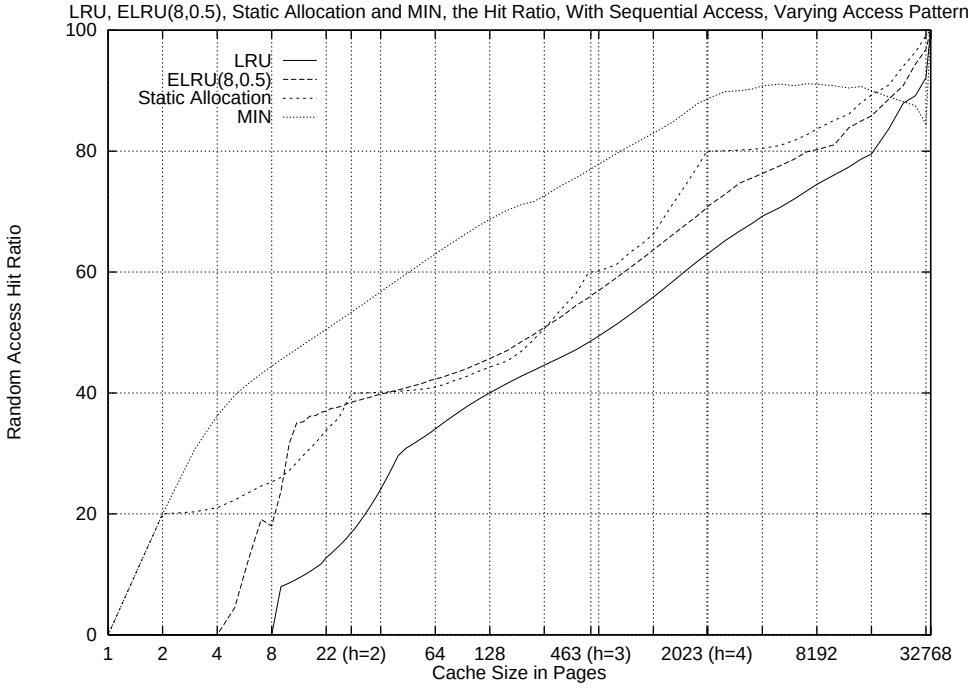




Figure 31: Comparison of all simulated algorithms, with sequential access.

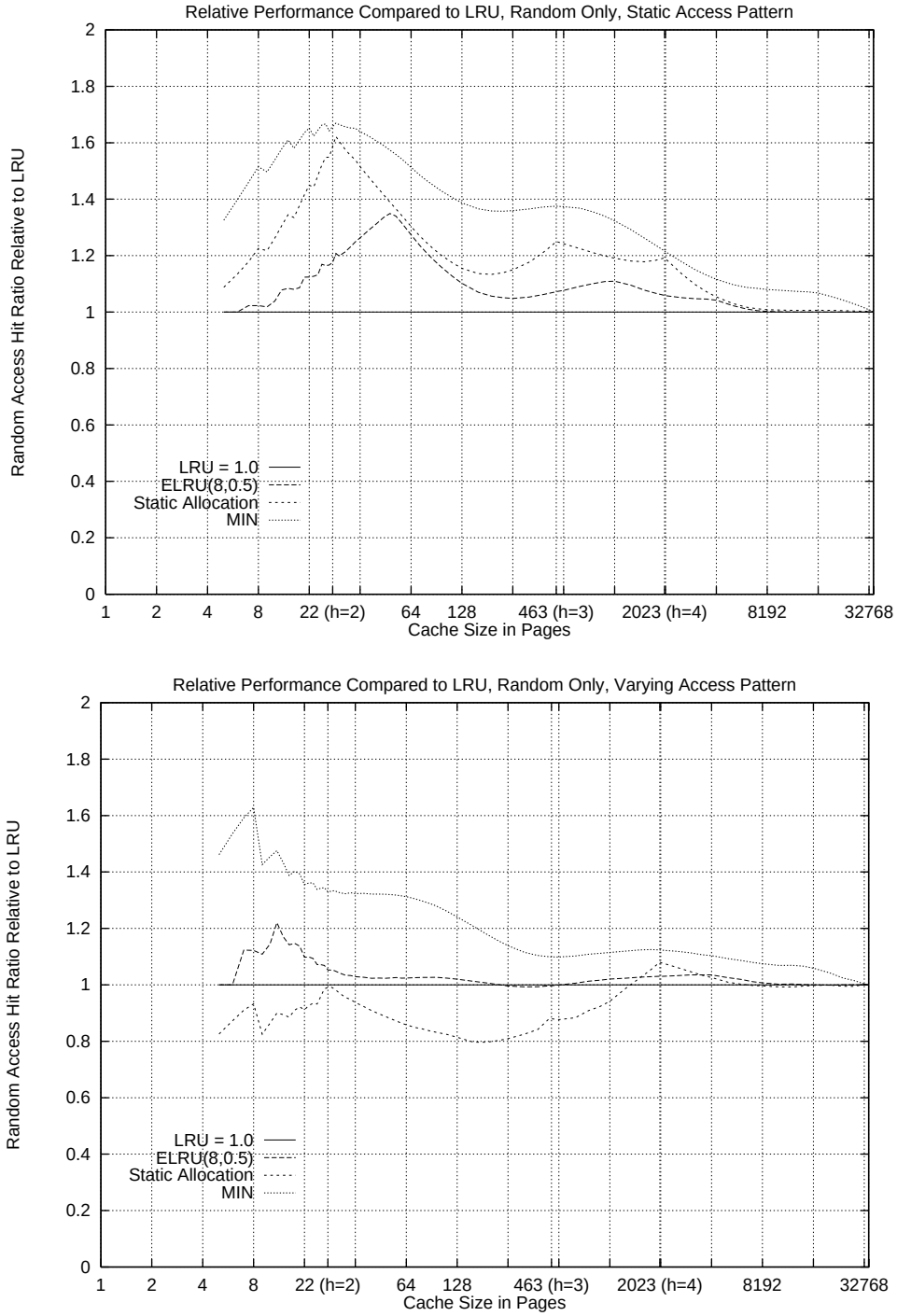



Figure 32: Comparison of all simulated algorithms, random access only, relative to LRU.

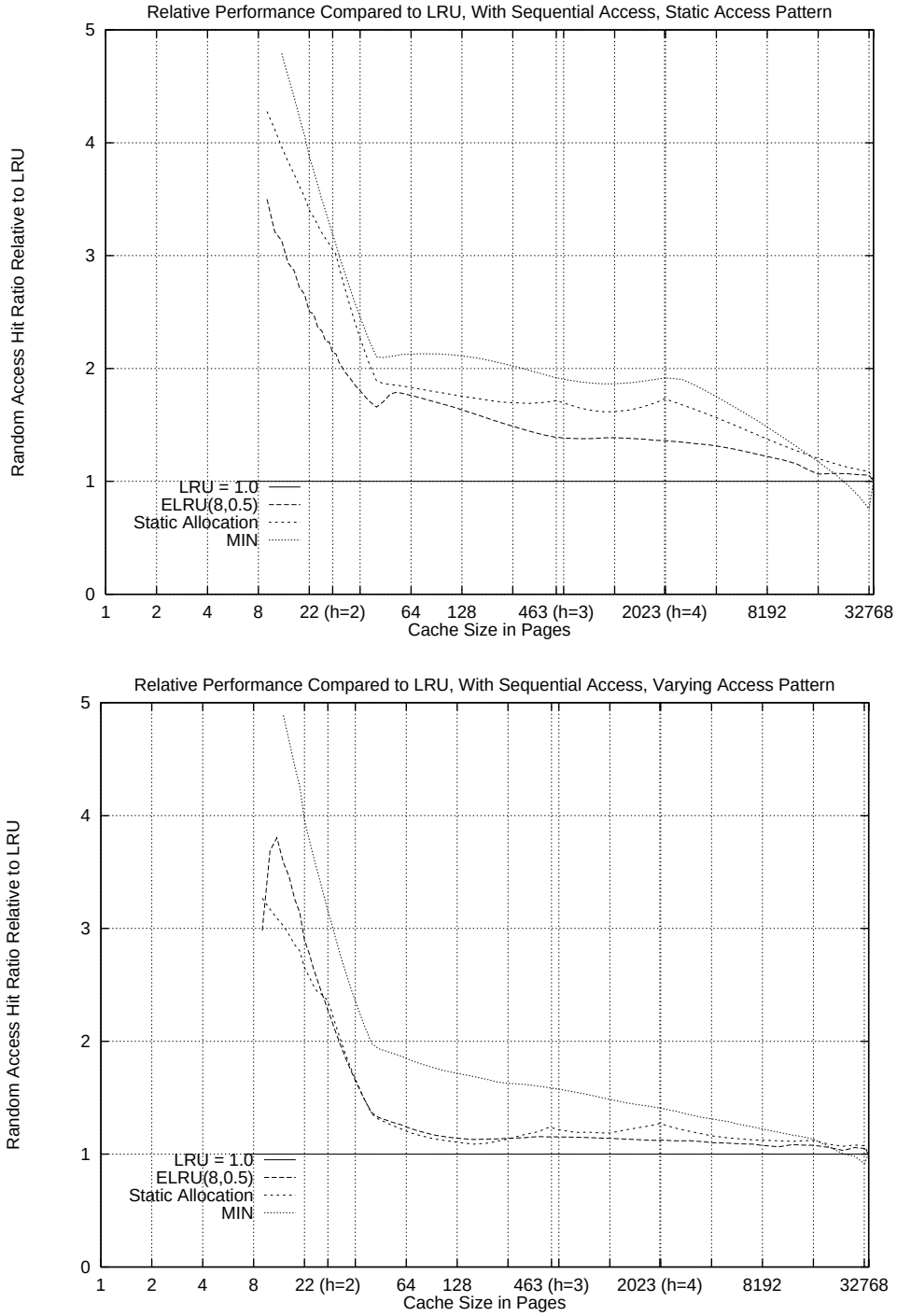



Figure 33: Comparison of all simulated algorithms, with sequential access, relative to LRU.

# B ELRU pseudocode

The data structures use by ELRU are:

**table** Rotary table of LRU lists. Each table entry contains an LRU list and count of pages in it.

**bottom_priority, insert_priority, top_priority** Indexes to the table.

**page(s)** structures for each page, containing necessary pointers and current priority value.

**pages, pages_above_insert_priority** Counters for number of pages in the ELRU structure.

*trim_lists* is a support function needed by coming ELRU functions.

**trim_lists**

```
if bottom_priority ≠ insert_priority then
  while table[bottom_priority] is empty do
    bottom_priority++;
    top_priority++;
  done
endif
if insert_priority < top_priority
    and pages_above_insert_priority − table[insert_priority].pages
      ≥ pages * threshold_above_insert_priority then
  pages_above_insert_priority − = table[insert_priority].pages;
  insert_priority++;
endif
```

ELRUinsert inserts a page into the cache.

**ELRUinsert**

```
page.priority = insert_priority;
insert_tail(table[page.priority], page);
pages++;
pages_above_insert_priority++;
```

ELRUtouch is executed for a page when it is accessed.

**ELRUtouch**

```
new_priority = page.priority + 1;
if new_priority ≤ top_priority and cache_size ≥ 2 then
  remove(table[page.priority], page);
  page.priority = new_priority;
  insert_tail(table[page.priority], page);
  if page.priority < insert_priority and new_priority ≥ insert_priority then
    pages_above_insert_priority++;
  endif
  trim_lists();
else
  remove(table[page.priority], page);
  insert_tail(table[page.priority], page);
endif
```

ELRUreplace is called to replace a page from an ELRU cache.

**ELRUreplace**

```
for priority = bottom_priority .. insert_priority do
  if table[priority] contains pages then
    page = table[priority].head;
    remove(table[priority], page);
  endif
done
```

ELRUtouch operation can be modified to get sligtly better resolution by adding two optimizations to avoid empty lists being generated by large access frequency differences:

**ELRUtouch2**

```
new_priority = page.priority + 1;
if new_priority ≤ top_priority and cache_size ≥ 2 then
  if page.priority ≠ insert_priority and new_priority ≠ insert_priority then
    if table[new_priority].pages = 1 and table[page.priority].pages = 1 then
```

```
        swap pages on table[new_priority] and table[page.priority]
    else if page.priority = insert_priority
        or new_priority = insert_priority
        or table[new_priority].pages > 0
        or table[page.priority].pages ≠ 1 then
      remove(table[page.priority], page);
      page.priority = new_priority;
      insert_tail(table[page.priority], page);
      if page.priority < insert_priority and new_priority ≥ insert_priority then
        pages_above_insert_priority++;
      endif
    endif
    trim_lists();
else
    remove(table[page.priority], page);
    insert_tail(table[page.priority], page);
endif
```

# C LRU list pseudocode

This appendix contains the implementation of LRU list using the same interfaces as used for ELRU, for comparison. The data structures necessary are a single list, and page structures for each page, similar to ELRU but no priority value.

**LRUinsert**

```
insert_tail(list, page);
```

**LRUtouch**

```
remove(list, page);
insert_tail(list, page);
```

**LRUreplace**

```
page = list.head;
remove(list, page);
```

# References


[1] R. Alonso, D. Barbara, and H. Garcia-Molina. Data Caching Issues in an Information Retrieval System. In *ACM Transactions on Database Systems*, volume 15, pages 359–384. Princeton University, September 1990.

[2] S. E. Bach, B. R. Pierce, and S. F. Saya. Modified LRU page replacement algorithm. *IBM Technical Disclosure Bulletin*, 23(8):3817–3818, January 1981.

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

[4] J. Bell, D. Casasent, and C. G. Bell. An investigation of alternative cache organizations. *IEEE Transactions on Computers*, C-23(4):346–351, April 1974.

[5] A. K. Bhide, A. Dan, and D. M. Dias. A Simple Analysis of the LRU Buffer Policy and Its Relationship to Buffer Warm-Up Transient. In *Proceedings of the Ninth Internaltional Conference on Data Engineering*, pages 125–133, Vienna, Austria, April 1993. IEEE.

[6] D. L. Black, D. B. Golub, D. P. Julin, R. F. Rashid, R. P. Draves, R. W. Dean, A. Forin, J. Barrera, H. Tokuda, G. Malan, and D. Bohman. Microkernel operating system architecture and mach. In *Micro-kernels and Other Kernel Architectures*, USENIX Workshop Proceedings, pages 11–30. USENIX Association, April 1992.

[7] M. Burrows, C. Jerian, B. Lampson, and T. Mann. On-line data compression in a log-structured file system. In *ASPLOS V Proceedings*, pages 2–9. ACM, October 1992.

[8] M. J. Carey, M. J. Franklin, and M. Zaharioudakis. Fine-grained sharing in a page server OODBMS. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, pages 359–370, Minneapolis, Minnesota, 1994. ACM.

[9] M. J. Carey, R. Jauhari, and M. Livny. Priority in DBMS resource scheduling. In *Proceedings of the Fifteenth International Conference on Very Large Data Bases*, pages 397–410, Amsterdam, 1989.

[10] R. W. Carr and J. L. Hennessy. Wsclock – a simple and effective algorithm for virtual memory management. In *Proceedings of the Eigth Symposium on Operating System Principles*, volume 15, pages 87–95, December 1981.

[11] C. Y. Chan, B. C. Ooi, and H. Lu. Extensible buffer management of indexes. In *Proceedings of the 18th International Conference on Very Large Data Bases*, pages 444–454, 1992.

[12] T.-F. Chen and J.-L. Baer. Reducing memory latency via non-blocking and prefetching caches. In *ASPLOS V Proceedings*, pages 51–61, 1992.

[13] H.-T. Chou and D. J. DeWitt. An evaluation of buffer management strategies. In *Proceedings of the 11th VLDB Conference*, pages 127–141, Stockholm, Sweden, 1985.

[14] W. W. Chu and H. Opderbeck. The page fault frequency replacement algorithm. In *AFIPS proceedings, Fall Joint Computer Conference*, pages 597–609, 1972.

[15] A. Dan, D. M. Dias, and P. S. Yu. Database buffer model for the data sharing environment. In *Proceedings of the Sixth International Conference on Data Engineering*, pages 538–544, Los Angeles, California, USA, February 1990. IEEE. ISBN 0-8186-2025-0.

[16] A. Dan, P. S. Yu, and J.-Y. Chung. Characterization of database access skew in a transaction processing environment. Research Report RC 17436 (#76194), IBM Research Division, T. J. Watson Research Center, Yorktown Heights, NY 10598, September 1991.

[17] P. J. Denning. The working set model for program behavior. *Communications of the ACM*, 11(5):323–333, May 1968.

[18] P. J. Denning. Working sets past and present. *IEEE Transactions on Software Engineering*, SE-6(1):64–84, January 1980.

[19] Y. Deville. A low-cost usage-based replacement algorithm for cache memories. *Computer Architecture News*, 18(4):52–58, December 1990.

[20] R. P. Draves. Page replacement and reference bit emulation in mach. In *USENIX Mach Symposium*, pages 201–212. USENIX Association, November 1991.

[21] W. Effelsberg. Principles of database buffer management. *ACM Transactions on Database Systems*, 9(4):560–595, December 1984.

[22] K. Elhardt and R. Bayer. A database cache for high performance and fast restart in database systems. *ACM Transactions on Database Systems*, 9(4):503–525, December 1984.

[23] E. B. Fernández, T. Lang, and C. Wood. Effect of replacement algorithms on a paged buffer database system. *IBM Journal of Research and Development*, 22(2):185–196, March 1978.

[24] M. H. Fogel. The VMOS paging algorithm — a practical implementation of the working set model. *Operating Systems Review*, 8(1):8–17, January 1974.

[25] J. Gray. *The Benchmark Handbook.* Morgan Kaufmann Publishers, Inc., 2929 Campus Drive, Suita 260, San Mateo, CA 94403, 1993.

[26] J. Gray and A. Reuter. *Transaction Processing: Concepts and Techniques.* The Morgan Kaufmann series in data management systems. Morgan Kaufmann Publishers, Inc., 2929 Campus Drive, Suite 260, San Mateo, CA 94403, U.S.A., 1993.

[27] L. M. Haas, W. Chang, G. M. Lohman, J. McPherson, P. F. Wilms, G. Lapis, B. Lindsay, H. Pirahesh, M. J. Carey, and E. Shekita. Starburst mid-flight: As the dust clears. *IEEE Transactions on Knowledge and Data Engineering*, 2(1):143–160, March 1990.

[28] P. Heidelberger and H. S. Stone. Parallel trace-driven cache simulation by time partitioning. Research Report RC 1550 (#68960), IBM Research Division, T. J. Watson Research Center, Yorktown Heights, NY 10598, February 1990.

[29] A. Hospodor. Hit ratio of caching disk buffers. In *Digest of Papers, Spring Compcon '92*, pages 427–432. IEEE, February 1992.

[30] Minimum-serialization, approximate-global-LRU, strict-local LRU, page replacement algorithm. IBM Technical Disclosure Bulletin, volume 27, number 12, May 1985.

[31] R. Jauhari, M. J. Carey, and M. Livny. Priority-hints: An algorithm for priority-based buffer management. In *Proceedings of the 16th VLDB Conference*, pages 708–721, Brisbane, Australia, 1990.

[32] j. John M. Thorington and J. D. Irwin. An adaptive replacement algorithm for paged-memory computer systems. *IEEE Transactions on Computers*, C-21(10):1053–1061, October 1972.

[33] E. E. Johnson. Working set prefetching for cache memories. *Computer Architecture News*, 17(6):137–141, December 1989.

[34] N. Kamel and R. King. Intelligent database caching through the use of page-answers and page-traces. *ACM Transactions on Database Systems*, 17(4):601–646, December 1992.

[35] K. R. Kaplan and R. O. Winder. Cache-based computer systems. *IEEE Computer*, 6(3):30–36, March 1973.

[36] P. Krueger and R. Chawla. The stealth distributed scheduler. In *Proceedings of The 11th International Conference on Distributed Computing Systems*, pages 336–343. IEEE, May 1991.

[37] M. S. Lam, E. E. Rothberg, and M. E. Wolf. The cache performance and optimizations of blocked algorithms. In *APSLOS-IV Proceedings*, pages 63–74. ACM, April 1991.

[38] T. Lang, C. Wood, and E. B. Fernández. Database buffer paging in virtual storage systems. *ACM Transactions on Database Systems*, 2(4):339–351, December 1977.

[39] S. J. Leffler, M. K. McKusick, M. J. Karels, and J. S. Quarterman. *The Design and Implementation of the 4.3BSD UNIX Operating System.* Addison-Wesley Publishing Company, Inc, Reading, Massachusetts, 1989.

[40] J. V. Lindley. Simplified LRU buffer select subroutine. *IBM Technical Disclosure Bulletin*, 26(9):4463–4464, February 1984.

[41] J. loup Gailly. algorithm.doc. included in GNU gzip 1.2.4 source code, 1993.

[42] H. Lu and K. Tan. Buffer and load balancing in locally distributed database systems. In *Proceedings Sixth International Conference on Data Engineering*, pages 545–552, Los Angeles, California, USA, February 1990. IEEE.

[43] K. Maruyama. mLRU page replacement algorithm in terms of the reference matrix. *IBM Technical Disclosure Bulletin*, 17(10):3101–3106, March 1975.

[44] D. McNamee and K. Armstrong. Extending the mach external pager interface to accommodate user-level page replacement policies. In *MACH*, USENIX Workshop Proceedings, pages 17–29. USENIX Association, October 1990.

[45] C. Mohan and I. Narang. Efficient locking and caching of data in the multisystem shared disks transaction environment. Research Report RJ 8301 (75662), Data Base Technology Institute, IBM Almaden Research Center, 650 Harry Road, San Jose, CA 95120, USA, August 1991.

[46] R. Ng, C. Faloutsos, and T. Sellis. Flexible buffer allocation based on marginal gains. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, volume 20, pages 387–396. ACM, June 1991.

[47] V. F. Nicola, A. Dan, and D. M. Dias. Analysis of the generalized clock buffer replacement scheme for database transaction processing. Research Report RC 17225 (#76312), IBM Research Division, T. J. Watson Research Center, Yorktown Heights, NY 10598, 1991.

[48] C. Nyberg, T. Barclay, Z. Cvetanovic, J. Gray, and D. Lomet. Alphasort: A risc machine sort. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, pages 233–242, Minneapolis, Minnesota, 1994. ACM.

[49] K. Oksanen. All RAM is accessible, but some RAM is more accessible than the other. In preparation, Helsinki University of Technology, Laboratory of Information Processing Science, Otakaari 1, 02150 Espoo, Finland, 1995.

[50] E. J. O'Neil, P. E. O'Neil, and G. Weikum. The LRU-K page replacement algorithm for database disk buffering. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, pages 297–306. ACM, May 1993.

[51] H. Opderbeck and W. W. Chu. Performance of the page fault frequency replacement algorithm in a multiprogramming environment. In J. L. Rosenfeld, editor, *Information Processing 74*, pages 235–241. IFIP, North-Holland, August 1974.

[52] M. Palmer and S. B. Zdonik. Fido: A cache that learns to fetch. In *Proceedings of the 17th International Conference on Very Large Data Bases*, pages 255–264, September 1991.

[53] B. G. Priave and R. S. Fabry. VMIN — an optimal variable-space page replacement algorithm. *Communications of the ACM*, 19(5):295–297, May 1976.

[54] J. L. Rivero. Method of modeling least recently used algorithms. *IBM Technical Disclosure Bulletin*, 25(4):1876–1878, September 1982.

[55] G. M. Sacco and M. Schkolnick. A mechanism for managing the buffer pool in a relational database system using the hot set model. In *Proceedings of the Eigth International Conference on Very Large Data Bases*, pages 257–262, September 1982.

[56] G. M. Sacco and M. Schkolnick. Buffer management in relational database systems. *ACM Transactions on Database Systems*, 11(4):473–498, December 1986.

[57] A. J. Smith. Bibliography on paging and related topics. *Operating Systems Review*, 12(4):39–56, October 1978.

[58] A. J. Smith. Sequentiality and prefetching in database systems. *ACM Transactions on Database Systems*, 3(3):223–247, September 1978.

[59] A. J. Smith. Cache memories. *Computing Surveys*, 14(3):473–530, September 1982.

[60] A. J. Smith. Bibliography and readings on CPU cache memories and related topics. *Computer Architecture News*, 14(1):22–42, January 1986.

[61] J. A. Solworth and C. U. Orji. Write-only disk caches. *Proceedings of the ACM SIGMOD International Conference on Management of Data*, 19(2):123–132, June 1990.

[62] J. R. Spirn and P. J. Denning. Experiments with program locality. In *AFIPS Proceedings, Fall Joint Computer Conference*, pages 611–621, 1972.

[63] H. S. Stone, J. Ture, and J. L. Wolf. Optimal partitioning of cache memory. *IEEE Transactions on Computers*, 41(9):1054–1068, September 1992.

[64] A. Swami and K. B. Schiefer. Estimating page fetches for index scans with finite LRU buffers. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, pages 173–184, Minneapolis, Minnesota, 1994. ACM.

[65] D. Thiébaut, H. S. Stone, and J. L. Wolf. Improving disk cache hit-ratios through cache partitioning. *IEEE Transactions on Computers*, 41(6):665–676, June 1992.

[66] R. Turner and B. Strecker. Use of the LRU stack depth distribution for simulation of paging behavior. *Communications of the ACM*, 20(11):795–798, November 1977.

[67] Y. Wang and L. A. Rowe. Cache consistency and concurrency control in a client/server DBMS architecture. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, volume 20, pages 367–376, June 1991.

[68] I. W.F. King. Analysis of demand paging algorithms. *Information Processing*, (71):485–490, 1972.

[69] S. J. White and D. J. DeWitt. QuickStore: A high performance mapped object store. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, pages 395–406, Minneapolis, Minnesota, 1994. ACM.

[70] W. S. Wong and R. J. T. Morris. Benchmark synthesis using the LRU cache hit function. *IEEE Transactions on Computers*, 37(6):637–645, June 1988.

[71] T. Ylönen. *Shadow Paging is Feasible.* Licentiate's thesis, Laboratory of Information Processing Science, Helsinki University of Technology, Finland, 1994.

[72] T. Ylönen and H. Suonsivu. A multi-user shadow paging transaction manager on Mach 3.0. In J. Helander and H. Saikkonen, editors, *Distributed Operating Systems and Mach: Proceedings of the Spring 1992 Graduate Seminar on Distributed Systems.* Helsinki University of Technology, Finland, 1993. Available as report 1993/TKO-C61.

[73] M. Young, A. Tevanian, R. Rashid, D. Golub, J. Eppinger, J. Chew, W. Bolosky, D. Black, and R. Baron. The Duality of Memory and Communication in the implementation of a Multiprocessor Operating System. In *The Proceedings of the 11th Symposium on Operating System Principles*, November 1987.

# D Glossary

This is a short dictionary on terms and concepts related to caches.

**Access string:** The string of data accesses made by the system; a list of pages which were referenced, in the order of reference.

**Asynchronous write engine:** A thread which takes care of writing dirty pages to the disk, usually by writing all pages which are dirty and have not been touched for a certain time. Normally the write engine works relatively independently from actual page replacement. The time limit is usually considerably lower than the age of pages actually replaced.

**Cache hit:** An access to a cache which succeeded: the data requested is in the cache.

**Cache miss:** An access to a cache which failed: the data requested is not in the cache and has to be loaded from the slower storage.

**Database access skew:** Normally a database access is more concentrated on certain data. For example, in banking applications, some accounts are much more frequently accessed than others, and branch records are more often updated than account records [16]. In addition, index pages are accessed more often than data pages. The resulting non-uniform access pattern is referred to as database access skew.

**External thrashing:** In a multitasking system, processes compete for the same set of pages. Even if the hot set of each process would fit in the memory, all hot sets combined might not. This could cause all processes miss, request a reload of their pages, which will be taken from another process, which will, in turn, be running next, and repeating the process.

**FIXing and UNFIXing:** Fixing a page means that the page is taken into use, and unfixing is releasing it. Normally, a page cannot be replaced during the time between each fix and unfix pair. Often this is characterized as "the page has got references pointing to it".

**Invalidation:** In a distributed or multiprocessor system, data items being cached may become modified in the secondary storage. Thus, the data in the cache becomes *invalid*, and it needs to be removed from the cache or reloaded into it. This is called "Invalidation". See *snooping*.

**Global LRU:** This term actually is what is now often called LRU. The reason to the term's existence is from various papers, which addressed LRU replacement which was implemented per-process or per-transaction, each of

which had its own share of the memory. Thus, when the buffer cache was combined, term "Global LRU" was coined to describe this new concept.

**Hit ratio:** The average ratio of cache hits out of the total number of references (hits+misses).

**Hot point:** In a hit ratio / cache size graph, discontinuities are visible. At these points, increasing the cache size over the point, a large difference in performance can be detected. This is due to the fact that the hot set fits in the cache completely, and thus number of cache misses is greatly reduced. There may be several hot points in a graph generated from complicated queries, for example when one more of the repeatedly accessed indices of the query fit in the cache. Performance improvement between two hot points is flat, if there is only looping and sequential access in the system. Tree structures like indexes generate smoother discontinuities. See Figures 35 and *stable interval*.

**Hot set:** The Hot set is the set of the data which is accessed frequently. Its opposite is the Cold Set, the set of data which is randomly or sequentially accessed. The difference between the hot and cold set in a real-world system is usually vague, the same data is accessed both sequentially and randomly.

**Internal thrashing:** When a process cycles through a set of pages, which is larger than the set of pages that can fit in the buffer. If an LRU page replacement algorithm is used, every new reference to a page will be a miss.

**Locality model:** A model of program and its access pattern. A simple model could be defined as "80% of accesses go into 20% of the data". Locality models are discussed in 2.2.

**LRU list:** A simple and efficient data structure for implementing LRU for buffer caches. The pages or their descriptor blocks in memory are linked together in a bidirectional linked list, and when a page is accessed, it is taken from its current position and moved into the end of the list. Thus to replace using true LRU policy, simply replace the first page in the list. See Figure 34.

**Memory pollution:** [In context of prefetching data to the cache] Many proposed prefetching algorithms are approximative, in other words, they do not know what they are supposed to prefetch, but are doing prefetching on the basis of assumptions, previous history, guesswork or similar imperfect methods. Thus, not all the data these algorithms prefetch is actually needed. This undesired effect is called "Memory pollution". The less memory pollution occurs, the better the prefetching algorithm.

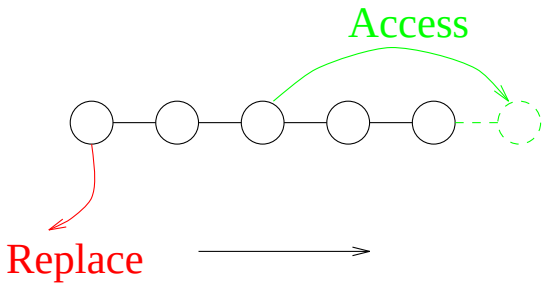



Figure 34: The LRU list. LRU replacement policy can be implemented in software using a single bidirectional list, on which the page is reinserted each time it is accessed, thus keeping the list in LRU order.

**Page fault:** A trap generated when a virtual memory which is not in the physical memory is accessed. The missing page is loaded from the swap or the file system. This is often called "Faulting in a page".

**Prefetch algorithm:** The algorithm which is used to determine which data items should be fetched into the cache in advance, before the actual request for them is made.

**Quasi-copies:** copies of data objects, which may have deviated from the original, but are still within certain limits and thus can be used for queries which do not require exact information. The term was introduced in [1]. This may also be referred as "Lazy Caching".

**Random access skew:** In a normal database system, even though accesses would be random by nature, actual access pattern to physical data is denser on certain types of information than it is on other types. This uneven access pattern is called random access skew. As an example, pages of an index may be accessed more frequently than the pages of the table the index is pointing to. Similarly in TPCB benchmark the branch table has higher access density than the accounts table.

**Replacement algorithm:** The algorithm which implements the *replacement policy*.

**Reference density:** Reference density is the frequency a page is being referenced, usually measured as number of references to the page during a certain specified time window.

**Replacement policy:** The policy about which data items are removed from the cache when space is needed for new items.

**Replacement string:** The string of data replaced from the cache, a list of pages which were replaced, in the order of replacement.

**Snooping:** A processor cache data item needs to be invalidated if any data in it is modified in memory by other processors in the same I/O or memory bus or an I/O device. This may be implemented by a technique called *snooping*. The idea is that the processor's memory interface monitors the I/O or memory bus and invalidates all addresses it sees.

**Stable interval:** In a hit ratio as a function of cache size graph, a flat area is called stable interval. This is an area, where cache size increment has little or no effect on the hit rate. Between intervals, there are discontinuities, called hot points. It may be waste to allocate more space for the cache if no next hot point can be reached by the increase. See Figure 35 and *hot point*.

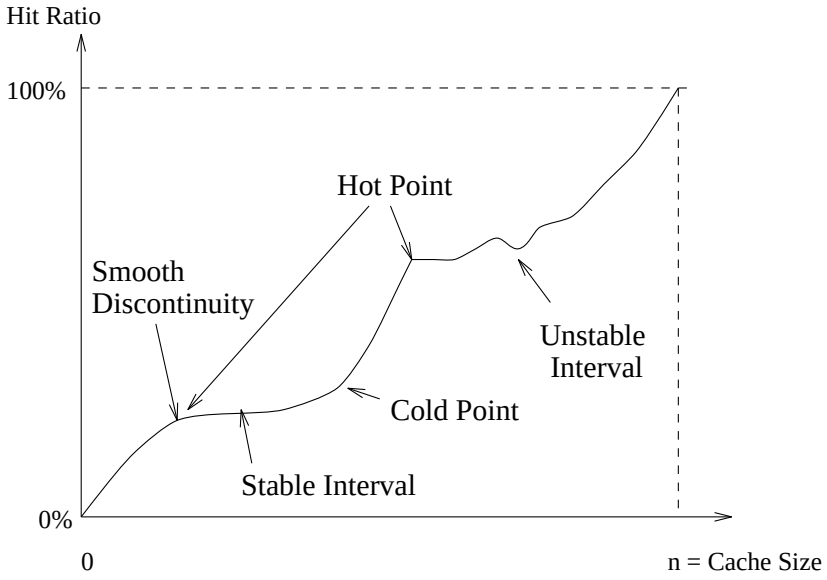



Figure 35: Hit ratio as a function of cache size. Intervals and discontinuities.

**Thrashing:** A state in which almost all cache accesses miss and the hit ratio drops near zero. This is usually due to combined effect of a failing replacement algorithm and too small cache.

**Warming up:** [In the context of caches] loading the initial data items into the cache; the time from starting up the system to the moment when the hit ratio reaches stable or "run" state. Sometimes it may be useful to load the data at the startup time, if it is known which data will probably be needed the most. The time to warm up the cache may be long [5].

# Index