Flashing Up The Storage Hierarchy

Ioannis Koltsidas

Doctor of Philosophy

Laboratory for Foundations of Computer Science

School of Informatics

University of Edinburgh

2010

Abstract

The focus of this thesis is on systems that employ both flash and magnetic disks as storage media. Considering the widely disparate I/O costs of flash disks currently on the market, our approach is a cost-aware one: we explore techniques that exploit the I/O costs of the underlying storage devices to improve I/O performance. We also study the asymmetric I/O properties of magnetic and flash disks and propose algorithms that take advantage of this asymmetry. Our work is geared towards database systems; however, most of the ideas presented in this thesis can be generalised to any data-intensive application.

For the case of low-end, inexpensive flash devices with large capacities, we propose using them at the same level of the memory hierarchy as magnetic disks. In such setups, we study the problem of data placement, that is, on which type of storage medium each data page should be stored. We present a family of online algorithms that can be used to dynamically decide the optimal placement of each page. Our algorithms adapt to changing workloads for maximum I/O efficiency. We found that substantial performance benefits can be gained with such a design, especially for queries touching large sets of pages with read-intensive workloads.

Moving one level higher in the storage hierarchy, we study the problem of buffer allocation in databases that store data across multiple storage devices. We present our novel approach to per-device memory allocation, under which both the I/O costs of the storage devices and the cache behaviour of the data stored on each medium determine the size of the main memory buffers that will be allocated to each device. Towards informed decisions, we found that the ability to predict the cache behaviour of devices under various cache sizes is of paramount importance. In light of this, we study the problem of efficiently tracking the hit ratio curve for each device and introduce a low-overhead technique that provides high accuracy.

The price and performance characteristics of high-end flash disks make them perfectly suitable for use as caches between the main memory and the magnetic disk(s) of a storage system. In this context, we primarily focus on the problem of deciding which data should be placed in the flash cache of a system: how the data flows from one level of the memory hierarchy to the others is crucial for the performance of such a system. Considering such decisions, we found that the I/O costs of the flash cache play a major role. We also study several implementation issues such as the optimal size of flash pages and the properties of the page directory of a flash cache.

Finally, we explore sorting in external memory using external merge-sort, as the latter employs access patterns that can take full advantage of the I/O characteristics of flash memory. We study the problem of sorting hierarchical data, as such is necessary for a wide variety of applications including archiving scientific data and dealing with large XML datasets. The proposed algorithm efficiently exploits the hierarchical structure in order to minimize the number of disk accesses and optimise the utilization of available memory. Our proposals are not specific to sorting over flash memory: the presented techniques are highly efficient over magnetic disks as well.

Acknowledgements

During my stay in Edinburgh for the last three years, both my life and my work have been greatly enhanced by the support and friendship of certain people. Thanking them in the following few lines is the least I can do for them; never alienating them and always offering them my friendship and support, is the only way I can show them my gratitude.

Stratis Viglas was my advisor and the person who has influenced my research and work more than any one else in my whole academic life. In the first place, he taught me how to learn what is truly valuable and helped me stay focused on my work when I was losing my target. He has been a tremendous source of inspiration and motivation. His work ethics have shown me beyond any doubt how research would be conducted in an ideal world. His theoretical and technical insight has always helped my work advance substantially when I found myself stuck. The most pleasant surprise in Edinburgh so far, though, was that Stratis has been to me a father and a true friend. I will never forget his friendship and support, both at times of joy and sorrow, and I truly am grateful to him for that. Most of all, I would like to thank him for the journey and for the time he has devoted to me. I will never forget that.

Peter Buneman has been my second supervisor and I found out quite early how lucky I am for that. He has always offered me solid scientific advice and insight. His office door has always been open for me and my questions and I am very thankful for that. All the members of the database group have always offered me their support; the fruitful discussions I have had with them have played a major role in shaping my own ideas. I am grateful to Wenfei Fan for his comments and insight. I am grateful to Heiko Müller for all the hard work he has put into our collaboration and for being open to my ideas and tolerant to my naive questions. Many thanks also to Anastasios Kemensietsidis, Irini Foundulaki and Floris Geerts for helping me out when I have needed it. I would also like to thank my fellow students, Vasileios Vassaitis, Shuai Ma and Shunichi Amano for their friendship and all the good discussions we have had at the lab. I would also like to express my gratitude to the School of Informatics for giving me the opportunity to pursue a PhD and for the financial support it has offered me during my studies; I also very thankful to the Engineering and Physical Sciences Research Council for that.

I owe a lot to my professors at the National Technical University of Athens, especially to professor Timos Sellis, who was my diploma thesis supervisor and a major influence towards my decision to pursue a PhD at the University of Edinburgh. Had it not been for him, I certainly would not be in this position now. I thank him for his teaching, his academic and technical guidance and his willingness to help me, even when I bothered him with my trivial questions. I also owe a lot to professor Nikos Papaspyrou, for introducing me to the world of research early on in my undergraduate studies and inspiring me passion for it. I thank him for his teaching and openness.

During my PhD I spent two summers working as a research intern at the IBM Almaden Research Center in San Jose, California. Working with the storage group there has had tremendous influence on my work, as it helped me become more focused and productive and showed me how different real-world problems are from academic ones. I am grateful to Prasenjit Sarkar for making me a member of his team and sharing with me his technical knowledge, his experience and his advice, which I have found most useful for my work and life. I am also thankful to Karan Gupta for being my mentor and helping me out with every problem that hindered my efforts. Many thanks to Prashant Pandey and Himabindu Pucha for their collaboration and guidance they offered me.

Konstantinos Krikellas has been my flatmate and officemate for two and a half years. I am most thankful to him for the friendship and support he has offered me and, primarily, for putting up with my character. My work and life in Edinburgh would be much different had it been not for him. With Savvas Makalias and Apostolos Apostolidis we have also shared common office space for most of my time in the university. My work life would not have been as fun if it had not been for them; I am thankful to them for their friendship. Most of the joy and truly happy times I have had in Edinburgh were due to the aforementioned three people and Manos Glynos, Konstantinos Kyritsis and Gerasimos Skouvaklis. The countless hours we have spent together made me feel really at home and showed me how valuable friends can be in life. I would also like to thank Georgia Vakalopoulou, Georgia Papakleovoulou, Iro Fourtouna, Sofia Pediaditaki and Panagiotis Douvaras for their friendship. I feel really honoured to have had such friends and could not ask for a better group of people to be part of during my studies in Edinburgh. I wish all of them best of luck in whatever they do.

I am truly grateful to Panagiotis Papadimitriou for his friendship. Together with Panagiotis we have made our first steps in the academia and in research. He has always been a true friend and I thank both him and his wife, Maria Daltayanni, for the times we have spent both in Greece and in California during my internships. I wish they are always as happy as I left them. During my internships, I got to have some really close friends at Stanford. Especially, I would like to thank Georgios Papadimitriou, Dimitris Tsamis, Yannis Antonellis, Dimitris Antonellis, Foivos Karachalios and Marcela Junguito. I wish them all the best.

I can only be grateful to Thomas Daskalakis, one of my closest friends in life since our high-school years, and I can only feel happy for that. He has always been there for me; I can only hope the same for myself and him. I also feel truly happy for having met Konstantina Palla. She is probably the person I have had the most and lengthiest on-line conversations with. Hours and hours she has stood by me and given me support and true joy. I thank her for being so open with me and letting me get to substantially know her.

My life in Edinburgh would have been much worse if it had not been for Myrto Kanakaki. Although she was not physically present, only her existence back home was enough to calm me and fulfil my life. She has given me all I could ask for from a partner in life and taught me how to be practical, organised and cheerful about my life. To Myrto I owe some of my happiest moments in life; her patience and personal sacrifice in certain circumstances have been invaluable to me. I am grateful to her for everything we lived together. She will always have a spacial place in my heart.

Above all, I thank my loving family for everything they have provided me with; I would not be the same person had it not been for them. My siblings Panagiotis, Sissy and Maria and my parents, Evangelos and Drosoula, have always done the best for me, offering their unconditional love and support I needed. My parents have always set the example for me, showing me how life is about other things apart from professional and academic achievements and at the same time supporting this and all my endeavors. My siblings have done everything in their power to help me in all ways imaginable. My twin sister, Maria, has always understood me and stood by me, probably before we were even born. It is their love and affection that keeps me going through my hardest times and that has made me miss home so much. I can only hope that my family is as proud of me as I am of them.

Declaration

I declare that this thesis was composed by myself, that the work contained herein is my own except where explicitly stated otherwise in the text, and that this work has not been submitted for any other degree or professional qualification except as specified.

(*Ioannis Koltsidas*)

To my parents,
Evangelos and Drosoula.

Knowledge comes through suffering.

– Aeschylus

Table of Contents

1 Introduction **1**

1.1 Flash Memory . . . 2

1.2 Flash Disks . . . 5

1.3 Hybrid Systems . . . 10

1.3.1 Data Placement . . . 11

1.3.2 Buffer Allocation . . . 11

1.3.3 Caching on flash memory . . . 12

1.3.4 Sorting hierarchical data in external memory . . . 12

1.4 Organisation . . . 13

2 Related Work **15**

2.1 Flash Disk Performance Evaluation . . . 15

2.2 Accelerating Random Writes . . . 16

2.3 Databases over Flash Memory . . . 20

2.3.1 Database Storage . . . 20

2.3.2 Indexing . . . 21

2.3.3 Query Execution . . . 22

2.3.4 Main Memory Buffering . . . 24

2.3.5 Caching on Flash Memory . . . 26

3 Data Placement **27**

3.1 Preliminaries . . . 27

3.2 Page Placement . . . 31

3.2.1 Conservative Algorithm . . . 31

3.2.2 Optimistic Algorithm . . . 34

3.2.3 Hybrid Algorithm . . . 36

3.3 Experimental study . . . 38

3.3.1 Experimental Setup 38
3.3.2 Impact of using both disks 40
3.3.3 Comparison of page placement algorithms 41
3.3.4 Mixed workloads 46
3.4 Discussion 47

4 Buffer Allocation 49
4.1 Introduction 49
4.2 Static Allocation 52
4.2.1 The effect of λ 55
4.3 Dynamic Allocation 58
4.3.1 Theoretical Background 58
4.4 Problem Formulation 60
4.4.1 Eviction Cost Classes 61
4.4.2 Adaptability 64
4.5 Algorithms 65
4.5.1 Baseline Resizing Algorithm 65
4.5.2 Reference-based Utility Criterion (RUC) 68
4.5.3 Hit-based Utility Criterion (HUC) 69
4.5.4 Hit Ratio Curve Aware Resizing (HRCA) 73
4.6 Discussion 80
4.7 Experiments 88
4.7.1 Stack Distance Measurement 90
4.7.2 Synthetic Workloads 94
4.7.3 TPC-C Workloads 101
4.7.4 Optimal Allocation 103

5 Caching On Flash Memory 107
5.1 Introduction 107
5.2 Problem Statement 108
5.3 Page Flow Schemes 110
5.3.1 The inclusive scheme 110
5.3.2 The exclusive scheme 112
5.3.3 The lazy scheme 113
5.3.4 Comparison 116
5.4 Implementation Issues 118

5.4.1 Using larger pages for flash 118
5.4.2 Storing the page directory on flash 121
5.4.3 How much flash? How much RAM? 122
5.5 Experimental Study 122
5.5.1 Impact of Cache Size on Hit Ratio 123
5.5.2 Impact of Flash Cache Size on RAM Hit Ratio 124
5.5.3 Validation of the Cost Formulae 125
5.5.4 Comparison of Page Flow Schemes 127
5.5.5 Impact of Flash Block Size 131
5.5.6 Caching Only Clean Pages 132
5.6 Discussion 133

6 Sorting Hierarchical Data In External Memory **137**
6.1 Introduction 137
6.2 Related Work 141
6.3 Sorting Hierarchical Data 143
6.3.1 Sort Keys 143
6.3.2 The HERMES Algorithm 146
6.4 Algorithm Analysis 154
6.4.1 Run Size 155
6.4.2 I/O Behavior 157
6.4.3 Improvements 158
6.5 Experimental Results 161
6.5.1 Impact of Input Size 161
6.5.2 Impact of Available Main Memory 163
6.5.3 Impact of Tree Depth 164
6.5.4 Impact of Key Length 165
6.5.5 Performance over a flash disk 166

7 Conclusions **169**
7.1 Future Work 174

8 Experimental Data **177**

Bibliography **193**

Chapter 1

Introduction

In this thesis we explore the design of systems equipped with both magnetic and flash disks. In particular, we study how a database system, or any other data-intensive application can take advantage of the properties of flash disks in a hybrid setup to increase I/O performance.

Flash memory has emerged as a high-performing and viable alternative to magnetic disks for data-intensive applications. In the near future, commodity hardware is expected to incorporate both flash and magnetic disks as storage media. In light of this development, fundamental principles of data storage and management need to be revisited, as all existing database systems and algorithms have been designed with disks consisting of rotating platters in mind: as such, they aim to avoid random access patterns at all cost. This is no longer a requirement if the systems are to operate over flash memory or over both magnetic and flash disks. What is more, traditional algorithms do not account for the update in-place inefficiency of flash memory and assume symmetric read and write costs: this is not at all the case for flash memory. The work presented in this thesis is geared towards hybrid storage systems, *i.e.*, systems equipped with both flash and magnetic disks. In particular:

- We explore how a flash disk can be used at the same level of the memory hierarchy as a magnetic disk in a hybrid setup and propose algorithms to adaptively decide placement of data in the most efficient medium, according to the workload of data.

- We study the problem of main memory buffer allocation in systems that store data across multiple storage devices to minimize the I/O cost by taking into account both the cache behaviour of the workload of the system and the I/O costs of the storage devices.

- We present an analytical study about how a system can utilize a flash disk as a cache layer between the main memory buffer pool and the magnetic disk(s).
- We study the problem of sorting hierarchical data in external memory using a generalisation of external merge-sort, as the latter results in access patterns that can take full advantage of the merits of flash memory. Our techniques are not specific to flash-equipped systems, but generally applicable to sorting tree-structured databases.

1.1 Flash Memory

Flash chips. Flash memory is a type of electronic memory that stores information in arrays of memory cells, called *flash cells*. Flash memory cells are made of floating-gate transistors; each transistor has two gates, a control gate and a float gate. A flash memory cell is shown in Figure 1.1. The float gate is insulated all around by an oxide layer and is placed between the control gate and the MOSFET channel. The control gate is on top of it. The oxide layer serves to electronically isolate the float gate and thus the gate can trap any electrons placed on it for a very long time, ten or more years depending on the environmental conditions. With the float gate charged, the electric field from the control gate is partially cancelled and the threshold voltage of the cell changes. In order to read the stored bit, a voltage between the possible threshold values is applied to the control gate. Then, the charge on the float gate determines whether the MOSFET channel will become conducting or remain insulating. As a result, by sensing the flow through the channel one can reproduce the stored value.

The type of flash cells that can only sense the presence or absence of current flow through the channel are referred to as *Single Level Cells* and devices that use them are therefore called SLC devices. Each single level cell can thus store one bit of information. Flash cells that can sense the amount of current flow through the channel are called *Multi Level Cells* and the devices that use them are called MLC devices. By sensing the amount of current flow, the cell is able to precisely determine the level of charge on the float gate; therefore such cells typically utilize four levels of voltage. As a result, each one of these cells can store two bits of information, which leads to MLC devices that are much more dense than SLC ones. However, both reading and writing to a multi level cell takes longer than a single level one, as there are more voltage levels

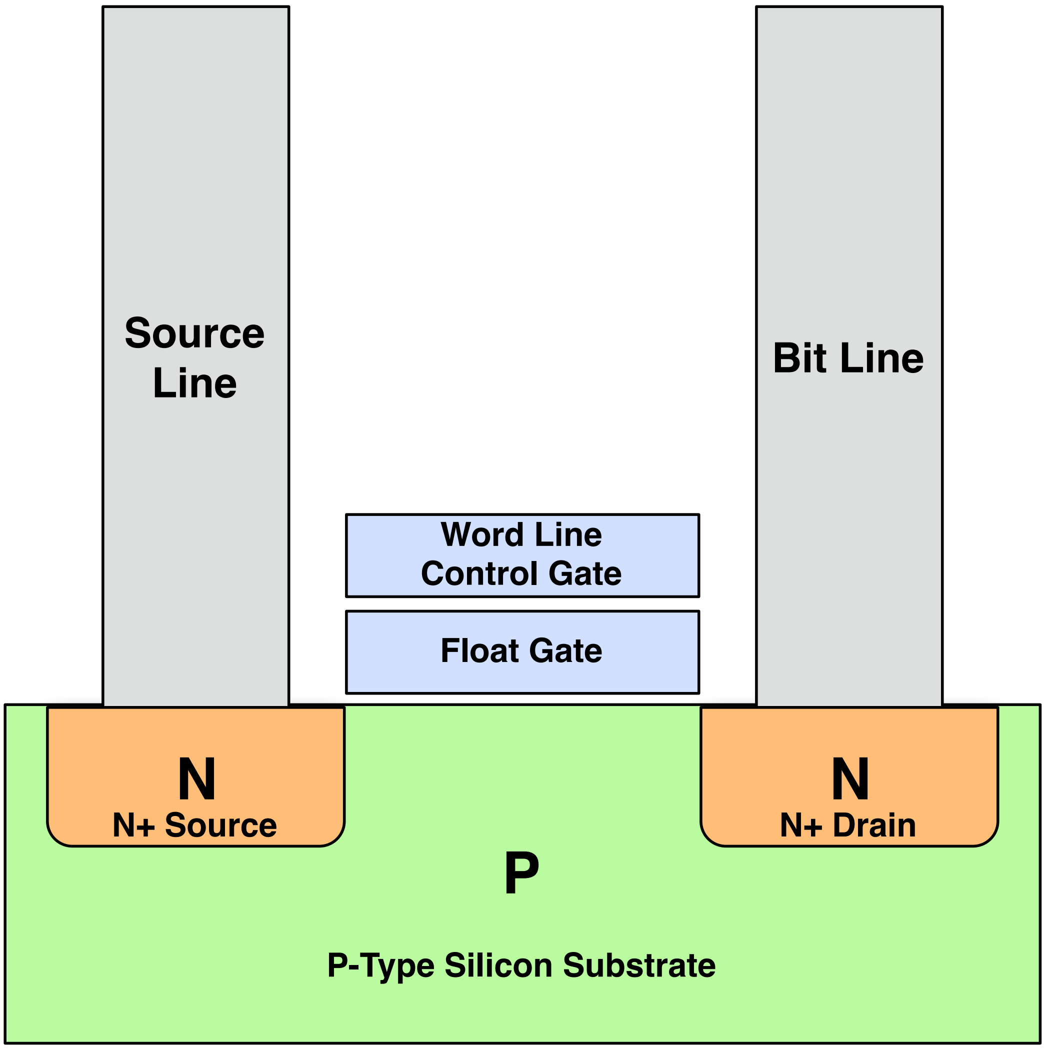


Figure 1.1: The structure of a flash memory cell.

to be dealt with. For these reasons, SLC devices are mainly used for high performance purposes, while MLC ones typically apply where large capacity is required.

Flash memory is divided into NOR flash and NAND flash, depending on how flash cells are connected to form arrays. Since only NAND flash is suitable for storage devices, however, for the rest of this work we refer to NAND flash simply as “flash” and only focus on that type of flash memory. Groups of cells are organized into pages, with the size of a page being typically 2kB or 4kB in most modern devices. A flash page is the smallest structure that is either readable or writable on a flash disk. An empty page has all its bits set. Writing the page implies resetting some of its bits (turning them to 0). The opposite, *i.e.*, setting some of the bits in the page, is not possible; rather, the whole page has to be reset before it can be re-written to. Resetting all the bits of

Operation	SLC NAND flash (μs)	MLC NAND flash (μs)
Read page	25	50
Erase block	2000	2000
Write erased page	250	900

Table 1.1: Operation costs for typical flash chips [AnandTech, 2009].

a page is referred to as *erasing*, but as explained below, cannot be done on a per-page basis. Flash pages are grouped together into blocks, with a block typically consisting of 128 pages (although 64 or 256 pages per block are quite common as well). A flash block is the smallest structure of flash memory that can be erased. Consequently, after a block has been erased, each one of its pages can be written to only once. In order to overwrite a page, the whole block needs to be erased first. The typical read, write and erase costs for a bare chip are shown in Table 1.1, both for SLC and MLC flash. The electrical properties of flash cells result in read operations being much faster than write operations: when the value of a NAND cell is changed, it takes some time before it reaches a stable state. Erasing takes two orders of magnitude more time than reading and one order of magnitude more than writing. Therefore, the need to erase incurs a severe penalty on flash chip performance; details are discussed later on in this work.

Wearing. An important limitation of flash memory is that each flash block is only capable of a finite number of erase cycles. After that, the block becomes unerasable and therefore unusable, as it can never be re-written to. This limitation is referred to as *memory wearing*. SLC chips have better write endurance than MLC ones; a typical value for an SLC chip is 100k erase cycles, while of an MLC one it is about 10k erase cycles. Lately, devices that use SLC flash chips capable of 1M erase operations have been announced [Micron Technology Inc., 2008].

Prices. The price of flash memory has been rapidly declining over the last few years, at an annual rate of 30%-40% or more [StorageSearch.com, 2006]. Market analysts predict that this price drop will continue for the coming years, although it will not be as dramatic [Denali, 2009]. The per-gigabyte price of bare MLC flash memory chips is shown in Figure 1.2 for the last few years; projected prices are also shown for the near future [StorageSearch.com, 2008], [Denali, 2009]. From more than 1000 $/GB, the price of MLC flash memory has now dropped to less than 2 $/GB and is expected to reach 0.5 $/GB by the end of 2013. Projections for SLC prices follow a similar trend, being about twice as expensive as for MLC chips. The cost of flash memory is thus

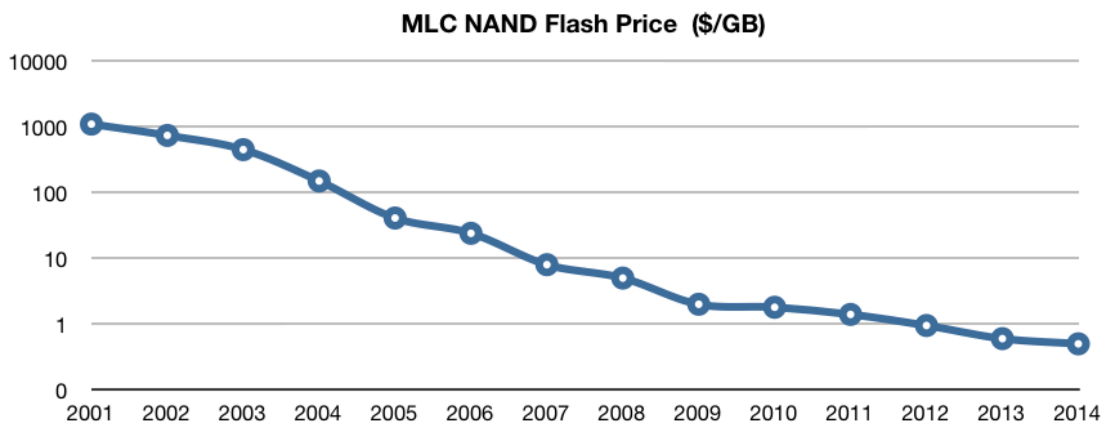


Figure 1.2: Flash memory price trend.

approaching that of magnetic disks, thereby making SSDs a cost effective alternative to traditional storage media.

1.2 Flash Disks

Flash disks, also known as *Solid State Drives*, or SSDs, are storage devices that package multiple flash memory chips into a single enclosure. Apart from the bare chips, such devices also include a controller, supported by several DRAM buffers. The controller translates system read and write requests to actual read, erase and write operations on the flash chips. In addition, the disk controller employs sophisticated algorithms and techniques (discussed later on) to hide the complexities of flash memory from the user and improve I/O performance. A sample flash disk, in the form factor of 3.5 inches in diameter, is shown in Figure 1.3 (as shown in [Wikipedia, 2009]).

The most important characteristics of flash disks can be summarised as follows:

- **I/O interface**. At the operating system level, SSDs behave identically to magnetic disks, as they are accessed through the same I/O interface. The on-disk controller provides a standard I/O interface, usually either an IDE or a SATA one, and translates system commands to flash memory operations. Typically, the unit of I/O operations on a flash disk is a sector of 512 bytes, which is equal to the size of a magnetic disk sector (for most magnetic disks).

- **No mechanical latency.** Flash disks are purely electronic devices and have no mechanical moving parts. The time needed to access a data item on a flash disk is independent of its position on the physical medium, *i.e.*, access latency does

Figure 1.3: A sample flash disk.

not depend on the access pattern. Additionally, access latency is orders of magnitude less than the random access latency for mechanical disks. Both properties present great opportunities for performance gains over magnetic disks.

- **I/O asymmetry**. Due to the electrical properties of flash chips, reading is much faster than writing, even when writing to clean pages (*i.e.*, ones that have been already erased), as becomes evident from Table 1.1. For most flash disks the read speed is two to four times as much as their write speed.

- **Erase-before-write limitation.** The most important limitation of flash disks is due to the erase-before-write limitation of flash pages: a sector cannot be overwritten. Rather, the whole flash block to which it belongs has to be erased first. Before erasing, all sectors of the flash block, even the ones that have not been updated, need to be read and stored in-memory. After the erase operation these sectors and the updated one(s) need to be rewritten. Considering that an erase operation is at least an order of magnitude more expensive than a read or a write, updating a disk sector becomes quite costly.

- **Wear levelling.** The disk controller employs *wear-levelling* techniques to alleviate the effect of flash chip wearing on the lifetime of the device. Such techniques spread writes evenly across all flash chips in the device and across all flash blocks in the same chip, thereby preventing some blocks from wearing out too early, and prolonging the overall lifetime of the disk. Assuming an 128GB flash disk, capable at writing at 80MB/s and a write endurance of 100k times, it would take 5 years of continuous writing before the flash chips wear out using wear-levelling. Considering that constantly writing to the disk for 5 years is not likely to occur in practice, the actual lifetime of the device would be much longer. Nevertheless, 5 years is already longer than most magnetic disks are used for, as they usually get replaced sooner than that for reasons that have to do with capacity, performance and energy efficiency. Thus, flash chip wearing does not limit the suitability of flash disk as a replacement for magnetic disks.

- **Energy efficiency.** Being purely electronic devices, solid state disks have low power consumption and generate little heat when in use. These features make SSDs ideal for mobile devices such as mobile phones, laptops, MP3 players and sensors. Energy efficiency is also desirable in enterprise environments; electricity costs amount up to 70% of the operational and cooling costs of a modern data center [HP Labs, 2006]. Recently, a great deal of research focuses on using SSDs in servers, as they have been shown to consume almost an order of magnitude less power than enterprise-class magnetic disks [Narayanan et al., 2009], while at the same time reduce the need for cooling.

- **Physical properties.** The lack of mechanical moving parts gives flash disks some additional advantages over traditional magnetic disks. Since no spin-up is required, they incur faster start-up times than magnetic disks. Also, flash memory is able to endure extreme shock, vibration, temperature and high altitude, making flash disks resistant to extreme environmental conditions. The operation of flash disks is completely silent and they are smaller and lighter than magnetic disks. In addition to their low power consumption, these properties make SSDs perfectly suitable for the requirements of mobile devices.

Performance and prices. Manufacturers are striving to produce high-performance flash disks at low prices; a wide range of devices have appeared on the market, with various price and performance characteristics. Each device incorporates many dice of flash chips to reach a capacity of tens or hundreds of gigabytes, as the capacity of each die is usually in the order of 8-16 gigabytes. The flash chips used in all SSDs have the same performance characteristics, as the latter are determined by the electrical properties of flash cells, *i.e.*, all bare SLC chips have almost identical performance as do all MLC chips, too. The performance of SSDs therefore only depends on the number of flash chips they use, the disk controller and the size of on-disk DRAM.

The peak performance of each individual chip ranges between 5 - 40 MB/s. In order to achieve higher performance, suitable for data intensive applications, flash disks employ parallelism: they read and write in parallel to many flash chips. The controller spreads data to many flash chips in parallel, or even to the whole drive. Then, for reads and writes in large enough chunks, the performance boost is proportional to the degree of parallelism. Controllers that support 4-8 channels can be found in commodity disks today, while enterprise products can have 10 channels or even more. The disk controller employs sophisticated techniques to avoid as many block erasures as possible. Some controllers use the on-disk DRAM to cache user data for faster access, while others choose to only store page mapping tables in DRAM, *i.e.*, the logical-to-physical page mapping. Larger DRAM buffers allow for a finer-grained mapping, thus giving the controller more opportunities for optimisations when writing data. Most disks also use redundant flash chips, *i.e.*, the total capacity of the flash chips in the drive is more than the capacity the user sees. The redundant flash chips are used for overwriting dirty blocks: the controller will place incoming data to a spare erased block without having to stall until the dirty block has been erased; dirty blocks are erased in the background.

Random writes. Performing small random writes on a flash disk is the most inefficient way of writing data. Assume a flash disk in which each block is 512kB and there is one 512kB DRAM buffer for the disk controller to use. When performing large sequential writes, *i.e.*, ones larger than 512kB, for a block write request of 512kB the controller needs to erase at most one block, if there are no clean blocks left on the device or the logical-to-physical address mapping is one of block granularity. On the other hand, when writing 512kB in packets of 4kB scattered all over the logical address space, there will be 128 write requests. If the controller logical-to-physical mapping is one of block granularity and all 4kB requests target different logical blocks, 128 block erase operations have to be performed; before each such operation, existing data on the block need to be read and merged with the new data and after the erase operation, the whole block has to be re-written. Thus, writing 512kB randomly in chunks of 4kB results in 128 block reads, erasures and writes. The situation can be alleviated if the logical-to-physical address mapping is a finer one, that is of page granularity; the controller waits until it has received 128 chunks of 4kB each, and then writes on the same physical block with at most one erase operation and at the same time keeps track of which logical block each 4kB chunk belongs to. This technique however is not very

Device	Seq. Read (MB/s)	Seq. Write (MB/s)	Random Read (MB/s)	Random Write (MB/s)	Price $/GB
FusionIO ioDrive	564	440	408	404	30
Intel X25-E	240.1	191.7	56.5	31.7	20
Intel X25-M	230.2	71	54.2	23.1	8.1
OCZ Vertex	250.1	93.4	32	2.41	4.5
OCZ Summit	208.6	195.2	29.1	0.77	2.8
JMicron JMF602B	134.7	87.1	16.2	0.02	2.69
Samsung	101.4	83.5	21.4	0.53	2.7
WD VelociRaptor	118	118.9	0.55	1.63	0.76
Seagate Momentus	77.9	76.6	0.28	0.81	0.15

Table 1.2: Sample SSD performance and cost characteristics (4KB I/O operations).

practical, as this mapping requires 128 times more DRAM memory buffers to maintain the mapping. It may also slow down reads, as in order to read a logical block, many physical ones will have to be read. In general, accelerating flash writes is a very hard problem and many tradeoffs are involved. Techniques towards that goal are discussed in more detail in Chapter 2.

In Table 1.2 we show the price and performance characteristics for seven SSDs and for two magnetic disks currently in the market ([AnandTech, 2009], [FusionIO, 2008]). The top seven disks on the table are SSDs, while the bottom two are magnetic ones. All operations are carried out in chunks of 4kB: random reads were performed across the whole disk, while random write tests span 8GB on each disk (except for the FusionIO disk, for which both tests span the whole disk). For sequential access patterns, the performance of all flash disks varies by less than an order of magnitude. The disparity in random read performance across flash disks is wider, but no more than two orders of magnitude. For random writes, on the other hand, performance ranges from 0.02MB/s to 404MB/s, that is, within more than four orders of magnitude. The per-gigabyte price of the SSDs also varies within a little more than an order of magnitude. The FusionIO SSD is an enterprise-class device; it supports an impressive random write throughput at a rather high price. The remaining six flash disks are consumer-class, although some of them are suitable for commodity servers as well. The WD VelociRaptor is an enterprise class 10000 RPM magnetic disk, while the Seagate Momentus is a commodity magnetic disk, spinning at 5400 RPM.

Clearly, there are many different classes of flash disk, especially with respect to their random write performance and price. Some of those outperform magnetic disks at randomly writing data by several orders of magnitude. Others, on the other hand, are almost two orders of magnitude slower at random writes than magnetic disks. There are SSDs that are faster at both sequentially reading and writing than magnetic ones; some SSDs are faster at sequentially reading but not at writing; others are worse than magnetic disks at both reading and writing sequentially. As a result, the term "flash disk" incorporates many different classes of device with respect to performance and price. Therefore, one cannot make specific assumptions about the efficiency of flash disks in general on a given access pattern. What is more, one can find many flash disks in the market that are outperformed by magnetic ones under various workloads. It is only random read workloads that all flash disks outperform magnetic ones.

1.3 Hybrid Systems

In this work we explore and evaluate how the designer of a database system can take advantage of this merit of flash disks to boost the I/O performance in a hybrid setup, *i.e.*, when both magnetic and flash disks are present. Deciding the role of a flash disk in the memory hierarchy of a hybrid system is not straightforward. The performance of RAM memory both in terms of access latency and transfer throughput is better than the performance of flash memory. The same holds for its spatial density, power consumption and cooling costs [Graefe, 2007]. Not surprisingly, its price is also higher. On the other hand, magnetic disks are much cheaper than flash disks; of course, their performance and operational characteristics are much worse. Nevertheless, there are notable exceptions to this, as shown in Table 1.2, in which magnetic disks outperform SSDs in random writes.

The design process for such a hybrid system is not a simple one. Should the designer consider the SSD as a part of main memory, *i.e.*, as an extended buffer, or as persistent storage? Given a specific budget, is one better off investing in DRAM only? Should one buy a small but very fast solid state disk or a larger (but slower) one? Such questions are crucial for the performance of the system, but cannot be given universally optimal answers. For instance, if one can buy enough DRAM to fit the working set of all the workloads that are likely to run on the system, then obviously one should go only with DRAM. Barring that, if the workloads are write-intensive it makes sense to invest in a high-performance flash disk to use as a cache, instead of an inexpensive one. In the general case it is neither easy nor safe to make decisions based on intuition;

even more so as both price and performance characteristics of flash disks are constantly changing.

1.3.1 Data Placement

In the first part of this work, we focus on hybrid systems that incorporate low-end flash disks with high capacities, *i.e.*, ones that outperform magnetic disks at random reads, but fall behind when writing randomly. We propose using the flash disk and the magnetic disk at the same level of the memory hierarchy, *i.e.*, the flash disk is not used as a cache for the magnetic disk. We show that important performance benefits can be gained with such a design, especially for queries touching large sets of pages with read-intensive workloads. We study the problem of optimal placement of each data page (*i.e.*, whether it should reside on the flash or on the magnetic disk) both from a practical and a theoretical perspective. We present a family of online algorithms that can be used to dynamically decide the optimal placement of each data page. Our algorithms adapt to changing workloads for maximum I/O efficiency. We have implemented all proposed algorithms and conducted an extensive experimental study. The results show that our algorithms can significantly improve I/O performance over both magnetic-disk-only and flash-disk-only setups, and for database workloads that frequently occur in practice.

1.3.2 Buffer Allocation

Next, we study the problem of buffer allocation in databases that store data across multiple storage devices. In such systems, devices share the same main memory for caching. The in-memory pages are managed independently for each device. We present our novel approach to per-device memory allocation. We introduce the metric of device caching utility for the data of each device. It is determined by the hit ratio for that data and the I/O costs of the host device. We propose a technique for measuring hit distances in the cache, which enables fast and accurate tracking of the hit ratio curve for each device. Our technique is applicable to all systems that can benefit from hit ratio curve tracking. As a first step, we present a static, parameter driven algorithm for partitioning the cache among devices. Then, we generalize this to a dynamic, on-line algorithm for near-optimal buffer allocation to devices, which takes into account device I/O characteristics. The dynamic algorithm adapts to changing workloads and is self-tuning: no parameters need to be set externally. Our experimental evaluation in a variety of configurations shows that utility-aware memory buffer allocation yields substantial -in some cases dramatic- improvement in both synthetic and real-world workloads. Equally important is that our techniques are able to effectively offset wrong data placement decisions.

1.3.3 Caching on flash memory

Next, we explore how a flash disk can efficiently act as a page cache between the main memory and the magnetic disk. We study the problem of deciding which data should be placed in the flash cache of a system. We identify and propose three invariants for the sets of pages cached either in main memory or on flash. For each invariant, the flow of pages between levels of the memory hierarchy is different. We present the page flow scheme of each invariant and an I/O-based cost model for the scheme. We discuss several implementation issues that arise when using a flash disk as a cache: (*a*) the page directory for the cache, (*b*) the size of flash pages, and (*c*) caching only pages that satisfy specific predicates; we show the correlation between each alternative and the properties of the flash disk. We have implemented our proposals and conducted an extensive experimental study. Our results show that most questions regarding flash-resident caches cannot be given universally optimal answers; rather, our cost model should be used to answer such questions for each individual case with confidence.

1.3.4 Sorting hierarchical data in external memory

We also study the problem of sorting in external memory using external merge-sort, as the latter employs access patterns that can take full advantage of the I/O characteristics of flash memory. We generalise the problem of sorting to hierarchical data, as such is necessary for a wide variety of applications including archiving scientific data and dealing with large XML datasets. We do not specifically focus on sorting using flash memory; our work is applicable to traditional systems as well. An algorithm that generalises the most widely-used techniques for sorting flat data in external memory is presented. The algorithm efficiently exploits the hierarchical structure in order to minimize the number of disk accesses and optimise the utilization of available memory. We extract and verify the theoretical bounds of the algorithm with respect to the structure of the hierarchical dataset. We implemented the algorithm and conducted a detailed experimental study of its performance for both archiving and stand-alone sorting; we include a comparison to the state-of-the-art approaches. Our results show that our algorithm outperforms the competition by a large margin and its performance is the one expected from its theoretical analysis. Though motivated by sorting scientific datasets for archiving purposes, the algorithm is general and efficient enough to be applicable in a variety of problems where the need for sorting arbitrary hierarchical datasets arises.

1.4 Organisation

The rest of this thesis is organised as follows. Related work in the area of data storage and management on flash memory is presented in Chapter 2. In Chapter 3 we present our work on efficient data placement in hybrid systems and in Chapter 4 we introduce our techniques for main memory buffer allocation in systems that use many storage devices. Our analytical study for the case of using a flash disk as a cache to the underlying magnetic storage is presented in Chapter 5. Our approach to sorting hierarchical data in external memory is presented in Chapter 6. We conclude and discuss future work in Chapter 7.

Chapter 2

Related Work

In this chapter we present previous research work on flash disks with respect to flash disk internals, storage over flash disks in general and database specific operations, such as indexing and query evaluation.

2.1 Flash Disk Performance Evaluation

In [Bouganim et al., 2009], the authors systematically study the performance characteristics of flash disks to identify the most favourable I/O patterns with respect to database workloads. They propose a benchmarking methodology for flash devices, provide an I/O benchmark that takes into account the particular characteristics of devices, and use it to evaluate a multitude of SSDs. A small number of performance indicators have been found adequate to accurately capture the performance characteristics of SSDs; in concert with our remarks of Chapter 1, the authors have found the performance discrepancy between high-end SSDs and low-end ones very significant in a multitude of workloads. The latency incurred by flash disks was found to be non-negligible, even for read operations; especially when writing, applications should opt for large I/Os. Applying the five minute rule [Graefe, 2007], the authors conclude that I/O should be done in blocks of 32KB; what is more, by aligning these blocks to flash memory pages, a significant performance improvement can be realised. Random writes were found to have a detrimental effect on performance; however, when restricted within a logical address range of 4-16MB, random writes performed nearly as well as sequential ones. On the other hand, concurrent sequential writes to more than 4-8 partitions caused severe performance degradation; thus, concurrent sequential writes should be limited to few partitions.

The performance of flash disks and its impact on algorithms is discussed extensively in [Ajwani et al., 2008]. The authors found flash disks to yield a better random read performance than magnetic disks, but a much worse random write one. Also, the algorithms designed for both main memory and magnetic disks were found to perform sub-optimally over flash memory, as they do not exploit the full potential of the novel medium. In accordance with the findings of [Bouganim et al., 2009], aligning write requests to block boundaries was experimentally verified to substantially improve random write performance; block alignment, however, did not help at random and sequential reads and random writes. In the authors' view, modelling the I/O cost of an algorithm on flash memory could use the standard external memory model as a base and distinguish between read and write operations. By applying a penalty for write accesses or different transfer block sizes the model could account for the special properties of flash memory. With a similar goal, the author of [Ross, 2008] proposes the "Erase-Once-Write-Many" model for analysing algorithms operating over flash disks. The analysis utilises the incremental 1-to-0 in-place updates that flash memory cells are capable of: an on-disk counter can be implemented, for instance, in a unary form by the number of zero bits indicating the count. In that case updating the value of the counter does not require re-writing the page. Departing from this simple idea, the author develops more efficient schemes for representing counters and more complex data structures such as linked lists, bloom filters and B-trees.

2.2 Accelerating Random Writes

A great deal of research has focused on increasing the throughput of flash disks when writing small chunks in a random fashion across a large portion of the address space. As discussed in Chapter 1, the erase-before-write limitation makes random writes inefficient. Flash disk controllers try to minimise the effect of this limitation on random writes; to that end, they employ a software layer called the *Flash Translation Layer* (FTL). Its main purpose is to provide logical-to-physical address mapping, power-off recovery, and wear-levelling. In [Agrawal *et al.*, 2008] the authors conducted a simulation-based evaluation of flash disks using traces extracted from real hardware: they found that the hardware and software components of flash disks can equally affect the overall performance. In the authors' view, appropriately-designed flash disks can be used to support OLTP workloads, such as TPC-C. Thus, it is conceivable that the appliances used today for such purposes, which typically employ hundreds of spinning disks, will be replaced by high-performance flash disks.

Different FTL algorithms are studied in [Chung et al., 2006]; with regard to their logical-to-physical address mapping they are categorised into three categories: sector mapping algorithms, block mapping algorithms and hybrid mapping algorithms. In the sector mapping case, the disk controller maintains a mapping from each logical sector to the physical flash sector to which it has been written. The controller is free to place a logical sector anywhere on the disk, as long as it keeps track of its location; thus, by choosing to erase a flash block only when a block's worth of logical data has been received, the disk can improve its random write performance significantly. As discussed in Chapter 1, however, sector mapping is not very practical, especially for embedded devices, as it requires a very large DRAM memory space for the mapping to be maintained. With block mapping, on the other hand, the logical sector offset in a logical block is identical to the physical sector offset within the physical block. In this case, a mapping only at block granularity is required and therefore few DRAM buffer suffice. The downside is, of course, that if the system issues many writes to the same logical sectors, too many erase operations are required and write performance degrades heavily. The above give rise to a hybrid mapping scheme: the physical block to which a logical sector belongs is first located using a block mapping table and then a sector mapping technique to locate the sector in that physical block. The sector mapping is not maintained in main memory for each block; rather it is stored persistently on the block itself.

A hybrid mapping scheme is also presented in [Kim et al., 2002]. The authors propose maintaining a small number of log blocks as temporary storage for overwrites, with each logical sector being mapped only to a certain log block (block level associativity); a logical sector can be placed anywhere within the physical log block. The experimental results presented show this technique to yield a substantial random write performance improvement over block-level mapping as well as insensitivity to workload characteristics. In [Lee et al., 2007], the authors identify patterns under which the aforementioned approach would result in very low space utilisation of the log buffers, leading to log block thrashing and, consequently, many erase operations. The block-level associativity is identified as the root cause for this weakness, and therefore the authors of [Lee et al., 2007] propose a scheme for fully associative translation between logical sectors and log blocks; under such a mapping the space utilisation of log blocks is increased. The improved log block utilisation guarantees a reasonable performance even with a small number of log blocks. In addition, the fully associative design enables the controller to avoid some block merging operations. Experiments show that write performance can be improved by as much as 50%. In [Birrell et al., 2007], it is argued that increasing the amount of volatile RAM on flash disks is the only way to achieve acceptable random access write performance and present a design for flash disks with large volatile buffers.

Write Performance vs Wear Levelling. To allow for low-latency write operations, a flash disk requires some block to be clean upon the write request. The more clean blocks are available, the more write requests can be served without waiting for an erasure. Therefore, the flash disk controller tries to continuously reclaim invalidated (or, obsolete) sectors, *i.e.*, sectors that have been updated, with their up-to-date version having been written in a different block. This procedure is referred to as *garbage collection* or *block reclamation*. For instance, if 10 blocks, having 640 sectors in total, only contain 150 valid sectors, the controller will gather the 150 valid sectors into its main memory and erase all 10 blocks. The 150 sectors will then be packed into 3 of the newly erased blocks and the remaining 7 blocks will remain clean for the controller to use in future writes.

Interestingly, as pointed out in [Gal and Toledo, 2005], the goals of write performance and wear levelling are often contradictory with respect to garbage collection. Assuming a block that contains data that remain relatively unchanged over time, *i.e.*, the block does not get overwritten, it is in the interest of the garbage collection algorithm not to touch that block: since its sectors are valid, that is, up-to-date, no space can be reclaimed from that block. Hence, reading, erasing and rewriting the sectors of the block incurs cost that will never be amortised. The more static the data of the block, the less frequently the garbage collection mechanism should touch it. In the interest of wear levelling, on the other hand, by reclaiming the static block, its data can be moved to another erase unit that has been heavily erased in the past, thereby reducing the future wear for the other unit over time. Hence, garbage collection and wear levelling essentially have contradicting goals. The additional write operations incurred due to garbage collection and wear levelling are referred to as *write amplification*. In [Hu et al., 2009] the authors present a probabilistic analysis of write amplification. Their simulation results show that write amplification heavily affects both the random write performance and the aging of the SSD. Distinguishing static from dynamic data and storing it in separate flash blocks was found to significantly reduce amplification; more so, as the portion of static data grew. In addition, the authors study the effect of over-provisioning, *i.e.*, equipping the flash disk with more capacity than is visible to the user: the extra flash blocks can be used as log blocks to speed up writes. According to their simulation results, providing the disk with 20% extra flash blocks can reduce write amplification by more than 50% when random writes are uniformly distributed in the logical address space.

In [Lee et al., 2009], the authors describe the major architectural characteristics of three classes of flash disk manufactured by Samsung Electronics. The low-end class of SSDs, targeted at personal and mobile computing platforms, aims to match the sequential read/write performance of commodity magnetic disks. As such, it employs 4-channel parallelism, a limited DRAM buffer (less than 1MB in size) and thin provisioning for extra flash blocks. The authors experimented with a device of this class in their previous work [Lee et al., 2008] and found this class of devices capable of improving the performance for the transaction log of a database, the rollback segment and for temporary data. However, due to the lack of write buffers and over-provisioning, random write performance was much worse than the random write performance of magnetic disks. In our work, we propose techniques to take advantage of this class of devices in a hybrid setup; we are not concerned with speeding up I/O operations on the device-level.

The second class of consumer-grade SSDs described in [Lee et al., 2009] aims at improving the random write performance for file systems, in which random writes occur in a very limited address space. Such disks are equipped with larger DRAM buffers (in the order of 32MB) and fat provisioning of flash storage. With the large write buffer the number of physical writes can be reduced; using a large number of extra blocks the number of erase operations per write is reduced as well. With these changes, devices of this class achieve an order of magnitude increase in random write throughput, when random writes are restricted within a one-gigabyte address space. Moving on to the high-end class of flash disks, targeted at enterprise database systems with a large number of concurrent I/O operations, the size of the on-disk DRAM is further increased to about 128MB, while the number of channels is increased from 4 to 8. More importantly, the 8 channels are now allowed to process different I/O requests in parallel; in this way, read operations do not need to wait for write operations to finish before they can be processed. Furthermore, the high-end class of disks supports native command queuing, *i.e.*, an internal queue of at most 32 requests is maintained: the controller may dynamically reschedule and reorder these requests to make the physical access pattern more flash friendly. Devices of this class outperform magnetic disks by several orders of magnitude.

2.3 Databases over Flash Memory

2.3.1 Database Storage

FTL algorithms, primarily designed with file systems in mind, are not well-suited for database workloads. As discussed in [Lee and Moon, 2007], random write operations in file systems are mostly required for metadata. When executing typical database workloads, however, DBMSs perform random write operations that are scattered over the whole disk address space. To improve write efficiency for flash-based databases, an *in-page logging* (IPL) scheme is proposed in [Lee and Moon, 2007]: changes made to a data page are not written directly to disk, but to log records associated with the page. Changes are logged on a per-page basis, while each data page and its log records are located in the same physical block of the disk *i.e.*, in the same erase unit. Each erase unit is divided into a number of data pages and a number of log sectors for the log records of the pages. In-memory representation of a page includes an in-memory log sector (of the same size as the flash log sector). When a page is dirtied in memory, its contents need not be written back to disk; only the log records for the page need to be appended to the log sector for the page on disk. When the erase unit is out of free log sectors, the logged changes are applied to the corresponding data pages in the unit. Then, the data pages are written to a new erase unit (that has its log sectors erased). That way, page updates only involve writing already erased log sectors. Block erasure is required only when the log sectors of a block become full. Simulation results of the IPL scheme show that it improves the random write efficiency of flash disks by an order of magnitude for typical database workloads. The authors also propose an IPL-based recovery mechanism for transactions that minimizes the cost of system recovery. Such logging schemes are orthogonal to our proposals; they can be used complementary and will most likely result in further I/O efficiency.

To avoid reading unnecessary attributes during scan selections and projections, the authors of [Tsirogiannis et al., 2009] advocate storing relations using a column-based layout. Specifically, they propose using PAX [Ailamaki et al., 2002], according to which each data page of an n-attribute relation is divided into *n minipages*; each minipage stores the values of a column contiguously. Thus, values that belong to different columns are physically separated and database operators can selectively access only the attribute values that are required for a query. Seeking from minipage to minipage incurs random I/O, but when operating over flash memory the cost of the random seeks is negligible. A scanning operator, termed *FlashScan*, is introduced, that reads projected attributes of a selection and produces tuples in row format. Although the techniques proposed are not novel and their efficiency has been thoroughly studied in the literature, their experimental results verify the applicability of these techniques in flash-based deployments. Substantial performance improvement over a row store was observed, more so as the selectivity and projectivity of queries were lower.

2.3.2 Indexing

In [Nath and Gibbons, 2008] the authors emphasise design principles for flash memory storage access methods. In addition to avoiding in-place updates and random writes, the designers of access methods should avoid sub-block deletions as well: when deleting a portion of a block any undeleted data in the same block need to be first copied to a new block. Hence, sub-block deletions can be two orders of magnitude more time- and energy-consuming than block deletions. Instead, access methods should employ *semi-random* writes, according to which individual pages within a block are written sequentially from the start of the block; the write pattern can select blocks in any order. The experimental evaluation of this principle was found to yield performance similar to the one of sequential writes.

Research has also been conducted in the area of flash-aware tree indexing. In their work presented in [Wu et al., 2007], the authors propose storing B^+-tree nodes as a sequence of log records spread over multiple disk blocks. To update a page, one appends log records to this sequence, thereby not having to erase the entire block for each page update. The evaluation of the approach shows that it improves performance both in terms of time and energy efficiency. In a similar fashion, the authors of [Li et al., 2009] propose the FD-tree, a flash-friendly B^+-tree variant hierarchical index structure. This structure consists of a small B^+-tree on the top and a few levels of sorted runs at the bottom. Operations on the tree aim to reduce the number of random writes on the disk. Updates are applied to the top tree which, due to its small size, is likely to fit in main memory; subsequently the changes are merged to the lower level sorted runs in batches. Pointers between the sorted runs of different levels are utilised to speed up searches. The experimental evaluation of this technique shows that it outperforms all other B^+-tree variants at both search- and update-intensive workloads. The techniques presented in this thesis are applicable to the block layer of a storage system and, as such, are oblivious to the file structures used. Thus, any such index structure can be used on top of the techniques presented in this thesis.

On the same topic, authors in [Nath and Kansal, 2007] propose a self-tuning B^+-tree that provides indexing functionality to the storage manager of a flash-based database system. The index dynamically adapts its storage structure according to the database workload and the underlying storage device. Specifically, nodes of the B^+-tree are stored either in *log* mode or in *disk* mode. In *disk* mode, the entire node is written in consecutive disk pages, while nodes in *log* mode are stored as log entries, that may be spread over multiple disk pages. Pages in *log* mode are written very efficiently (as updates do not incur overwriting physical blocks), while read operations require all log entries for the page to be gathered, so that the page can be reconstructed. On the contrary, pages in *disk*-mode can be read efficiently (by just reading a page from disk), but writing a *disk* mode page requires erasing the physical block first. Switching between modes incurs a specific cost, therefore authors propose an on-line algorithm to decide the optimal mode of a page.

In [Li et al., 2008] the authors propose a hash index for flash-resident data that avoids deleting records in place. Rather, modifications are logged in the hash buckets. Two variations of a linear hashing index are presented; one is similar to the standard linear hash index and is geared towards search-intensive workloads. In the second variation, buckets are split lazily, *i.e.*, bucket splits are processed in batches, so as to reduce the cost of writes at the expense, of course, of some additional cost when searching the index; thus, the second variation is more suitable for update-intensive workloads. Geared towards wireless sensor devices that use flash storage, a hash index, termed *MicroHash*, is proposed by the authors of [Zeinalipour-Yazti et al., 2005]. The goal of the design is to provide efficient equality and temporal queries in a computing environment with severely constrained processing capabilities; therefore, it is not very relevant to this thesis. Also in the context of sensor networks, although with wider applicability, the authors of [Nath and Gibbons, 2008] study the problem of efficiently maintaining a large random sample (in the order of hundreds of megabytes) of a data stream. Techniques are provided for maintaining both guaranteed uniform random samples and biased ones, such as weighted and age-decaying samples.

2.3.3 Query Execution

The use of indexes during query execution is discussed in [Myers, 2007], where the author studies how the selectivity of a query should be taken into account by the optimiser to select the access method. Both magnetic and flash disks are considered.

Depending on the selectivity of the query and the I/O costs of each storage device, it may be optimal to do an index scan or a full table scan either from the magnetic disk or the flash disk. The author also evaluated the standard join algorithms over magnetic and flash disks. Of particular interest is the case in which the flash disk has worse sequential read bandwidth than the magnetic disk. In that case the flash disk has better performance at low selectivity joins in the presence of indexes: the full potential of the fast random I/O of the flash disk can be exploited. However, at high selectivities a full scan of the relation on the magnetic disk performs better due to its superior sequential read performance. Nevertheless, it is clear from the results of [Myers, 2007] that the I/O costs of the flash disk used in each particular case need to be taken into account in order to reach informed decisions regarding access methods.

Query execution over flash memory is also studied in [Tsirogiannis et al., 2009]. The authors propose a join operator, termed *FlashJoin*, that aims (*a*) to reduce the I/O cost incurred by join evaluations by minimising the number of passes over the participating tables and (*b*) to minimise the I/O required to fetch attributes for the query result. Towards the first goal, the operator only accesses join attributes; other projected attributes are fetched only for rows that participate in the result. The operator processes join indexes instead of all projected attributes from each relation and therefore has a small memory footprint. Thus, a join can be computed in one-pass using much less main memory than with a standard hybrid-hash join kernel. Towards the second goal, a late materialisation strategy is employed: retrieving projected attributes is postponed for as far down-stream the query plan as possible. Each join produces only the necessary attributes required by the remaining operators and therefore, even if multiple passes are required, the partitioning cost for the extra passes is reduced.

The most common *ad hoc* join algorithms are revisited in [Do and Patel, 2009] with respect to execution over flash memory. The authors show how previous results for magnetic disks continue to hold for flash drives. The buffer allocation strategy is found to have a critical impact for both types of storage media when executing ad hoc join algorithms. The authors conclude that using blocked I/O can significantly improve performance of joins on SSDs as it reduces the number of erase operations and the overhead incurred by the FTL software. Also, the authors point out that with flash disks, the CPU time can be the dominating factor for the overall cost of a join algorithm and, thus, both I/O costs and CPU times should be optimised for in a system that uses flash memory for storage.

During query execution, a portion of the incurred I/O cost is due to maintaining the transaction log, the rollback data and operations over temporary storage spaces. The authors of [Lee et al., 2008] experimentally evaluated the overhead of such database operations and found it to be substantial. Therefore, they suggest that towards improving the performance of transaction processing systems, one should not optimise only for tables and indexes, but for the rest of the storage spaces as well. In their evaluation they use a commercial database system and study the access patterns involved in the transaction log, in the rollback segments used by the multi-version concurrency control mechanism of the DBMS and in the temporary storage spaces used for external sorting and partitioning. They identified that in these storage spaces sequential writes and random reads are the dominant access patterns, *i.e.*, these spaces are well-suited for use with flash memory. Replacing the magnetic disk with a commodity flash disk for these storage spaces, they observed an order of magnitude improvement in transaction throughput for the transaction log and the concurrency control rollback segment and more than a factor of two improvement in response time for external sorting. What is more, since it is common practice for database systems to have physically separate storage spaces for the tables, the indexes, the transaction log and the temporary data, the proposals of [Lee et al., 2008] are immediately applicable in most deployments. Transactional logging using flash memory is also studied in [Chen, 2009]. However, instead of an internal flash disk, the author proposes a solution that utilises multiple USB flash drives. Compared to an internal flash disk, the multiple USB drives are found to have comparable performance at a much lower price. The performance during an OLTP workload was dramatically increased using the proposed solution instead of a magnetic disk. Using parallel device scans on the flash drives also resulted in a more efficient recovery process.

2.3.4 Main Memory Buffering

Previous research has considered buffer management over flash disks, *i.e.*, for the case that the flash disk is used for persistent storage and a subset of the pages stored on the flash disk are cached in main memory. In [Kim and Ahn, 2008] the authors propose BPLRU, a scheme for the on-disk buffer cache of flash disks. This scheme treats the buffer as a write cache and groups RAM buffers in blocks that are equal in size to the flash erase-unit; page replacement is performed with erase-unit granularity (using LRU). If not all sectors of a dirty victim page are present in-memory, the absent ones are read from disk so that the whole block can be written to a new flash location without the need for an in-place update. Additionally, a block that was written sequentially is moved to the tail of the LRU list and becomes the next victim. Experimental evaluation shows this technique to be very promising. Similarly, the authors of [Park et al., 2006] propose that the buffer cache choose for replacement a clean page over a dirty one and therefore trade the number of writes with the number of reads. In our work, we have generalised this concept for a system in which the same buffer pool holds pages from both the flash and the magnetic disks. In that case, not only the dirtyness of the page, but also the read/write costs of the storage medium and the access history of the page are considered when choosing a page to replace.

In [Ou et al., 2009] the authors extend the ideas presented in [Park et al., 2006]: they not only aim to minimise the number of writes to the flash disk, but also they aim to exploit the spatial locality of victim pages. Dirty pages whose page numbers are close to each other, and therefore are likely to be physically stored near one another, are clustered together in page clusters. The algorithm maintains a priority queue of all page clusters in the system and evicts the one that has been referenced least recently. Thus, write operations become more flash-friendly, although this technique bears a higher complexity due to the system having to maintain a priority queue of length proportional to the number of clusters. In a similar fashion, the authors of [Stoica et al., 2009] present a technique for transforming random writes of dirty pages into sequential ones. They have created a shim layer within the storage manager of a DBMS that writes dirty pages, evicted by the buffer manager, sequentially in multiples of the erase block size. Hence, the physical medium is utilised in an append-only manner. Naturally, this introduces the need for erase block reclamation at the storage manager level and also introduces some overhead when reading. An analytical model is presented for this storage layout and the potential benefit of the technique is demonstrated experimentally for a high-end SSD. The specific device allows the user to manually erase blocks [Jonathan Corbet, 2009]. Note, however, that this technique is not generally applicable to all devices, as in most flash disks on the market the user has no control on block reclamation and physical data placement. Our work is oblivious to the techniques employed by the device firmware and controller structure; therefore they can be used with any device, provided that its average read and write costs are known.

2.3.5 Caching on Flash Memory

In [Narayanan et al., 2009] the authors discuss how flash disks can be incorporated in the enterprise storage hierarchy in terms of performance, capacity, power consumption and reliability. They describe an automated tool that can decide what storage hardware configuration is optimal for a specific workload, using multiple metrics. The tool operates in an offline fashion and makes decisions based on specific workloads. Among other things, the authors consider the use of the flash disk as a read cache, although specific properties of such a cache are not studied. The use of the flash disk as write-ahead log is also considered. The authors conclude that while SSDs are suitable as read caches and for write-ahead logging, their price/performance characteristics do not make them preferable to magnetic disk as an alternative for persistent storage. Thus, they conclude that the price-per-gigabyte cost of flash memory will have to drop before SSDs can replace magnetic disks in a cost-effective way. Interestingly enough, the authors of [Lee et al., 2009] found a flash disk to outperform a level-0 RAID with 8 enterprise-class 15k-RPM magnetic disks on the TPC-C benchmark with respect to transaction throughput, cost effectiveness and energy consumption. In this thesis we explore how an SSD can be used as cache irrespectively of cost, power consumption and reliability; we are mostly concerned with identifying which data should be cached on the SSD for maximum performance.

Another piece of work relevant to caching on flash memory – which, however, is not geared towards database workloads – is outlined in [Leventhal, 2008] and implemented in the ZFS filesystem [Sun Microsystems., 2008]. The flash disk is used as a cache for the magnetic disk with the goal of improving the performance of random read workloads. In that setup there is no eviction from main memory to the flash disk; rather, the flash cache stores main memory pages before they are evicted. Filling the flash cache with pages is performed asynchronously, thereby avoiding write latencies on main memory evictions. By employing large sequential writes to predictively push data to the flash cache, the system avoids paying the cost of random writes and increases the flash write bandwidth. Also, the flash cache never stores dirty pages and therefore no write-back to disk is required for flash pages. Our work is an analytical study of the behaviour of flash caches; the techniques engineered in ZFS are thus complementary to our work and can provide a suitable implementation basis.

Chapter 3

Data Placement

3.1 Preliminaries

In this chapter we study the problem of data placement in a commodity hybrid system, *i.e.*, one that is equipped with both a magnetic and a flash disk. Our design is not geared towards high-end flash disks that completely outperform magnetic disks. Rather, we consider low-end flash disks that use MLC flash memory; such disks have greater capacity at a fraction of the cost of SLC ones, but quite worse write performance. Given the higher storage capacity (comparable to that of magnetic disks) and lower cost of MLC flash disks, commodity hardware is expected to incorporate that type of disk, which clearly cannot outperform magnetic disks with respect to random write performance. In this chapter we consider such flash disks and explore how data placement can take advantage of their merits in a hybrid setup.

One idea is to use the flash disk as a cache for the magnetic disk, *i.e.*, as an extended buffer. While this design might be reasonable for a file system, it can prove suboptimal for database workloads as it disregards the writing inefficiency of the flash disk. The reading efficiency of the flash disk, on the other hand, is an argument for using it for persistent storage [Graefe, 2007]. Given this discrepancy, and considering the growing capacity of flash disks, we propose to use both types of disk at the same level of the memory hierarchy, *i.e.*, a database page can reside *either* on the flash disk *or* on the magnetic disk, but not on both. We present algorithms for optimally placing a page according to its workload. Pages with a read-intensive workload are placed on the flash disk, while pages with a write-intensive workload are placed on the magnetic disk. We propose ways of accurately predicting the workload of a page in an adaptive fashion.

The types of system that can benefit from our proposal include (but are not limited to): (*a*) (parts of) database systems with well-defined workloads, especially when a portion of the data is very frequently accessed but only scarcely updated *e.g.*, the database catalog, typical access paths, *etc.*; (*b*) archiving systems, where a percentage of data appearing only in the latest versions are frequently accessed, whereas a larger percentage of the archive (also referred to as the *deep* archive) is infrequently used; (*c*) file systems, where both kinds of disk are transparently handled by the operating system, but user data is organized according to its I/O workload for maximum efficiency; (*d*) hybrid hard disks, *i.e.*, magnetic disks that are equipped with flash memory, which they use as non-volatile cache [Wikipedia, 2008]. Our algorithms can be employed by the controllers of such disks to boost performance.

The high-level architecture of our system is shown in Figure 5.1. A magnetic disk and a NAND flash disk operate at the same level of the memory hierarchy. Each data page exists only on the flash, or on the magnetic disk at any given time. The storage manager decides the optimal placement for each page according to the workload of the page. Pages with a read-intensive workload are placed on the flash disk, while pages with a write- or update-intensive workload are placed on the magnetic disk. Thus, reads are faster than a magnetic-disk-only system, and writes are faster than a flash-disk-only system. In this manner the total I/O cost is reduced.

The main challenge in this approach is how one can predict the future workload of a page based on past accesses to the page. Also of paramount importance is the ability to self-tune, *i.e.*, adapt the placement choice for each page when its workload changes from read-intensive to write-intensive and vice-versa. Considering that moving a page from one disk to another incurs significant I/O cost, the prediction of a page's future workload has to be as accurate as possible. Failure to achieve an acceptable level of accuracy means that the I/O cost will be heavily penalised, as the page will migrate from disk to disk before the migration cost has been expensed.

We present a family of algorithms to decide the optimal placement of data pages. We employ a typical buffer pool: pages are fetched on demand from disk to main memory. Read and write operations that are served in main memory are referred to as *logical* hereafter, while ones that reach the disk are referred to as *physical*. Whenever the buffer pool is out of space, a page is selected to be replaced according to the buffer manager's replacement policy. At that point the system needs to decide the placement of the page, *i.e.*, whether the page will be stored on the flash or on the magnetic disk. The decision is made dynamically and on a per-page basis, it depends only on the

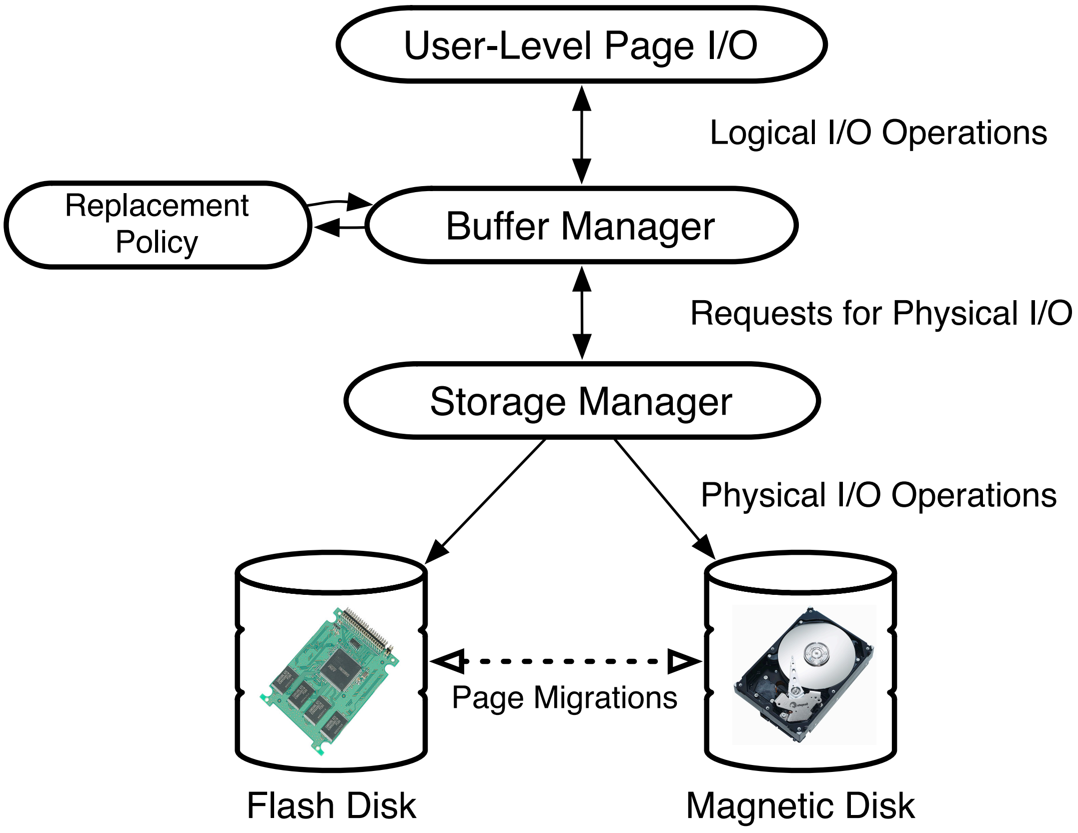


Figure 3.1: An overview of our system

history of the page, and is independent of all other pages. Our system keeps track of the location of each page (so that it knows which disk to read it from) and statistics about its workload.

The decision algorithm is an on-line one. We model the decision process for each page as a two-state task system [Borodin et al., 1992], depicted in Figure 3.2. The two *states* of the system are f and m, representing that a page is on the flash disk or on the magnetic disk, respectively. The cost for reading a random page from the magnetic disk is r_m, while the cost for writing a page to a random position is w_m; r_f and w_f are the respective costs for the flash disk. The *transition cost* from one state to the other is equal to the cost of writing a page to the other disk (when a transition occurs, the page has already been read). The *tasks* in our task system are I/O operations. The cost of processing a read request is r_f when the system is in state f and r_m when it is in state m (resp. w_f and w_m for write/update operations).

The algorithm we propose for page placement is also an on-line one. The problem we are solving is an instance of the page migration problem: deciding the optimal node of a network to store a data page, so as to minimise the total cost of serving requests

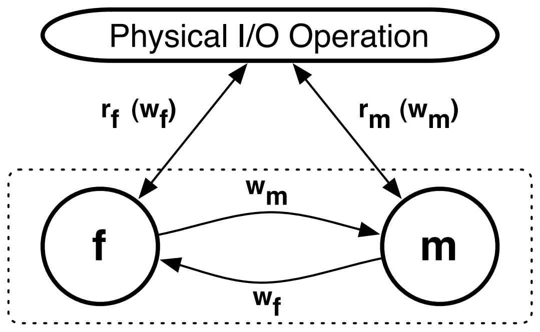


Figure 3.2: Abstraction as a two-state task system

for the page from other nodes of the network. In [Black and Sleator, 1989], the authors thoroughly study this problem and present competitive algorithms for different network topologies. In [Borodin et al., 1992], the authors study metrical task systems similar to the one we use to model our system and provide a $(2n-1)$-competitive online algorithm (n is the number of states of the task system). However, our task system is not metrical, due to the write cost discrepancy in the two types of storage media: we have used an algorithm similar to the one proposed in [Borodin et al., 1992] and shown that it is $(2n-1)$-competitive in our non-metrical task system as well. Page migration in graphs with arbitrary edge distances is studied in [Westbrook, 1992] where the authors propose a randomised algorithm that approaches 2.62-competitiveness against an oblivious adversary.

The decision problem we study in this chapter resembles the page migration problem on an arbitrary tree [Black and Sleator, 1989]. The key difference is that in our case the page migration cost depends on the direction of the migration. Our proposed solution resembles the algorithm given in [Nath and Kansal, 2007] as the mode-deciding algorithm. However, one cannot simply adapt that approach if one wants to realistically model the problem we solve. The reason is that one needs to make the crucial distinction between logical and physical I/O operations. The interaction between physical and logical operations is not clear, unless buffer pool and storage management parameters are taken into account. This is due to the actual I/O cost being decided by physical operations, while application-level I/O requests are merely logical. This salient distinction is elegantly captured in our model. As we shall see in Section 3.3, our results prove this non-trivial extension necessary in an implementation with real-world workloads (and not in a simulation).

```
Algorithm conservative (Page $pg$)

1.  if ($pg$ is a new page)
2.      $pg$.state $\leftarrow m$, $pg.C \leftarrow 0$
3.  After each physical read of the page:
4.      $pg.C \leftarrow pg.C + (r - r')$
5.  Upon eviction of the page:
6.  if ($pg$.dirtybit $= 1$)
7.      $pg.C \leftarrow pg.C + (w - w')$
8.  if ($pg.C > w_f + w_m$)
9.      $pg.state \leftarrow$ other state
10.     $pg.C \leftarrow 0$
11.     $pg$.dirtybit $\leftarrow 1$
```

Figure 3.3: The conservative algorithm

3.2 Page Placement

We present a family of on-line algorithms that have the same structure, but use different cost metrics.

3.2.1 Conservative Algorithm

The first algorithm, which we refer to as conservative, is given in Figure 3.3. For each page in the system, the algorithm maintains a counter C that is updated after each physical operation. The cost of reading the page from the current disk is r, while the cost of reading the page from the other disk is r' (resp. w and w' for writing). When a page is *physically* read, C is incremented by the cost difference $r - r'$ (line 3), that represents the cost units that would have been saved, had the page been read from the other disk (if $r - r' < 0$ the page was read from the read-efficient disk). The same happens when a dirty page is to be evicted from the buffer pool: the cost counter is incremented by $w - w'$. Upon eviction, C is examined (line 8) and if it is greater than the cost of two migrations ($w_f + w_m$), the page migrates to the other disk (by changing its *state* value and setting its dirty bit to 1 – lines 9 to 11).

The conservativity of the algorithm lies in two points: (a) The algorithm initiates a migration only after the accumulated cost for a page has surpassed $w_f + w_m$. This is the earliest point in time that the algorithm can be certain that the page is not on the optimal storage medium. For the time period during which the counter accumulated the $w_f + w_m$ cost units, the cost would have been less if the page was stored on the other disk. This is because the cost of migrating to the other disk and back has been already reached during the last physical operations and those physical operations would have been served more efficiently by the other disk. Note that at this point the algorithm can also be certain that in the worst case (that the workload of the page immediately changes in favour of the previous disk) the maximum extra cost incurred by the wrong decision is at most $2(w_f + w_m)$. (b) The algorithm takes into account only physical operations on pages, not logical ones. The physical cost is the actual cost paid by the system and therefore the conservative algorithm does not try to induce the physical access pattern from the logical one. Rather, it waits until the logical access pattern has been translated into physical accesses. Note that due to the lack of any access history for new pages, they are always written to the magnetic disk for the first time, since the magnetic drive is more write-efficient (line 1 in Figure 3.3).

An off-line algorithm that knows the exact workload for each page beforehand can decide the optimal placement of the page *i.e.*, it incurs the minimum I/O cost for the workload. We refer to such an algorithm as the *optimal offline algorithm* and to the cost incurred by it as OPT. For any on-line algorithm A with cost C_A, we say that A is *c-competitive* with respect to the optimal offline algorithm if the cost incurred by A is at most c times the cost of the optimal algorithm, *i.e.*, $C_A < c \cdot OPT + c_0$, where c_0 is a constant. We now prove that the cost incurred by the an algorithm like conservative is at most three times the cost incurred by the optimal offline algorithm, *i.e.*, conservative is 3-competitive:

Theorem 2.1: *The conservative algorithm is* 3*-competitive with respect to the optimal off-line adversary.* □

Proof. We will show that conservative is 3-competitive *w.r.t.* the optimal off-line adversary, *i.e.*, at any time t, $CONS(t) < 3OPT(t) + C_0$, where $CONS(t)$ is the total cost incurred by conservative up to that time, $OPT(t)$ is the total cost incurred by the optimal on-line algorithm, and C_0 is a constant. First we will show that conservative is 3-competitive in a metric space *i.e.*, when the migration cost is the same for both directions and equal to $K = \frac{w_f + w_m}{2}$. A migration happens when $C = w_f + w_m = 2K$ (to comply with this our algorithm could trivially set $C = 2K$ when $C > 2K$; this would yield the same decisions). The accumulated cost of the page we are running the algorithm for is C; at time t suppose that conservative is in state s_1, while optimal is in state s_2. We define a potential function $\phi(t)$ as follows:

$$\phi(t) = \begin{cases} 2C & \text{if } s_1 = s_2 \\ 3K - C & \text{otherwise} \end{cases}$$

Observe that $\phi(t) \geq 0$ and $\phi(0) = 0$. Also $CONS(0) = OPT(0) = 0$. For each possible event at a time t, we will show that $\Delta_{CONS} + \Delta_\phi \leq 3\Delta_{OPT}$, in which Δ_X indicates the change in the value of X as a result of the event. By summing over all events we obtain the desired inequality (since $\phi \geq 0$). Possible events are:

1. Transition of conservative. Then $\Delta_{CONS} = K$ and $\Delta_{OPT} = 0$. Before the transition $C = 2K$ holds, and after the transition $C = 0$ holds. Also:

   $$\Delta_\phi = \phi(t+1) - \phi(t) = (3K\text{-}0) - 2 \cdot 2K = -K, \text{ if } s_1 = s_2$$

   and

   $$\Delta_\phi = \phi(t+1) - \phi(t) = 0 - (3K - 2K) = -K, \text{ if } s_1 \neq s_2.$$

   In both cases $\Delta_{CONS} + \Delta_\phi = K - K = 0 \leq 3\Delta_{OPT} = 0$.

2. Transition of the optimal off-line algorithm. Then, $\Delta_{CONS} = 0$ and $\Delta_{OPT} = K$. Also:

   $$\Delta_\phi = (3K - C) - 2C = 3K - 3C \leq 3K, \text{ if } s_1 = s_2$$

   and

   $$\Delta_\phi = 2C - (3K - C) = 3C - 3K \leq 6K - 3K = 3K, \text{ if } s_1 \neq s_2.$$

   Hence $\Delta_{CONS} + \Delta_\phi = 0 + 3K = 3K = 3\Delta_{OPT}$.

3. The last event is serving a read/write request. Let c_1 be the cost of serving the request in state s_1 and c_2 in state s_2. Then $\Delta_{CONS} = c_1$ and $\Delta_C = c_1 - c_2$. If $s_1 = s_2$, then $\Delta_{OPT} = c_1$ and $\Delta_\phi = 2 \cdot \Delta_C \leq 2c_1$ hold, since $\Delta_C = c_1 - c_2 \leq c_1$. Thus:

   $$\Delta_{CONS} + \Delta_\phi \leq c_1 + 2c_1 = 3c_1 = 3\Delta_{OPT}$$

   If $s_1 \neq s_2$, then $\Delta_{OPT} = c_2$ and $\Delta_\phi = -\Delta_C = c_2 - c_1$. Therefore:

   $$\Delta_{CONS} + \Delta_\phi \leq c_1 + c_2 - c_1 = c_2 \leq 3c_2 = 3\Delta_{OPT}$$

Thus conservative is 3-competitive for the symmetric graph H with transition costs equal to K for both states. We will show that conservative is also 3-competitive for our asymmetric graph G that has a transition cost of w_f when moving to f and w_m when moving to m. Since the algorithm is the same (and starts from the same state), it will perform the same transitions in G as it would in H, in which case it would be 3-competitive. However, two consecutive transitions on H would cost the same as they cost on G (because the cost of a cycle is $w_f + w_m$ in both graphs). Thus, the algorithm has the same cost in both graphs if it performs an even number of transitions, and an extra cost of $w_f - \frac{w_f - w_m}{2}$ for G and an odd number of transitions. This extra cost is constant, so conservative is 3-competitive in G as well.

□

Our evaluation shows, however, that the cost of conservative remains more than 1.5 times less than the cost of the optimal algorithm for realistic workloads.

3.2.2 Optimistic Algorithm

Though physical operations capture the actual cost paid by the system, their sequence is dictated by logical operations and the replacement policy of the buffer pool. Moreover, while the page remains in the buffer pool (*i.e.*, between two physical operations on the page) many logical operations may occur. The conservative algorithm, will only record two (or one) physical operations on the page, and thus, if the workload changes, it will take many physical operations before conservative adapts. This gives rise to an "optimistic" version of the algorithm that works only on logical page operations and adapts to new workloads as quickly as possible; the algorithm is presented in Figure 3.4.

For each page pg the optimistic algorithm maintains a read counter (pg.reads) and a write counter (pg.writes). Each counter is incremented when a logical read or write operation occurs, respectively. These counters hold the total logical read and write operations on the page since its last migration. Upon eviction, the algorithm computes the total cost the system would pay if these operations were physical, for each of the two disks (c_f for the flash disk, c_m for the magnetic one – lines 10 and 11 in Figure 3.4). The page migrates to the disk with the least total cost, if it is not already there, and its read and write counters are reset (lines 11 to 14). When a new page is created, the algorithm does not account for logical operations until the page has been physically written for the first time (line 3). This is because most newly created pages will be

```
Algorithm optimistic (Page $pg$)

1.  if ($pg$ is a new page)
2.    $pg$.state $\leftarrow m$, $pg$.reads $\leftarrow 0$, $pg$.writes $\leftarrow 0$
3.    No accounting until after the first physical write
4.  After each logical read of the page:
5.    $pg$.reads $\leftarrow pg$.reads $+ 1$
6.  After each logical write of the page:
7.    $pg$.writes $\leftarrow pg$.writes $+ 1$
8.  Upon eviction of the page:
9.  $c_f \leftarrow pg.\text{reads} \cdot r_f + pg.\text{writes} \cdot w_f$
10. $c_m \leftarrow pg.\text{reads} \cdot r_m + pg.\text{writes} \cdot w_m$
11. if (($c_f > c_m$ and $pg$.state $= f$) or ($c_m > c_f$ and $pg$.state $= m$))
12.   $pg$.state $\leftarrow$ other state
13.   $pg$.reads $\leftarrow 0$, $pg$.writes $\leftarrow 0$
14.   $pg$.dirtybit $\leftarrow 1$
```

Figure 3.4: The optimistic algorithm

logically written to many times when they are created (e.g. after a B^+-tree node split). These logical writes do not reflect the normal workload for the page and are therefore not logged.

The optimistic algorithm is not conservative in the number of migrations. It assumes that when the workload of a page changes from read-intensive to write-intensive (or vice-versa), the migration cost will be amortised, *i.e.*, changes to the workload of the page are not frequent. Thus, optimistic adapts quickly to changing workloads but when changes do not last long enough for the migration cost to be expended, the overall cost paid by the system increases. Our experimental results verify these observations.

Another caveat is that optimistic tries to minimise the cost of future physical operations on the page based on its history of logical operations. Consider a page p having been brought into the buffer pool at time t_1 and evicted at time t_2, after having been logically read a large number of times: its workload upon eviction is found to be strongly read-intensive and the page will be written to the flash disk. Then, the workload of the page changes to write-intensive and optimistic needs to see some k logical writes on p before deciding it is now write-intensive. If the page is frequently replaced by the buffer manager until the k logical writes have been served, these k logical writes will have been realised as physical ones (since the page is frequently evicted). Reasons for these frequent evictions include the buffer manager deciding to assign fewer pages to the file p belongs to, or some other file becoming hot, or the time between writes on p being much longer than the time between reads, (*i.e.*, the page becomes cold). In this scenario, not only is the benefit from the migration never realised (since read operations are very scarce after the initial eviction), but also the system pays a very high penalty by writing the page to the flash disk, before the write-intensive workload has been identified.

3.2.3 Hybrid Algorithm

To minimise the total cost of physical operations, one needs both physical *and* logical operations on data pages to be taken into account. We introduce a hybrid algorithm that combines the strong points of conservative and optimistic, at the same time avoiding their weak points. The basic idea is that a physical operation on a page has more impact on the decision of the algorithm than a logical one. This is because physical operations on a page are typically fewer than logical ones, but at the same time they are the ones to affect the actual cost.

The probability that a logical operation will not be realised as a physical one is proportional to the size of the buffer pool. Let n be the number of pages in a file and b the number of pages the buffer manager has dedicated to the file: the probability that a logical operation on a page will be served in-memory is b/n. The probability that a logical operation on a page will affect the total I/O cost is equal to the probability of the logical operation resulting in a physical one. Thus, the probability that a logical operation will have an impact on the I/O cost is equal to $(1-b/n)$. We use this probability to scale the impact of a logical operation.

The hybrid algorithm, shown in Figure 3.5, maintains four counters per page: lr and lw count logical reads and writes since the last migration, respectively; pr and pw count physical reads and writes since the last migration, respectively. Newly created pages are written to the magnetic disk and counters are not modified until the page has been written for the first time (lines 1-4). For each logical or physical operation on the page, the corresponding counter is incremented (lines 5-10). Upon eviction, the algorithm computes the total cost physical and logical operations would incur for each disk (lines 16-17). As mentioned, the cost of logical operations is scaled by $1-b/n$.

```
Algorithm hybrid (Page $pg$)

1.  if ($pg$ is a new page)
2.      $pg$.state $\leftarrow m$
3.      $pg.lr \leftarrow 0$, $pg.lw \leftarrow 0$, $pg.pr \leftarrow 0$, $pg.pw \leftarrow 0$
4.      No accounting until after the first physical write
5.  After each logical read of the page:
6.      $pg.lr \leftarrow pg.lr + 1$
7.  After each physical read of the page:
8.      $pg.pr \leftarrow pg.pr + 1$
9.  After each logical write of the page:
10.     $pg.lw \leftarrow pg.lw + 1$
11. Upon eviction of the page:
12. if ($pg$.dirtybit $= 1$)
13.     $pg.pw \leftarrow pg.pw + 1$
14. $q \leftarrow 1 - b/n$
15. $c_f \leftarrow (pg.lr \cdot q + pg.pr) \cdot r_f + (pg.lw \cdot q + pg.pw) \cdot w_f$
16. $c_m \leftarrow (pg.lr \cdot q + pg.pr) \cdot r_m + (pg.lw \cdot q + pg.pw) \cdot w_m$
17. if (($c_f - c_m > w_f + w_m$ and $pg$.state $= f$) or
       ($c_m - c_f > w_f + w_m$ and $pg$.state $= m$))
18.     $pg$.state $\leftarrow$ other state
19.     $pg.lr \leftarrow 0$, $pg.lw \leftarrow 0$, $pg.pr \leftarrow 0$, $pg.pw \leftarrow 0$
20.     $pg$.dirtybit $\leftarrow 1$
```

Figure 3.5: The hybrid algorithm

A page migrates to the other disk if the accumulated cost for the current disk surpasses the cost for the other disk by $w_f + w_m$ cost units (line 18).

Accounting for logical operations when deciding the placement of a page allows hybrid to recognise changes in the workload of the page very early, as does optimistic. However, hybrid is not as eager as optimistic to trigger page migration. It decides that a page should migrate only after it is certain that the page is on the wrong disk (*i.e.*, the cost of migrating to the other disk and back has been already paid). In that sense, hybrid resembles conservative. By taking into account physical costs (*i.e.*, actual costs) the system has a realistic view of the effect of the buffer pool on logical operations.

Note that we have not studied the competitiveness ratio of optimistic and hybrid: our experiments, presented below, show both algorithms to be more competitive than conservative in most cases.

3.3 Experimental study

3.3.1 Experimental Setup

We implemented our algorithms to evaluate their performance under various workloads. Our system consists of a storage manager and a buffer manager and uses B^+-trees for storing data. Though we have implemented other file structures as well (*i.e.*, heap files and linear hash files), we only present results with B^+-trees since they are the most commonly used database structures and make our presentation more succinct. Moreover, B^+-trees have the extra property of exhibiting both random access patterns (*e.g.*, when descending the levels of the tree) and sequential ones (*e.g.*, when scanning the leaves). The system was implemented in C++ and was running on an Intel Pentium 4 box clocked at 2.26GHz with 1.5GB of physical memory. The operating system was Debian GNU/Linux with the 2.6.21 kernel. The system has two magnetic disks and a flash disk. Our system and the operating system ran from one of the magnetic disks. The other magnetic disk (referred to simply as the magnetic disk hereafter) and the flash disk were used to store data pages. The magnetic disk was a 300GB Maxtor DiamondMax 6L300R0 with 16MB of cache memory. The flash disk was a Samsung MCAQE32G5APP, an MLC NAND flash disk with a capacity of 32GB. Both disks were connected to the system using the IDE interface. To reduce the effects of operating system caching we used both storage media as raw devices. Therefore, the operating system did not cache data pages, pages were never double buffered and our system had absolute control of physical I/O operations.

Metadata. As discussed in Section 3.2, the storage manager keeps accounting information for each page. For the conservative algorithm this information is nine bytes per page, of which one byte represents the state of the page and the rest hold a 64-bit integer that represents the accumulated cost for the page. The optimistic algorithm needs one byte for the state of the page and eight bytes for two integers counting logical reads and writes, for a total of nine bytes. The hybrid algorithm requires one byte for the state and sixteen bytes for four 32-bit integers counting logical and physical reads and writes. For a data file of size n bytes, the metadata is $\lceil \frac{n}{4096} \rceil \cdot 9$ bytes for conservative and optimistic and $\lceil \frac{n}{4096} \rceil \cdot 17$ bytes for hybrid, if 4096-byte pages are used. To reduce this amount, one can increase the page size. However the size of extra data is negligible for most practical purposes, as it is three orders of magnitude less than the size of the data. For instance, for the hybrid algorithm (which has the largest requirements), the metadata for a 10GB table are only 44MB. All accounting information is stored on the hard disk with the operating system (*i.e.*, file pages contain only raw data).

We assume the capacity of either disk is enough to hold all data placed on it. The address for a page is the same for both disks (*i.e.*, we only used the first 32GB of the magnetic disk) and no explicit mapping is necessary. In a real deployment, metadata for files, pages, page mappings, and free space on each of the disks would be maintained. This is necessary even for systems using just one disk, so standard file system techniques could be used for that purpose without any additional overhead. To keep the experimental study as simple as possible we chose not to implement these structures, since they would not affect our measurements. Given the current capacities of flash drives, it is conceivable that the flash disk is not large enough to accommodate all read-intensive pages. For such cases, ranking algorithms are required to capture the utility of each read-intensive page being kept on the flash disk.

Raw performance of disks. We measured the read costs for each disk by computing the average of 10^6 read requests of 4096 bytes each at random offsets on the disk. Requested page offsets span the whole disk address space; this is particularly important for the magnetic disk, as measured costs need to reflect the average rotational latency of a read. We similarly measured the average time for random writes for each disk. The results are shown in Table 3.1. The second column is the measured average times, while the third column is the costs normalized by the read time of the flash disk. The flash disk was 23 times faster than the magnetic disk at reading random pages; the magnetic disk was 10 times faster than the flash disk at writing to random locations. It is clear from the relative cost differences that when pages are placed on the correct medium (according to their workload), I/O cost will significantly drop – almost regardless of the access pattern.

Datasets and workloads. We tested the efficiency of our system under a multitude of workloads. The record layout consisted of a key that was sixteen bytes long and a payload of eighty bytes. The B^+-tree contained one million records and had a size of 140MB. To minimize the effect of on-disk caches, the pages of the tree were not stored consecutively on disk, but separated by nine-page intervals (*i.e.*, the pages of the B^+-

Operation	Time (CPU cycles $\times 10^3$)	Cost units
Flash read	915	1
Flash write	108987	118
Magnetic read	21470	23
Magnetic write	10983	12

Table 3.1: I/O costs

tree spanned 1.4GB). We experimented with trees and buffer pools of various sizes. Across all configurations the results were consistent. To avoid repetition we present the results for the aforementioned B^+-tree size and a buffer pool of 20MB. We choose to show the results for this setup as it is more in line with the discrepancy between disk and main memory capacities; one can expect a difference of three orders of magnitude between the two in current configurations. The workloads on the B^+-tree consisted of reads, *i.e.*, lookups and range queries, and writes, *i.e.*, insertions and updates. Each insertion or update to the B^+-tree results in the destination leaf being both read and written, and internal nodes on the path from the root of the tree to the leaf being at least read and potentially written (*i.e.*, in the case of a split). We focus on these simple operations as we aim to show that the placement of the page on the right medium (in addition to our buffer pool replacement policy) is what primarily makes a difference. Our algorithms prove this point true for such basic workloads; we conjecture that it will continue to hold for any complex workload that will use all these primitives.

3.3.2 Impact of using both disks

In the first set of experiments we measure the performance improvement gained by using both types of disk over using only one. We ran the same set of queries in three different setups: (*a*) using only the flash disk, (*b*) using only the magnetic disk, and (*c*) using both disks. In all cases the conservative algorithm decided the placement of pages. Since the workload of a page does not change, the conservative algorithm gives the least performance improvement among all three algorithms. Additionally, we used LRU as the buffer pool replacement policy as it is applicable in all three setups. In the first experiment, we executed a set of $50,000$ read queries (80% of which were lookups and 20% range queries) that targeted all leaf nodes of the B^+-tree. We executed this set of queries 15 times (emptying the buffer pool after each execution) and measured the wall clock time of each run. Results are shown in Figure 3.6 (a) with the query set run shown on the x-axis and total execution time shown on the y-axis (the raw data are given in Figure 8.1). Our system is shown as "*M/F*"; the system using only the flash disk is shown as "*F*"; and the system using only the magnetic disk is shown as "*M*".

As expected, F is much faster than M for reads. The performance of our system is initially equal to that of M, since all pages are first on the magnetic disk. A large number of frequently accessed pages (*e.g.*, the internal nodes of the B^+-tree and some hot leaf pages) migrate to the flash disk during the 4th execution of the query set, while the remaining read-intensive leaf node pages migrate to the flash disk during the 7th execution. Since we are using the conservative algorithm the point in time at which pages migrate depends only on the number of physical accesses to the page. Next, we executed a set of $50,000$ insert/update queries (30% insertions, 70% updates) using the same buffer pool size. Results are shown in Figure 3.6 (b) (the raw data are given in Figure 8.2). In this case, F is one order of magnitude slower than M, as expected. Our system has initially the same performance with M. At the 4th execution the pages storing the internal nodes of the B^+-tree migrate to the flash disk and performance improves slightly (due to the number of insertions being relatively small, internal nodes have mostly a read workload). Had the size of the buffer pool been too small to fit all internal node pages, the performance boost would have been much greater.

Next, we generated mixed query sets including both read and write queries. In the first set, 40% of pages are read-only, 40% are write-only and the remaining 20% have a 50% probability of being read and a 50% probability of being updated. Results are shown in Figure 3.7 (a) (the raw data are given in Figure 8.3). Then, we altered the query set, so that 70% of the pages are read-only and 30% are update-only. The results are shown in Figure 3.7 (b) (the raw data are given in Figure 8.4). As the ratio of pages with a read workload grows, the performance of both F and *M/F* improves, while the performance of M remains almost constant. Clearly, using both a magnetic and a flash disk is more I/O efficient than using only one type of disk, provided that pages are placed on the disk that best suits their workload.

3.3.3 Comparison of page placement algorithms

We then moved on to study how well the page placement algorithms adapt to changing workloads. We created a set of 100,000 read queries and a set of 100,000 update queries. Using these two query sets, we created two different B^+-tree query sequences and executed them using the conservative, optimistic and hybrid algorithms. Addition-

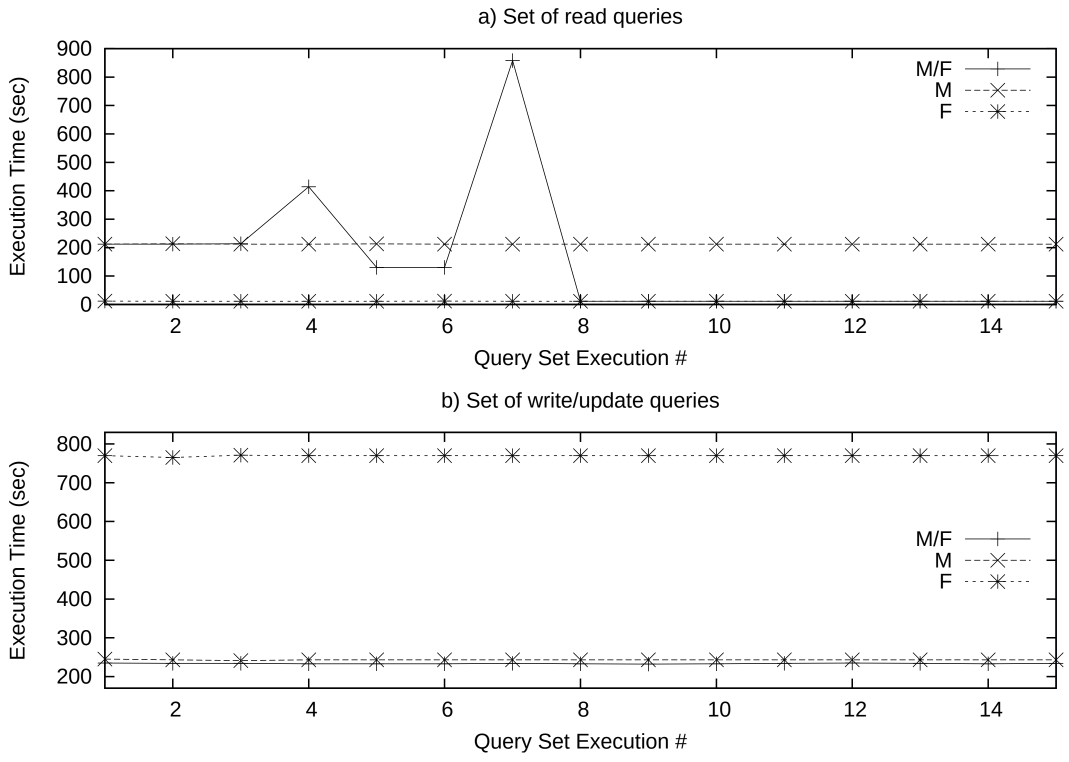


Figure 3.6: Read-only and write-only sets of queries

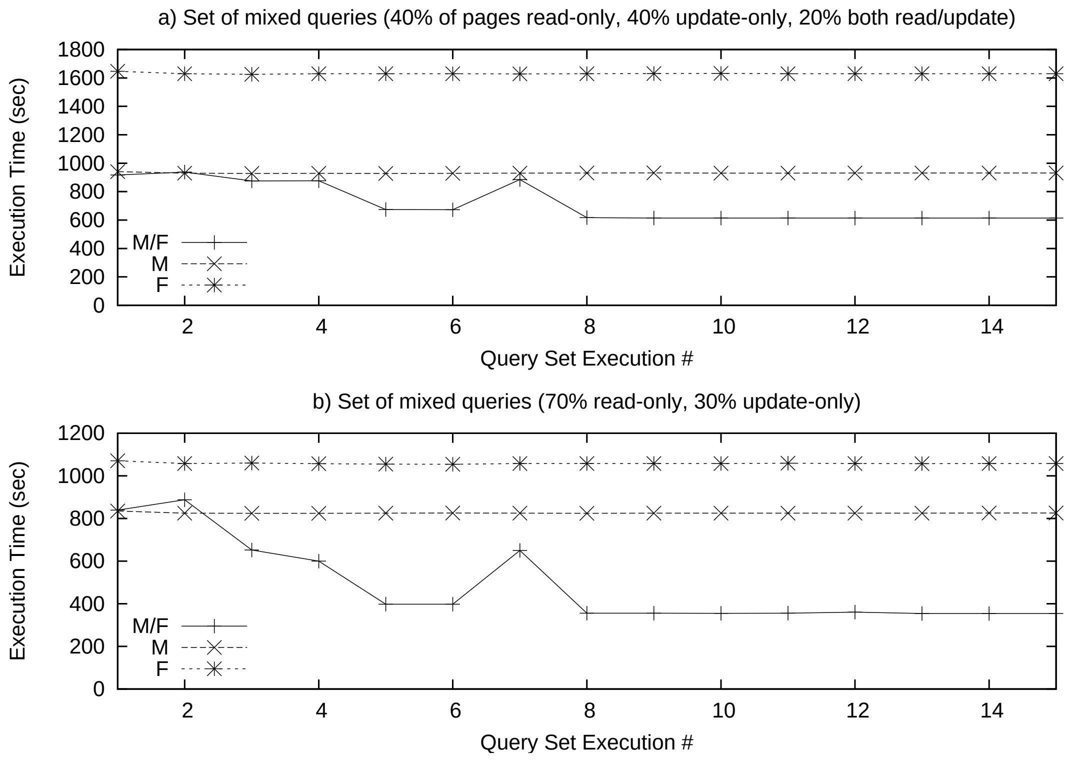


Figure 3.7: Execution of mixed queries

ally, each query sequence was executed using the optimal placement for each page, which we computed off-line. The difference between the two query sequences is the frequency with which the page workload changes. In the first sequence, the set of read queries is executed 10 times, followed by 10 executions of the update query set; then, the read query set is executed again 20 times. In the second sequence, 3 executions of the read query set are followed by 3 executions of the update query set and vice-versa for a total of 18 query set runs. The buffer pool was emptied after each execution. Neither sequence is very likely to occur in real-world workloads; however, they highlight the difference between the three algorithms and their relationship to the optimal one in terms of their adaptability to changing workloads. The results of the two execution sequences are shown in Figure 3.8 and in Figure 3.9 respectively. The raw data are given in Figure 8.5 and Figure 8.6 for the top and bottom graphs of Figure 3.8. For Figure 3.9 the raw are given in Figure 8.7 and Figure 8.8, respectively. On the x-axis, r's stand for read query set executions and u's stand for executions of the update query set. In addition to showing the total execution time on the y-axis, we also show an alternative plot with the y-axis denoting the number of pages being placed on the flash disk by the different placement algorithms, as this gives a more succinct picture of their decisions.

The first sequence shows that optimistic performs nearly optimally, while conservative is the slowest algorithm to adapt to workload changes. The performance of hybrid lies between the performance of optimistic and conservative, *i.e.*, hybrid adapts to workload changes more gracefully than optimistic, but more eagerly than conservative. Updates have a higher impact than reads on the decisions of the algorithms. This is due to update costs for the two disks differing by 107 cost units, while read costs differ by 22 cost units. This is why all algorithms adapt very quickly to the update workload. Also, observe that conservative and hybrid adapt very quickly to the initial read workload, but they adapt much more slowly after the execution of the ten update query sets. This is due to the cost threshold of $w_f + w_m$ having to be surpassed before a migration is triggered. When the workload changes from read- to update-intensive, some pages are read from the flash disk and written to the magnetic disk, which is the best case in terms of I/O efficiency. This explains why the first execution of the update query set, after the 10 executions of the read query set, is executed faster than the following 9 update query executions. Note that the total time for conservative was 9,784 seconds, for optimistic it was 7,003 seconds, for hybrid it was 8,338 seconds, while executing the queries according to the optimal off-line algorithm took 6,366 seconds. The time for executing this sequence only on the magnetic disk was 13,920 seconds and 12,760

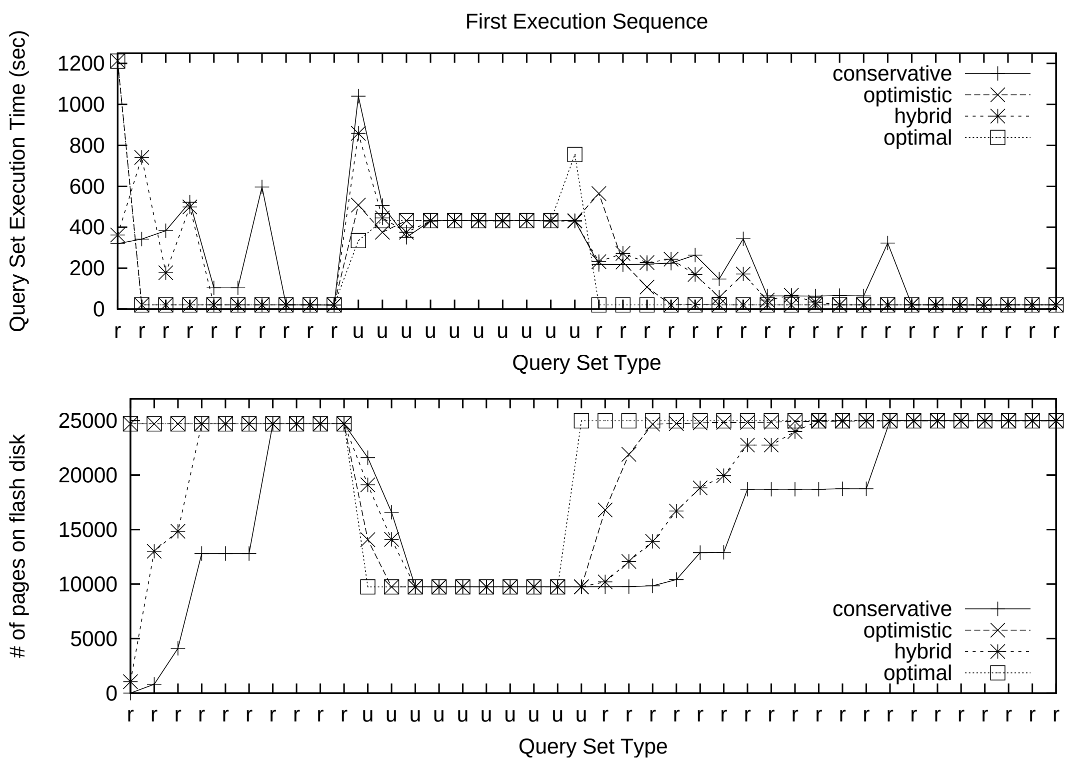


Figure 3.8: Infrequently changing workloads

for executing it only on the flash drive, amounting to a substantial improvement in all cases.

In the second sequence of runs, the workload changes every three executions. Since the cost of a page migration from the magnetic disk to the flash disk is not expensed by the three successive read query set executions, the optimal placement for pages is on the magnetic disk (except for the internal nodes of the B^+-tree). For this reason, the optimal algorithm only places internal node pages on the flash disk. The conservative algorithm places only a small number of pages on the flash disk during the third read query set execution, but places them back on the magnetic disk after the first set of update queries. The optimistic algorithm eagerly places pages on the flash disk, which incurs a great cost when the workload changes. Of course, not all read pages are placed on the flash disk by optimistic, but only the ones that are logically read many times by the read query set. For hybrid, some pages migrate to the flash disk during the second and third query set executions. However, after the first three update query sets, all pages migrate back to the magnetic disk (except for the internal node pages of the B^+-tree). The total times were 3,932 seconds for conservative, 4,560 seconds for optimistic, and 4,060 seconds for hybrid, while executing the queries according to

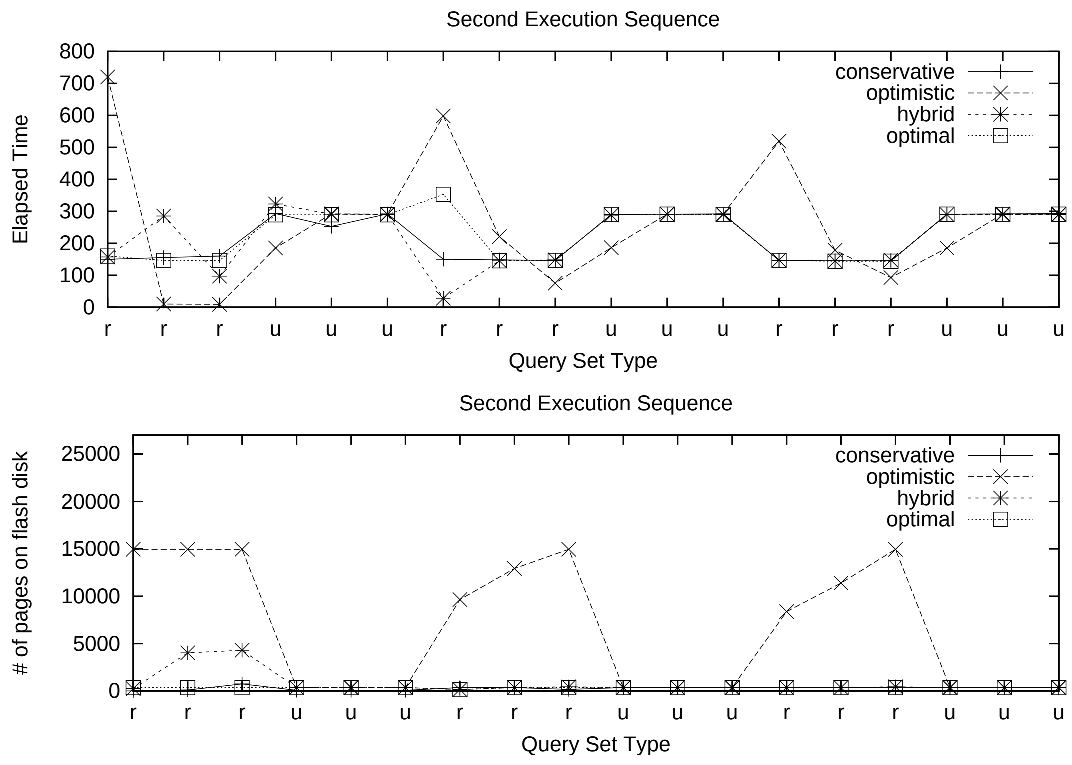


Figure 3.9: Alternating workloads execution

the optimal algorithm took 3,920 seconds. The total time for executing this sequence using only the magnetic disk was 4,150 seconds and 6,552 seconds when using only the flash disk.

Of all on-line algorithms, optimistic gives the greatest performance improvement when it makes the right decisions. However, when workload changes do not last long enough for the migration cost to be paid off, optimistic introduces extra I/O cost due to wrong migration decisions. On the contrary, conservative decides migrations only after workload changes persist for a number of future accesses. Thus, conservative is less likely to make the wrong decision and does not migrate pages with frequently changing workloads. For this reason, however, it improves performance less than optimistic. The hybrid algorithm, by taking into account the decision criteria of both optimistic and conservative, manages to balance its adaptivity between the aggressive behavior of optimistic and the defensive behavior of conservative. Thus, hybrid is more I/O-efficient than conservative, without taking the risks of optimistic that could lead to very poor performance. Therefore, we believe that hybrid is the most appropriate algorithm to decide on page migration and we focus on that algorithm for the remaining sections.

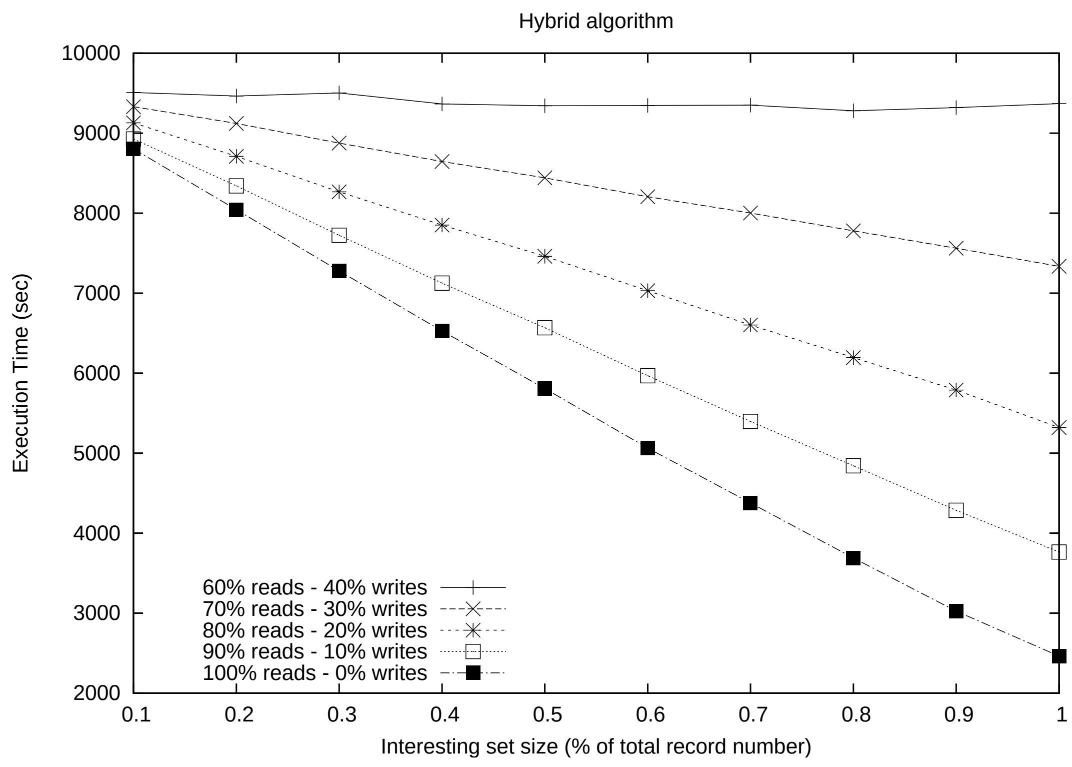


Figure 3.10: Performance in mixed workloads

3.3.4 Mixed workloads

For the next set of experiments we created query sets that have mixed-type queries. We picked a range of record key values (which we refer to as the *interesting* set) and performed B^+-tree operations on these key values with a predetermined probability. All other records had equal 50% read and update probabilities. All records (both the ones in the interesting set and all remaining) had the same probability of appearing in the query set. No set of pages was made artificially hot and the only thing that changed was the ratio between reads and updates among the pages in the interesting set. We varied the read and update probabilities, as well as the range of key values for records in the interesting set. We executed 1,000,000 queries using the hybrid algorithm using a 20MB buffer pool and measured the total execution time. Results are shown in Figure 3.10, while the raw data for the graph are given in Figure 8.9.

Performance improves when the workload of the interesting set pages becomes more read-intensive. For a given number of records in the interesting set, the number of pages that migrate to the flash disk is constant across workloads with different read/update probabilities. However, as the update probability grows, pages of the interesting set (most of which are on the flash disk) become more frequently updated, thus decreasing the performance gain of having them on the flash disk. When the workload is more than than 70% read-intensive, one can see that performance improves as the size of the interesting set increases. This is because more pages migrate to the flash disk where read operations are more efficient. One can also see that when records in the interesting set are updated more than 30% of the time, performance does not improve as the size of the interesting set grows. This is due to leaf nodes that store records of the interesting set not migrating to the flash disk. In this case only internal nodes are placed on the flash disk and thus performance is slightly improved over a system employing only a magnetic disk. When using the magnetic disk only, execution time is comparable to that of the "60% read - 40% write" workload for all workloads and interesting set sizes (with only slight deviations). When using only the flash drive, execution time is much higher due to the pages not belonging to the interesting set being updated 50% of the time (except when the entire dataset is interesting).

3.4 Discussion

Deferred page migrations. The solid state disk we used in our experiments is equipped with a DRAM buffer. The buffer is partitioned into a number of segments (a typical size for each segment is 512kB to 1MB). Its purpose is to temporarily hold the contents of updated blocks (erase units) in order to avoid some erase operations, *i.e.*, to act as a write cache. Each DRAM segment can store a number of *contiguous* blocks. Thus, when writes to sectors are sequential, page sectors that belong to the same block are buffered in a DRAM segment. When the segment becomes full, all updates to the sectors of the block are performed with a single erase operation. Our benchmarks show that writing pages sequentially to the flash disk is over 10 times faster than writing them in a random fashion.

Our system can take advantage of this to further reduce the I/O cost and improve response time. During normal operation, the system can only *mark* pages that should migrate to the flash disk and perform all write operations on the magnetic disk. Then, migrations can be executed sequentially and in the background *e.g.*, when the system load is lower, or when execution of the query that marked them for migration has finished. Such a strategy is sensible for pages that are scarcely updated, or else the benefit of using the flash disk is cancelled.

Accelerating flash writes. As shown in Section 3.3.4, the I/O cost improvement of our system shrinks as the frequency of updates to the flash disk pages grows. To minimize this effect one can employ logging techniques such as the ones presented in [Lee and Moon, 2007] and [Nath and Kansal, 2007], which are complementary to ours. We conjecture that the combination of both techniques will lead to increased I/O efficiency.

Sequential access patterns. A typical access pattern of a database system is a sequential scan. The magnetic disk is more efficient at both reading and writing sequential data. It is conceivable to have the query engine supply hints to the buffer and storage managers whenever such patterns are encountered. As in [Graefe, 1993, Stonebraker, 1981], the buffer manager can use sequential access hints not only for page replacement but also for page placement. In particular, it can employ sequential access costs in the page placement algorithm as opposed to random access ones, thereby favoring the magnetic disk and ensuring that sequentially accessed pages do not migrate to the flash disk. Such hints can be used by the buffer manager as well, since most of times a file is scanned sequentially, pages need not remain in memory after they are accessed, as discussed in [Stonebraker, 1981], [Graefe, 1993].

Chapter 4

Buffer Allocation

4.1 Introduction

Databases and data-intensive applications generally use more than one physical devices to store their data. These devices may have similar I/O cost characteristics or not. In such setups, the solid state drive may be used either as persistent storage ([Koltsidas and Viglas, 2008], [Lee and Moon, 2007]) or as a caching layer between the hard disk and the main memory ([Narayanan et al., 2009]). Even in a flash-only system, the I/O costs of the flash disks vary widely as discussed in Chapter 1. Taking into account the disparate I/O cost characteristics of devices in such setups is of paramount importance if informed decisions with respect to performance are to be made. To a lesser extent, the same holds for homogeneous systems as well, *e.g.*, those using only magnetic disks. At any given time, the I/O efficiency of a disk is heavily affected by (*a*) the access pattern, (*b*) how many concurrent operations access its data (*i.e.*, how long the request queue for the disk is) and (*c*) what portion of the address space of the device is accessed by these requests. Consequently, even identical disks may exhibit very different I/O throughput during some time periods. What is more, the throughput of a faulty disk or a disk under RAID reconstruction is seriously degraded.

The motivating impetus for our work is the observation that storage devices in a multi-device setup compete for main memory buffer space. The system needs to drive this competition based on informed decisions that carefully exploit the I/O cost discrepancy between devices. Our goal is to improve the performance of such a system by allocating the optimal number of main memory buffers to each device. This requires taking into account not only the I/O characteristics of devices, but also the behaviour of the data cached from any device, across different cache sizes. We therefore introduce the notion of device caching *utility* that captures the cost savings realised by allocating page frames to a device. This utility metric is subsequently used to decide how many page frames (that is, what portion of the buffer pool) will be dedicated to the data of each device.

This problem bears some similarity to the problem of partitioning a cache across competing applications (or processes or queries in a DBMS) that store their data on a single device. However, there are two crucial distinctions to be made: (*a*) in our case, device costs play a major role in the I/O cost resulting from each different memory allocation, while in the application domain, only the hit ratio of the data cached by each application affects the I/O cost and (*b*) in the application domain, the competing entities (the applications) are solely responsible for their cache behaviour, *i.e.*, they generate their own workloads. Thus, the workload of each application can be regarded as relatively static through time with respect to its cache behaviour; as such, specific assumptions can be made about it per application. The same holds for each query in a database; it is easy to model the cache behaviour of a query for specific access methods (as in [Chou and DeWitt, 1985], [Ng et al., 1995]). In the device domain, on the other hand, the workload the devices see is generated by applications; most importantly, many applications (or, database queries) may access data from the same device concurrently. Therefore, the workload seen at the device level will be much different from the one generated by each application, tending to be more random as more applications/queries operate on the device. Additionally, changes to the workload seen by a device will be more frequent and more dramatic, than they are for an application, as, for instance, new queries enter concurrent execution in a DBMS and old ones terminate (while the cache behaviour of each such query remains almost the same each time it is executed). The applications' cache behaviour is determined by the *way* it manipulates its data, which is relatively fixed for each application. The devices' cache behaviour is determined by the *data* of the device used at each point in time, which changes frequently and radically. For these reasons, some of the solutions proposed for the application domain are unsuitable for the problem we are studying. An important contribution of our work, however, is that the techniques we present for tracking the hit ratio curve in a cache can be used in the application domain as well, to provide more informed memory partitioning.

Buffer allocation is a problem orthogonal to data placement: placement of pages across the storage media can be decided by the storage manager using arbitrary criteria. We assume data processing that requires demand paging: pages are brought into main

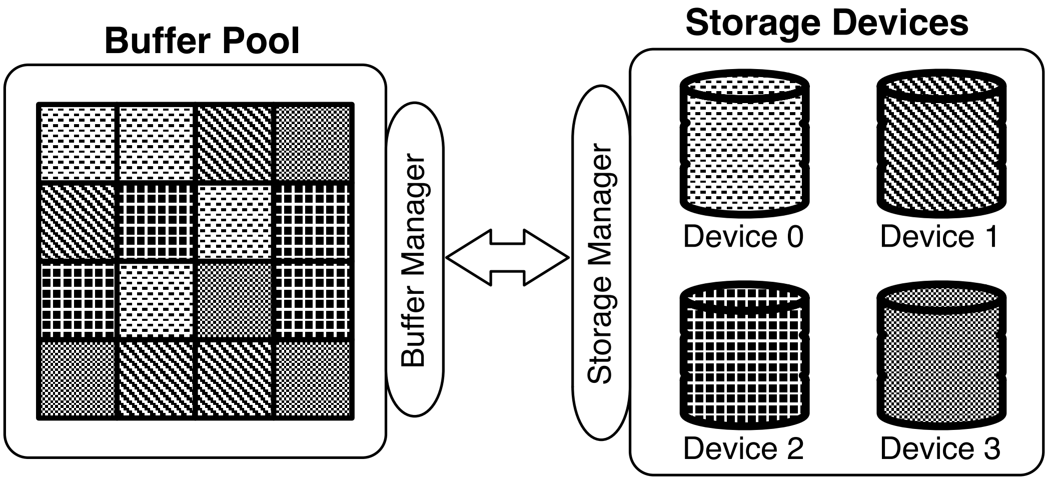


Figure 4.1: Buffering pages from multiple devices

memory before processing, and this happens only on page referencing. When a page is referenced, it is read from the disk on which it is persistently stored into a main memory buffer. Main memory buffers are managed by the buffer manager of the system. When a page needs to be replaced, the buffer manager selects the *victim* page for replacement, according to a replacement policy. If the victim page is *dirty*, *i.e.*, has been written to since it was brought into the buffer pool, it is written back to the storage medium to which it belongs. Then, it is removed from main memory. Such a system, in which the main memory buffer pool contains pages that belong to four different storage devices, is pictured in Figure 4.1.

Caching data pages in the main memory buffer pool (also referred to as *page cache* hereafter) reduces the number of physical I/O operations. A read operation is required whenever a referenced page is not found in the cache, *i.e.*, a *cache miss* occurs, while an additional write operation is required whenever a dirty page is evicted from main memory. The total I/O cost paid for a specific workload is determined by the number of read and write operations on the storage media of the system and the cost of each such operation. If all the underlying media have the same read and write costs, then the I/O cost paid by the system only depends on the miss ratio of the cache and the rate at which pages of the system are being dirtied. However, if the underlying media have varying I/O costs, the total cost paid by the system is determined by the read cost of the missed pages and the write cost of the dirty victim pages. For instance, consider a system like the one studied in Chapter 3, in which the flash SSD is orders of magnitude more read-efficient than the magnetic disk. Consider a clean page p_1 that is stored on the flash disk and a clean page p_2 that is stored on the magnetic disk. These two pages are both cached in-memory at some point in time and assume that they are the ones that have the least probability of reference by future requests among all pages in the buffer pool. If p_1 is chosen for eviction, at the next read operation on p_1 the system will pay the cost of reading from the flash disk; if p_2 is chosen, the system will pay the cost of reading from the magnetic disk, which is many times higher. The best choice for eviction is p_1: if p_2 has not been evicted by the next access to it, the system will have avoided paying the high cost of a magnetic disk read.

In this chapter we study the problem of deciding how many data pages should be cached from each device and on what principle one should replace pages; our goal is to minimise the total I/O cost paid by the system. We start by introducing a static caching scheme, appropriate for hybrid setups such as the one of Chapter 3: buffer allocation and replacement decisions are primarily based on the I/O cost of the storage devices. Then, we formally define the utility metric of caching for each device and propose dynamic and adaptive allocation algorithms based on that.

4.2 Static Allocation

Our goal is to improve I/O efficiency by reducing page misses in the cache (*i.e.*, the total number of I/O operations), while at the same time reducing the I/O cost of each miss. We achieve the latter by taking into account the I/O cost for each page eviction when choosing the next page to evict. We propose a static buffer replacement policy, termed *Cost Based Replacement*, or CBR, that decides on page replacement based not only on access recency, but also on the I/O cost that the replacement decision is likely to incur. CBR is geared towards a hybrid system like the one studied in Chapter 3; in the following we assume such a setup.

The buffer pool is logically divided into two segments: the *time* segment and the *cost* segment, as shown in Figure 4.2. Pages in the time segment are sorted on their timestamp (*i.e.*, the time of their last access). Pages in the cost segment are sorted on their cost of eviction. If the buffer pool is B pages in size, the cost segment size is λB, $\lambda \leq 1$. Of all pages in the buffer pool, the cost segment contains the λB least recently used ones, at any given time. According to our replacement policy, the next page to be evicted is always selected from the cost segment, while new pages fetched from one of the disks are always inserted in the time segment.

The *eviction cost* of a page is equal to the I/O cost of evicting a page and re-fetching

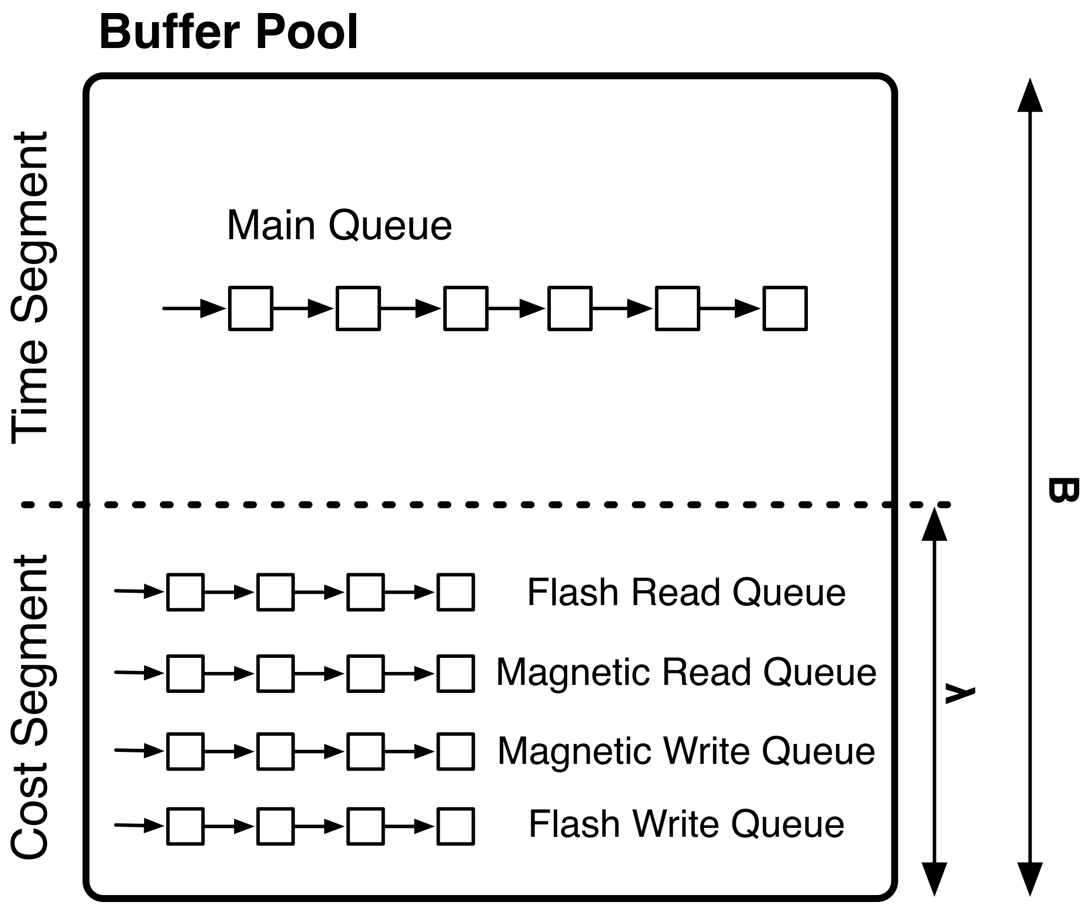


Figure 4.2: Buffer Pool Segments

it on its next reference. Let c_i^r, c_i^w denote the I/O cost of reading a page from device i and writing a page to device i, respectively. Then, the eviction cost of a page is c_f^r if the page is stored on the flash disk ($c_f^r + c_f^w$ if it is dirty) and c_m^r if it is stored on the magnetic disk ($c_m^r + c_m^w$ if dirty). Pages in the time segment are sorted on their timestamp in typical Least-Recently-Used (LRU) fashion. Implementation-wise, a queue (termed *main* queue) is maintained with the timestamps of pages and pointers to them. The front of the queue always refers to the page with the minimum timestamp (*i.e.*, the least recently used page). When a page is accessed (and therefore has the greatest timestamp), it is put in the back of the queue.

The cost segment consists of four queues, one for each eviction cost class, as shown in Figure 4.2: (*a*) the flash read queue (FRQ) holds pointers to non-dirty pages that are stored on the flash disk, (*b*) the flash write queue (FWQ) holds pointers to dirty pages on the flash disk; (*c*) the magnetic read queue (MRQ) and (*d*) the magnetic write queue (MWQ) hold clean and dirty pages stored on the magnetic disk, respectively. Each queue holds its elements sorted on their timestamp, just like the main queue.

The buffer manager maintains a hash index on pages by their page identifier. If a page is in the buffer pool, a lookup in this index returns the queue element that represents the page (which can be an element of any one of the five queues). On access, a page is inserted into the buffer pool as in algorithm fetchPage-static of Figure 4.3. A hash index lookup is performed to check if the page is in the buffer pool (line 1). If it is in the pool and in the main queue (*i.e.*, in the time segment), it is given the current timestamp and moved to the back of the queue (lines 2-4). If it is in a queue of the cost segment, it is removed from that queue, given the current timestamp and inserted to the back of the main queue (lines 5-8). Then, the least recently used page of the time segment (*i.e.*, the front of the main queue) is removed from the main queue and inserted to the back of the cost segment queue that holds pages with the same eviction cost (lines 9-10).

If the page is not in the buffer pool it is read from the disk on which it resides. If the pool is full, a page is evicted using algorithm evictPage-static of Figure 4.4 (line 13). The requested page is read from disk, given the current timestamp, and added to the back of the main queue (*i.e.*, in the time segment). If the size of the time segment is less than $(1-\lambda)B$, it means there is room in the time segment so the page is inserted there. Otherwise, all new pages will be inserted into the cost segment, until the pool is full. Thus, when the pool becomes full, the size of the time segment is $(1-\lambda)B$ and the size of the cost segment is λB. After page eviction, the size of the time segment is less than $(1-\lambda)B$ (*i.e.*, $(1-\lambda)B-1$) while the size of the cost segment remains λB.

Page evictions are decided by algorithm evictPage-static shown in Figure 4.4. The page to be evicted is the front element from the non-empty queue that holds pages with the least eviction cost (lines 2-5). If the page is dirty (*i.e.*, it comes from either MWQ or FWQ), it is written to disk (line 6). Then, it is removed from the queue in which it resided and deleted from main memory (lines 7-8). Finally, the least recently used page of the time segment is removed and placed into the cost segment, by being appended to the back of the appropriate queue (lines 9-10). The cost segment maintains the same size and a free page is created, so the page to be read after the eviction can be inserted in the time segment.

Observe that for each page access or page eviction the complexity of our algorithm is constant in the size of the buffer pool. All operations on queues are $O(1)$ and both fetchPage-static and evictPage-static incur a constant number of operations on queues (one at least, two at most). Additionally, each hash index lookup is also $O(1)$. Conse-

```
Algorithm fetchPage-static (PageId pid)

1.  pg ← hash_lookup(pid)
2.  if (pg found in main queue)
3.     give pg a new timestamp
4.     move pg to the back of main queue
5.  else if (pg found in cost segment queue q)
6.     remove pg from q
7.     give pg a new timestamp
8.     add pg to the back of main queue
9.     pg' ← the front element of the main queue
10.    insert pg' to the back of cost segment queue q'
       that holds pages with cost evict_cost(pg')
11. else
12.    if (buffer pool is full)
13.       evictPage-static ()
14.    read pid from disk into pg
15.    give pg a new timestamp
16.    if (size of main queue < $(1-\lambda)B$)
17.       add pg to the back of main queue
18.    else
19.       insert pg to the back of cost segment queue
          that holds pages with cost evict_cost(pg)
```

Figure 4.3: Algorithm fetchPage-static

quently, the complexity of our algorithm is only greater than the complexity of LRU by some constant c.

4.2.1 The effect of λ

The value of λ affects the efficiency of the algorithm. For simplicity, consider that pages in the buffer pool are clean. As λ grows, the number of magnetic disk pages in the buffer pool grows, while the number of flash disk pages in the buffer pool shrinks (because FRQ pages are evicted first, they will typically be found only in the main

```
Algorithm evictPage-static ()

1.  Page $pg$;
2.  if (FRQ is not empty) $pg \leftarrow$ front of FRQ
3.  else if (MRQ is not empty) $pg \leftarrow$ front of MRQ
4.  else if (MWQ is not empty) $pg \leftarrow$ front of MWQ
5.  else  $pg \leftarrow$ front of FWQ
6.  if ($pg$ is dirty) write $pg$ to disk
7.  remove $pg$ from the queue it belongs to
8.  delete $pg$
9.  $pg' \leftarrow$ front of main queue
10. insert $pg'$ to the back of cost segment queue $q'$
    that holds pages with cost evict_cost($pg'$)
```

Figure 4.4: Algorithm evictPage-static

queue which decreases in size with increasing λ), *i.e.*, the hit probability of a magnetic disk page increases, but so does the miss probability of a flash disk page.

Using the experimental setup of Chapter 3, we run a set of micro-benchmarks on a B^+-tree to measure the impact of CBR and the effect of λ on the I/O cost. We experimented with two different query sets, using different values for λ. In the first query set (query set A), 50% of B^+-tree leaf nodes have a read workload and the remaining 50% of leaf nodes are read and updated with equal probability. Therefore half of the leaf pages are placed on the flash disk and the remaining are placed on the magnetic disk (of course, internal pages are placed on the flash disk, since no insertions occur). All measurements are taken after pages have been placed on the appropriate disk. All leaf nodes have an equal probability of being referenced by a query. Query set A consists of 100,000 queries. We measured the total execution time as we varied λ and plotted the results in the top part of Figure 4.5. When $\lambda = 0$, the replacement policy degenerates to simple LRU, while when $\lambda = 1$ the whole buffer pool is used as a cost segment, with page replacement decided only by the eviction cost of a page.

One can see that performance improves as the value of λ increases, for values of λ up to 0.9. This is because pages on the flash disk and pages on the magnetic disk are accessed with the same probability. When pages corresponding to internal nodes of the tree (stored on the flash disk) do not fit in the time segment, *i.e.*, for $\lambda > 0.9$, they are

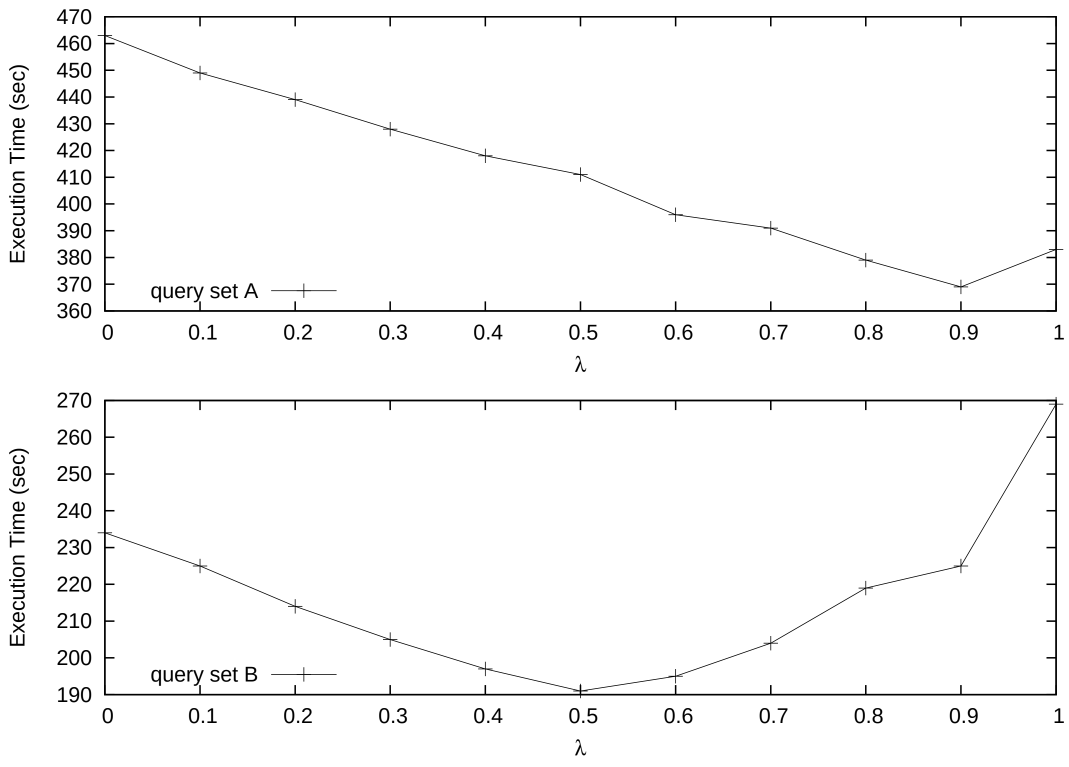


Figure 4.5: Replacement policy for different λ

evicted in favour of pages on the magnetic disk. Then, each leaf page access requires 3 physical flash read operations (the depth of the tree was 4), and thus performance degrades. Performance with $\lambda = 1$ remaining better than with simple LRU has to do with flash pages not being particularly hot. The performance benefit of our replacement policy reaches 21% in this case.

In the second set of queries (query set B), 10% of the B^+-tree leaf nodes have a read workload, while the rest are read and updated with equal probability. The set consists of 150,000 queries. However, only 15,000 queries access leaf pages on the magnetic disk, while the remaining $135,000$ access leaf pages that are on the flash disk (*i.e.*, 10% of all leaf pages); thus, 10% of the B^+-tree leaf pages are hot. The results, as we varied λ, are shown in the bottom part of Figure 4.5. The best performance is for $\lambda = 0.5$: 15% better than the performance of LRU. As λ grows greater than 0.5 performance degrades. More than half of the buffer pool fills up with magnetic disk pages that are scarcely accessed, while for most accesses to flash disk pages (which amount to 90% of all accesses) a miss occurs. When internal node pages do not fit in the time segment (for some $\lambda > 0.9$), performance again degrades and becomes even worse than the performance of LRU.

The problem with CBR is that its performance depends heavily on a wise choice for the value of λ for a particular workload. For instance, if the low eviction cost pages (*e.g.*, clean flash pages) are particularly hot in a workload, while the pages stored on the magnetic disk are cold, then a very small value for λ serves best in terms of efficiency. On the other hand, the hotter the high-eviction-cost pages are, the greater the value of λ should be to force more pages of the high-cost classes to remain in main memory. Moreover, λ determines the portion of the buffer pool that will be managed based on cost, but does not determine the number of pages cached from each device on a *per-device* basis; thus, even if the workload of devices were known CBR could not enforce that a specific number of pages from the cost segment be given to a device that sees high heat.

4.3 Dynamic Allocation

We now turn to dynamic allocation algorithms. Our goal is to overcome the weaknesses of CBR, most importantly that (*a*) the wise choice for λ is crucial to the performance of the system under a specific workload and (*b*) CBR does not adapt λ when the workload changes. To that end, the system needs to keep track of specific characteristics of the workload of each device. Before introducing our algorithms for dynamic allocation, we present an overview of existing techniques for tracking the behaviour of caches.

4.3.1 Theoretical Background

To allocate the optimal number of buffers to a specific device, being able to predict the hit ratio for the data of the devices under various caches sizes is of paramount importance, if informed decisions are to be made. In [Mattson et al., 1970] Mattson *et al.* propose an algorithm that tracks the hit ratio for a workload for caches of all sizes up to S in a single pass, by running it using a cache of size S. The only constraint for this algorithm is that it requires a stack replacement policy, such as LRU, while most modern replacement policies have not been shown to be such. In [Kim et al., 2000] authors argue that Mattson's algorithm is impractical due to the overhead of measuring hit distances and propose a method for analytical approximation of the hit ratio curve, based on Belady's lifetime function. The same approach is followed by [Choi et al., 2000]. Their analytical approximation, however, is also specific to LRU and based on assumptions on the shape of the hit ratio curve. In this work, on the contrary, we provide an efficient and accurate technique for measuring hit distances, with constant time complexity, which is applicable to all stack replacement algorithms. In this manner we eliminate the overhead preventing the use of Mattson's algorithm in [Kim et al., 2000], [Choi et al., 2000]. In [Zhou et al., 2004] the authors also propose a technique for estimating hit distances efficiently; an experimental comparison with our own technique suggests that ours is preferable for use in real-world deployments (see Section 4.7.1).

A similar problem to ours is the one of allocating memory to different processes according to their cache hit rates. In that context, the authors of [Stone et al., 1992] assume that the miss-rate functions of processes are convex and propose a cache partitioning algorithm for both fully- and set-associative processor caches. The key idea is that the optimal partitioning appears at a point where the miss-rate of the competing processes derivatives are equal, which was proven in [Ghanem, 1975]. They assume LRU as the replacement policy and approximate the miss ratio curve for each process analytically. In [Thiébaut et al., 1992] the authors extend that work to disk caches and study how hit counts and statistics should "age" with time, to provide adaptability. The problem of allocating buffers to processes is also studied in [Zhou et al., 2004] and authors propose a greedy algorithm for memory partitioning, once the miss ratio curve is known. In [Soundararajan et al., 2008] the authors study the same problem in the context of database applications running on a storage system. They collect samples of each application's latency for different cache partitioning configurations and employ support vector machine regression to approximate the per-application performance models. Their goal is to meet the QoS requirements of the applications and further maximise the revenue for the service provider, if that is possible. In [Soundararajan et al., 2009] their work is extended to also provide disk bandwidth partitioning among applications. The main drawback of their approach is that they use statistical tools that require sampling for each application. In their experiments they report that the time to collect the samples reached 30 minutes, which is not acceptable for the case of per-device cache partitioning. Also, their main goal is not to improve throughput, but to meet the QoS requirements for each application. In their work, applications compete for both buffer space and disk bandwidth, while in our case, the buffer manager has complete control of the bus.

In the database world, authors of [Ng et al., 1995] propose a method for allocating buffers to queries, based on the marginal gain that each query will get if more buffers are allocated to it. Their methods though, assume specific database access patterns. The same approach is followed by [Chou and DeWitt, 1985]. However, in our case, where buffers are allocated per storage device, multiple queries may operate on the same device at the same time and therefore specific access patterns cannot be assumed. Thus, hit rates for per-device data cannot be predicted. Of course, such techniques are applicable one level higher, that is, within the buffer space allocated for each device.

4.4 Problem Formulation

Terminology. Departing from the system studied in Chapter 3 and Section 4.2, we now turn to a generic system that consists of many storage devices. In the general case, the storage component of the system consists of n storage devices, while there are S buffer pool pages available in main memory (page frames). Let c_i^r be the cost of reading a random data page from device i into a main memory buffer and c_i^w be the cost of writing a data page from a main memory buffer to device i. Similarly to the case of static allocation, we define the *eviction cost* of a page x, or $e(x)$, to be the cost of evicting the page from the buffer pool and re-fetching it on its next reference. Assuming that page x is stored on device i, it follows that $e(x) = c_i^r$ if the page is clean at the time of eviction and $e(x) = c_i^w + c_i^r$ if the page is dirty. It also follows that all clean pages (and dirty ones, respectively) that belong to the same device have the same eviction cost. Additionally, let s_i be the portion of the cache devoted to device i at some point in time, *i.e.*, the number of page frames that are occupied by pages that belong to device i; we refer to s_i as the *cache size for device i*.

Page replacement may be done on a per-device basis, that is, the buffer manager may enforce a different replacement policy for the pages of each device. Let Y_i be the replacement policy based on which the buffer manager replaces pages of device i. Furthermore, let W be the global workload during a time interval, *i.e.*, the sequence of page references generated by the system during that time to pages that belong to all storage devices. Then, we define the *workload of device i*, or w_i, to be the subsequence of W that references pages stored on device i. We refer to the number of page hits and misses that occur in the cache for pages stored on device i as *device i hits*, or h_i, and *device i misses*, or m_i, respectively. The *heat* of a device is the percentage of total references that target pages of that device, or $\frac{|w_i|}{|W|}$. Assuming that during the execution of w_i, the device cache size s_i is fixed, then the number of hits and misses for device i only depends on the size of the device cache (s_i), the replacement policy used for that particular device (Y_i) and the device workload itself (w_i). With $H(Y,s,w)$ giving the hit ratio occurring under replacement policy Y for a cache with size s for the execution of reference sequence w, we have that $h_i = H(Y_i, s_i, w_i) \cdot |w_i|$.

I/O cost. For each page hit in the cache, no I/O cost is paid by the system, as the page access is served in-memory. If a page reference results in a page miss in the cache, and assuming normal operation (the cache is warmed up), the missed page will be fetched from the device to which it belongs and a victim page will be removed from the cache and written back to the device to which it belongs, if it is found to be dirty. With p_i^d denoting the probability that a page of device i is found dirty upon its eviction, the cost paid for the pages of device i is $C_i = (|w_i| - h_i) \cdot (c_i^r + p_i^d \cdot c_i^w)$ and the total I/O cost paid by the system is equal to:

$$C = \sum_{i=0}^{i=n-1} C_i = \sum_{i=0}^{i=n-1} (1 - H(Y_i, s_i, w_i))|w_i|(c_i^r + p_i^d \cdot c_i^w)$$

Inversely, the *utility* of the buffer pool is equal to the number of cost units saved by page hits. Thus, the cost units saved by caching s_i pages of device i is $C_i' = h_i \cdot (c_i^r + p_i^d \cdot c_i^w)$ and the total I/O cost saved is equal to:

$$C' = \sum_{i=0}^{i=n-1} C_i' = \sum_{i=0}^{i=n-1} H(Y_i, s_i, w_i)|w_i|(c_i^r + p_i^d \cdot c_i^w)$$

Our goal is to minimise the total cost paid by the system C, under the constraint that $\sum_{i=0}^{i=n-1} s_i = S$, or, equivalently, maximise C'. Naturally, the formula for C confirms the intuition that the total I/O cost drops when the hit ratio for device caches increases and when the read and write costs of the devices are low. In addition, the more dirty pages are chosen for eviction, the higher C will be. In what follows we assume that the replacement policy for each device is fixed, that is, either the "best" replacement policy has been chosen *a priori* for the workload of the device or the same replacement policy is used for the caches of all devices. Thus, the only parameter in the above equation tunable by the system is s_i, that is, the portion of the buffer pool that is dedicated to pages of device i, which effectively determines H and, possibly, p_i^d. Choosing the optimal value for each s_i is the goal of the rest of this chapter.

4.4.1 Eviction Cost Classes

As introduced above, the eviction cost of an in-memory page x that belongs to device i is equal to c_i^r if the page is clean and equal to $c_i^r + c_i^w$ if the page is dirty. This classifies the pages of a device cached in-memory into two classes *w.r.t.* their eviction cost: the

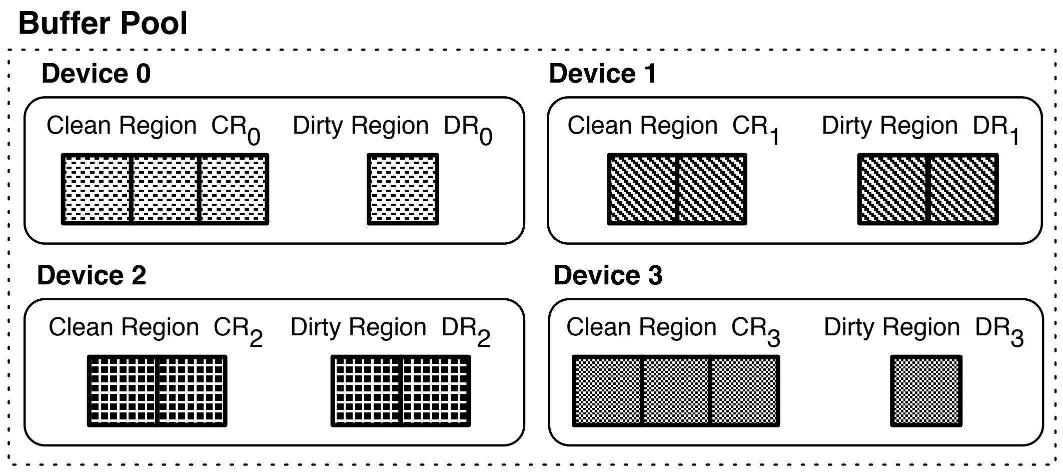


Figure 4.6: Regions for pages with the same eviction cost.

clean pages class and the *dirty pages class*. Conceptually, the portion of the buffer pool devoted to device i can be thought of as consisting of two separate *regions*: one that holds the clean pages of the device (the *clean region* or CR_i) and one that holds the dirty ones (the *dirty region* or DR_i). Observe that CR_i can be thought of as the cache of a read-only device, while DR_i can be thought of as the cache of a device, all the pages of which get dirtied when brought in-memory.

For reasons of simplicity in our analysis, we therefore use the notion of regions occupied by pages that belong to the same eviction class, *i.e.*, have the same eviction cost (whereas not all pages belonging to one device have the same eviction cost). In Figure 4.6 the buffer pool of Figure 4.1 is shown with $s_i = 4$ for all i. In this example, the system has assigned 3 page frames to the clean region of devices 0 and 3 and 1 page frame to their dirty regions, while 2 frames have been assigned to both the clean and dirty regions of devices 1 and 2.

Extending our previous definitions to account for an analysis based on regions, rather than devices, is straightforward. Upon a reference to a page that belongs to device i, the page is looked-up in both the clean and the dirty region of the device. Since the clean region only caches clean pages, while the dirty region only caches dirty pages, the referenced page will either be found in the clean region or the dirty region or in neither of the two. If it is found in one of the regions, a page hit occurs for the corresponding region. Otherwise, the page is read from the device and placed in the clean region (after replacing a page from that region, if it is found full). When a clean page gets dirtied by the workload, it is moved from the clean region of the device to its dirty region (replacing a dirty page if the dirty region is full). Note that each reference to a page of device i may result in a hit in exactly one of the two regions associated with the device or a miss in both regions. In other words, the concept of a page reference only makes sense when referring to a device, not a region, as references do not target a specific region, as the requested page may or may not be dirty. The same holds for page misses. For instance, if a referenced page is found neither in the clean region of the device nor in the dirty one, then one cannot characterise this exclusively as miss to either of the two.

In a system that uses n storage devices, there will be $2 \cdot n$ regions in the buffer pool, two for each device. In that sense, one can think of the buffer pool as a collection of $2 \cdot n$ regions, instead of a collection of n devices. Using similar terminology to the one used for devices, a region j has size s_j and an eviction cost e_j equal to c_i^r if it holds clean pages of device i or $c_i^r + c_i^w$ if it holds dirty pages from that device. The workload seen by that region is then $w_j \equiv w_i$, *i.e.*, the subsequence of W that references pages of device i and the replacement policy for the region is Y_j. Note that the clean region of device i may replace pages using a different replacement policy than the dirty region for that device; although there is no apparent reason for using a different policy for the two regions (since they see the same workload), our model provides the freedom to the user to do so. Then the hit ratio for a region j is equal to $H(Y_j, s_j, w_j)$ and the number of occurring hits $h_j = H(Y_j, s_j, w_j) \cdot |w_j|$. The cost units saved by a hit on the region are equal to the eviction cost of the region e_j. Thus, the utility of caching s_j pages in region j is measured by the cost units saved by hits on that region, that is, $C'_j = H(Y_j, s_j, w_j) \cdot |w_j| \cdot e_j$. In other words, the utility of caching pages of that region (a) grows with H, the effectiveness of the replacement policy (Y_j) for a cache of that size (s_j) and workload (w_j), (b) grows with the heat of the device i to which the region belongs ($|w_j| = |w_i|$) and (c) grows with the eviction cost for that region (e_j). The cost units saved by the whole buffer pool are given as follows:

$$C' = \sum_{j=0}^{j=2n-1} C'_j \Rightarrow C' = \sum_{j=0}^{j=2n-1} H(Y_j, s_j, w_j) \cdot |w_j| \cdot e_j$$

Our goal is to maximise the utility of the buffer pool, *i.e.*, maximise the cost units saved by hits on the $2 \cdot n$ regions. In other words, we aim to distribute the S page frames to the $2 \cdot n$ regions so as to maximise C'. Note that using a model based on regions instead of devices, the cost of writing back dirty pages is incorporated in the eviction cost of the regions, thereby eliminating the parameter p_d introduced in the analysis for the device-based model.

Replacement Policies. In this work we assume that replacement policies for all devices do not exhibit Belady's anomaly [Belady et al., 1969], *i.e.*, they are not based on FIFO queues. Therefore the hit ratio for the policy grows with cache size (that is H grows as s_j grows); this holds for virtually all modern page replacement algorithms. In the general case, however, one cannot know exactly how H changes with s_j (the *hit ratio curve*), unless very specific characteristics of the workload and the replacement policy are priorly known, which is infeasible in real-world systems. For stack replacement policies, such as LRU, the hit ratio curve can be efficiently computed for a specific workload [Mattson et al., 1970]. For such cases, we propose techniques that can take advantage of the hit ratio curve to make more informed decisions.

4.4.2 Adaptability

In our analysis so far we have only considered static workloads, that retain the same characteristics over time, with respect to page and device heat. For such a case, there is an optimal distribution of the S page frames to the $2 \cdot n$ regions, that maximises C' throughout the whole workload. However, this is not realistic in real-world workloads, in which not only the heat of devices, but also the heat of pages varies widely throughout the workload. For instance, assume a relation A stored on device 0 and relation B stored on device 1. If A is hotter than B, then increasing the size of CR_0 and DR_0, and decreasing CR_1 and DR_1 by an equal amount of page frames, will most probably yield a much higher hit ratio for the whole workload. However, if later in the workload B becomes hotter than A, keeping region sizes fixed is obviously a wrong decision; rather, the regions for device 1 should now grow, acquiring pages from the regions of device 0, which is now cold. A similar situation may arise even between the two regions of a device, when for instance the workload of the device changes from write-intensive to read-intensive and vice-versa. What is more, one cannot reach optimal decisions based only on hit ratios; rather, the eviction cost of each region should also be taken into account when trying to minimise the I/O cost of the system. For instance, if page hits for a region j are twice as many as the hits for a region j', but the eviction cost of j' is ten times higher than the eviction cost of j, then in order to minimise the total I/O cost, one should cache more pages of region j', rather than j. If the read and write costs of the storage devices change over time, the system needs to adapt to those changes as well.

It should be clear from the above that keeping the sizes of regions fixed throughout a workload is very unlikely to yield the optimal I/O cost for the system. Rather, the system should be adaptive and change region sizes as the characteristics of the workload change over time. In the following sections we propose algorithms that can detect workload changes and adjust region sizes accordingly. The key idea of the algorithms presented is that they estimate how the utility of each region changes through time and how it is expected to change when the size of each region changes. Then, regions with high utility are given more page frames (they *grow*), while regions with low utility are downsized (they *shrink*) giving some of their page frames to the growing ones. The key difference between the proposed algorithms is the metrics they used to capture the utility of regions and to predict how the utility of a region will change if the region either grows or shrinks.

4.5 Algorithms

We now turn to the presentation of our algorithms for maintaining the optimal size for each region in the system. First we present two algorithms suitable for any replacement policy and then we turn to algorithms for systems with stack replacement policies, that take advantage of the hit ratio curve to make more informed resizing decisions.

4.5.1 Baseline Resizing Algorithm

We propose two algorithms that adapt the size of each region using different criteria. However, the workflow for both algorithms is the same and, therefore, before describing in detail the two alternative criteria, we present the workflow of the baseline resizing algorithm. The baseline algorithm for fetching a page into the buffer pool is given in Figure 4.7. For a device i, we keep track of the number of page references r_i the device has seen and for a region j we keep track of the number of page hits that have occurred in that region, h_j. Initially, we determine the device to which the requested page belongs and increase the number of references for that device (Lines 1-2). Then, a lookup is done in the clean region of the device; if the requested page is found there, the hit counter for that region is incremented, the utility metric for the region is updated by updatePriority and the page is returned (Lines 3-6). Note that updatePriority is implemented differently for the two proposed algorithms and will be discussed separately for each. If the page was not found in the clean region, the same is done for the dirty region of the device (Lines 7-10). If the page is not found there, a page miss has occurred in the buffer pool.

```
Algorithm fetchPage (Page $pg$)

1.  $i := device(pg)$
2.  $r_i := r_i + 1$
3.  if $pg$ found in $CR_i$
4.      $h_{CR_i} := h_{CR_i} + 1$
5.      updatePriority ($CR_i$)
6.      return $pg$
7.  if $pg$ found in $DR_i$
8.      $h_{DR_i} := h_{DR_i} + 1$
9.      updatePriority ($DR_i$)
10.     return $pg$
11. $j' :=$ PickEvictionRegion()
12. Evict a page from $j'$ according to its replacement
    policy, writing it back to the device it belongs to,
    if it is dirty, that is, if $j'$ is a dirty region
13. $s_{j'} := s_{j'} - 1$
14. $s_{CR_i} := s_{CR_i} + 1$
15. updatePriority ($j'$)
16. updatePriority ($CR_i$)
17. Read $pg$ from device $i$
18. Insert $pg$ into $CR_i$
19. return pg
```

Figure 4.7: Baseline algorithm for fetching a page

Assuming normal operation, *i.e.*, the buffer pool is full, at that point a page will need to be evicted, to make room for the new page *pg*. The region from which the page will be evicted will shrink by 1 page frame, while the region into which the new page will be inserted will grow by 1 page frame. Therefore, at that point the algorithm decides which region should shrink; the one to grow will always be the clean region of the hit device. The region that will shrink is chosen by pickEvictionRegion based on the cost utility of each region; at this point the two proposed algorithms also differ and therefore pickEvictionRegion will be discussed separately for each. This procedure will return the region with the least cost utility among all regions. Note that the returned region may be the hit one, if it is found to be the least cost effective. After the eviction region is picked, the victim page is chosen by the replacement policy for that region. If the eviction region is a dirty one, then the victim page will be dirty and will need to be written back to the device which it belongs to (Lines 11-12). Now the free page

```
Algorithm touchPage (Page $pg$)

1.  $i := device(pg)$
2.  if $pg$ found in $DR_i$
3.      return
4.  else /* $pg$ is in $CR_i$ */
5.      Remove $pg$ from $CR_i$
6.      $s_{CR_i} := s_{CR_i} - 1$
7.      $s_{DR_i} := s_{DR_i} + 1$
8.      updatePriority ($CR_i$)
9.      updatePriority ($DR_i$)
10.     Insert $pg$ into $DR_i$
```

Figure 4.8: Baseline algorithm for dirtying a page

frame of the eviction region is given to the hit region, *i.e.*, the clean region of the hit device (Lines 13-14). The utilities for regions j' and CR_i have now potentially changed and are therefore updated by updatePriority (Lines 15-16). Next, the referenced page can be read from device i into the free page frame that was given to CR_i and can be returned to the user (Lines 17-19).

The algorithm for making a page dirty is shown in Figure 4.8. Note that when touchPage is called the page will already have been fetched into the buffer pool and will therefore reside either in the clean region or the dirty region of the device the page-to-be-dirtied belongs to. Dirtying a page means moving it to the dirty region of the device it belongs to. Initially, that device is identified and a lookup is done to its dirty region; if it is found there, nothing more needs to be done (Lines 1-3). Otherwise, the page is in the clean region of the device. It is thus removed from it and inserted into the dirty region of the device. As a result, the clean region shrinks by one page frame, while the dirty region grows by one (Lines 4-7). Then the utilities for the two resized regions are updated by updatePriority, as they may have changed (Lines 8-9). Note that if decreasing the size of the clean region by one is a “wrong” decision from a cost utility point of view, at the next call of pickEvictionRegion on a reference to a page of that region, this will be identified and the clean region will grow again. Last, the dirtied page is inserted in the dirty region for the device (Line 10).

Capturing Utility. Following the analysis of Section 4.4, the most straightforward thing to do is to capture the cost utility of each region by calculating $C'_j = H(Y_j, s_j, w_j) \cdot |w_j| \cdot e_j$ at each call to updatePriority. Using the terminology of the algorithms, at each call to updatePriority one would need to compute $h_j \cdot e_j$ in order to rank the regions by their cost utility . Then, the region with the minimum cost utility would be selected by pickEvictionRegion and returned to fetchPage, as it would be the one that should shrink. On the other hand, the hit region would grow, since it would be more (or, equally) cost effective as the one returned by pickEvictionRegion.

We experimented with this criterion for cost utility, but found it not to be adaptive in most cases. The reason is that it takes into account the number of hits a region has seen when deciding its utility. Although this is not fundamentally flawed, imagine the following situation: a region becomes cold at some point and sees no references (and thus hits) and therefore shrinks to a very small size, or even to 0. Then, suppose that this region subsequently becomes hot again. Since its size is very small, very few hits will occur, or even none at all, regardless of the heat of the region and the effectiveness of the replacement policy. Therefore, this region will always be found to be the least cost effective and will never get a chance to grow, although it has become a very hot one that could potentially give a very large number of hits. To avoid this situation, we provide two alternative criteria for estimating the cost utility of a region: (*a*) one that does away with page hits in the first place and (*b*) one that uses ghost caching to allow growing of regions that have reached a very small size.

4.5.2 Reference-based Utility Criterion (RUC)

The first, and simplest, of the algorithms, termed RUC, makes the following assumption regarding $H(Y_j, s_j, w_j)$:

Assumption 5.0: The hit ratio function $H(Y,s,w)$ depends only on s, *i.e.*, $H(Y,s,w) = H(s)$ and is the same for all regions. □

In other words, the effect of Y_j and w_j is the same on H for all regions and thus the hit ratio of two regions j, j' is expected to be the same when $s_j = s_{j'}$; the number of hits a region sees, thus, only depends on the size of the region and the heat of the device it belongs to. This assumption is valid when all regions use the same replacement policy and have similar workloads. Of course, it is not very accurate for many real-world workloads. Our experiments of Section 4.7, however, verify that, even for real-world workloads, RUC performs very well, even under this simplifying assumption.

As discussed in Section 4.5.1 determining the utility of a region by the number of hits it has seen is very prone to leading to a non-adaptive behaviour. Under RUC, the utility of region j is decided by the heat of the region (which under our assumption determines the number of hits for a specific size of that region), that is, the number of references it has seen, and its eviction cost. Furthermore, RUC adopts the concept of *per-page utility*, *i.e.*, the cost utility that each page contributes to the utility of the region. The intuition behind this is that if two regions have the same utility, then the one that has less per-page utility (more pages) should shrink, as it utilises its page frames less efficiently. Therefore, the measure of cost utility used by RUC for region j is computed as $U_{RUC}(j) = \frac{r_j \cdot e_j}{s_j}$.

The pseudocode for updatePriority and pickEvictionRegion for RUC is given in Figure 4.9. We use a Red-Black tree to hold the utilities of regions, with each element of the tree holding the utility of region j and a pointer to the region itself. In updatePriority, the utility for the region passed as parameter is computed and its entry is updated in the Red-Black tree *rbt*. Then, in pickEvictionRegion the element with the minimum value is extracted from *rbt* and the region to which it belongs is returned to the caller. Our choice to use a Red-Black tree to hold the utilities of regions is due to our algorithm requiring efficient updates to the values for regions' utilities and efficiently finding the minimum of those. At both these operations the Red-Black tree is most efficient: both updating an element and extracting the minimum is logarithmic to the size of the tree. Since there are $2 \cdot n$ regions in the system, both updatePriority and pickEvictionRegion are $O(log(2n))$. Note that all operations of fetchPage and touchPage are of constant complexity, apart from updatePriority, pickEvictionRegion and the replacement policy. For virtually all modern policies, the complexity is constant-time as well; thus, the complexity for RUC is $O(log(2n))$ for n devices. If replacement policies with complexity greater than $O(log(2n))$ are used for some regions, then the complexity of RUC is equal to those.

4.5.3 Hit-based Utility Criterion (HUC)

The design goal for HUC is to drop the assumption that RUC makes about the hit ratio function across regions; rather, for HUC, the hit ratio function for each region is arbitrary. What is more, HUC takes into account the hit ratio when deciding the utility of a region. As discussed in Section 4.5.1, however, just using $C'_j = H(Y_j, s_j, w_j) \cdot |w_j| \cdot e_j$ as a measure of utility leads to a system that adapts very poorly to changing workloads. In order to make an informed decision before resizing a region, one needs to know how many more or less hits the region is expected to yield if it grows or shrinks, respectively.

```
Algorithm RUC

rbt : a red-black tree on regions' utilities.
Procedure updatePriority (Region $j$)
1. $U_{RUC}(j) := \frac{r_j \cdot e_j}{s_j}$
2. Update entry for $j$ in rbt with $U_{RUC}(j)$

Procedure pickEvictionRegion ()
1. $j$ := get element of rbt with minimum $U_{RUC}$
2. return $j$
```

Figure 4.9: The RUC algorithm

Since we make no assumption on H, apart from that it grows with s, we cannot analytically compute how the hit ratio will change beforehand. As an alternative, to find out what the hit ratio is expected to be after resizing a region, HUC uses *ghost caching* (also known as *shadow caching*), *i.e.*, the cache directory for the region HUC holds a superset of the pages that are actually buffered. In particular, except for the directory of the buffered pages, we maintain another two directories that hold ghost pages, with each simulating the state at which the region would be if it had a different size. For ghost pages we only maintain their metadata in cache directory – no I/O is involved with ghost pages. The one of these two directories simulates a cache that is d page frames smaller than the current size of the region and is therefore termed the *small ghost directory* (or simply SGD). The other one simulates a cache that is d page frames larger than the current size of the region and is termed *large ghost directory* (or simply LGD). For both ghost directories we use the *same replacement policy* as for the directory of the region itself and both these ghost directories see the same workload as the region itself. Therefore, the number of hits occurring for each one of the two ghost directories is equal to the number of hits that would occur for the region, if its size was the same as the size of either of the ghost caches. The directories maintained for a region j by HUC are visualised in Figure 4.10.

Apart from the number of hits h_j seen by region j, HUC also keeps track of the number of hits occurring in each one of the two ghost caches, h_j^{SGD} and h_j^{LGD}, respectively. The key idea in HUC is that if the region is resized from s_j to $s_j - d$ (the size of the SGD), then the number of hits will drop from h_j to h_j^{SGD}. Similarly, if region j grows from s_j to $s_j + d$ (the size of the LGD), the number of hits seen will increase

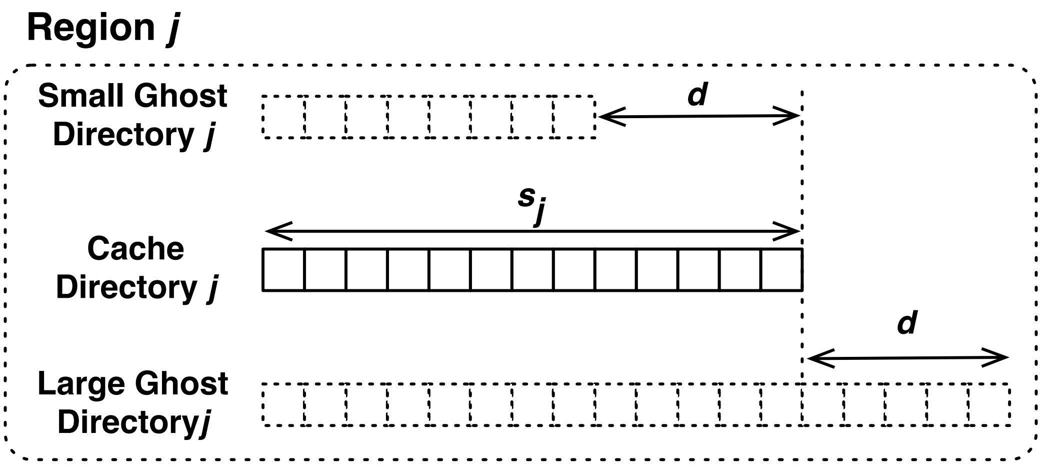


Figure 4.10: Bookkeeping for a region under HUC

from h_j to h_j^{LGD}. We define Δh_s to be the number of hits that the region will miss by shrinking from s_j to $s_j - d$, that is $\Delta h_s = h_j - h_j^{SGD}$ and Δh_g the number of hits the region will gain by growing from s_j to $s_j + d$, that is $\Delta h_g = h_j^{SGD} - h_j$.

For each region in the system, HUC tracks the projected decrease in the utility of the region if it shrinks by d page frames, referred to as *shrink utility* (ΔU^s_{HUC}) and the projected increase in the utility of the region if it grows by d page frames, termed *grow utility* (ΔU^g_{HUC}). By our definition, the utility for a region is proportional to the number of hits it sees ($H(Y_j, s_j, w_j) \cdot |w_j|$) and to the eviction cost for the region (e_j). Following that definition, we have that $\Delta U^s_{HUC} = \Delta h_s \cdot e_j$ and $\Delta U^g_{HUC} = \Delta h_g \cdot e_j$. In other words, the shrink utility ΔU^s_{HUC} represents the cost units that the region will fail to save if its size shrinks by d, while the grow utility ΔU^g_{HUC} represents the extra cost units that will be saved by region j if it grows by d. The intuition behind HUC is that the regions with low shrink utility should give some of their page frames to regions with high grow utility; the former will only fail to save a few cost units, while the latter will be able to save many more and therefore the overall utility of the system's buffer pool will increase.

Before giving the algorithmic details for HUC we should note that the size of the SGD is always kept equal to $s_j - d$ and the size of the LGD is always kept equal to $s_j + d$. This detail is omitted in the baseline algorithm for simplicity of presentation. The only modification required to account for that, however, is that whenever the size of a region is decreased or increased by one page frame, the same happens for both ghost directories of that region (this occurs in Lines 13, 14 of fetchPage and Lines 6, 7 of touchPage). Also, when a lookup is done on region j (Lines 3, 7 in fetchPage), a lookup is done in each one of the ghost directories and the hit counters for these directories (h_j^{SGD} and h_j^{LGD}, respectively) are incremented if the lookup results in a hit in the respective directory. Independently of the lookup in the region, a lookup in the ghost caches may yield a hit or miss. In the event of a miss, the referenced page identifier is brought into the ghost cache directory, following an eviction of a page if required; the victim page in that case is chosen by running the replacement policy for the ghost cache. This is required to ensure that the contents of the ghost caches are the ones that the region would have, if its size were equal to the size of the ghost caches. This is also omitted from the fetchPage algorithm to keep the presentation simple.

The updatePriority and pickEvictionRegion procedures for HUC are given in Figure 4.11. Similarly to RUC, we use a Red-Black tree to hold the shrink utilities of the regions, as we need to efficiently update the shrink utilities of regions and find their minimum value. In updatePriority, we compute the shrink utility for a region, using the hit counter for the region and the hit counter for its small ghost directory. The updated shrink utility for that region is then stored in the *rbt*. In the pickEvictionRegion procedure, we first compute the grow utility of the hit region, *i.e.*, the region to which the referenced page is to be inserted, using the hit counter for that region and the hit counter for its large ghost directory (Lines 1-3). Then, we use the *rbt* to extract the region with the minimum shrink utility among all regions (Line 4). Next, we compare the grow utility of the hit region to that minimum shrink utility. If the former is found higher than the latter, the hit region should grow and therefore the region with the minimum shrink utility is returned as the region from which the eviction will take place. Otherwise, the hit region will retain the same size, that is both the eviction and the insertion of the referenced page will take place in the hit region (Lines 5-8).

The computational complexity for HUC is the same as the one for RUC, that is $O(log(2n))$ for a buffer pool over n devices. The space overhead for HUC, however, is slightly higher as it uses 3 page directories for any region j, holding metadata for $3 \cdot s_j$ pages in total, while RUC only maintains metadata for s_j pages. One may argue that HUC is much more expensive in terms of CPU cost than RUC, as it requires 3 lookups when looking up a page in a region (one for the region itself and another one for each one of the ghost directories). Also, each time a page reference results in a miss in one of the ghost directories, the replacement policy needs to be run for that directory to pick a ghost page for eviction; this adds extra CPU overhead. However, all operations to the ghost directories may be parallelized with the operations on the regions page

```
Algorithm HUC

rbt : a red-black tree on regions' shrink utilities.
Procedure updatePriority (Region j)
1. $\Delta h_s := h_j - h_j^{SGD}$
2. $U_{RUC}^s(j) := \Delta h_s \cdot e_j$
3. Update entry for j in rbt with $U_{RUC}^s(j)$

Procedure pickEvictionRegion ()
1. $j' :=$ the hit region
2. $\Delta h_g := h_{j'}^{LGD} - h_{j'}$
3. $U_{RUC}^g(j') := \Delta h_g \cdot e_j$
4. $j :=$ get element of rbt with minimum $U_{RUC}^s(j)$
5. if $U_{RUC}^g(j') > U_{RUC}^s(j)$
6.    return j
7. else
8.    return j'
```

Figure 4.11: The HUC algorithm

directory, as they operate on different data. What is more, ghost directory updates may also be overlapped with the I/O operations issued by the region, thereby having negligible effect in the CPU cost of the algorithm.

4.5.4 Hit Ratio Curve Aware Resizing (HRCA)

In this section we explore how one can compute the hit ratio curve of a replacement policy for a specific workload and how this curve can be used to reach more informed resizing decisions. Therefore we assume that the replacement policy is a *stack replacement policy*, as for this class of replacement policies the hit ratio curve can be tracked with one pass, for various cache sizes [Mattson et al., 1970]. A replacement algorithm is a stack algorithm if it exhibits the *inclusion property*. The latter states that for any workload the contents of a cache of size s are a subset of the contents of a cache with size $s' > s$ [Mattson et al., 1970]. As discussed in [Mattson et al., 1970], any replacement algorithm that induces a total ordering on all previously referenced pages and uses this ordering to make replacement decisions is a stack algorithm. For instance, LRU orders accessed pages by their time of last access and always selects the one least recently accessed one for replacement – therefore it is a stack algorithm. In a similar fashion the Least Frequently Used (LFU) algorithm orders accessed pages by their frequency of reference and always evicts the least frequently accessed one. For stack replacement algorithms, given a cache memory of size S, one can efficiently compute the hit ratio for any cache memory of size less than S for a specific workload, using Mattson's algorithm [Mattson et al., 1970]. In the following we assume LRU as the replacement policy, for reasons of simplicity. On the other hand, most modern replacement algorithms (such as ARC [Megiddo and Modha, 2003], CAR [Bansal and Modha, 2004]) use multiple page queues and do not define a total ordering on accessed pages, therefore they cannot be classified as stack algorithms. For such cases, more generic algorithms such as RUC and HUC can be used.

4.5.4.1 Mattson's Algorithm

This algorithm keeps track of the position in the LRU stack at which each hit occurs, *i.e.*, it records at which stack *distance* the hit page was, before it is brought to the front of the LRU queue (as the hit page is now the most recently accessed page). The algorithm maintains an array of S counters, which we refer to as the *distances* array or $D[]$. Upon a hit to a page at distance i (from the front of the LRU queue), the counter $d[i]$ is incremented by one. Notice that this hit would never have occurred if the size of the cache was $i-1$, or less. Essentially, $D[i]$ represents the number of hits that would occur in a cache of size i, but *not* in a cache of size less than i. We refer to $D[]$ as the *hit-distance* distribution. Therefore, to compute the total number of hits a cache of size $k < S$ would see for a specific workload W, one only needs to run the workload for a cache of size S and then compute $h(i) = \sum_{k=1}^{k=i} D[k]$, for any i of interest. We refer to $h(i)$ as the *hit-size* distribution. Of course, the hit ratio for a cache of size k may be computed as follows:

$$H(LRU,k,W) = \frac{h(i)}{|W|} = \frac{\sum_{k=1}^{k=i} D[k]}{|W|}$$

In implementing the above algorithm for large caches, there are two main challenges that have to be dealt with. First, the size of the distances array is equal to the number of pages that fit in the cache and therefore for large caches it occupies (and thus, wastes) too much memory space. Second, upon each and every hit in the cache the algorithm needs to find out at what distance the page was in the LRU queue. The naive way to do that is to linearly search the queue until the referenced page is found; however, a linear search on every hit is unacceptable from a performance perspective. To deal with the first problem, we tracked hit distances at a coarser granularity than that of a page. Particularly, we divided the cache into c chunks and assumed that all pages that belong to a chunk have approximately the same distance from the front of the queue. Therefore, the distances array only needs to keep track of c counters. For instance, for a cache of 1000 pages we would divide it into 50 chunks of 20 pages each. Then, for a hit at depth 83 we would increment the counter at $83/20$, that is $D[5]$, and in this way $D[i]$ would represent all hits occurred in depth greater than $20 \cdot (i-1)$ and less than $20 \cdot i$. Giving up enough accuracy in the recorded distances, one can reduce the size of $D[]$ as much as one wishes, and vice versa.

4.5.4.2 Measuring Hit Distances

To deal with the second challenge mentioned above, that is, to efficiently measure the distance of a hit page in the LRU stack, we made the following observation: for LRU, the distance of a page in the queue is, by definition, proportional to the recency of its last access. Thus, one can use the recency of a page to estimate its distance in the LRU stack. To that end, for each page in the cache, we store the timestamp of its last access (for practically all real systems this timestamp is kept track of anyway, so this scheme does not impose any additional overhead). When page i (with timestamp of last access t_i) is hit, we estimate its stack distance (d_i) using the timestamp of the previously most recently accessed page (t_{max}) as follows (the snapshot is shown in Figure 4.12(a)):

$$\overline{d_i} = t_{max} - t_i + 1$$

The above is, however, not very accurate for most practical cases. The reason is that each hit in the cache results in the hit page being moved to the front of the queue and assigned a new timestamp. Thus, after a hit $t_{max} - t_{min} + 1$ is greater by one than the actual size of the cache (assuming that t_{min} is the timestamp of the LRU page), since t_{max} has increased, t_{min} has remained the same and no page has been inserted to or evicted from the cache). Depending on the distance of the hit page, the quantity $t_{max} - t_i + 1$ used to estimate d_i may also be erroneous after a hit. An example is shown in Figure 4.12(b), in which no hit has yet occurred and therefore timestamps are continuous as one traverses the queue. If page with timestamp 4 is hit, however, then the state of the stack is the one shown in Figure 4.12(c). In that state, if one wants to find the distance of page i_3 with timestamp 3, both $t_{max} - t_{i_3} + 1$ and $t_{max} - t_{min} + 1$ are off by 1 *w.r.t.* the actual distances of page i_3 and the LRU page, respectively. Had the hit occurred for the page with timestamp 2 instead, the state of the stack would be the one shown in Figure 4.12(d). Computing the distance of the page with timestamp 3 in that

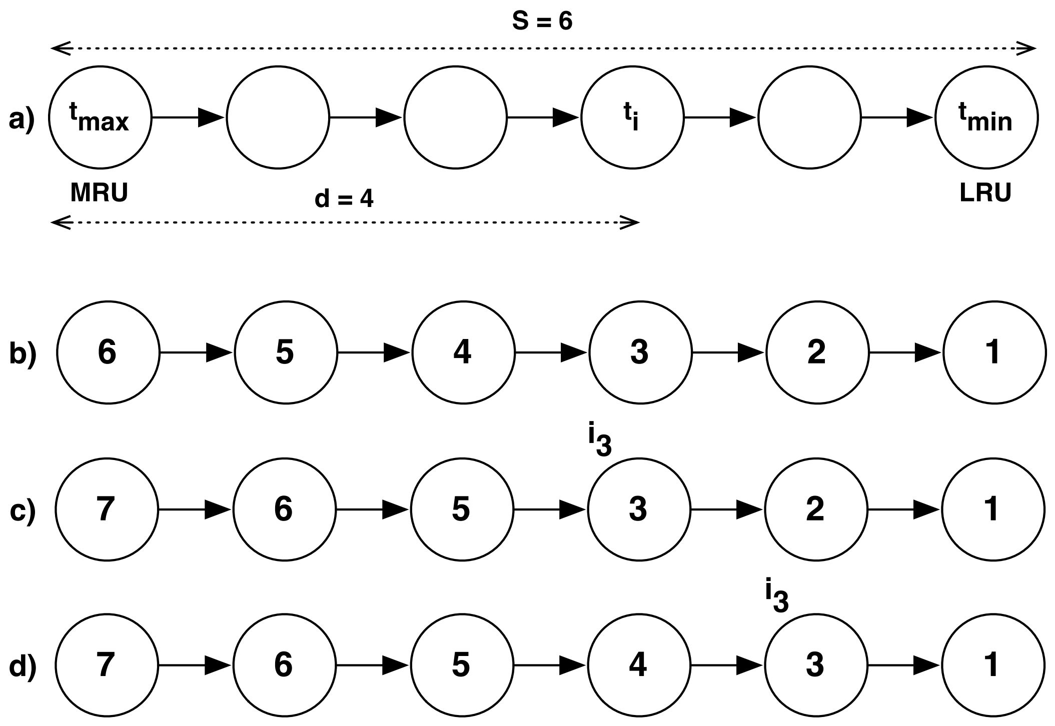


Figure 4.12: Estimating the distance of the hit page

case, would yield the right value for the term $t_{max} - t_{i_3} + 1$ (the value of $t_{max} - t_{min} + 1$ would still be greater than the distance of the MRU and LRU pages).

Since each hit moves the hit page to the front of the stack, each hit introduces a *discontinuity* in the timestamps as one traverses the queue. This discontinuity introduces an error of one distance unit in the quantity $\overline{d_i} = t_{max} - t_i + 1$, for all pages i that are deeper in the stack than the hit page. In other words, the error in estimating d_i by $\overline{d_i}$ is equal to the number of hits that have previously occurred for pages with distance $d' < d_i$. We refer to this error as Δd_i; it follows that $d_i = \overline{d_i} - \Delta d_i$. In the following we assume that page hits are uniformly distributed in the distances space, *i.e.*, a hit may occur at any distance with the same probability. Therefore, the number of hits expected to occur in distances less than d_i, denoted as $h_<(i)$, is proportional to $\frac{d_i}{S}$, that is, $h_<(i) = \rho\frac{d_i}{S}$. Each such hit at distance $j < d_i$ introduces one discontinuity. Thus, it contributes one error unit to $\overline{d_{j'}}$, for all pages $j' > j$. Then, in order to compute the number of error units in $\overline{d_i}$ one needs to sum these discontinuities up to d_i. Hence:

$$\Delta d_i = \sum_{k=1}^{d_i} \rho \frac{k}{S} = \frac{\rho}{S} \cdot \frac{d_i(d_i+1)}{2}$$

To find ρ, we leverage the least recently used page, which we know is at distance

$d_{min} = S$. Thus:

$$\begin{aligned}\Delta d_S &= \overline{d_S} - d_S \Rightarrow \frac{\rho}{S} \cdot \frac{S(S+1)}{2} = t_{max} - t_{min} + 1 - S \\ \Rightarrow \rho &= 2 \cdot \frac{t_{max} - t_{min} + 1 - S}{S+1}\end{aligned}$$

Knowing ρ, we can compute d_i for any i by solving the following equation:

$$d_i = \overline{d_i} - \Delta d_i \Rightarrow d_i = t_{max} - t_i + 1 - \frac{\rho}{S} \cdot \frac{d_i(d_i+1)}{2}$$

The method presented above has constant time complexity and as our experimental evaluation (presented in Section 4.7.1) shows, it provides very accurate results. Even under the assumption of uniformly distributed hit distances, our method provided very accurate results for both randomly generated and real-world workloads. Note that if hit distances are priorly known not to follow a uniform distribution, but their distribution is known, it can be used to derive the formula for Δd_i as we do above for the case of a uniform distribution.

4.5.4.3 The HRCA algorithm

In order to decide the optimal size for each region, we need to know the number of hits each region is expected to achieve for all different region sizes. For the reasons stated in Section 4.5.4.1, the total memory of S pages is divided in c chunks; each region may be allocated any number of these chunks. Considering that the optimal size for each region may vary from 0 to c chunks, we need to know the number of hits any region would experience for any number of chunks from 0 to c. Therefore, each region is equipped with a ghost directory of size S, that uses the same stack replacement algorithm as the region itself. Similarly to HUC, the ghost directory sees the same operations as the cache itself, the only difference being that when a hit occurs in the ghost directory, its hit distance is measured using the technique of Section 4.5.4.2. Using Mattson's algorithm, we compute for each region i the array $D_i(s_i)$, which gives the number of extra hits that would have occurred for region i if its size was increased from $s_i - 1$ to s_i chunks, for all $0 < s_i \leq S$.

Let $h_i(s_i)$ denote the hit-size distribution for region i, *i.e.*, the total number of hits that would occur in region i if its size was s_i. Then D_i is the derivative of h_i. By our discussion about region utilities of Section 4.4, it follows that the utility of allocating s chunks to region i can be computed as follows:

$$C_i'(s_i) = h_i(s_i) \cdot e_i$$

```
Algorithm setTargetSizes

1.  for all $0 \leq i \leq 2n-1$
2.      $ts[i] := 0$
3.  $max := 0$
4.      $chunksLeft := S$
5.  while ($chunksLeft > 0$)
6.      for all $0 \leq i \leq 2n-1$
7.          if $D_i[ts[i]+1] > D_{max}[ts[max]+1]$
8.              $max := i$
9.              $ts[max] := ts[max]+1$
10.             $chunksLeft := chunksLeft - 1$
```

Figure 4.13: Computing the optimal region sizes

Our goal is then to maximise the quantity $\sum_{i=0}^{i=2n-1} C_i'(s_i)$ under the constraint that $\sum_{i=0}^{i=2n-1} s_i = S$. In this respect, our problem is a resource allocation problem that has been shown to be NP-hard [Rajkumar et al., 1997]. Similarly to [Zhou et al., 2004], we use a greedy algorithm that allocates each memory chunk to the region that is going to see the greatest increase in its hit ratio from that chunk. The input to the algorithm is the $2n-1$ $h_i(s_i)$ arrays (one for each region) and the total available memory S. The output is an vector $ts[i]$ that holds the *target* size of region i. The target size is of course, the number of chunks that the greedy algorithm has decided is the “optimal” for that region. The algorithm is given in Figure 4.13. The time is divided in fixed-length *epochs* and at the end of each epoch new target sizes for the regions are computed using setTargetSizes.

The workflow of HRCA is very similar to the baseline algorithm. When a miss occurs for a region its current size and its target size are considered. If its target size is less than its current size, then a victim page is selected from that region for eviction and the referenced page is read in from the device; the size of the region remains the same in this case. If the current size of the region is less than its target size, then the region should grow. A page frame of another region is vacated and allocated to the hit region. When deciding which one of the other regions should shrink (*i.e.*, which one should evict a page), the only candidate regions are the ones for which the target size is less than their current size (there will always exist at least one such region, since the hit one has a target size greater than its current one). For these candidate regions we take into account: (a) how “far” the candidate region i is from reaching its target size,

```
Algorithm HRCA

rbt : a red-black tree on regions' eviction priority.
Procedure updatePriority (Region j)
1. if $s_j < ts[j]$
2.     Remove entry for j from the rbt
3. else
1.     $V_j := \frac{s_j - ts[j]}{e_j}$
2.     Update entry for $j$ in rbt with $V_j$

Procedure pickEvictionRegion ()
1. $j' :=$ the hit region
2. if $ts[j'] < s_{j'}$
3.     return $j'$
4. else
5.     $j :=$ get element of rbt with maximum $V_j$
6.     return $j$
```

Figure 4.14: The HRCA algorithm

that is, the difference $s_i - ts[i]$ and (b) the eviction cost of the region. We compute the *eviction priority* V_i for the candidate region as:

$$V_i = \frac{s_i - ts[i]}{e_i}$$

and choose the one with the highest eviction priority. The intuition behind this is that the greater $s_i - ts[i]$ is, the less the utility of the region has been found by the setTargetSizes algorithm and thus the more urgent it is to bring the region down to its target size. Also, for two candidate regions j, j' with $s_j i - ts[j] = s_{j'} - ts[j']$, the one with the least eviction cost has the highest priority, as a miss on that region will result in less I/O cost paid by the system. Similarly to RUC and HUC, the eviction priorities are stored in a red-black tree that allows very efficient retrieval of the maximum value. Then, in pickEvictionRegion the one with the highest eviction priority is returned. The pseudocode for updatePriority and pickEvictionRegion is given in Figure 4.14.

The computational complexity of HRCA is the same as the one for RUC and HUC on each page access, that is, $O(log(2n))$ for a buffer pool of n devices. The only difference is that at the end of each epoch the setTargetSizes algorithm is run to compute the new target sizes. Its complexity is $O(c)$, *i.e.*, it is linear to the number of chunks used. For all practical epoch lengths and chunk numbers we used, the computational overhead was found to be negligible. In terms of memory overhead, HRCA uses one page directory of size s_i for the actual cache and a directory of size S that holds the metadata for the ghost pages.

4.6 Discussion

HRCA issues. As noted in [Zhou et al., 2004] using a greedy algorithm like setTargetSizes to compute the solution in a resource allocation problem yields the optimal solution when the utility functions are convex. In our case this is equivalent to the hit ratio curve (or, the hit-size distribution) being convex. Indeed, this is usually the case for hit ratio curves of real workloads. However, in some cases, hit ratio curves are not convex in specific intervals. If this is the case for very small hit distances, then the given algorithm may result in a very inefficient allocation. We stumbled upon such a case when running a TPC-C workload using two devices in read-only mode; we only had two regions, one clean region per device (we set eviction costs equal to one for both devices). In Figure 4.15 we show the hit-size distribution (on the bottom graph) and its derivative, the hit-distance distribution (on the top graph), for a specific epoch during the execution (simulation) of the workload (the raw data are given in Figure 8.11 and Figure 8.10, respectively). It is clear that in this case, all memory chunks should be allocated to Region 1, as it utilises them far better that Region 2. However, observe that $D_1[1] < D_2[i]$ for all i. Therefore, the greedy algorithm will never allocate a chunk to Region 1 and all chunks will be allocated to Region 2. Although this case seems quite skewed, it did occur for real workloads. The allocation decision of the greedy algorithm in this case is the worst possible.

To alleviate the situation we adopted the following approach: instead of only taking into account the value of the derivative function at the next-chunk-to-be-allocated, we also compute the following quantity:

$$D'_i = \sum_{j=ts[i]}^{j=ts[i]+chunksLeft} D[j]$$

which represents the number of hits the region will see if all remaining chunks are allocated to it. This computation is performed in the loop starting at Line 6 of setTargetSizes. We compute both D'_i and D'_{max} and compare $f = \frac{D_i}{D_{max}}$ to $f' = \frac{D'_{max}}{D'_i}$. If $f' > f$, it means that f is probably skewed and therefore Lines 8-10 are not executed, leaving max with its current value. Using this conservative technique, our algorithm reached optimal allocation decisions for such cases.

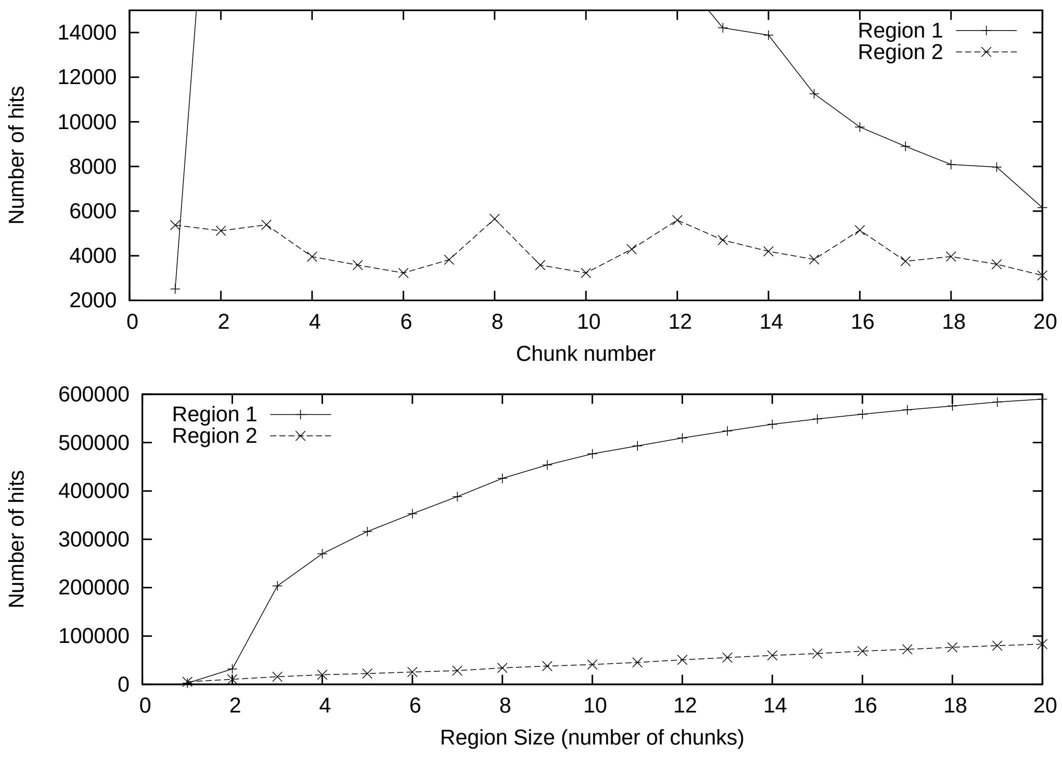


Figure 4.15: Hit distances for an epoch of the TPC-C workload

Time considerations. RUC keeps track of the number of references for each device, while both HUC and HRCA keep track of the number of hits occurred for the ghost cache(s) of each region. For our algorithms to be adaptive to changing workloads, they need to make sure that the most recent behaviour outweighs the older behaviour. For instance, a region may become very hot at some point in time and experience a large number of hits. All three algorithms will increase the region size, as a result. However if the region subsequently becomes cold, or if the access pattern changes to an inefficient one, then its size should shrink to favour other regions. Therefore, the recorded statistics need to "age" with time – otherwise the algorithms have no means of telling that the workload has changed. To that end, we follow the approach of [Thiébaut et al., 1992] and use time windows of length τ. For HRCA an epoch is the time window we use. For RUC and HUC a window of similar length is adopted. At the end of each window, that is, every τ page references, the statistics for each region (reference counts for RUC, hit counts for HUC, HRCA) are divided by a constant Γ_τ. Consequently, at any point in time the statistics of the current window contribute to resizing decisions with a weight of 1, the ones of the previous window with weight $\frac{1}{\Gamma_\tau}$, the ones of the window before the previous one with weight $\frac{1}{\Gamma_\tau^2}$, *etc*. The length of the window is kept fixed, as we found this approach to work very well in our experiments. In [Fagin, 1977], the author studies what the window size should be to achieve a specific hit ratio; however, specific assumptions are made about the reference pattern and therefore we do not explore this direction further.

Comparison. All three algorithms take into account the read and write costs of the storage devices in addition to the system workload characteristics to decide the optimal number of pages to cache from each device. RUC tends to cache more pages from page classes that see a lot of references, *i.e.*, are hot, and have a high eviction cost. On the other hand, HUC and HRCA cache more pages from the regions that exhibit a high hit ratio, in addition to a high eviction cost. Essentially, HUC and HRCA take into account the effectiveness of the replacement policy for the given workload and cache size, while RUC assumes that the hottest regions yield the highest hit ratios. Therefore, we expect the performance of the three algorithms to be roughly the same when this assumption holds. However, in all other cases HUC and HRCA are expected to perform better, as they make informed resizing decisions. In particular, we expect HRCA to be more efficient than HUC, as it knows the hit-size distribution for all possible sizes of each region and can therefore approach the globally optimal region sizes. On the other hand, HUC only knows the local values of the hit-ratio distribution, given by the small- and large ghost directories. The bad news for HRCA is that it can only work for systems that use stack replacement algorithms, like LRU. In all other cases, generic algorithms like RUC and HUC are the only option.

I/O costs. In this work we assume that on a page miss the buffer manager fetches a single page from the disk, while at the time of a dirty eviction it writes a single page to the disk. Thus, the I/O pattern of interest is randomly reading a page and randomly writing a page on any one of the storage devices; the cost of randomly reading a page or randomly writing a page is the cost we use to calculate the eviction cost for any page. Of course, for various storage media, these costs may vary across time as, for instance, the cost of random writes on flash disks [Bouganim et al., 2009]. In this work, however, we are interested in the average read and write costs of devices – no accuracy is required for specific accesses. Since our goal is to increase the throughput of the system, the read and write costs of the storage devices represent the number of random read and write I/O operations, respectively, these devices can serve (IOPS). The IOPS capability of a disk serving random I/O requests primarily depends on the address space that these requests span and the I/O queue depth on the device, *i.e.*, the number of outstanding I/O requests that have been queued at any given time (using a given elevator algorithm). Typically, the smaller the address space is, the better the throughput is, as the on-disk caches can be more effective. The same holds for the I/O queue depth: the longer it is, the more opportunities arise for the controller to merge requests, thereby reducing, for instance, the number of arm movements for magnetic disks and the number of flash block erase operations on flash disks.

In our experiments we have used four different disks, two flash disks and two magnetic ones. For these disks we have experimentally measured their IOPS capability for various sizes of address space that the requests span and various queue depths. We varied the size of the file being randomly accessed (*i.e.*, the requests address space) from 128MB to 64GB for the magnetic disks (for sizes greater than that I/O throughput of devices remains the same) and up to 28GB for the flash disks (both were 32GB disks). We used `fio` [Axboe, 2009] to measure the number of IOPS for each configuration and have plotted the results as 3-d surfaces. In all cases, the reported numbers are for 4KB pages. In Figure 4.16 and Figure 4.17 we have plotted the random read and random write IOPS, respectively, for the Seagate ST3808110AS magnetic disk. The corresponding plots for the Maxtor 6L300R0 magnetic disk are shown in Figure 4.18 and Figure 4.19. As expected, for both disks I/O throughput increases as the requests are restricted to a small portion of the device address space. Both magnetic disks exhibited better write performance than read performance – our guess is that the on-disk cache was used more aggressively for writes, buffering successive requests before they were committed to the medium. The Maxtor disk was equipped with a buffer cache of 16MB whereas the Seagate disk had an 8MB buffer; this is why the random write throughput for small address spaces is much better for the Maxtor disk. The I/O throughput increased slightly with the length of the I/O queue for both magnetic disks; substantial increases were observed only when the address space was kept very small (less than 4GB) and primarily for reads.

We took similar measurements for the two flash disks used with our system. The first one was a high performance Intel X25-E SLC flash disk. The plots for its random read and write I/O throughput are shown in Figure 4.20 and Figure 4.21, respectively. The second was a low cost MLC flash disk, the Samsung MCAQE32G5APP; the plots for its random read and write throughput are shown in Figure 4.22 and Figure 4.23 respectively. The first observation is that the Intel disk is orders of magnitude more I/O efficient for all address space sizes and queue depths; the Intel disk is an enterprise-level SLC disk selling for about $20 per GB, while the Samsung flash disk is an inexpensive MLC device. For both disks, the read throughput is orders of magnitude greater than

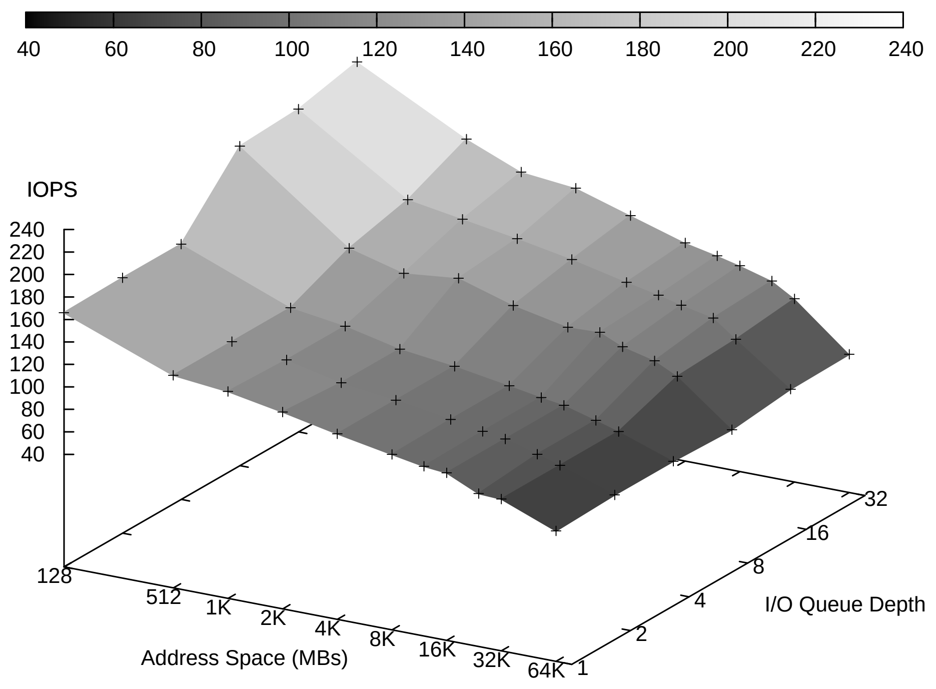


Figure 4.16: Read IOPS for Seagate ST3808110AS

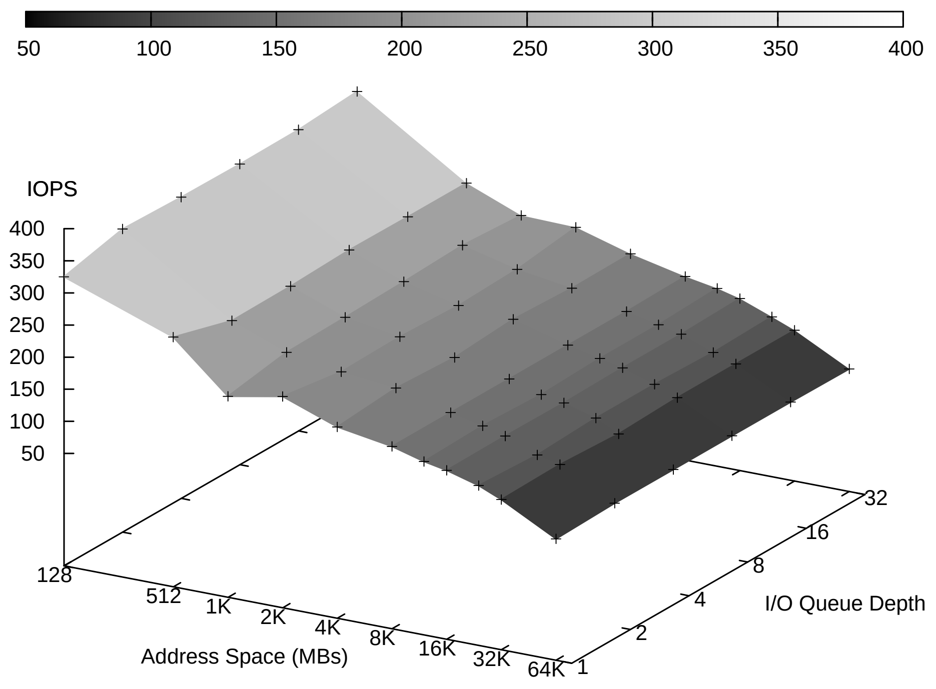


Figure 4.17: Write IOPS for Seagate ST3808110AS

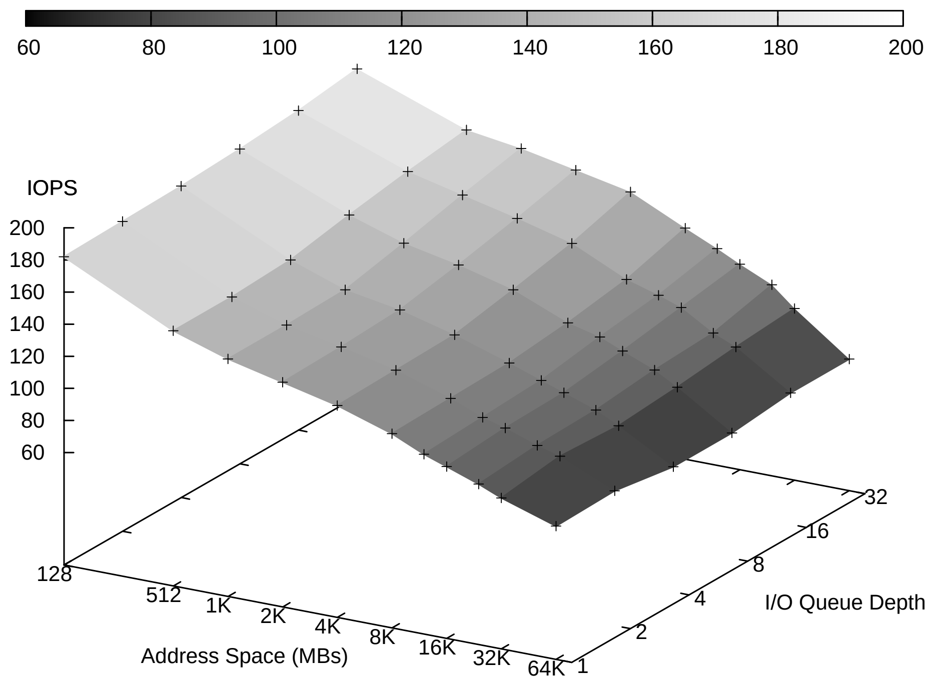


Figure 4.18: Read IOPS for Maxtor 6L300R0

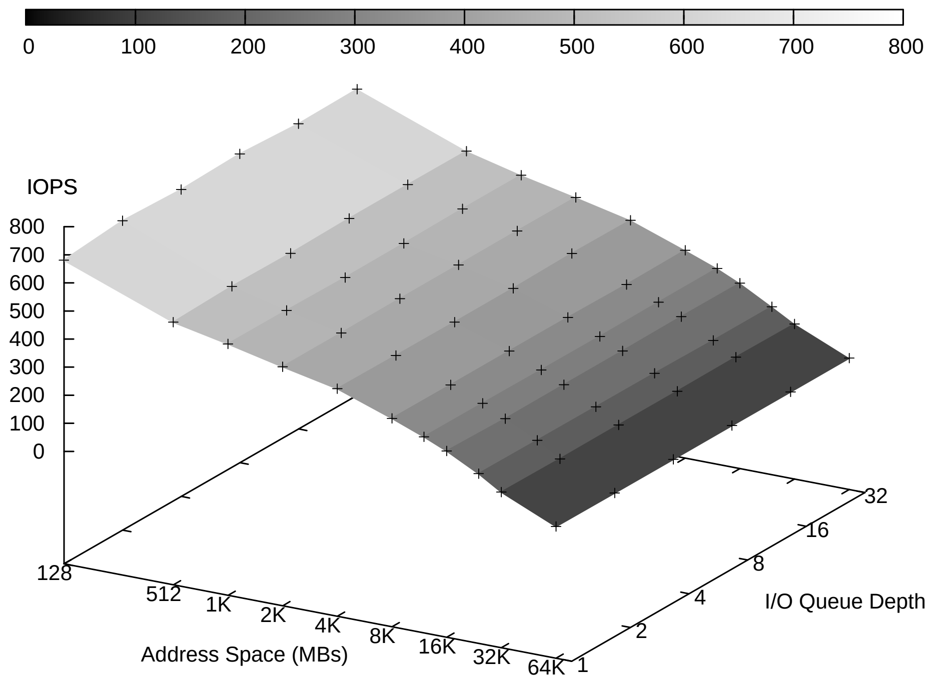


Figure 4.19: Write IOPS for Maxtor 6L300R0

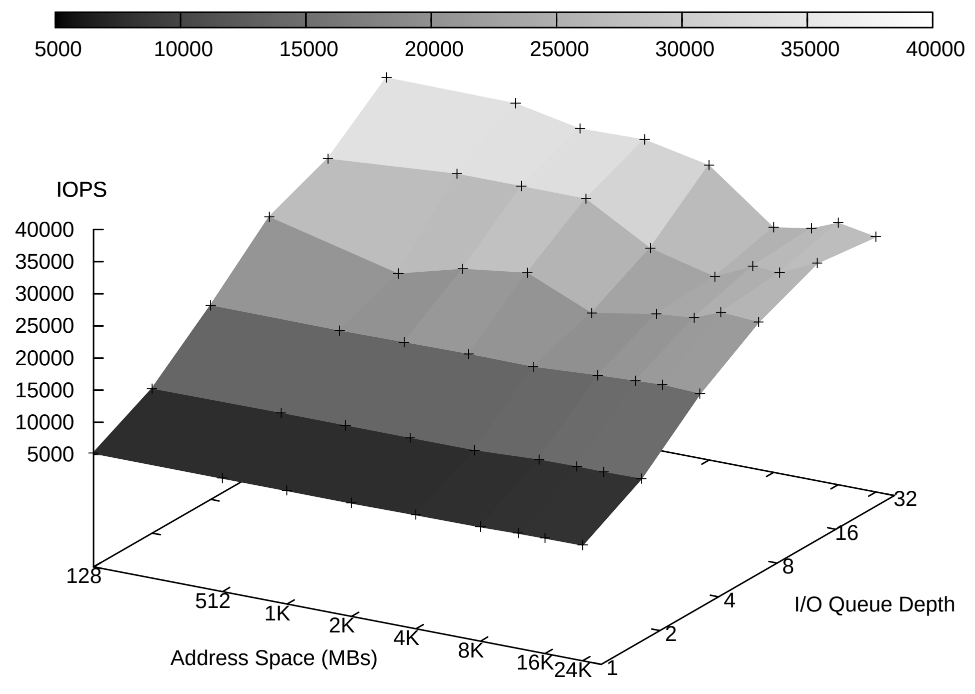


Figure 4.20: Read IOPS for Intel X25-E

their write throughput, more than 100 times greater for the Samsung disk and up to 10 times greater for the Intel disk. Of course, this is due to the erase-before-write limitation of the flash chips. We found the random write throughput of both devices to be substantially higher for very small address spaces; this is consistent with the findings of [Bouganim et al., 2009]. For the Intel disk, random read efficiency did not depend on the size of the address space very heavily, an indication that the controller of the disk uses the underlying flash dice in a massively parallel fashion – which also explains its high performance overall. For the Samsung disk this was not the case and throughput dropped significantly for large address spaces, as the on-disk buffers were no good for such sizes. Increasing the queue depth had absolutely no effect on write throughput for the Samsung disk and a rather light effect on the read throughput. On the other hand, the length of the queue depth for the Intel disk had a dramatic effect on its read throughput, which also implies a high degree of parallelism in reading data from the flash chips. The effect of queue depth on its write throughput was substantial only for a very small address space – for larger ones erase times dominated.

Our system uses plots such as the ones given above to calculate the costs for each storage device. In any case we are not interested in the latency of specific requests, but

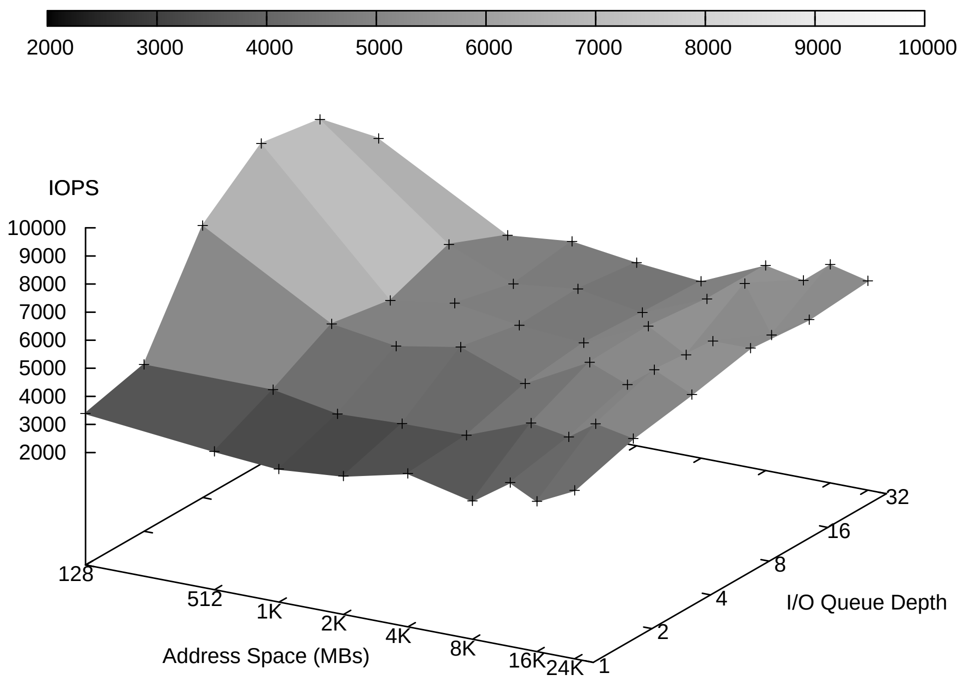


Figure 4.21: Write IOPS for Intel X25-E

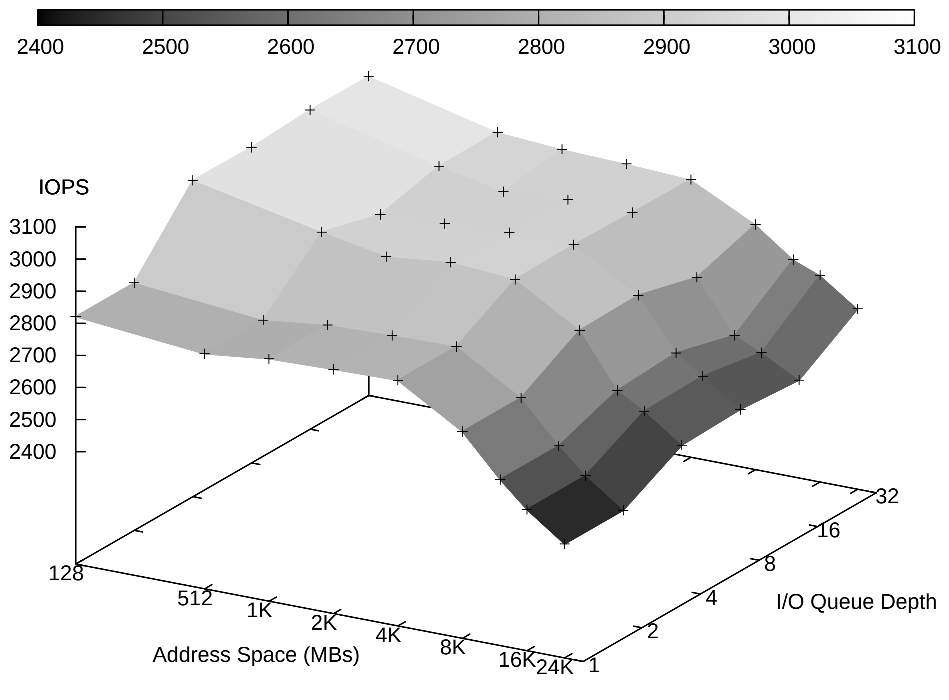


Figure 4.22: Read IOPS for Samsung MCAQE32G5APP

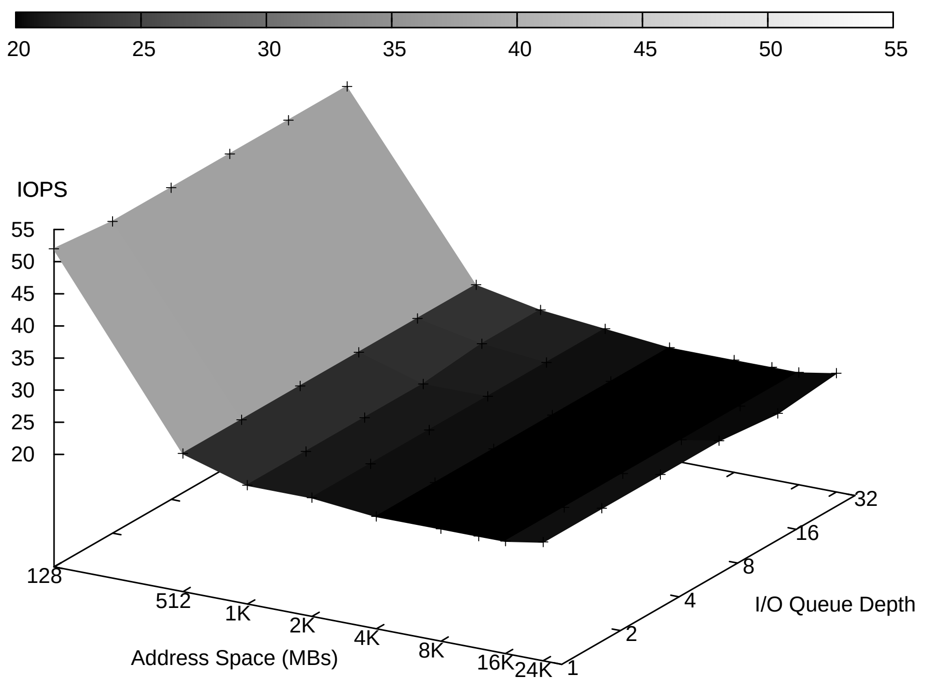


Figure 4.23: Write IOPS for Samsung MCAQE32G5APP

in the average throughput the system is expected to have for a specific address space size and a specific queue depth. If more advanced I/O techniques are used, such as clustering or prefetching, then the average cost per page served in each request should be the one given to our system. The throughput of the devices may also be sampled at real time. Large deviations from the predicted values could possibly imply that the device has failed or is about to fail and therefore its cost should be re-adjusted. For instance, during RAID reconstruction in a disk array, read and write costs become greater; if that is detected by the system, it will cache more pages from the faulty device, thereby reducing the effect of the failure on the throughput of the system.

4.7 Experiments

Setup. We implemented our algorithms to evaluate their performance under various workloads, both using real storage devices and simulated ones. We used a quad-core Intel Xeon E5420 box (“System A”) clocking at 2.5GHz with 4GB of physical memory, equipped with a magnetic and a flash disk dedicated for the data of our system. The magnetic disk was an 80GB Seagate Baracuda 7200 with 8MB of cache, while the flash one was a 32GB Intel X25-E SLC disk. Both disks were connected using the SATA II interface. We also used an Intel Pentium 4 box ("System B") clocking at 2.26GHz with 1.5GB of physical memory, also equipped with a flash and a magnetic disk dedicated solely to our experiments. The magnetic disk was a 300GB Maxtor DiamondMax 6L300R0 with 16MB of cache memory. The flash disk was a Samsung MCAQE32G5APP, an MLCNAND flash disk with a capacity of 32GB. Both disks were connected to the system using the IDE interface. The Operating System was Debian GNU/Linux with the 2.6.26 kernel. The system was implemented in C++ and compiled using the GNU GCC compiler. As shown in the graphs of Section 4.6 and for the address space size of the data we used, for System A the flash disk was about 50 times more efficient at reading than the magnetic one and about 20 times at writing. For System B, the flash disk was 23 times faster than the magnetic one when reading, while the magnetic disk was about 10 times faster when writing. Details for the performance one the disks we used can be found in Section 4.6.

For the non-simulated experiments, we used the magnetic disk and the flash disk as the storage devices in the two systems. To eliminate OS caching effects we used both storage media as raw devices: the OS did not cache data pages, pages were never double buffered and our system had absolute control of physical I/O. Read and write costs were estimated as described in Section 4.6. The operating system and our system itself ran from a third disk, which was not used as a storage device. In the experiments, we compare the performance of the three proposed algorithms, RUC, HUC and HRCA, to the performance of a buffer pool that does not distinguish between pages of different devices, *i.e.*, a single region employing a global replacement strategy is used for the whole buffer pool (this case is referred to as *global buffer pool* or GBP).

Workloads. We used two different kinds of workloads. The first category, referred to as IRP, includes synthetic workloads, in which all pages that belong to the same device have the same probability of reference (they follow an Independent Reference Pattern). In some experiments all devices have the same probability of reference, that is, they see the same heat, while in other experiments some devices are hotter than others. If we want to increase only the heat of a region and not its hit ratio, we also increase its address space size by the same portion. When we want to increase the hit ratio for a specific device, but not its heat, we shrink its address space (while maintaining the same number of references to the device). We varied the probability of a page being read or written to and created workloads with varying dirtiness ratios. Details for each workload are given in the corresponding subsection. For the second workload, referred to as TPC-C, we ran the TPC-C benchmark on the PostgreSQL database and collected a trace of all page references, which we then translated to disk offsets. The results of this section are the execution of these traces by our system after using different criteria to place data pages on the storage devices. Details are given in the corresponding subsections.

4.7.1 Stack Distance Measurement

Initially, we wanted to evaluate the stack distance measurement technique presented in Section 4.5.4.2, which uses page timestamps to estimate the distance at which a hit page was found. We refer to this technique as TS in the following. Our goal is to experimentally evaluate the accuracy of this technique and its overhead and compare it with existing approaches. We refer to the naive algorithm, that linearly searches the LRU queue to find the distance of the hit page, as NAIVE; of course, NAIVE is perfectly accurate. We also compare our algorithm to the one proposed by the authors of [Zhou et al., 2004] (see Section 4.3.1). We refer to the latter as GROUPS. We implemented GROUPS as described in the original paper [Zhou et al., 2004]. The LRU queue is divided in page groups and two pointer arrays track the start and end of each page group, while each page has a pointer that points to the group it belongs. Pages in the same group are assumed to have the same stack distance. The time is divided in epochs and a list of pages (the *scan list*) keeps track of the pages that have been hit during the current epoch. At the end of an epoch, the scan list is processed and for each hit page the hit counter of the group it belongs to is incremented. The group pointer of the page is set to point to the first group (if it doesn't already do). For all page groups between the first one and the group to which the page previously belonged, their last elements are pushed to the next page group. Also, for each page miss, the last element of each page group needs to be pushed to the next group. Of course the new page brought to the cache after a miss is assigned to the first group. Therefore, the running time of the GROUPS algorithm is linear to the number of page groups used.

In the first experiment we evaluated the accuracy and the running time of the three algorithms, running an IRP workload using all three algorithms. Experiments did not involve any I/O, only the LRU queue operations were executed and therefore the running time reflects the CPU cost for the stack distance measurement. The running time of GROUPS depends on the epoch length, while neither NAIVE nor TS use epochs. For GROUPS, we varied the epoch length from 10 to 10^4 operations. The size of the cache

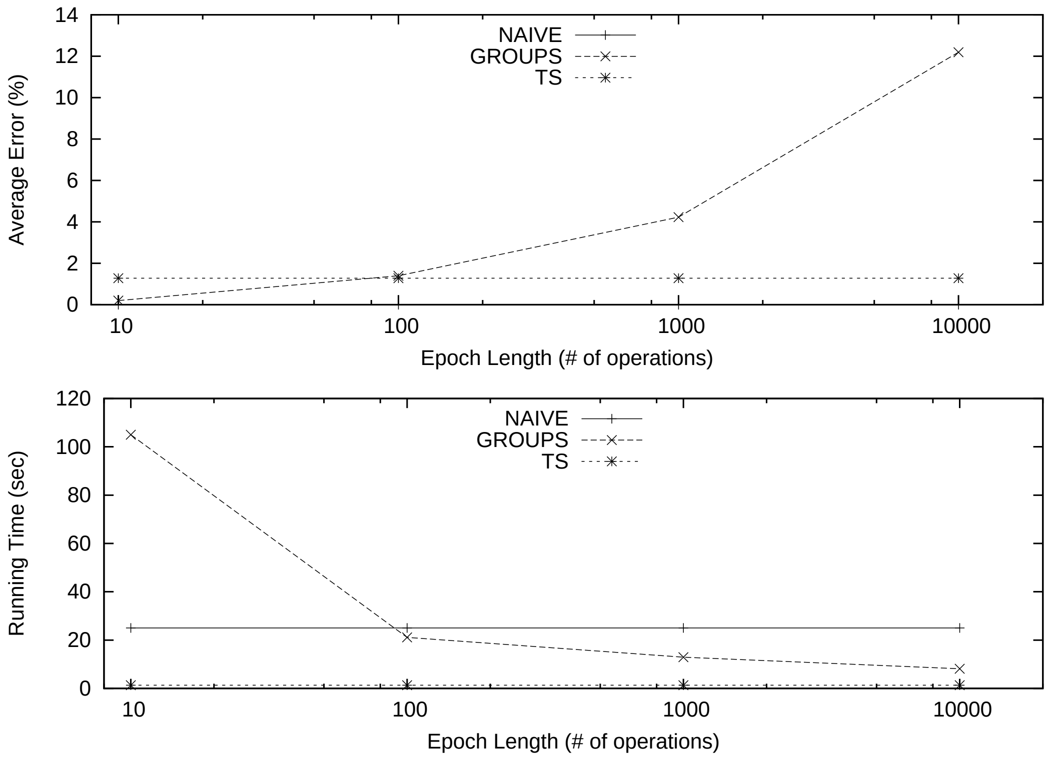


Figure 4.24: Effect of epoch length on running time and accuracy.

was 5k pages. For each run of the algorithms, we measured the running time. The average over five runs is reported in bottom graph of Figure 4.24 (the raw data are given in Figure 8.13). For GROUPS and TS we also computed the average error on the measured distance. We report the errors in the top graph of Figure 4.24, as a percentage of the actual distance (the raw data are given in Figure 8.12). The running time for TS is negligible, as it only requires $O(1)$ time per page hit. At the same time, TS achieves good accuracy, with an average error of less than 2%. For NAIVE, the running overhead is prohibitively high; this is due to a linear search taking place upon a page hit. The overhead for GROUPS grows as the epoch length shrinks; for very short epochs the overhead is heavier than the overhead of NAIVE. The accuracy of GROUPS increases as the epoch length shrinks; on the other hand, for long epochs GROUPS demonstrates poor accuracy, with the average error climbing up to 12%. For both very short and very long epochs, either the average error or the running overhead of GROUPS become unacceptably high; in all other cases TS exhibits both better accuracy and running time.

Next, we evaluated the accuracy of GROUPS and TS under different workloads. We ran an IRP workload and the TPC-C workload and calculated the average errors in distance measurements. For GROUPS we used an epoch of 10^4 operations, as for

Workload	GROUPS	TS
IRP (Δ)	12.19%	1.2%
TPCC (Δ)	41.21%	22.67%
IRP	2.75%	0.01%
TPCC	4.17%	0.1%

Table 4.1: Average Error (%) in distance estimation

shorter epochs the running time of the algorithm was comparable to the one of NAIVE (and thus using GROUPS would not make any sense anyway). The size of the cache was 5k pages. The results are shown in Table 4.1. In the top two rows (marked with a Δ) we report the error in the number of hits measured at each distance, averaged over all distances, that is the average error in $D[i]$ elements (the derivatives of the hit ratio curves at all different distances), using the terminology of Section 4.5.4. In the bottom two rows we report the average error of the total number of hits measured by the algorithms for different cache sizes, that is the average error of $\sum_{j=0}^{i} D[j]$ for all cache sizes j (the hit ratio curve values for different cache sizes). For both workloads TS achieved remarkably better accuracy than GROUPS, both for the derivative of the hit ratio curve and the hit ratio curve itself. For IRP workloads the TS was an order of magnitude more accurate than GROUPS. For TPC-C, TS was not as accurate as it was for IRP; this was expected due to our assumption that hit distances are uniformly distributed, which is not valid for TPC-C. As discussed in Section 4.5.4, the accuracy of TS for such workloads will improve if we compute the distribution of hit distances for these workloads and use it to estimate the error Δd_i. From the table one can also see that when summing the hit ratio curve derivatives to get the hit ratio curve, most of the errors are counterbalanced and therefore the average error for the curve is much less than the one for the derivatives. For both IRP and TPC-C the hit ratio curve error was less than 0.1% for TS, *i.e.*, it was practically as accurate as NAIVE.

We then moved on to study the effect of the size of the cache on the running time of the three algorithms. We used an IRP workload and varied the size of the cache from 100 to 40000 pages. We ran all three algorithms five times each; the average running times are shown on the top chart of Figure 4.25 (notice that both axis are in logarithmic scale). The raw data are given in Figure 8.14. As expected, the running time of TS is constant to the size of the cache. The running time of NAIVE is quadratic to the size of the cache: measuring the hit distance for each hit is linear to the size of the cache

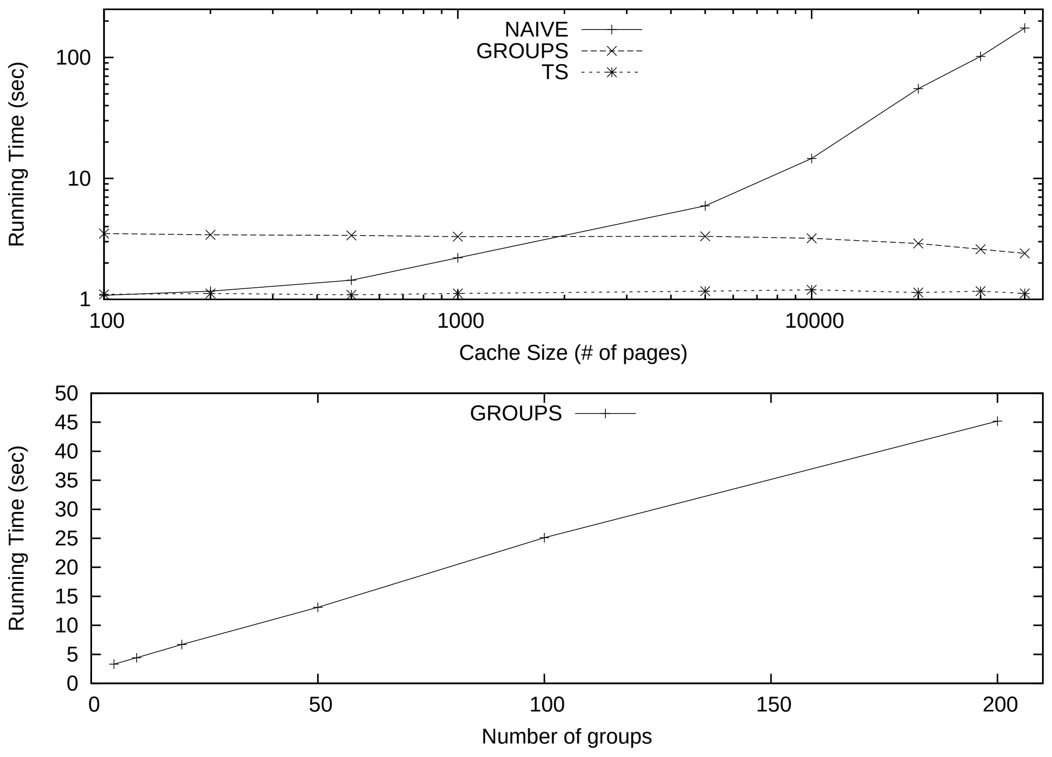


Figure 4.25: Effect of cache size and number of groups on running time.

and the hit ratio (and, thus, the number of hits) is also linear to the size of the cache (we used a random workload). For GROUPS, the running time drops as the size of the cache increases, because the number of page misses drops and therefore it has to do less processing (pushing the last page of each group to the next upon each miss). We also experimented with how the number of groups affects the running time of GROUPS, for a given cache size. Therefore, we kept the cache size fixed at 5k pages and varied the number of page groups from 5 to 200. The running times of GROUPS are shown in the bottom chart of Figure 4.25 (the raw data are given in Figure 8.15). As one would expect, the running time is linear to the number of groups used, since the running time of GROUPS is dominated by the time required to adjust the group borders.

In total, the accuracy/overhead ratio for TS proved to be far greater than the one for GROUPS. Of course, using NAIVE is out of the question for real-world caches due to its prohibitive overhead. Furthermore, TS is easier to implement than GROUPS and requires less bookkeeping: GROUPS needs every page to keep track of the group it belongs to, which results in more memory being wasted. On the other hand, TS uses timestamps which are typically maintained for each page anyway in real-world systems.

4.7.2 Synthetic Workloads

We then went on to evaluate the proposed region resizing algorithms using synthetic workloads, simulating a number of different storage device configurations. We created different scenarios that allowed us to test various properties of our algorithms. In all cases the page size was 4kB.

4.7.2.1 Effect of device cost discrepancy

In the first experiment, all devices had the same heat and the same address space, while references to pages of a device followed an independent reference pattern, that is all devices had the same number of hits. However, the read/write costs varied across devices. The read and write costs for device *i* were equal, *i.e.*, devices where symmetric *w.r.t.* I/O costs. We used two different setups: (*a*) the *linear* setup in which I/O cost of devices grow linearly with the device id, that is device 2 has twice the cost of device 1, device 3 has three times the cost of device 1, *etc* and (*b*) the *exponential* setup in which I/O cost of devices grow exponentially with the device id, that is device 2 has 10 times the cost of device 1, device 3 has 100 times the cost of device 1, *etc*. Of course, neither of the setups reflects the costs of real-world setup; however, it serves well to give some insight about how device cost discrepancy affects the behaviour of our algorithms. We used four devices in both setups. The whole dataset contained 1M pages, spread evenly among devices, while the workload consisted of 10M references to these pages. We used 50k pages as a main memory buffer pool.

The results are shown in Figure 4.26, in which all costs are represented as their percentage improvement over the cost of GBP (the raw data are given in Figure 8.16). As expected, the greater the discrepancy of I/O costs among the devices, the more I/O cost units our algorithms save, by allocating more page frames to the regions with high eviction cost. Since all regions have the same heat and hit ratio, the optimal strategy in this case is to allocate all memory to the region with the highest eviction cost. Under RUC, the cache size for devices was proportional to their read/write cost, as all devices saw the same number of references. Under HUC as many pages of the highest cost device as possible were cached, as under this strategy it saves the most cost units, given that the number of hits for all devices is the same. The same applies for HRCA. The best value for the λ parameter for CBR was $\lambda = 1.0$, *i.e.*, the cost segment spanned the whole buffer pool and cached pages from the high-cost devices. Notice that for the exponential setup, the performance of RUC matches the one of the rest of

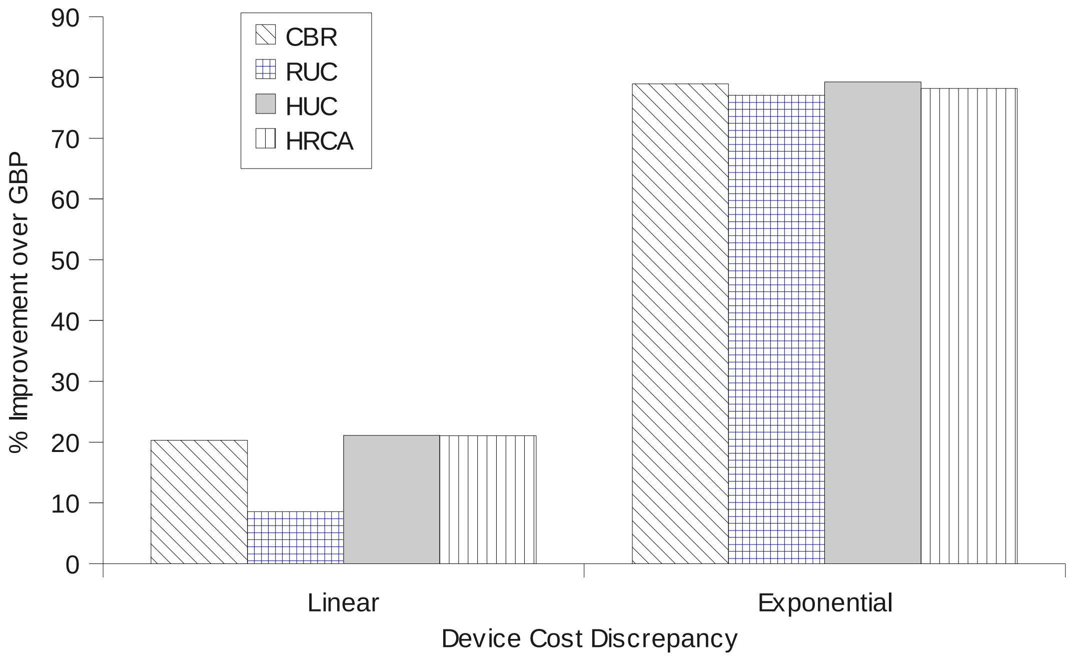


Figure 4.26: Effect of device cost discrepancy (read-only)

the algorithms; since the cost of the high-cost region was exponentially higher than the one of the rest of the devices, RUC allocated an equally high portion of the page frames to that device. All algorithms provided a substantial improvement over a global buffer pool, even for the case that all regions have the same heat and hit ratio. Of course, for CBR, the optimal value for λ had to be chosen manually. The latter is not a desirable property for real-world deployments.

4.7.2.2 Effect of device heat

We then went on to measure how the heat of a single device affects the behaviour of our algorithms. We experimented with 8 devices and three device cost configurations. In the first of those, Setup 1, all devices had the same read and write cost. In Setup 2, device 0 had the I/O cost of the Intel flash disk, while the rest 7 devices simulated magnetic disks. In Setup 3 device 0 simulated a magnetic disk, while the rest 7 devices simulated flash disks. The whole dataset contained 1M pages while the workload consisted of 10M references to these pages. We used 100k pages as a main memory buffer pool. Device 0 receives 40% of all references, with the rest 60% of references spread evenly among the other 7 devices. The workload only consisted of read requests. The results we collected are shown in Figure 4.27 (the raw data are given in Figure 8.17).

For Setup 1 the cost of CBR was equal to the cost of the global algorithm, as the

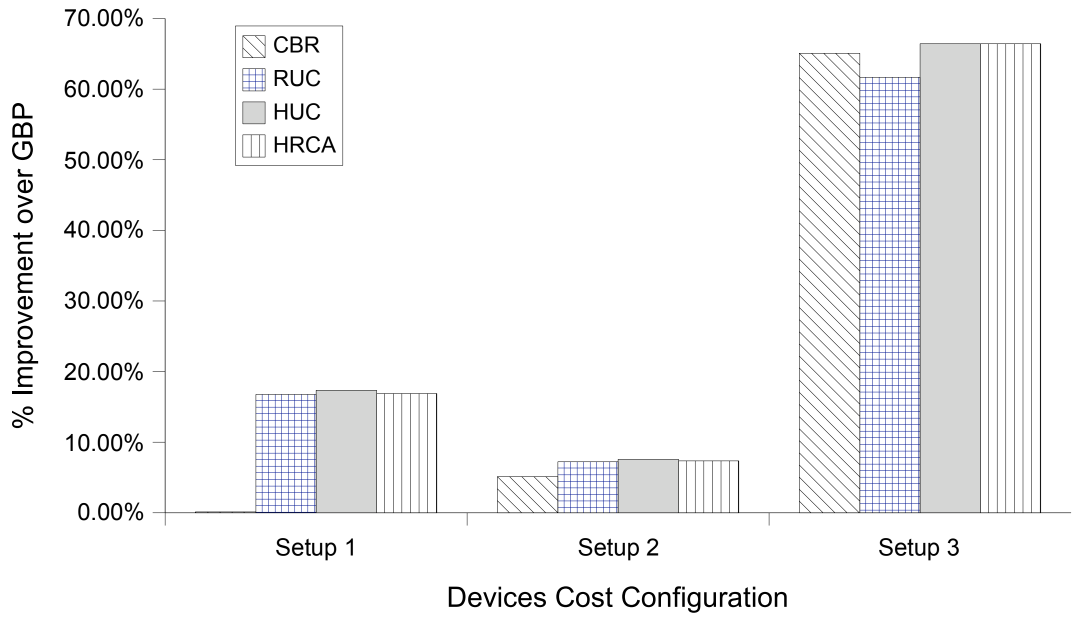


Figure 4.27: Effect of device heat

costs of all devices were the same. RUC detected the high heat of device 1 and allocated page frames to regions proportional to their heat. Similarly, HUC and HRCA detected the increased number of hits for device 0 and increased the number of frames allocated to it (note that the hit ratio for all devices was the same). For Setup 2, device costs came into play. The I/O cost of the hot device was 50 times less than the one of the rest of the devices, while the number of hits it exhibited was about 5 times more than the other devices (it received 40% of the references, while each one of the rest of the devices received $\frac{60\%}{7} \approx 8\%$ of the references). Therefore, the optimal behaviour is not to cache pages from the hot device at all and allocate all page frames to the high-cost devices instead. The best performance for CBR was for a $\lambda = 1.0$, which made the cost segment span the whole cache. RUC increased the size of the high-cost (albeit, cold) regions, based on the heat/device costs ratio. HUC and HRCA did the same, but their decision was based on the number of hits/device costs ratio. In the third Setup, the hot device was the high-cost one, while the rest were much faster. Obviously, this case leaves much more room for performance improvement. Based on the same reasons as the ones given for Setup 2, the algorithms now decided that the whole buffer pool should be allocated to the region of the hot device, which, of course, is the optimal decision. Improvement of I/O cost reached 67% in this case. Naturally, the placement of data in this setup is fundamentally flawed, as the hot pages should have been placed on the efficient device. Nevertheless, our results show that with utility-aware memory

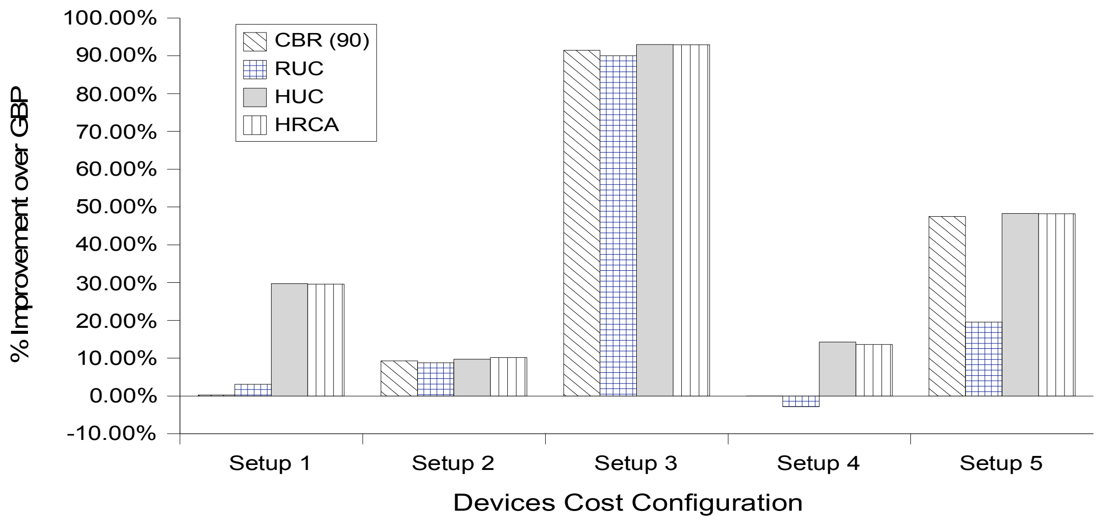


Figure 4.28: Effect of hit ratio

buffer allocation, a large portion of the extra I/O cost due to the wrong placement can be counterbalanced.

4.7.2.3 Effect of device hit ratio

In the next experiment, we studied the behaviour of the algorithms when regions receive the same heat, but have different hit ratios. The whole dataset contained 1M pages, while the workload consisted of 10M references to these pages. We used 100k pages as a main memory buffer pool. We simulated different cost setups for 2 devices. In the first of those, Setup 1, both devices had the same read and write cost. In Setup 2, device 0 had the I/O cost of the Intel flash disk, while device 1 simulated a magnetic disk. In Setup 3 device 0 simulated a magnetic disk, while device 1 simulated the Intel disk. We also used Setup 4 and 5 that employed arbitrary costs. For Setup 4 the read/write costs of device 0 were half the ones of device 1, and in Setup 5 the situation was vice-versa. The address space for device 0 was 5 times less than the one for device 1 (for all setups). As a result, the cache regions for device 0 exhibited 5 times higher hit ratio than the regions of device 1.

The results for this set of experiments are shown in Figure 4.28 (the raw data are given in Figure 8.18). For Setup 1, CBR had the same cost as the global buffer pool, because both devices had the same cost. The same holds for RUC, that was unable to distinguish between the different workloads of the two devices, as their heat was the same. Both HUC and HRCA detected the increased hit ratio of device 0 and increased the size for its regions, achieving a performance improvement of 30% against GBP.

Under the second Setup, the I/O costs of device 0 were about 50 times less than the ones for device 0, while the number of hits it exhibited was 5 times more than the hits for device 1; thus, the optimal allocation was to only cache pages from device 1. This is what CBR did for a $\lambda = 1.0$. RUC measured the same heat for both devices and therefore allocated 50 times more page frames to device 1 (proportionally to its I/O cost *w.r.t.* device 0). Both HUC and HRCA detected the high hit rate of device 0, but based on the cost discrepancy, correctly allocated all page frames to the regions of device 1. All algorithms achieved an improvement of 10% over GBP – caching more device 1 pages than GBP did not give a dramatic improvement as the hit ratio for device 1 was low anyway; however the decisions made by the algorithms were optimal. In Setup 3, similarly to the Setup 3 of Section 4.7.2.2, the placement of data on disks was very inefficient, as the data that gave the high hit ratio were placed on the slow device. The latter gave much room for improvement for all algorithms that made their decisions on the same principles as for Setup 2. By allocating all page frames to the high hit ratio pages of the slow device, they were able to realise a dramatic improvement of nearly 92%. Note that for both Setup 2 and 3 the optimal strategy is to allocate all memory to the high-cost device. This is not the case for Setup 4, in which pages of device 0 see 5 times more hits than the pages of device 1, but device 1 is only 2 times slower than device 0. Thus, as many pages as possible from device 0 should be cached. HUC and HRCA correctly identify the situation and make the optimal decision. For CBR the optimal value for λ is 0, *i.e.*, CBR effectively degenerates to GBP. RUC sees the same heat for both devices and therefore caches twice as many pages from device 1 as it does from device 0. This is obviously a wrong decision and, not surprisingly, RUC does worse than GBP. In Setup 5 device 0 has twice the cost of device 1 and therefore all page frames should be allocated to pages of device 1. The optimal value for CBR is thus $\lambda = 1.0$, using which it matches the performance of HUC and HRCA, that achieve about 48% improvement. RUC assigns $2/3$ of the buffer pool to the regions of device 0 and the rest to device 1. Again, this is only due to the costs of the two devices, not the properties of their workloads.

4.7.2.4 Effect of read / write ratio

Using the workload of Section 4.7.2.3, we then experimented with how the write intensity of the workload affects performance. As previously, accesses to device 0 gave a hit ratio 5 times higher than the one for device 1. Device 0 simulated the Samsung flash disk, while Device 1 simulated a magnetic disk. We run three series of experiments,

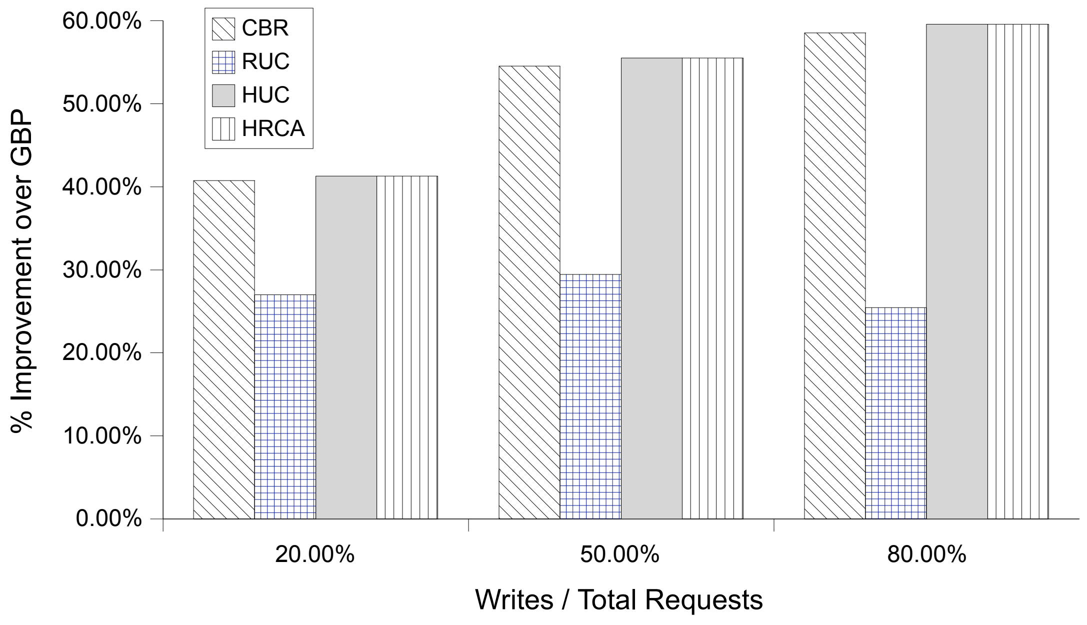


Figure 4.29: Effect of read / write ratio

each time varying the number of write requests. In the first case 20% of the requests were writes and the rest were reads. In the second case this percentage was 50%, while in the third case it was 80%. The size of the whole cache was 100k pages. The results we collected are shown in Figure 4.29 (the raw data are given in Figure 8.19).

For all three cases the optimal strategy was to allocate most of the buffer pool to the dirty region of device 0, as it had the highest hit ratio (along with the clean region of device 0) and by far the highest eviction cost. For CBR, the best value for λ was 1.0 in all three cases and the improvement it achieved matched the one of HUC and HRCA. Of course the value for λ had to be chosen manually. On the other hand, RUC could not distinguish the different hit ratios for the two devices, since they both saw the same number of references and therefore allocated page frames to regions proportionally to their eviction cost. As a result, almost $1/3$ of the buffer pool was allocated to the regions of device 1, that yielded much lower hit ratio. For this reason, irrespectively of the write ratio, RUC assigned the same portion of the cache to each region in all three cases. The rest three algorithms were able to increase the improvement over GBP as we increased the write ratio, by caching more dirty pages of the high-hit-ratio and high-cost device (0). For the specific setup, the more write-intensive the workload becomes, the less data should be placed on the flash disk, if optimising data placement is of interest (as in Chapter 3, [Koltsidas and Viglas, 2008]). Our cache allocation techniques however can offset some of the I/O cost that would be imposed due to the wrong

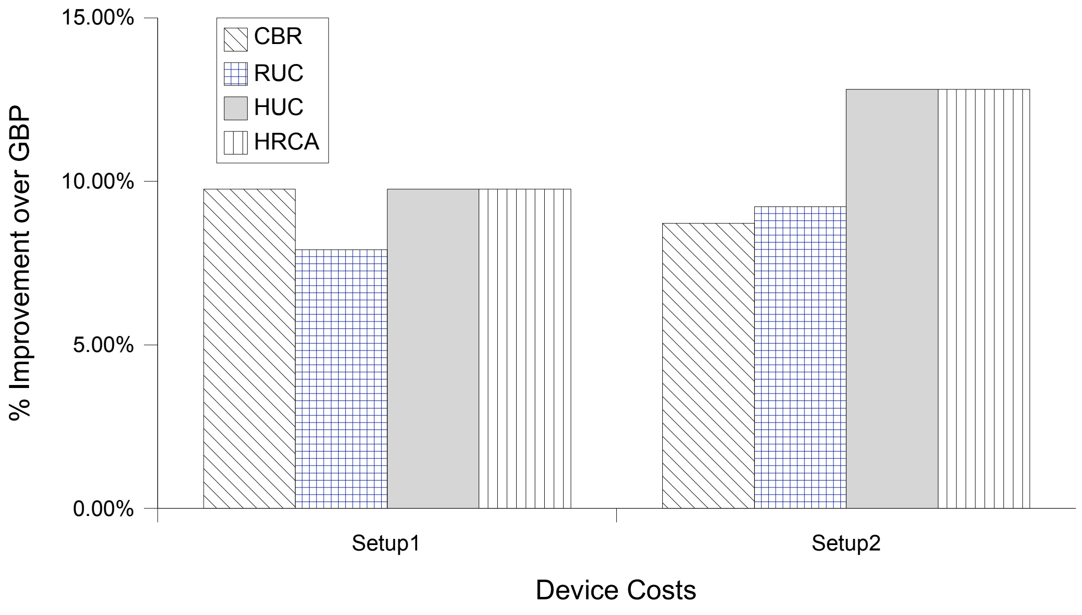


Figure 4.30: Changing Workload

placement by identifying which data are placed in the wrong medium and allocating more page frames to them in memory.

4.7.2.5 Changing workload

Next we experimented with a workload changing over time. We used 8 devices. In the first quarter of the workload, device 0 receives 40% of all references, with the rest of the references distributed equally to the rest of the devices. In the second quarter, all devices receive the same heat, but pages of device 3 give 5 times higher hit ratio than the one of the rest of the devices. The same occurs for device 4 in the third quarter of the workload. In the last quarter, device 0 becomes hot again, as in the first quarter. We used two different setups. In Setup 1 the hot device (device 0) and the high hit-ratio devices (3 & 5) simulated Intel flash disks and the rest of the devices simulated magnetic disks. In the second setup devices 0, 2, 4 and 6 were Intel flash disks, while the rest were magnetic disks. The total size of the cache was 100K pages. The results are shown in Figure 4.30 (the raw data are given in Figure 8.20).

In both setups, the allocation decisions of HUC and HRCA are optimal and therefore they achieve the highest performance increase over the global algorithm, 10% in Setup 1 and 13% in Setup 2. In Setup 1, CBR performed best for $\lambda = 1.0$, effectively allocating the whole buffer pool to the magnetic disks and matched the performance of HUC and RUC. In the second setup, however, the two devices that at some point had a high hit ratio in the workload are magnetic disks (two of the total four such disks). Since CBR allocates page frames according to device cost only, the page frames are distributed evenly among the magnetic disks and therefore disks 3 and 5 get less than the optimal in-memory space. This is why in Setup 2 CBR does worse than HUC and HRCA. RUC, on the other hand, fails to identify the high hit ratio of devices 3 and 5 in both cases and therefore its performance is worse than HUC and HRCA.

4.7.3 TPC-C Workloads

Next, we evaluated the performance of our algorithms with the TPC-C workload, using different data placement schemes. In all cases we used two storage devices. We used two different data placement schemes: (*a*) in the first one, referred to as *P*1, the placement of pages was decided using the data placement algorithms presented in [Koltsidas and Viglas, 2008] and in Chapter 3 using the costs of the Samsung flash disk, and (*b*) in the second, *P*2, the 15% most frequently accessed pages were stored on the flash disk, while the rest were stored on the magnetic disk, that is, *P*2 corresponds to the case that the flash disk itself is used as a cache.

P1. We ran the TPC-C trace using 10k pages of buffer pool, both with the cost of System A and System B. The results are shown in Figure 4.31 (the raw data are given in Figure 8.21). In both cases HRCA achieved the best performance, which outperformed the global algorithm by about 12% for System A and 17% for System B. Note that the I/O cost discrepancy for System B was much greater than the one for System A, due to the read/write asymmetry of the MLC flash disk. This is why the performance improvement is greater for System B. The optimal value for λ did not grant optimal behaviour to CBR in neither of the cases. RUC and HUC performed similarly for both setups. In the second case, however, HUC performed much worse than HRCA, an indication that the local maxima found by HUC in the hit-distance space (using the small and large caches) were not global ones and therefore its decisions were not optimal.

We also wanted to evaluate how accurate the total I/O costs reported by the simulator were. Therefore we ran this set of experiments on the real hardware (System A and B) and measured the total running time. Then we calculated the improvement using the measured time for all algorithms. In Figure 4.32 we have plotted the error between the projected improvement and the actual improvement measured (the raw data are given in Figure 8.22). In the vast majority, the error remained 3% or less, which means that if, for instance the projected improvement is 17%, as in System B of Figure 4.31 for

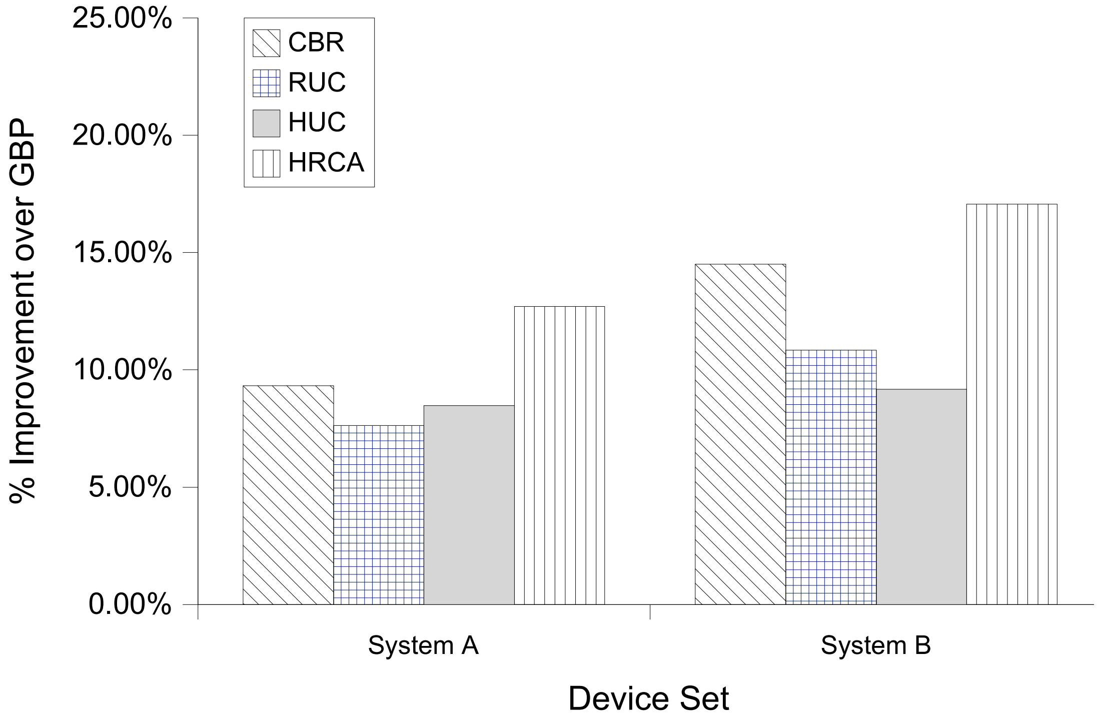


Figure 4.31: TPC-C with $P1$ placement

HRCA, then the real improvement will be between 16.49% and 17.51%. Thus, very safe conclusions can be drawn using the projected values. For all IRP workloads we ran in real hardware (not shown here) that error remained less than 1%, as in those all I/O was purely random, while for TPC-C short sequential access patterns were observed as well.

P2 Under this placement, the flash disk received 3 times more heat than the magnetic disk and 3 times higher hit ratio as well. In total it gave 9 times more hits than the magnetic disk. The results are shown in Figure 4.33 (the raw data are given in Figure 8.23). For System A, the flash disk is more than 50 times faster than the magnetic disk and therefore the optimal behaviour was to only cache pages from the magnetic disk (although better allocations may exist for specific time windows in the workload). For System B the optimal scheme is to allocate most of the cache to the dirty region of the flash disk, as it exhibits more heat and hit ratio than both regions of the magnetic disk, as well as, a much higher eviction cost. All algorithms are close to the optimal decision.

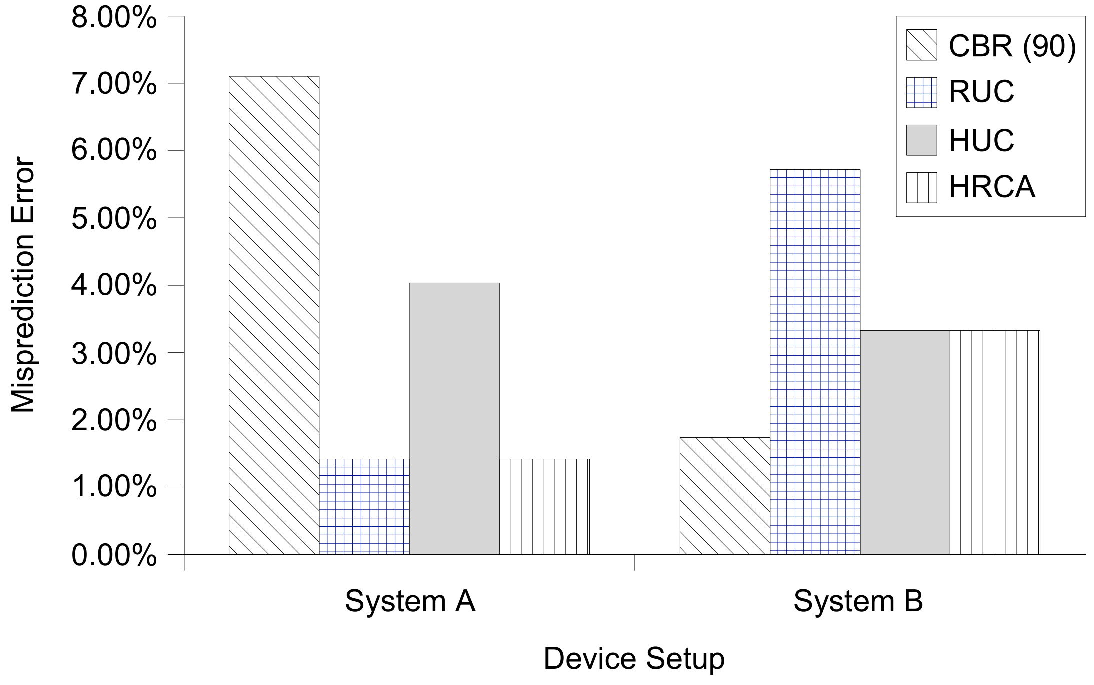


Figure 4.32: Misprediction Error

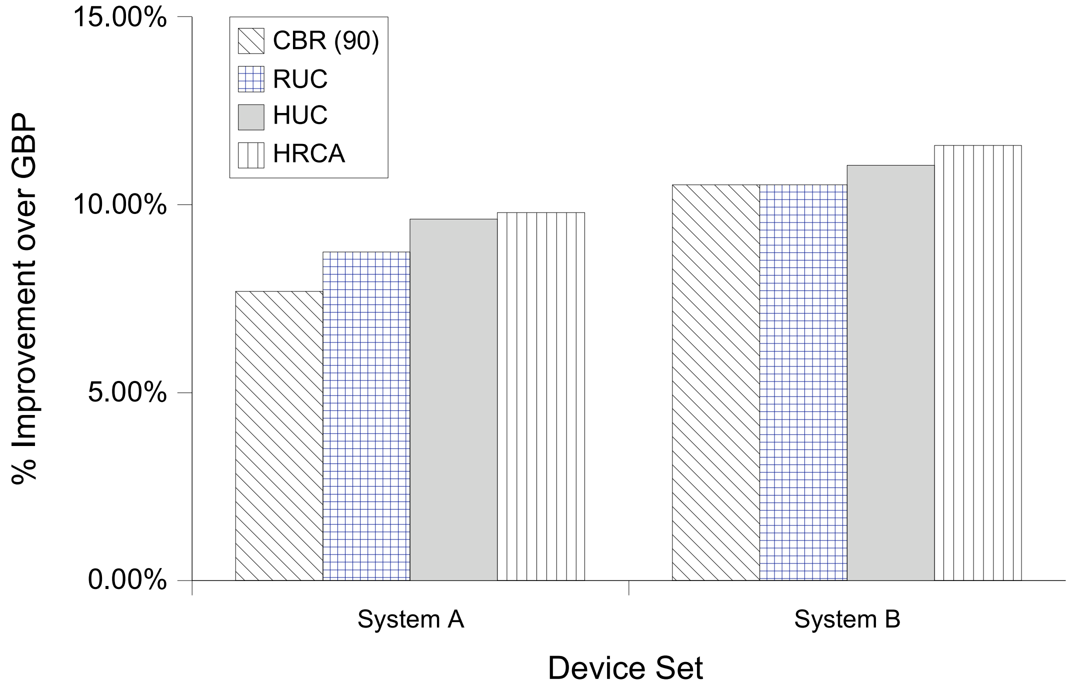


Figure 4.33: TPC-C with $P2$ placement

4.7.4 Optimal Allocation

Next, we studied how the greedy algorithm employed by HRCA for chunk allocations compares to the optimal allocation algorithm. The latter, which we refer to as OPT, ex-

Algorithm	Improvement (%)	CPU Time (sec)
HRCA	-	69
OPT	8.79%	358
OPT_TS	8.65%	69

Table 4.2: Using optimal allocation

haustively searches all possible allocations of chunks to regions and can therefore pick the globally optimal (at an exponential running cost, though). We used OPT along with the NAIVE distance measurement algorithm so that we get the best possible accuracy. In addition, we used TS distance measurement to get an idea of how OPT would behave with that; we refer to that as OPT_TS. We run the TPC-C workload with $P1$ placement for the System A setup (*i.e.*, there were 4 regions) and the improvement over HRCA and the running time of each allocation algorithm. We used 20 memory chunks. In Table 4.2 we report the results.

Using the greedy algorithm for the TPC-C workload is, of course, not optimal (while it is for IRP workloads). The OPT algorithm achieved an 8.79% better performance than HRCA, while OPT_TS reached an 8.65% improvement over HRCA; exhaustively searching the allocation space can thus give substantial improvement. An important observation is that the improvement is not due to measuring hit distances accurately (with NAIVE), as OPT_TS realises almost the same improvement using TS for measuring hit distances. Another thing to note is that for 20 memory chunks and 2 devices (*i.e.*, 4 regions) running the optimal allocation algorithm is feasible with practically no overhead (there are only 1771 different allocations to be considered at the end of each epoch). Therefore, for such cases OPT_TS should be used instead of HRCA. Of course, for more devices, or memory chunks, this is not possible. Using 20 chunks and 16 devices, for instance, yields over 32 billion different allocations.

Remarks. For each synthetic workload we used (except for the one used in Section 4.7.2.5), its characteristics with respect to device heat and hit ratios remained fixed throughout the workload; the adaptability of the algorithms was not tested. For such workloads HUC and HRCA did equally well in all cases. Of course this is due to the hit-distance distributions for the regions being constant functions and therefore the locally optimal allocation that HUC decided was globally optimal and thus matched the performance of HRCA. For most such workloads, CBR can also match the performance of HUC and HRCA *provided that* the optimal value for λ is priorly known; this of course is not something one can count on in real world deployments and as such one cannot expect CBR to make optimal decisions in the general case. It is, however, true that in most cases it did at least as well as GBP (for moderate values of λ). The performance of RUC was not as good as the one of HUC and HRCA, but in many cases it was very close. The worst cases for RUC were when devices saw high hit ratios without being any hotter than the rest of the devices. As expected, the higher the degree of discrepancy of the device costs, the more our algorithms could improve over the global algorithm. Last but not least, our experiments showed that with the correct allocation principles, the high I/O cost due to wrong placement of data on storage devices can be mitigated; our algorithms demonstrated dramatic improvement over GBP in such cases.

Chapter 5

Caching On Flash Memory

5.1 Introduction

In this chapter we explore the design principles for a system that uses a flash disk as a cache for the underlying storage, typically one or more magnetic disks. The motivating impetus of our work is the observation that flash disks exhibit low latency and high random read efficiency; this makes them ideal for use as read caches. By comparing the price and performance characteristics of high-end flash disks to those of DRAM and magnetic disks, it follows that a flash disk is ideal to serve as a cache layer between the main memory and the magnetic disk. This implies a 3-tier memory hierarchy. In this chapter we study analytical tools that enable the designer to decide with high confidence the optimal setup for such a system.

When designing a system with a 3-tier memory hierarchy like the one discussed here, one of the crucial decisions is determining the sizes of the main memory and the flash disk caches. As of September 2009, the cost of DRAM is about \$16/GB; the cost of flash disks varies from about \$1.6/GB for the low-end ones [TigerDirect.com, 2009], to about \$8/GB for the high-performance consumer flash disks [AnandTech, 2008], and to about \$30/GB for enterprise-level solutions [TGDaily, 2008]. As discussed in Chapter 1, the performance of flash disks in this price range varies by two orders of magnitude for reads and four orders of magnitude for random writes (refer to Table 1.2 of Chapter 1 for details). Considering the price/performance trade-off for the two types of cache, and given a specific budget, the question of what main memory and flash disk capacities minimise the price/performance ratio is not a straightforward one to answer.

At the next level, the designer is to decide which data will be cached on the flash disk. In contrast to buffering in main memory, pages do not need to be brought into the flash cache before they are processed. That is, a page may go directly from the magnetic disk to the memory and may well never be written to flash. Thus, deciding how data should flow from one level of the memory hierarchy to the others is not straightforward. A set of rules dictates the flow of data pages from one level to the others: we term this a *page flow scheme*. Relevant issues include how the workload of a page affects the decision about caching the page or not. For instance, in the ZFS filesystem [Sun Microsystems., 2008] dirty pages are never cached on flash. From an implementation perspective questions arise about the directory of pages cached on the flash disk and the optimal page size to use on flash. We show why these questions are crucial and provide the tools to address them. Our proposals and results are independent of the page replacement algorithm used by either cache and we therefore do not study page replacement at all in this context.

5.2 Problem Statement

Consider a database, or any other data processing system, with three components for data storage and staging: (*a*) RAM memory (*e.g.*, DRAM chips), (*b*) one or more flash disks, and (*c*) persistent storage, *i.e.*, a single hard disk, an array of hard disks, or any other collection of storage media. Data processing requires demand paging: pages are brought into main memory before processing, and this happens only on page referencing. The high-level architecture of such a system is shown in Figure 5.1. We refer to main memory as *RAM* and to the main memory buffer pool as *RAM cache*. We use *FLASH* to refer to the system's flash disk(s) used as a page cache (the on-flash cache is called *FLASH cache*); *HDD* refers to the underlying long-term storage.

When designing a page cache, the principal decision is *which* pages will be cached; *for how long* pages are cached for is determined by the replacement policy, which we do not consider. For a system of only a RAM cache and a hard disk, the former decision is not hard to make: *all* referenced pages will be written to the RAM cache, as this is required to use them. For a system with a FLASH cache in addition to the RAM one, however, there is no such requirement. As our goal is to reduce I/O operations to and from the HDD, the most reasonable thing to do under demand paging is to store on the FLASH cache the "hot" portion of the dataset that cannot fit in RAM. Let $P_{RAM}(t)$ be the set of pages stored in the RAM cache at some point in time t, and $P_{FLASH}(t)$ be the set of pages on the FLASH cache (for all practical cases, $|P_{RAM}(t)| < |P_{FLASH}(t)|$). We have identified the following three potential invariants:

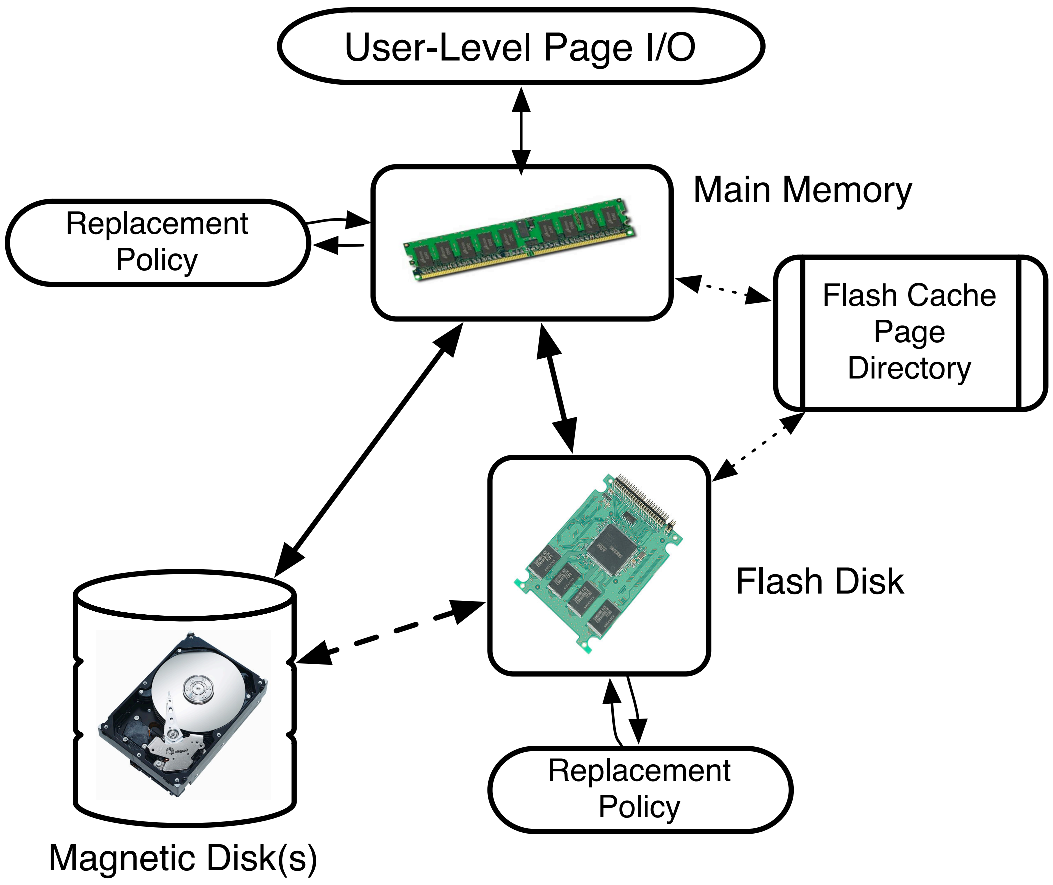


Figure 5.1: An overview of our system

1. $\forall t\ P_{RAM}(t) \bigcap P_{FLASH}(t) = P_{RAM}(t)$

   Whenever a page is in RAM, it is also cached on FLASH. This is analogous to the case of *inclusive* cache memory hierarchies of processors.

2. $\forall t\ P_{RAM}(t) \bigcap P_{FLASH}(t) = \emptyset$

   No page is stored on *both* RAM and FLASH at any time. A page brought from FLASH to RAM is removed from FLASH (and vice versa). Specifically, a RAM victim is stored in the frame of the page hit on FLASH, *i.e.*, a RAM page is *swapped* with a FLASH page.

3. $\forall t\ P_{RAM}(t) \bigcap P_{FLASH}(t) \subseteq P_{RAM}(t)$

   A page on RAM may or may not be cached on FLASH, depending on criteria either set by the user or decided based on the current workload.

Enforcing any one of the above invariants results in a different flow of pages across the levels of the memory hierarchy; thus, we define three different *page flow schemes*. Each scheme incurs a different I/O cost for a given workload. We detail all three schemes and model their I/O costs, without assuming any specific replacement policy for either the RAM or the FLASH cache. Page replacement is orthogonal to deciding which pages should be cached and where, which is the problem we study here. Therefore, our schemes may be used with any replacement policy.

5.3 Page Flow Schemes

We describe different page flow schemes to be used in a system employing a flash disk as a page cache between the main memory buffer pool and the hard disk.

5.3.1 The inclusive scheme

The inclusive scheme enforces the first invariant where the set of pages cached in RAM is always a subset of the pages cached on FLASH. The algorithm for fetching a page under this scheme is given in Figure 5.2. On a page reference, we look the page up in the directory for the main memory cache. If the page is found it is served in-memory. Else, we need to bring it in main memory and evict a page if memory is full. Given the invariant, the page must have been cached on flash, so it is written back only if it is dirty. We look the page up in the FLASH cache directory and, if the page is found, we read it from FLASH and put it into the RAM cache; else the page is read from HDD, written to the FLASH cache and then to the RAM cache. If the FLASH cache is full, a page v_f needs to be evicted from FLASH; if dirty, it will be written to HDD. Since $|P_{RAM}(t)| < |P_{FLASH}(t)|$, v_f will not exist in RAM if both caches use the same page replacement algorithm; if this is not true the FLASH replacement policy needs to ensure that it never evicts a page currently in RAM.

Let h_r, m_r, h_f and m_f respectively be the total number of RAM hits, RAM misses, FLASH hits and FLASH misses incurred by the whole workload. Also, let F_R, F_W, D_R, D_W be the average cost of a flash read, a flash write, an HDD read and an HDD write, respectively. These include the cost of writing the page to RAM or reading the page from RAM. Furthermore, consider the probability that a page in RAM is dirty before its eviction and let this probability be p_d. Let R_{RAM} be the cost of running the replacement algorithm for the RAM cache and R_{FLASH} be the corresponding cost for the FLASH cache. For the remainder of this chapter, we assume constant time replacement algorithms, *i.e.*, R_{RAM} and R_{FLASH} are negligible; still, we include them in the cost formulae for completeness.

```
Algorithm inclusive fetchPage (Page $pg$)

1.  if ($pg$ in RAM buffer pool)
2.    return $pg$
3.  else
4.    Evict a victim page $v_r$ from RAM
5.    Write $v_r$ to FLASH, iff it is dirty
6.    if ($pg$ in FLASH cache)
7.      Read $pg$ from FLASH
8.      return $pg$
9.    else
10.     Evict a victim page $v_f$ from FLASH
11.     Write $v_f$ to HDD, iff it is dirty
12.     Read $pg$ from HDD
13.     Write $pg$ to FLASH
14.     return $pg$
```

Figure 5.2: The inclusive page flow scheme

For each RAM hit, there is no I/O cost. For each RAM miss either a FLASH hit or a FLASH miss occurs (*i.e.*, $m_r = h_f + m_f$). For each FLASH hit, a page is evicted from RAM with cost $R_{RAM} + p_d F_W$ and a page is read from FLASH with cost F_R. For each FLASH miss, a RAM page is evicted with cost $R_{RAM} + p_d F_W$; also, a flash page is evicted with cost $R_{FLASH} + p_d D_W$ and the referenced page needs to be read from disk and written to FLASH, with cost $D_R + F_W$. The cost C_1 of inclusive is:

$$\begin{aligned} C_1 &= h_f(F_R + R_{RAM} + p_d F_W) + m_f(R_{RAM} + p_d F_W + R_{FLASH} + p_d D_W + D_R + F_W) \\ \Rightarrow C_1 &= h_f F_R + m_r(R_{RAM} + p_d F_W)) + m_f(R_{FLASH} + p_d D_W + D_R + F_W) \end{aligned}$$

We have not taken into account the operations on the RAM and FLASH page directories. If the page directory is stored in-memory for both caches, the cost of a lookup or an update is $O(1)$ – at least for computationally cheap replacement policies like LRU. As we will discuss later, however, the page directory of the FLASH cache may require a substantial portion of the main memory. In systems with limited main memory it may be more efficient to store the FLASH directory on FLASH itself. On this ground, we also need to take into account the costs of lookups and updates to the FLASH cache directory. Let L be the cost of a lookup on the FLASH directory and U be the cost of a directory update. An update is either the insertion or deletion of a page, or a bookkeeping update (*e.g.*, moving the page to the MRU position, if LRU is used). On a RAM miss, inclusive pays the cost of a FLASH directory lookup; on RAM eviction, it pays the update cost if the victim page is dirty, *i.e.*, p_dU (no lookup is required as the RAM victim will also be on FLASH). For each FLASH hit the bookkeeping of the directory is updated, while for each FLASH miss two updates are required: one for the victim page and one for the new page that is fetched. The cost of maintaining the directory C_1^d for the whole workload is:

$$C_1^d = m_r(L+p_dU)+h_fU+m_f(U+U)=m_r(L+U+p_dU)+m_fU$$

Hence, the total cost for inclusive is $C_1' = C_1 + C_1^d$.

5.3.2 The exclusive scheme

The exclusive page flow scheme enforces Invariant 2: the set of pages cached in-memory and the set of pages cached on FLASH are disjoint. The exclusive algorithm for fetching a page is given in Figure 5.3. The case for a RAM hit is the same as for inclusive. When a RAM miss occurs, we look the page up in the FLASH cache directory; if found, the page is read from FLASH. If the RAM cache is full, a page will be evicted from RAM. The victim is selected by the replacement policy and written to FLASH (whether it is dirty or not), while the referenced page is deleted from FLASH and inserted into RAM. Effectively, we swap the on-flash referenced page with the RAM victim. For a FLASH miss, the RAM victim is written to FLASH and the referenced page is read from the HDD into main memory. If the FLASH cache is full we also need to evict a page from FLASH.

For each RAM hit, no I/O cost is paid. Each RAM miss results in either a FLASH hit or a FLASH miss. For each FLASH hit the cost of evicting from RAM is $R_{RAM}+F_W$. The referenced page is read from FLASH with cost F_R and the victim page is written to FLASH with cost F_W. For each FLASH miss, FLASH eviction costs $R_{FLASH}+p_dD_W$ on top of the $R_{RAM}+F_W$ cost of evicting from the RAM cache. Moreover, reading the referenced page from HDD adds a cost of D_R. Thus, the cost of exclusive is:

$$C_2 = h_f(R_{RAM}+F_R+F_W)+m_f(R_{RAM}+F_W+R_{FLASH}+p_dD_W+D_R)$$

Let us consider the FLASH cache directory maintenance cost for exclusive. For each RAM miss, L cost units are paid for a FLASH lookup. For a FLASH hit we pay U cost units: the hit page is replaced by the RAM victim and the directory bookkeeping is updated. In the event of a FLASH miss the cost is equal to $U+U$: a page is evicted

```
Algorithm exclusive fetchPage (Page $pg$)
1.  if ($pg$ in RAM buffer pool)
2.    return $pg$
3.  else if ($pg$ in FLASH cache)
4.      Read $pg$ from FLASH
5.      Pick a victim page $v_r$ from RAM
6.      Replace $pg$ with $v_r$ on FLASH
7.      return $pg$
8.    else
9.      Evict a victim page $v_f$ from FLASH
10.     Write $v_f$ to HDD, iff it is dirty
11.     Evict a victim page $v_r$ from RAM
12.     Write $v_r$ to FLASH
13.     Read $pg$ from HDD
14.     return $pg$
```

Figure 5.3: The exclusive page flow scheme

from FLASH to HDD and another is evicted from RAM and written to FLASH. Hence, the directory maintenance cost is equal to:

$$C_2^d = m_r L + h_f U + m_f(U+U) = m_r(L+U) + m_f U$$

The total cost for exclusive is equal to $C_2' = C_2 + C_2^d$.

5.3.3 The lazy scheme

The lazy page flow scheme enforces Invariant 3 by caching an arbitrary set of referenced pages in the FLASH cache. Generally, the system decides if a page will be cached on FLASH when it evicts the page from main memory, *i.e.*, after the system has an indication for the workload of a page by applying user-specified criteria. In the simple case, which we will focus on for the moment, no such criterion is used; rather, a RAM victim is always written to FLASH and stays there until evicted by the FLASH cache replacement policy. The lazy algorithm for fetching a page is given in Figure 5.4. A page is served in-memory if found in RAM. Else, we look it up in the FLASH directory. If a FLASH hit occurs, the page is read from FLASH (and the directory's bookkeeping is updated). If the RAM cache is full, a victim is picked by the RAM replacement policy and then evicted. We must then find out if the victim is also on FLASH. If so, it is writ-

```
Algorithm lazy fetchPage (Page $pg$)

1.  if ($pg$ in RAM buffer pool)
2.    return $pg$
3.  else if ($pg$ in FLASH cache)
4.      Read $pg$ from FLASH
5.      Evict a victim page $v_r$ from RAM
6.      if $v_r$ in FLASH cache
7.          Write $v_r$ back to FLASH, iff it is dirty
8.      else
9.          Evict a victim page $v_f$ from FLASH
10.         Write $v_f$ to HDD, iff it is dirty
11.         Write $v_r$ back to FLASH
12.     return $pg$
13.   else
14.     Evict a victim page $v_r$ from RAM
15.     if $v_r$ in FLASH cache
16.         Write $v_r$ back to FLASH, iff it is dirty
17.     else
18.         Evict a victim page $v_f$ from FLASH
19.         Write $v_f$ to HDD, iff it is dirty
20.         Write $v_r$ back to FLASH
21.     Read $pg$ from HDD into RAM
22.     return $pg$
```

Figure 5.4: The lazy page flow scheme

ten back only if it has been dirtied in RAM. Otherwise, a page is evicted from FLASH (and written back to HDD) to make room for the RAM victim to be written to FLASH. For a FLASH miss, a page is evicted from RAM and written to the FLASH cache in the same fashion as for a FLASH hit. The referenced page is read from HDD and brought directly into main memory. Note that on Line 9 one can apply any predicate based on the workload history for that page to decide whether the page should be cached on FLASH or not. We discuss such alternatives later on.

Consider now the cost of the lazy scheme. A main memory victim may or may not exist in the FLASH cache. Let the probability of a RAM victim being on FLASH be q. The cost associated with a RAM victim C_3^V (*i.e.*, the cost of Lines 5-11, 14-20) is equal to $R_{RAM} + p_d F_W$ if the page is on FLASH and $R_{RAM} + R_{FLASH} + p_d D_W + F_W$ otherwise.

Hence:

$$C_3^V = R_{RAM} + qp_dF_W + (1-q)(R_{FLASH} + p_dD_W + F_W)$$

For a FLASH hit the cost is $F_R + C_3^V$ and for a a FLASH miss the cost is $C_3^V + D_R$. The cost of this scheme is therefore:

$$C_3 \quad = \quad h_f(C_3^V + F_R) + m_f(C_3^V + D_R) = (h_f + m_f)C_3^V + h_fF_R + m_fD_R$$

As noted in Section 5.3.1, both h_f and m_f represent the total hits and misses for pages referenced in the workload. Thus, lookups in the FLASH index for the RAM victim are not accounted for by h_f and m_f. However, the probability of the RAM victim being on FLASH is expected to be equal to the probability of any referenced page being on FLASH: it does not depend on whether the looked-up page was in RAM at the time of the lookup. Therefore, $q = \frac{h_f}{h_f+m_f}$ and $1-q = \frac{m_f}{h_f+m_f}$, which gives that:

$$C_3^V \quad = \quad R_{RAM} + \frac{h_f}{h_f+m_f}p_dF_W + \frac{m_f}{h_f+m_f}(R_{FLASH} + p_dD_W + F_W)$$

Then:

$$\begin{aligned} C_3 &= (h_f+m_f)R_{RAM} + h_fp_dF_W + h_fF_R + m_f(R_{FLASH} + p_dD_W + F_W) + m_fD_R \\ \Rightarrow C_3 &= h_f(R_{RAM} + p_dF_W + F_R) + m_f(R_{RAM} + R_{FLASH} + p_dD_W + D_R + F_W) \end{aligned}$$

As for the maintenance cost of the FLASH page directory, lazy pays L cost units for each RAM miss to look the page up in the FLASH directory. For a FLASH hit, U cost units are paid to update the entry for the hit page, in addition to the cost $C_3^{V,d}$ of updating the directory entry for the RAM victim. If a FLASH miss occurs, only $C_3^{V,d}$ cost units are paid for directory maintenance. For $C_3^{V,d}$, we have that:

$$C_3^{V,d} \quad = \quad L + qp_dU + (1-q)(U+U)$$

Hence, we have that:

$$\begin{aligned} C_3^d &= m_rL + h_f(U + C_3^{V,d}) + m_fC_3^{V,d} \\ \Rightarrow C_3^d &= m_r(L + C_3^{V,d}) + h_fU \\ \Rightarrow C_3^d &= m_r(2L + qp_dU + 2(1-q)U) + h_fU \\ \Rightarrow C_3^d &= 2(h_f+m_f)L + h_fp_dU + 2m_fU) + h_fU \\ \Rightarrow C_3^d &= h_f(2L + (p_d+1)U) + 2m_f(U+L) \end{aligned}$$

And, for lazy: $C'_3 = C_3 + C_3^d$.

Various criteria can be applied to decide whether a RAM victim page should be cached on flash. For instance, if the flash disk used is poor in random writes, one can avoid some random writes by caching only clean pages. We experiment with this in Section 5.5.6. Alternatively, the access history for each FLASH page can be maintained, *e.g.*, by keeping track of the number of hits the page has seen, or the number of times it has been dirtied. The system could then maintain a set of the f hottest pages, where f is the number of pages that fit in FLASH. Then, only these f pages will be cached on flash, thus implementing a frequency-based replacement policy. Similarly, one may decide to cache the f pages that have the most read-intensive workload as in Chapter 3. Another alternative is to only cache pages that have been accessed at least twice while they were cached in RAM; that way, one can avoid polluting the flash cache when a file scan occurs. The mentioned options can even be combined; however, we will not study them further here, as they assume or define some aspects of the replacement policy of the cache.

5.3.4 Comparison

We compare the page flow schemes on the basis of their I/O costs. We assume (for now) that the FLASH cache directory is stored in main memory and we therefore do not consider directory maintenance costs. Given the cost formulae for C_1, C_2, C_3, it is tempting to factor out the common term $h_f F_R + m_f(R_{FLASH} + F_W + D_R) + m_r R_{RAM}$. However, this would make the invalid assumption that, for a fixed workload, h_f, m_f remain the same for all three page flow schemes. We will now show why this is not true.

Assume that a workload is executed on the same system three times, once with each page flow scheme presented. In all cases the RAM and FLASH replacement policies are the same. What is more, assume a stack replacement algorithm (not a FIFO one), *i.e.*, one that does not exhibit Belady's anomaly [Belady et al., 1969] and therefore the hit ratio for the policy grows with cache size (that is, with the number of available page frames); this holds for virtually all modern page replacement algorithms. Let r, f be the maximum capacity, in pages, of the RAM and FLASH caches, respectively. We define the *effective capacity* of a cache at level i to be the number of pages cached at level i that are guaranteed *not* to be cached at any level higher than i at the same time. The effective size of the RAM cache is $e_r = r$. For the FLASH cache, its effective size

e_f is equal to the number of pages cached on FLASH that are unique on FLASH (*i.e.*, none of them are cached in RAM at the same time). Consequently, the effective size of the FLASH cache is different for each page flow scheme. For inclusive the effective size of the FLASH cache is $e_f^1 = f - r$, while for exclusive it is $e_f^2 = f$. For lazy, the subset of FLASH pages also cached in RAM varies with the workload; however the following always holds:

$$f - r \leq e_f^3 \leq f$$

Observe that the FLASH cache hit ratio depends on the effective size of the cache, not on its capacity. Consider, *e.g.*, inclusive: when it looks a page up on FLASH, it is only likely to find the requested page in $f - r$ pages, for if the requested page was any of the r pages cached in RAM, no lookup on FLASH would be needed. Therefore, the hit ratio is a function of the page replacement policy, the effective size of the cache, and the workload (the reference pattern). For a given replacement policy Y and a workload W, let the hit ratio be $H = H(Y,W,e_f)$. We have that:

$$H(Y,W,e_f^1) \leq H(Y,W,e_f^3) \leq H(Y,W,e_f^2)$$

Taking into account that $h_f = H \cdot |W|$, we have that:

$$h_f^1 \leq h_f^3 \leq h_f^2 \Rightarrow m_f^1 \geq m_f^3 \geq m_f^2$$

for the three algorithms, since $m_r = h_f + m_f$. The effective size of the RAM cache is the same for all different schemes; the same applies for the RAM hit ratio.

One can only model the hit ratio for a page replacement policy if the characteristics of the workload are priorly known. Our evaluation shows that for a given policy the hit ratio varies widely across different workloads. This suggests that in a real deployment, where the characteristics of the page reference pattern are not known *a priori*, one cannot statically determine the optimal page flow scheme. In our system we continuously monitor the hit ratio with respect to the effective size of the FLASH cache and accordingly adapt the flow of pages in the memory hierarchy. Specifically, we keep track of FLASH hits and misses and the rate at which pages are dirtied (p_d). Based on the normalised read and write costs for the flash and the hard disks, which are known (or measured) in advance, we periodically evaluate the cost formula for each page flow alternative and adopt the one that minimises the total cost. In Section 5.6, we discuss some workload characteristics based on which one can decide the optimal scheme statically and with some confidence.

5.4 Implementation Issues

One important decision is the location of the FLASH cache page directory when the RAM cache is much smaller than the FLASH cache. For r pages cached in RAM, $b \cdot r$ is the size in bytes of the RAM directory, for b bytes per directory entry. Similarly, $b \cdot f$ bytes are needed for the FLASH directory. Let B be the size of a page in bytes, S_r be the size of RAM, and S_f be the size of the FLASH disk; we have that $f = \frac{S_f}{B}$. If the FLASH directory is stored in-memory, $S_r - bf$ bytes are left in main memory for caching. Hence, $m = \frac{S_r - bf}{B+b}$ pages are cached in RAM.

If the entire main memory is used for the RAM cache it fits $\frac{S_r}{B+b}$ pages. Given a replacement policy and a workload the following holds:

$$H\left(Y, W, \frac{S_r}{B+b}\right) \leq H\left(Y, W, \frac{S_r - bf}{B+b}\right)$$

Our experiments show the difference between these two hit ratios to be significant, more so as the discrepancy between the RAM and FLASH sizes grows. Thus, it may be desirable to reduce the portion of main memory used for the FLASH page directory. To that end there are two alternatives: (a) use a larger page size for FLASH, or (b) store the FLASH directory (or a part of it) on FLASH instead of RAM.

5.4.1 Using larger pages for flash

Let B_r and B_f be the RAM and FLASH page sizes respectively; b_r bytes are required for a RAM directory entry and b_f bytes for a FLASH directory one. Each directory entry holds, at the very least and in both cases, the HDD offset of the page (also serving as its identifier), a pointer to the page in the cache (a main memory pointer for a RAM page, or a disk offset for the FLASH cache) and a dirty bit. The replacement policy requires extra bytes for bookkeeping (*e.g.*, a pointer to the next page in an LRU queue) and for the in-memory hash table required for lookups. For the RAM directory some more bookkeeping is required for pinning/unpinning, concurrency control, *etc.*; we will not further elaborate on b_r as it is not our focus.

If $B_f > B_r$, each FLASH page has $\frac{B_f}{B_r}$ RAM pages; we use the term *block* to refer to such FLASH pages. All I/O between the flash disk and the HDD is in blocks of B_f bytes, while data movement from and to the RAM cache is in pages of B_r bytes. The RAM cache and all in-memory structures use the HDD *offset* of a page as its universal identifier. Thus, the RAM directory uses $\frac{offset}{B_r}$ as the page identifier, while the FLASH directory uses $\frac{offset}{B_f}$ as the identifier for a block stored at *offset* on HDD. Therefore

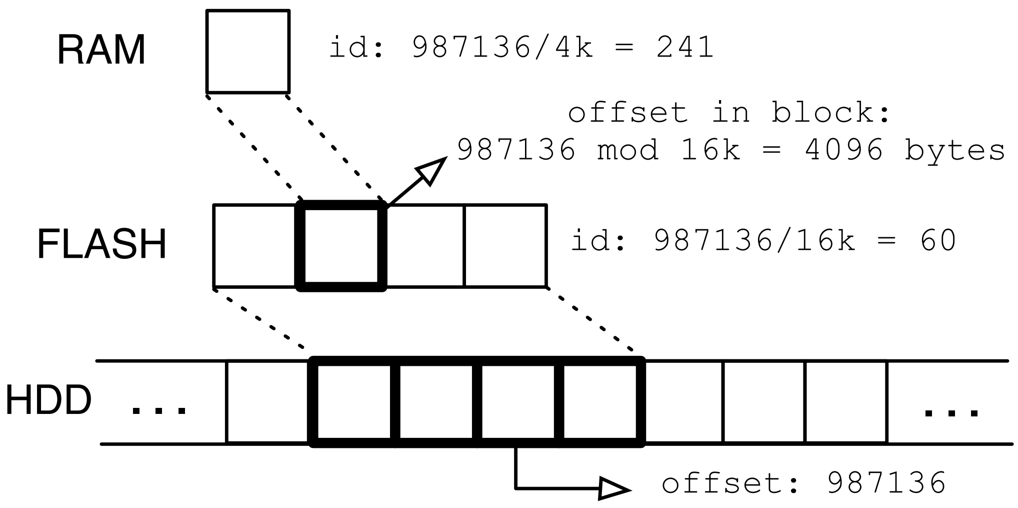


Figure 5.5: Using larger FLASH pages

$\log_2 \frac{offset}{B_f}$ bits are required to identify a page in the FLASH directory. By knowing the RAM directory identifier of a page, one can use B_f and B_r to obtain the identifier of the FLASH block where the RAM page belongs. Let p_r be a RAM page of FLASH block p_f. For each reference to p_r, we look it up in the FLASH directory. If p_f is there, then p_r is located at offset $(p_r B_r \mod B_f)$ within p_f, the offset of which is given by $(p_r B_r \div B_f)$. Otherwise, p_f is read from HDD into FLASH and p_r is computed the same way. FLASH evictions take place with B_f granularity. The case for $B_r = 4$kB and $B_f = 16$kB is shown in Figure 5.5.

If p_r is evicted from RAM to FLASH but p_f is not cached on FLASH at that time (a case that arises under exclusive and lazy), writing page p_r of block p_f to FLASH is not straightforward. One solution is to first fetch p_f from HDD into FLASH and overwrite its p_r page incurring one additional HDD read. We refer to this technique as overwriting. Note that when fetching the whole block from the HDD, some pages of the block may already be cached in RAM. Therefore, the invariant $\forall t P_{RAM}(t) \bigcap P_{FLASH}(t) = \emptyset$ of the exclusive scheme is not strictly enforced with this technique. Under inclusive this never arises: any page cached in RAM will have its host FLASH block cached on FLASH.

An alternative solution is to assign a block to p_f on FLASH, invalidate all its pages but p_r, and overwrite p_r. If block p_f is later read from HDD, only the invalid pages will be overwritten on FLASH; if it is written to HDD, only the valid pages will be written. We term this technique invalidating. Except for a slight implementation complexity the main drawback of this solution is that a large number of pages in a flash block may

Flash Page Size	overwriting (MBs)	invalidating (MBs)
4K	568	N/A
8K	280	284
16K	138	142
32K	68	72
64K	33.5	37.5
128K	16.5	20.5

Table 5.1: FLASH directory size

become invalid, and, thus, waste space. This is especially true if the reference pattern exhibits poor spatial locality. A solution is for invalid pages not to be stored on FLASH blocks, but only *marked* as invalid in the FLASH directory. The following example shows the impact of each technique on the size of the FLASH directory.

Example. Let $B_r = 4$kB, $B_f = 64$kB and assume a 64-bit offset for HDD and FLASH, and 32-bit main memory addressing; $\log_2 \frac{2^{64}}{64\text{k}} = 48$ bits are required for the block identifier. For a 128GB FLASH cache, there are 2M flash blocks. Thus, a FLASH block requires $\log_2(2\text{M}) = 21$ bits for flash addressing (*i.e.*, the flash disk offset where the page is stored). Also needed are: 1 dirtiness bit for each block; 32 bits for the main-memory pointer in the LRU queue; and 32 bits for the main memory hash index (a pointer to the directory entry) for simple LRU. In total a FLASH directory entry requires at least $48+21+1+32+32 = 134$ bits. Using overwriting the FLASH directory occupies $\frac{2\text{M}\cdot 134}{8\cdot 1\text{M}} = 33.5$MB. Using invalidating, one additionally needs $\frac{64k}{4k} = 16$ bits per FLASH block as validity bits (one per block page) These sum up to 150 bits per directory entry, or 37.5MB. If larger FLASH blocks were not employed at all, *i.e.*, 4kB pages were used for FLASH, the identifier of a page would require 52 bits and there would be 32M block addresses on flash, the representation of each of which would require 25 bits. Thus, $52+1+25+32+32 = 142$ bits would be required for each of the 32M pages, or 568MB in total. □

Table 5.1 shows the directory size for FLASH blocks of various sizes, for a 128GB flash disk. Using larger pages on FLASH saves a lot of main memory, which can be used for caching in RAM to increase the RAM hit ratio. Larger flash pages, however, reduce the paging (and thus caching) granularity, so the flash hit ratio is expected to drop, especially for workloads with poor spatial locality. We further explore this trade-off in Section 5.5.5. Note that, as studied in [Bouganim et al., 2009], writing on flash using a large block size (*e.g.*, 32 or 64KB) increases bandwidth and random write efficiency. Therefore, large flash blocks are not only a way to shrink the FLASH directory and increase RAM hits, but also a way to speed up random writes to flash.

5.4.2 Storing the page directory on flash

If the amount of main memory is small or the FLASH directory occupies too much memory, it may be preferable that the whole FLASH directory is stored on the flash disk (at least for low latency flash disks). Another reason is that, given the non-volatility of flash, if the FLASH directory is persistent, its contents can be preserved between crashes, thereby eliminating warm-up time. In this setup, the directory itself is stored on FLASH and some of its pages are buffered in main memory. Let f' be the number of pages cached on FLASH; hence the size of the directory is $b_f f'$. Then, $f' = \frac{S_f}{B_f + b_f}$ and the size of the directory is $\frac{b_f S_f}{B_f + b_f}$. Assuming that n bytes of the directory are buffered in main memory, with $b_r \leq n \leq b_f f'$, the size of main memory available for caching pages is $S_r - n$ and the number of pages cached is $r' = \frac{S_r - n}{B_r + b_r}$. It is clear that $f' < f$ and $r' > r$, where f, and r are for when the FLASH directory is kept in main memory, as in the previous sections. Therefore, $e'_f = e_f - b_f f' < e_f$ for all three page flow schemes if an external FLASH directory is used. Consequently, $H(Y, W, e'_f) \leq H(Y, W, e_f)$, *i.e.*, the hit ratio for the FLASH cache will drop. The size of RAM available for caching is now greater by $b_f f' - n$; thus, the RAM hit ratio will rise. Given that $S_r \ll S_f$, then, for small values of n, the increase of the RAM hit ratio will be greater than the drop of the FLASH hit ratio. Additionally, as n shrinks the cost of a directory lookup/update grows, as fewer pages of the directory are buffered. The hit ratio for directory pages is given by $H(Y, W_d, n)$, where W_d is the reference pattern for directory operations generated by workload W. If l directory page accesses are required for a lookup, u directory page accesses are required for an update, and the probability of a directory page being dirtied in RAM is p_d^d, we have:

$$L = l((1 - H(Y, W_d, n))F_R + p_d^d F_W)$$
$$U = u((1 - H(Y, W_d, n))F_R + p_d^d F_W)$$

neglecting the cost of directory page lookups/updates served in-memory. Substituting these formulae to the corresponding ones of Sections 5.3.1, 5.3.2, and 5.3.3, one can calculate the expected cost. Of course, the directory page miss ratio $1 - H(Y, W_d, n)$ and probability of a directory page being dirtied p_d^d will have to be monitored or otherwise estimated.

5.4.3 How much flash? How much RAM?

For the case that no FLASH cache is used, assume h_r RAM hits and m_r RAM misses occur for a workload. Then, the total cost C_0 for this case is $C_0 = m_r(D_R + p_d D_W)$. Using a simulator, one can simulate the cache behaviour of a system with varying RAM and FLASH cache sizes (or, even with no FLASH cache). Specifically, by running the simulator for various cache sizes and for the type of workload the system will process, values can be collected for h_r, m_r, h_f, m_f, and p_d. By using the values in the cost formulae, along with the read/write costs of specific flash and magnetic disks, one can determine which combination of cache sizes and hardware devices is the most I/O-efficient for workloads of the given type. Moreover, the price-to-I/O-cost ratio for each case gives the most cost-efficient solution. Alternatively, the 5-minute rule of [Graefe, 2007] can be used to determine the optimal memory and flash disk capacities required, assuming prior workload knowledge. Our cost formulae determine the type of flash disk that gives the best price/performance ratio for a type of workload. Naturally, the decision for the size of the main memory and the flash disk is an offline one and optimised for specific workloads. However, the optimal page flow scheme can be decided on-line, on a per-workload basis, by periodically evaluating the cost formulae. Our proposals are also applicable in database systems that employ per-file/relation buffer management: by monitoring the workload for each file and calculating the cost of each different scheme, our model may lead to different relations being buffered using different schemes.

5.5 Experimental Study

Setup. We implemented our algorithms to evaluate their performance under various workloads. Our system consists of a main memory buffer pool for caching in RAM, a page cache on a flash disk and a magnetic disk for persistent storage. Each page is identified by its disk offset on persistent storage. The system was implemented in C++ and we used an Intel Pentium 4 box clocked at 2.26GHz with 1.5GB of physical memory for our experiments. The Operating System was Debian GNU/Linux with the 2.6.26 kernel. The system had two magnetic disks and one flash disk. Our system and the OS ran from one of the magnetic disks and the other magnetic disk (referred to as the HDD hereafter) was used to store the dataset. The HDD was a 300GB Maxtor 6L300R0 with 16MB of cache. The flash disk was a Samsung MCAQE32G5APP, an MLC NAND flash

Disk Model	4kB Read IOPS	4kB Write IOPS	$/GB
Samsung	2500	21	1.6
Intel X25-M	12000	592	8.1
Intel X25-E	35000	3300	20
Fusion ioDrive	102000	101000	30

Table 5.2: Flash disks considered

disk with a capacity of 32GB. Both disks were using the IDE interface. To eliminate OS caching we used both storage media as raw devices: the OS did not cache data pages, pages were never double buffered and our system had absolute control of physical I/O. Read and write costs were estimated using the 3-d surfaces presented in Section 4.6.

Flash Disks. The flash disk we used has a poor write performance and is unsuitable as a cache. Therefore, we considered other flash disks, more suitable for caching, by using their read and write costs in the equations of Section 5.3. For the I/O costs of these disks we used published benchmarks and documents about their efficiency in IOPS ([TGDaily, 2008, AnandTech, 2008, TigerDirect.com, 2009]). We present the read/write costs of all flash disks considered in Table 5.2. Observe that random read performance varies up to two orders of magnitude among disks, while random write performance varies as much as four orders of magnitude.

Workloads. We used three different workloads. The first, referred to as IRP, follows an independent reference pattern where all pages in the dataset have the same probability of reference, *i.e.*, a random reference pattern. We varied the probability of a page being read or written to and created workloads with varying dirtiness ratios. For the second workload, referred to as TPC-C, we ran the TPC-C benchmark on the PostgreSQL database and collected a trace of all page references, which we then translated into HDD offsets. We did the same for the TPC-H benchmark to obtain the third workload. The results of this section are the execution of these traces by our system after varying its parameters. In all cases, the main memory page size was set to 4kB. For all experiments we used LRU as the page replacement policy (for both the RAM and the FLASH caches).

5.5.1 Impact of Cache Size on Hit Ratio

We measured the effect of the size of a page cache on its hit ratio, *i.e.*, how $H(Y,W,S)$ varies with S, the effective size of the cache, under LRU. We ran the three workloads for

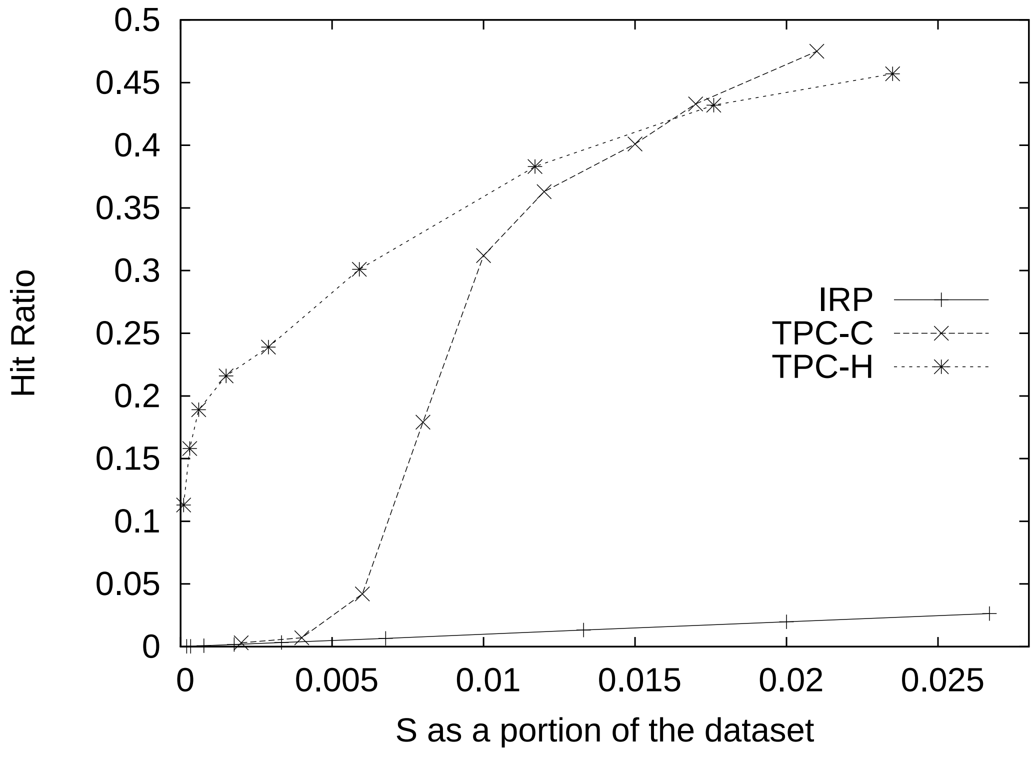


Figure 5.6: $H(Y,W,S)$ as a function of S.

different page cache sizes; we report the hit ratio in Figure 5.6 (the raw data are given in Figure 8.24). The x-axis is S shown as a percentage of the size of the whole dataset. In all cases the hit ratio grows with the size of the cache, as discussed in Section 5.3.4. The growth rate varies widely with the workload: it is linear for IRP and non-linear for TPC-C and TPC-H. This is expected since both TPC-C and TPC-H have a working set, albeit of different size, while IRP does not. But the most important observation is that, apart from H growing with S, one cannot make assumptions that hold for all workloads. Not that, for $S > 0.025$, the curves slope upwards monotonically and therefore the hit ratio for this interval is not shown here.

5.5.2 Impact of Flash Cache Size on RAM Hit Ratio

In our system, the directory for the FLASH cache is stored in main memory. As discussed in Section 5.4, as the size of the flash cache grows, the available main memory for the RAM cache shrinks and therefore the RAM hit ratio is expected to drop. We measured this effect by growing the size of the FLASH cache (and thus the FLASH directory) while keeping the size of the RAM cache fixed, and measuring the RAM hit ratio H. In addition, we ran the same workloads with no FLASH cache (and thus all main memory

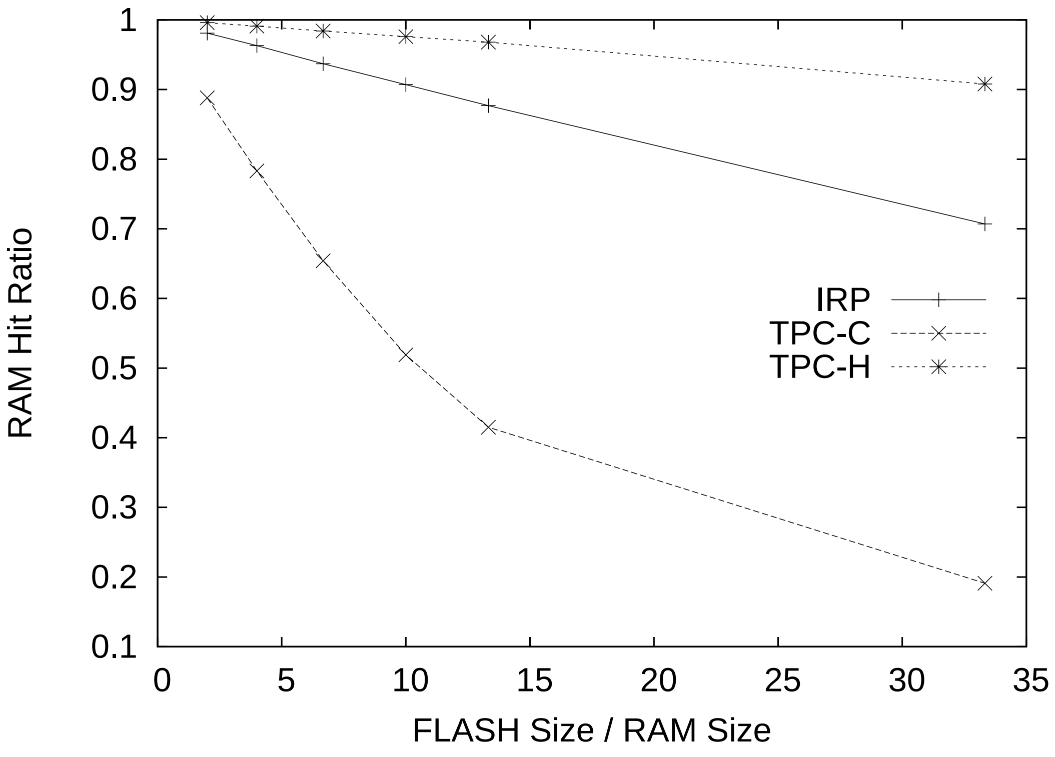


Figure 5.7: $\frac{H}{H'}$ for different sizes of the FLASH cache.

available for the RAM cache) and measured the RAM hit ratio H'. In Figure 5.7 we report $\frac{H}{H'}$ for different sizes of FLASH cache (the raw data are given in Figure 8.25). As evident, there is a drop in the hit ratio for all workloads. This drop is linear for IRP as it has no working set, and for TPC-H as its working set fits in RAM in all cases. For TPC-C the working set fits in main memory for small FLASH sizes, but for larger ones it does not; thus, the hit ratio drops very quickly, giving a curve that is the inverse of the curve of Figure 5.6 for TPC-C. In all cases, the main memory occupied by the FLASH index has a big impact on the RAM hit ratio.

5.5.3 Validation of the Cost Formulae

We then went on to empirically verify the validity of the cost formulae of Section 5.3. We executed a synthetic IRP workload using the Samsung disk and measured the running time for each page flow scheme. We also used the cost formulae of Section 5.3 with the I/O cost metrics for the particular flash disk and HDD to calculate the total cost of each scheme. We plot the ratio of the execution time for each physical run and the cost projected by the formulae *for that scheme* in Figure 5.8 (the raw data are given in Figure 8.26). Observe that the ratio of the real cost over the projected one remains

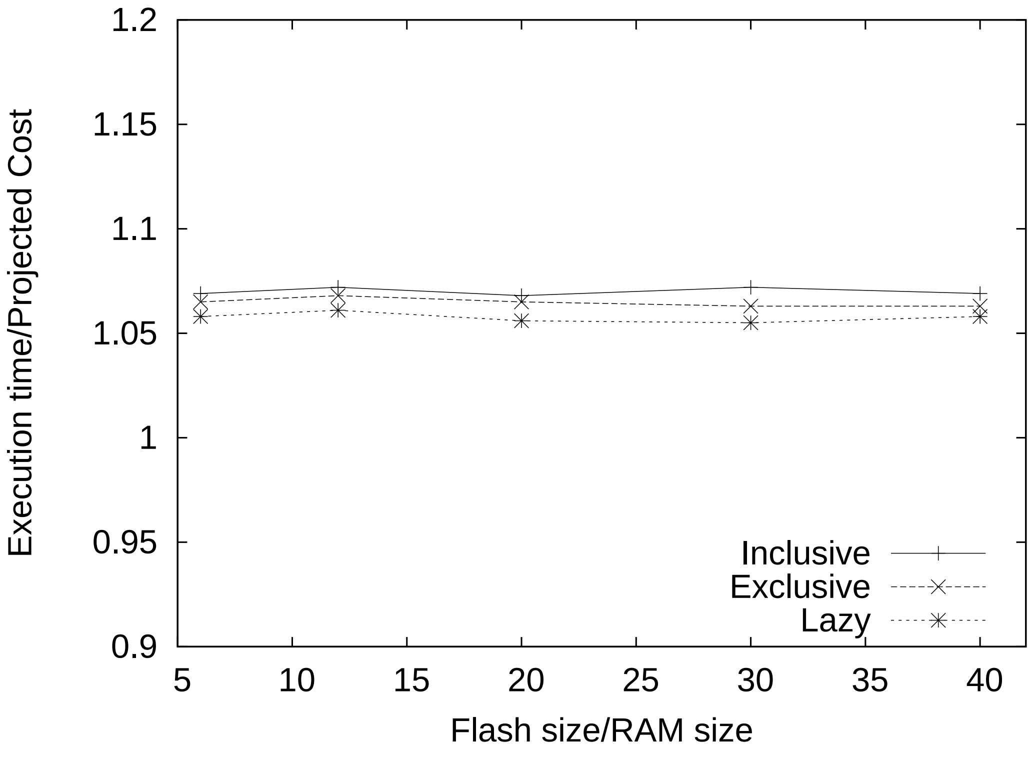


Figure 5.8: Validation of the cost formulae.

constant for all FLASH cache sizes. Also, this ratio remains the same across page flow schemes, indicating the consistency of the cost formulae. The ratio being 6% to 8% higher than 1 is due to our cost formulae not taking into account the warm-up time for the caches. Our formulae assume that each RAM miss results in a RAM eviction (and thus either a FLASH hit or a FLASH miss), which does not hold until after the RAM cache becomes full. The same applies for the warm-up time of the FLASH cache. Although one can adapt the formulae to account for this cost, as one can approximate after how many references each cache becomes full, we chose not to do so in the interest of simplicity; moreover, this cost is negligible for workloads of interest. Another caveat arises when using our formulae for very small datasets, in which the on-disk caches of the FLASH disk and the HDD affect the disk read/write costs. For all real-world workloads, however, our formulae were quite accurate in their cost estimation. As argued in [Bouganim et al., 2009], not all flash writes incur the same cost, just as is the case for magnetic disk writes. In this work we are interested in the cost of all writes throughout the workload, *i.e.*, in the average cost of random writes for blocks of specific size. This cost can be accurately approximated using the techniques of Section 4.6.

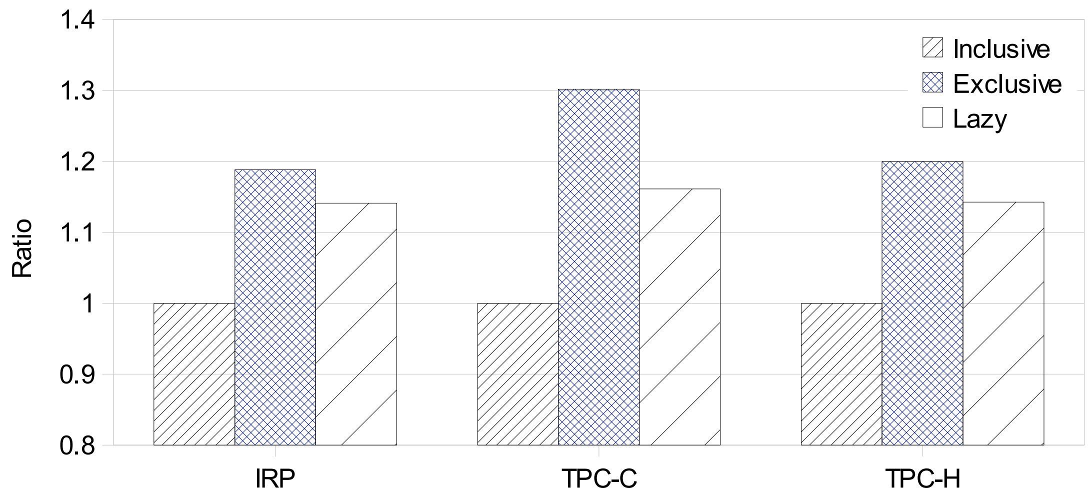


Figure 5.9: Flash hit ratios per scheme.

5.5.4 Comparison of Page Flow Schemes

We compare the three page flow schemes across the different workloads.

5.5.4.1 Flash Hit Ratio

We first measured the FLASH hit ratio for each scheme and workload. We ran the experiments for different RAM and FLASH sizes obtaining similar results; to avoid repetition, we only report in Figure 5.9 the results for the FLASH cache being 6 times the size of the RAM cache (the raw data are given in Figure 8.27). All hit ratios for each workload are normalised by the hit ratio of inclusive. As explained in Section 5.3.4, exclusive has the highest hit ratio for all workloads, while inclusive has the lowest. The hit ratio for lazy varies between the two. However, as we will see in the sequence, the highest hit ratio for exclusive does not always result in a lower I/O cost.

5.5.4.2 Total I/O cost

We ran TPC-H and TPC-C for a varying FLASH size and a fixed RAM size. We plotted the total I/O cost as calculated using the formulae of Section 5.3 for different flash disks. In the first experiment, we ran TPC-H with the FLASH cache size varying from 5 to 40 times the size of the RAM cache. The projected I/O cost of the FusionIO ioDrive is shown in Figure 5.10 (the raw data are given in Figure 8.28). In this case, the exclusive scheme performs better than the other two for all FLASH cache sizes; the following experiments will reveal that this is not always the case. Observe also that increasing

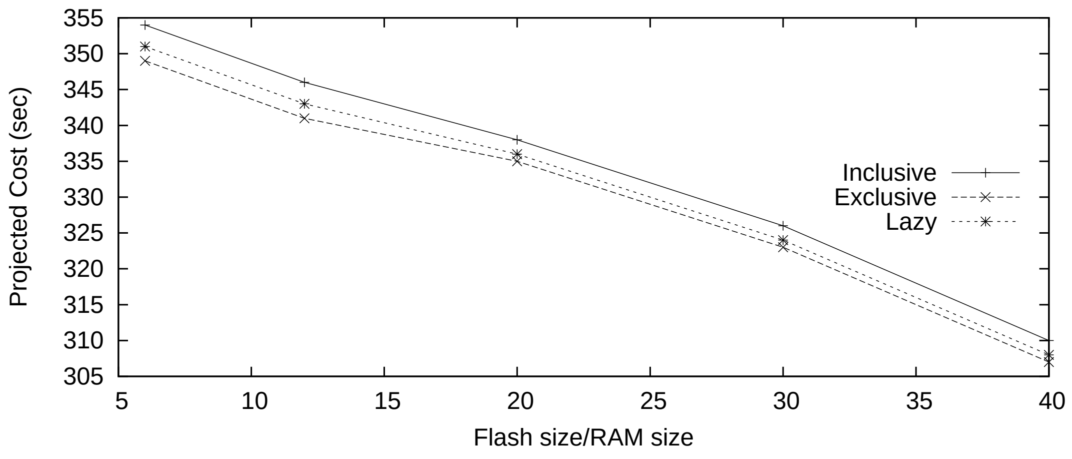


Figure 5.10: Total cost for TPC-H with Fusion ioDrive.

the size of the FLASH cache can benefit performance by a significant factor.

We then ran TPC-C for the same FLASH cache sizes as before and calculated the total cost based on the cost metrics of the Samsung flash disk; the results are shown in Figure 5.11(a) (the raw data are given in Figure 8.29). The exclusive algorithm is totally unsuitable in this case due to the disproportionally high write cost of the Samsung disk (as for each RAM eviction exclusive pays the cost of a flash write). As for inclusive and lazy, observe that while their cost is similar for large FLASH sizes, there is a performance gap for small FLASH sizes (or, big RAM sizes).

We repeated the cost calculations for TPC-C, but used the I/O costs of the Intel X25-E flash disk; the results are shown in Figure 5.11(b) (the raw data are given in Figure 8.30). When the FLASH cache is less than 15 times the size of the RAM cache, exclusive is the most efficient page flow scheme with a total I/O cost that is up to 30% lower than the I/O cost of inclusive and 14% lower than the I/O cost of lazy. On the other hand, for a FLASH cache size more than 35 times that of the RAM cache, lazy is the optimal scheme with an I/O cost that is 16% lower than the I/O cost of exclusive. Therefore, even for the same FLASH disk and workload the optimal scheme changes with the ratio of the FLASH cache size over the RAM cache size.

Next, we kept the FLASH and RAM cache sizes fixed and ran the TPC-C workload under each scheme and calculated the total I/O cost for all four disks. The results are shown in Figure 5.12 (the raw data are given in Figure 8.31); the optimal algorithm differs for each disk. Note that lazy is optimal for the two MLC disks (Samsung and Intel X25-M), while exclusive is optimal for higher-performance SLC devices (Intel X25-E and FusionIO). The graph confirms our hypothesis that no scheme is universally

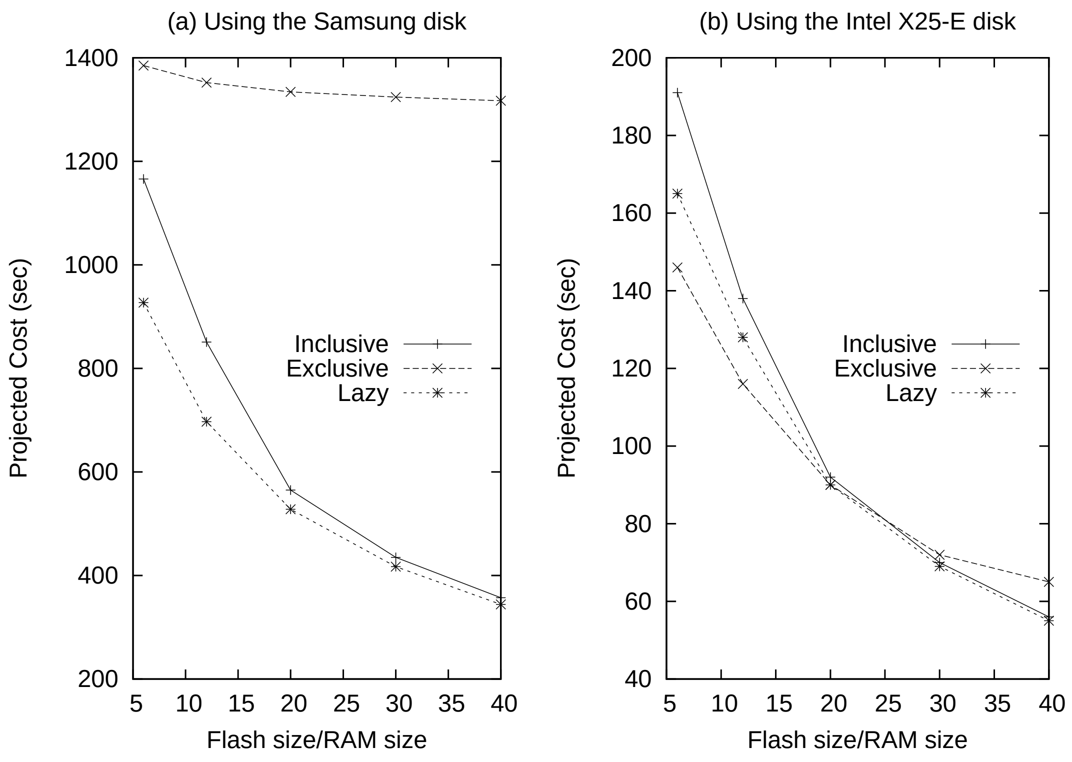


Figure 5.11: Cost for TPC-C under for each scheme.

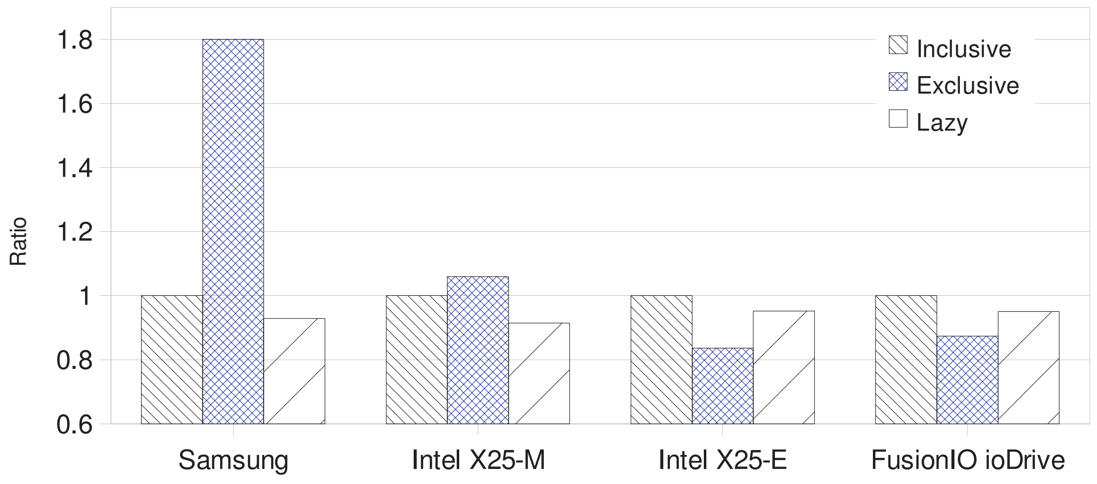
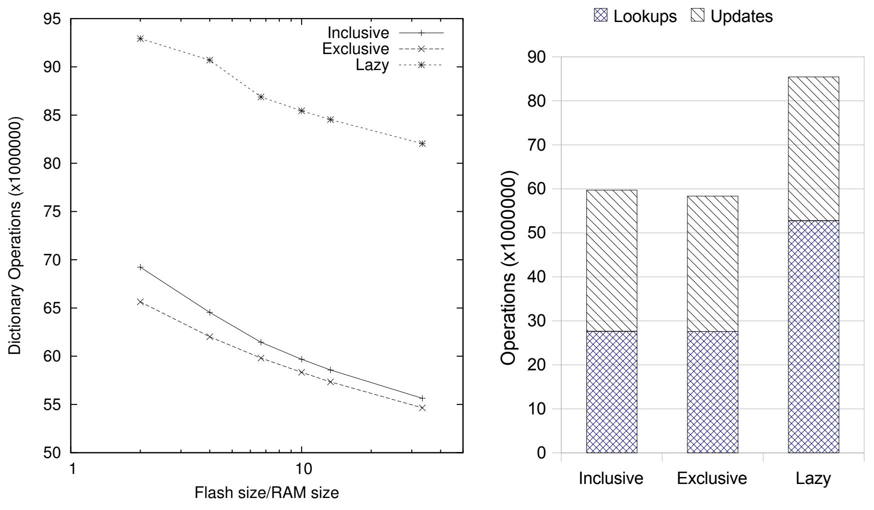


Figure 5.12: Total cost for TPC-C per flash disk.

optimal across all workloads and flash disks. It also appears that inclusive is never the best performer; however, this is not the case if directory maintenance costs are accounted for as well.



(a) Total operations (b) Cost breakdown

Figure 5.13: Directory operations for TPC-C

5.5.4.3 Directory Operations

We have so far assumed that the page directory for the FLASH cache resides in main memory. Hence, in our cost calculations we have not included directory maintenance costs. In Figure 5.13(a) the total number of directory operations, *i.e.*, lookups and updates is shown, for each page flow scheme for the TPC-C workload and for different sizes of FLASH cache (the raw data are given in Figure 8.32). In Figure 5.13(b) we break down the operations into lookups and updates for the FLASH size being 10 times the size of RAM (the raw data are given in Figure 8.33); in all other cases the results were similar and thus not reported here. The exclusive scheme is the most efficient, as it requires fewer directory operations than both lazy and inclusive. For all schemes, as the FLASH cache size grows, so does the FLASH hit ratio and, thus, the number of directory operations required for each scheme drops: a FLASH hit requires fewer operations than a FLASH miss. As shown in Figure 5.13(b), inclusive and exclusive incur the same number of lookups, while inclusive incurs one more update for each FLASH page dirtied in main memory. The lazy scheme requires more lookups, as each RAM eviction requires a FLASH lookup. If the FLASH directory was kept on the flash disk, directory maintenance operations would affect the total I/O cost and should be considered in the cost formulae. The general conclusion is that, similarly to the I/O cost, the cost of directory maintenance varies widely across different workloads, schemes, and flash disks; for most cases, the lazy scheme incurs the most directory operations and is thus the least suitable for the case of a flash-resident directory.

5.5.5 Impact of Flash Block Size

We then investigated how the size of the FLASH block affects performance. In all cases the RAM page size was set to 4kB. We first used a flash block size varying from 4kB to 128kB. For each block size we ran the TPC-H workload and measured the hit ratio for FLASH and the total number of HDD reads, using the overwriting technique of Section 5.4.1: if a page is evicted from RAM and its corresponding block is not on FLASH, then the whole block is brought from HDD to FLASH. We ran TPC-H using the inclusive and lazy schemes. In Figure 5.14 we show for each scheme the FLASH hit ratio (top graph) and the number of HDD reads (bottom graph). The raw data are given in Figure 8.34. Under inclusive, before a page is brought into RAM its flash block is written to the FLASH cache. Subsequent accesses to the pages of that block will be served from FLASH. Thus, the hit ratio for inclusive increases as the block size grows and overwriting acts as a prefetching mechanism, greatly affected by locality of reference. Under lazy, a block is written to flash when any RAM page that belongs to that block is evicted from RAM for the first time. Even for workloads with a high degree of locality, pages of the same flash block will have most likely been read into RAM before one of them is evicted to FLASH. Therefore, locality does not affect lazy as much (at least for small block sizes). As the block size grows, so does the granularity at which the replacement policy tracks the reference pattern through access recency (or frequency). Therefore, the hit ratio drops (for inclusive this effect is cancelled by the effect of prefetching). As shown in the bottom graph lazy performs about twice as many HDD reads as inclusive. This is not only due to its lower hit ratio: when a RAM victim is written to FLASH, the block it belongs to needs to be read from HDD if it is not cached on FLASH. Conversely, for inclusive, the first invariant guarantees that the block the page belongs to is on FLASH.

Next, we experimented with both overwriting and invalidating to gauge their performance under both TPC-C and TPC-H as we varied the flash block size from 4kB to 128kB; we used the lazy scheme in all cases. Let $h_{\mathsf{overwriting}}$ and $h_{\mathsf{invalidating}}$ be the hit ratios for overwriting and invalidating, respectively. In the top graph of Figure 5.15 we report the ratio $\lambda = \frac{h_{\mathsf{overwriting}}}{h_{\mathsf{invalidating}}}$ for the two workloads; in the bottom graph we show the corresponding ratio of HDD reads (the raw data are given in Figure 8.35). For both workloads, overwriting had a higher hit ratio than invalidating, due to the prefetching effect described earlier. This effect was more evident in TPC-H as it exhibits a higher degree of locality than TPC-C. As explained earlier, under the lazy scheme the hit ratio

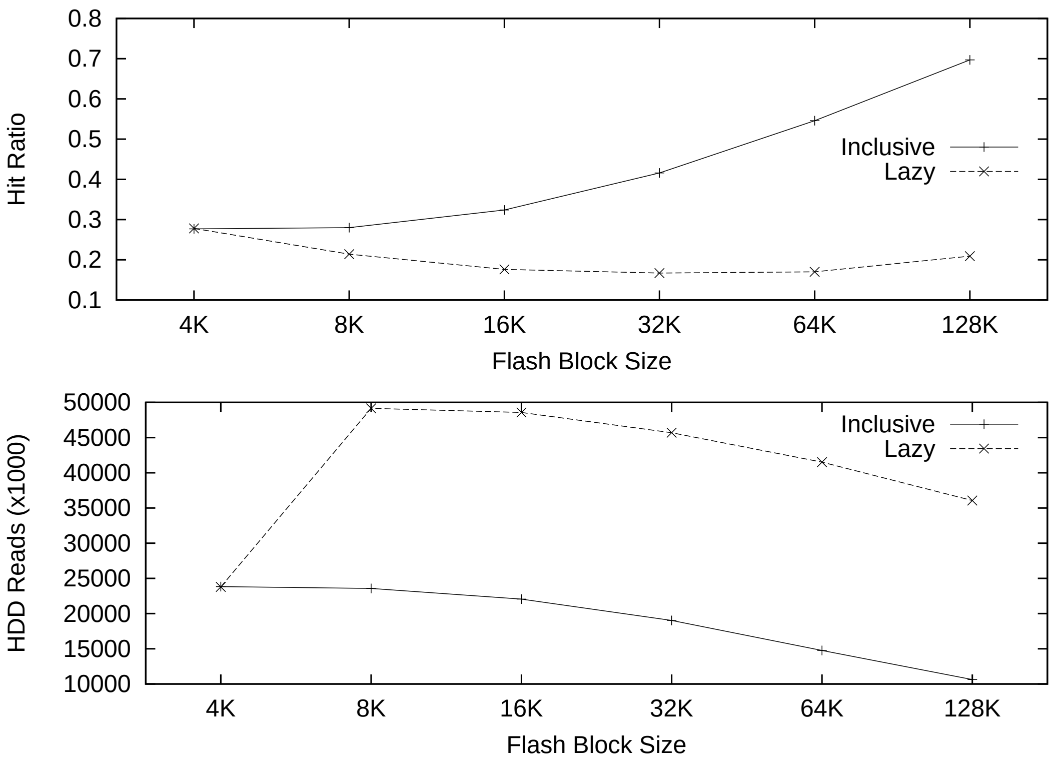


Figure 5.14: Impact of block size under overwriting.

for both overwriting and invalidating was less than the hit ratio for more fine grained replacement (*i.e.*, for 4kB blocks). For HDD operations, overwriting resulted in more HDD reads than invalidating for both workloads (ranging from twice to 1.3 times as many HDD reads). For all workloads with locality of reference, overwriting is expected to give a higher hit ratio than invalidating at the cost of extra HDD read operations. The optimal choice depends on the read efficiency of the flash disk and the HDD.

5.5.6 Caching Only Clean Pages

We evaluated the effect of caching dirty pages in the FLASH cache. Recall from Section 5.3 that, for the lazy scheme, one may apply any criterion to decide if a RAM victim page will be cached on FLASH or not. Dirty pages cached on FLASH are more likely to cause updates on the flash disk. Thus, if the flash disk is not efficient at random writes, it makes sense to restrict FLASH caching to clean pages only. Other criteria may be used as well, *e.g.*, the frequency of writes on a page. We used IRP workloads with different dirtiness ratios, *i.e.*, the probability of a page being dirtied on each next reference. Each workload was executed using the lazy scheme twice: once caching all RAM victims on FLASH and once caching only the clean ones. We used the Samsung flash

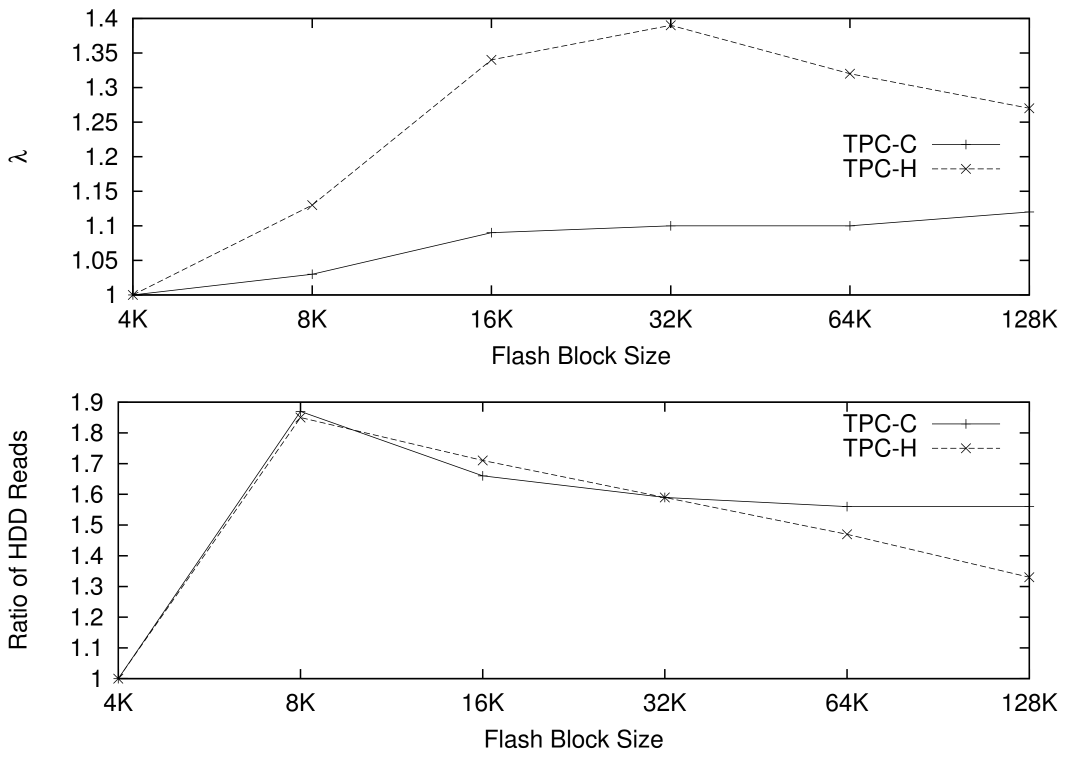


Figure 5.15: Comparison of overwriting and invalidating.

disk (which is inefficient at random writes) and measured the hit ratio for the FLASH cache and the total execution time for varying dirtiness ratios. Hit ratios are shown in the left graph of Figure 5.16 and execution times are shown in the right graph (the raw data are given in Figure 8.36). The hit ratio drops when only clean pages are cached, as some of the hot dirty pages are evicted to HDD. For small dirtiness ratios, the drop in the hit ratio is gradual, as the hottest of the dirty pages fit in RAM. For a dirtiness ratio greater than 0.7, the hit ratio drops substantially. On the other hand, the execution time is much less when caching only the clean pages, due to the write inefficiency of the flash disk we used. Observe that for dirtiness ratios between 0.1 and 0.7 the running time remains the same when caching only clean pages, *i.e.*, the increased miss ratio is counterbalanced by the time saved by avoiding flash writes. For greater dirtiness ratios the hit ratio drop results in about a 10% increase in execution time.

5.6 Discussion

Choosing the optimal scheme. Our evaluation shows that the I/O cost of a workload depends heavily on: (*a*) the workload itself, (*b*) the page flow scheme, and (*c*) the I/O

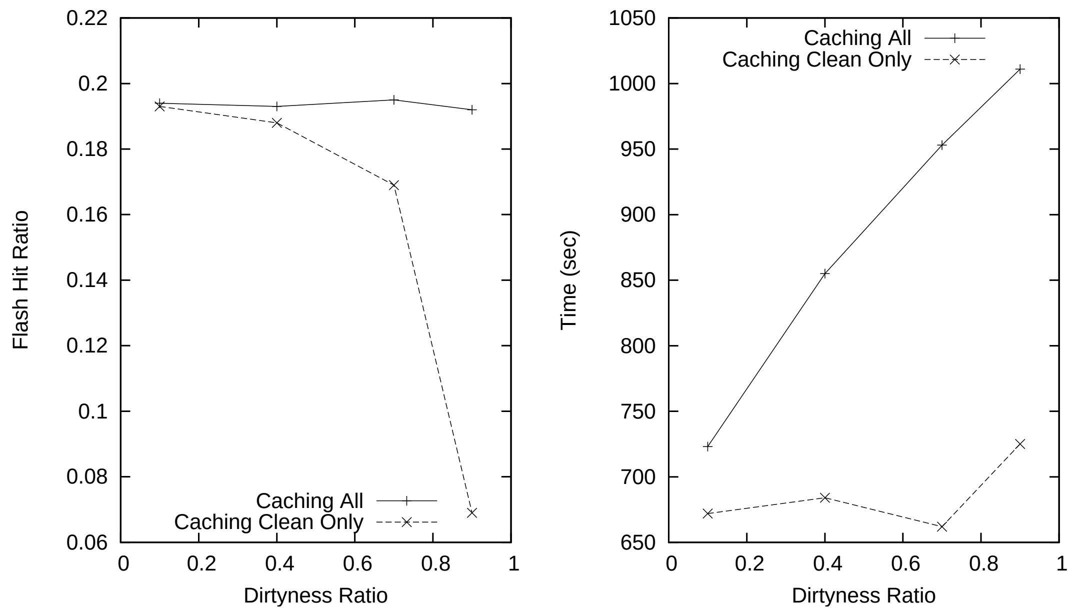


Figure 5.16: Effect of caching only clean pages.

costs of the flash disk. Thus, one cannot decide with confidence the optimal scheme *a priori* without evaluating our cost formulae. Given the workload and the flash disk, we can hypothesise about the optimal scheme. For instance, exclusive performs one flash write for each RAM miss, whether the victim page is dirty or not; inclusive and lazy do so only for dirty pages. Hence, for write-intensive workloads, exclusive is likely to be more expensive, more so if the flash disk is not write-efficient. Then, multiple flash writes can be avoided if only clean pages are cached on FLASH, (see Section 5.5). If the RAM cache is not a small percentage of the FLASH cache, exclusive is likely the best option: no page will be cached on both caches, saving space on FLASH. Using similar arguments it is sometimes possible to make the optimal choice if the characteristics of the workload are well-known.

The experimental results also verify our hypothesis that hit ratios alone cannot fully describe the system's I/O efficiency. For instance, as shown in Figures 5.9 and 5.12, although exclusive has the highest hit ratio for TPC-C, it is not the optimal scheme across all flash disks *w.r.t.* to the total I/O cost. This holds for all workloads we have tested. Moreover, as shown in Figure 5.11(b), even for a specific workload and flash disk, the optimal scheme changes for different FLASH or RAM cache sizes. Note also that inclusive appears to always be less I/O-efficient than lazy, if the same page size is used for RAM and FLASH. However, this changes radically for different page sizes, or directory maintenance costs (*i.e.*, when the FLASH directory is stored on the flash disk).

Flash writes. We have so far assumed that pages are written to the flash disk in a random fashion. In [Narayanan et al., 2009], the flash disk is split in a write cache and a read cache, with the write cache employing a log-structured filesystem to speed up random writes. Such techniques are complementary to ours, especially for flash disks with poor random write performance; for high-end ones random writes are as efficient as sequential ones, *i.e.*, they are converted to sequential ones by the disk controller. In [Leventhal, 2008], the authors use large asynchronous sequential writes instead of synchronous random ones. Flash writes can be asynchronous in our case too: pages to be cached on flash are only marked as such in main memory and moved to flash asynchronously. Such writes can also be performed sequentially in large chunks, as in ZFS; then, one would need to adjust the write costs. This is analogous to using a larger flash block, as described in Section 5.4.1. In our case, large flash blocks are fetched from disk when their first page is accessed (*e.g.*, for inclusive), effectively prefetching all other pages of the block. ZFS only writes to flash pages from main memory (evictees) and thus this effect is absent. As shown in Section 5.5.5, prefetching can greatly enhance performance – at least for database workloads. For high-performance flash disks with high random write throughput and low latency, *e.g.*, the FusionIO ioDrive, large sequential writes will not make much difference: sequential and random writes have almost the same throughput. Additionally, such devices have comparable read/write latencies. Thus, flash writes can be synchronous at RAM eviction time, without bogging the system down. Our work focuses on deciding the size and the contents of the RAM and FLASH caches, while [Leventhal, 2008] mainly focuses on implementation efficiency. Therefore, we believe that our approach is complementary to the ZFS approach. That said, we feel that a system should primarily decide on the contents of the flash cache, and secondarily on implementation principles.

Remarks. Techniques that speed up random writes on flash disks, *e.g.*, In-Page Logging [Lee and Moon, 2007], are complementary to our work and important for caching efficiency. An interesting question is what on-flash data structure or filesystem serves caching needs best, especially if the page directory and the replacement algorithm are external memory ones. The answer depends on how pages are written to flash and how they are replaced upon eviction; we do not study this further here. As for metadata persistency, standard techniques (*e.g.*, write-ahead logging) can be employed. Given the non-volatility of flash memory, one can do the same for the flash page directory. That way, after a system failure, the flash cache will warm up instantly, speeding up system recovery.

Chapter 6

Sorting Hierarchical Data In External Memory

6.1 Introduction

In this chapter we study the problem of external sorting. Sorting has always been important in data management. Its usefulness is even greater for database systems as sorting plays a significant role in a number of key query processing algorithms, including join evaluation, duplicate elimination, and aggregation, to name a few. Previous studies, like the one presented in [Lee et al., 2008], have found that algorithms like external mergesort generate access patterns that are dominated by large sequential writes and random reads. As discussed in Chapter 1 and Chapter 2, such access patterns are most favourable for flash memory and can exploit the full potential of the new medium. The vast majority of existing algorithms, however, focuses on flat datasets. Apart from evaluating the performance of sorting algorithms over flash memory, it is our goal to explore how existing algorithms can be generalised to operate on hierarchical data. Our proposals are oblivious to the storage medium, as we aim to give a generic solution to the problem of sorting hierarchical data, such as XML datasets. We evaluate the proposed techniques over both flash and magnetic disks; we also experimentally explore data staging in the context of sorting, for a hybrid system equipped with both types of storage media.

The problem of sorting hierarchical data has, surprisingly enough, received little attention from the research community. This is due to the relational data model being inherently flat. The need for sorting hierarchical data has re-emerged in the context of managing scientific data archives, which tend to be largely hierarchical, complicated in structure, and quite voluminous. In this chapter we present the *Hierarchical ExteRnal MErgeSort* (HERMES) algorithm for sorting hierarchical data in external memory. The algorithm takes into account the hierarchical structure and by exploiting it, it is able to efficiently sort large datasets while minimising disk I/O and, at the same time, using a minimal amount of main memory.

Archiving scientific data. A side-goal of this work is managing scientific data for archiving purposes. Scientific data sources on the Web play a major role for ongoing research efforts. Annotated protein databases like UniProt [Consortium, 2007], or sequence databases like EMBL [European Bioinformatics Institute, 2007], are the primary sources of information in, *e.g.*, selecting targets for conducting biological experiments, or in pharmaceutical research. As is the case in any kind of research, reproducibility of results is of paramount importance. Problems arise due to the dynamic nature of scientific databases: they continuously change as new results become available. Pitfalls include the identification of erroneous entries in a database, and therefore their modification, which results in invalidating scientific results that have used the erroneous entries as input. In addition, as research progresses, more accurate results are generated through improved experimental methods. It is common practice for scientific database providers to overwrite existing database states when changes occur and publish new releases of the data on the Web on a regular basis. Failure to archive earlier states of the data may lead to loss of scientific evidence, as the basis of findings may no longer be verifiable.

Scientific data is predominantly kept in well-organised hierarchical data formats. To support versioning, in [Buneman et al., 2004] the authors propose an archiving approach that efficiently stores multiple versions of hierarchical data in a compact archive. Version numbers denote time and become a first-class citizen of the process: time is added as an extra attribute to the data being archived. To generate a new version of the archive the authors propose the *nested merge* operator: multiple versions are *merged* on the time attribute, with the archiver storing each element only once in the merged hierarchy to reduce storage overhead. An archived element is annotated with timestamps representing the sequence of version numbers in which the element appears. By merging elements into a single data structure the archiver is able to retrieve any version from the archive in a single pass over the data.

In Figure 6.1 we see an archive A_{1-2} containing two versions of data and an incoming version V_3. For ease of presentation, we assume that nodes are compared on their values. Nodes in the archive are annotated with their version number (denoted by

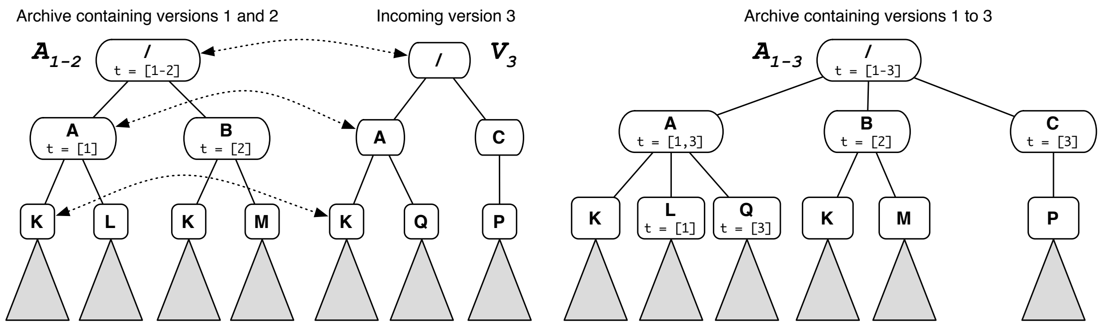


Figure 6.1: Merging an incoming version into an existing archive.

t in the figure); version numbers act as timestamps representing the points in time that a node is present in the archive. Nodes without a timestamp are assumed to inherit the timestamp of their parent. Corresponding elements are connected by dotted edges.

Starting from the root, corresponding nodes in V_3 and in A_{1-2} are merged recursively. When a node y from V_3 is merged with a node x from A_{1-2}, the timestamp of x is augmented with the new version number (*e.g.*, the root of the archive and node A). The subtrees of nodes x and y are then recursively merged by identifying correspondences between their children. Nodes in V_3 that do not have a corresponding node in A_{1-2} are added to A_{1-3} with the new version number as their timestamp (*e.g.*, node Q). Nodes in A_{1-2} that no longer exist in the current version V_3 have their timestamp terminated, *i.e.*, these nodes do not contain the new version number (*e.g.*, node B). The process is repeated for all levels.

With serialised hierarchical data formats, like XML, one usually traverses the data depth-first. The problem with nested merge as described in [Buneman et al., 2004] is that it does not manifest this natural access pattern. To identify correspondences between children of merged nodes, one must process complete subtrees. Thus, numerous passes over the data may be required. If, however, the nodes of the datasets are ordered on their keys the situation greatly improves. Assuming an ascending order, as shown in Figure 6.1, whenever two nodes x and y are to be merged, one can sequentially scan the children and compare their key values. The child with the smaller value is output to the new archive, after having been annotated with the proper timestamp. This ensures a total ordering among all children of any node. This process ensures that the archive is always sorted on node keys. More importantly, only the incoming version has to be sorted before nested merge. As the new version may be comparable in size to the archive, sorting is a salient operation.

Change detection. Apart from archiving, sorting different versions of hierarchical datasets enables efficient change detection. This is useful to systems that support incremental query evaluation and trigger condition evaluation. Existing approaches to change detection in XML documents (*e.g.*, [Chawathe et al., 1996, Cobena et al., 2002, Wang et al., 2003]) operate on unsorted documents and only work in main memory. However, an algorithm similar to *nested merge* can be employed to efficiently spot differences. With the input documents sorted, change detection can be supported for datasets larger than the size of main memory.

Sorting hierarchical data. The complexity of sorting large hierarchical datasets is shown to be below that of sorting flat data [Silberstein and Yang, 2004]. This is due to the smaller number of possible sorting outcomes, as in any sorted result the initial parent-child relationships from the original data have to be retained. For example, there is no need to compare nodes in different subtrees of the dataset, or located at different levels of the hierarchy. As with any external memory algorithm, the major challenge is to reduce the overall number of I/O operations.

The common approach to sorting large datasets is external mergesort and its variations [Knuth, 1998]. External mergesort splits the dataset into multiple *runs* that are sorted in main memory during a single pass over the data. Runs are then merged to generate the sorted output. Sorting hierarchical data is, however, not straightforward; the hierarchical structure has to be retained and each sorted run has to represent a proper hierarchy itself. An obvious approach would be to "*flatten*" the data by writing the complete set of root to leaf paths to a file and then sorting the entries in the file using standard external mergesort. As shown in [Silberstein and Yang, 2004], this approach does not exploit the hierarchical structure and is inefficient in terms of memory, storage space and processing power. Bottom-up approaches like the one of [Silberstein and Yang, 2004] for sorting hierarchical data, on the other hand, operate by splitting the input in complete subtrees that are sortable in main memory. These subtrees are stored as sorted runs in separate files. Once the children of each node are sorted, the data is output by reading the sorted subtrees from the run files. This employs a random access pattern: though each run will be sequentially scanned, entire runs will be read in a different order than the one they were generated. Such approaches do not perform well on the high-branching, wide-spread structure of scientific datasets. As an example, the current release of the EMBL Nucleotide Sequence Database [European Bioinformatics Institute, 2007] (Release 93, December 2007) has over 100 million entries below a single root node. The average size of each entry is four kilobytes. Therefore, during output, a large number of small files will have to be accessed in random order, which penalises I/O performance if a magnetic disk is used for storage. In the following we assume a magnetic disk as the storage medium. We evaluate our sorting algorithm over a flash disk in Section 6.5.5.

6.2 Related Work

Sorting is a fundamental computing problem and as such it has received considerable attention. Departing from internal memory implementations, the basis of most external memory sorting algorithms over flat, record-based datasets is external mergesort. Various extensions have been proposed over time, with [Knuth, 1998] presenting an extensive study of most external sorting techniques, while [Graefe, 1993] presents the details of implementing external mergesort as part of a relational database engine. There have been numerous proposals for improving the algorithm's performance, ranging from increasing its internal sorting efficiency, to enhancing its CPU utilisation, or to its parallelisation.

In [Zheng and Larson, 1996] the authors propose placing blocks from different runs in consecutive disk addresses to reduce the seek overhead during the merging phase (at the expense of additional seek cost during run creation). They also study reading strategies, like forecasting and double buffering, and propose a read planning technique. The latter uses heuristics to precompute the order in which records will be read from disk during merging. It then utilises this order to reduce seek overhead, based on knowledge of the physical location of the blocks on the medium. These improvements can be almost verbatim applied to our algorithm, provided they are adapted to hierarchical data (see Section 6.4.3 for a discussion on how this can be achieved).

In [Nyberg et al., 1995] the authors present AlphaSort, a cache-sensitive, memory-intensive external sort algorithm. AlphaSort groups the records as they arrive from disk and employs quicksort, due to its cache locality, as the main memory sorting algorithm for the creation of the initial runs. In addition, AlphaSort sorts $(key - prefix, pointer)$ pairs, rather than the records themselves, in order to reduce in-memory data movement and employs replacement selection for merging runs, which is ideal for memory constrained environments and has excellent cache behaviour when the merge tree is small. Borrowing concepts from AlphaSort is something that can help HErMeS achieve better performance whenever the length of keys (which, in the case of hierarchical data can become arbitrarily long) exceed the size of a cache lines.

Moving on to strictly hierarchical data models, the most wide-spread one is XML. It was used as the serialisation protocol for archiving scientific data and the definition of the nested merge operation [Buneman et al., 2004], which provided the motivation for the development of HERMES. Similar concepts were provided in [Tufte and Maier, 2001] and [Wanxia Wei and Mengchi Liu and Shijun Li, 2004] where the semantics of generalised XML tree merging were defined (albeit in different ways). Regardless of the exact semantics, efficient merging implementations depend on having their inputs sorted, and therefore our proposal is immediately applicable. Furthermore, XML query languages like XPath [Berglund, 2007] and XQuery [Boag, 2007] provide an *order by* clause that may be used in conjunction with a DTD to completely sort XML documents. However, the specification does not mention any particular implementation. We believe that our algorithm is one such possible implementation to be used by XML query engines.

The XML Toolkit (XMLTK) provides a tool named XSort for sorting XML documents [Avila-Campillo *et al.*, 2002]. XSort allows the specification of the context nodes the subtrees of which should be sorted. For each context node multiple XPath expressions identify the actual elements to be sorted. Only user-specified elements are sorted and the subtrees of these elements are not sorted recursively. Sorting proceeds by generating a global key for each element to be sorted. It then uses a standard external mergesort algorithm to sort elements based on the value of this global key. XSort does not exploit the hierarchical structure of the data. Indeed, it might not be possible to sort the entire document without making multiple calls to XSort. By collapsing hierarchical data to their flat counterparts, the hierarchy reconstruction step is left to the user. Our algorithm does not impose such restrictions.

The most relevant piece of work we are aware of, and the state-of-the-art in sorting XML datasets, is NEXSORT [Silberstein and Yang, 2004]. The NEXSORT algorithm takes into account the properties of hierarchical datasets and consists of two phases: sorting and output generation. During the sorting phase, NEXSORT scans the input document depth-first, detects complete subtrees, and decides, based on a user-given threshold, whether to sort these subtrees in main memory or not. Only subtrees of size no less than the specified threshold are sorted and stored on disk as a sorted run. Sorted subtrees are replaced in the tree by just their root and a pointer to the sorted run stored on disk. Conceptually, NEXSORT processes the input document bottom-up, collapsing subtrees into their roots until only the root of the entire tree remains. In the output phase, NEXSORT performs a depth-first traversal of the collapsed tree to generate the final sorted document. Generated sorted runs need to be accessed in a random fashion during the merging phase, therefore penalising I/O; even when operating over a flash disk, the secondary-storage data structures employed by NeXSort incur a substantial performance penalty. Furthermore, the choice of threshold is a critical part for the performance of NeXSort making performance dependent on the structure of the document. For documents like EMBL [European Bioinformatics Institute, 2007], where only a few subtrees are large, this approach is very inefficient. Our algorithm, by making efficient use of compression and carefully laying out runs on external memory, is able to achieve much better I/O performance, as we shall see in Section 6.5.

6.3 Sorting Hierarchical Data

We now present our algorithm: *Hierarchical ExteRnal MErgeSort* – HErMeS, an adaptation of external mergesort for hierarchical data. HErMeS runs in two phases: (*a*) first, the hierarchical document is "vertically" split into sorted runs on disk; (*b*) then, the runs are iteratively merged into greater ones until the final sorted output is generated. HErMeS extensively exploits the fact that one needs to perform key comparisons only for nodes having the same parent node (*i.e.*, siblings). Nodes belonging to different subtrees do not need to have their keys compared. This enables us to apply *local* replacement selection for every in-memory node.

6.3.1 Sort Keys

A hierarchical dataset is a tree whose nodes have an *identifier* (or label), a *type*, and an optional *value* (or payload). To sort hierarchical datasets we have to specify a sorting criterion for nodes. This criterion may include the node label, its value, a combination of the two, or a well-defined subset of the subtree rooted under that node.[1] We assume a hierarchical sort key specification (*key specification* for short) similar to [Buneman et al., 2001, Buneman et al., 2004]. The key specification K is a set of *key definitions* $k = (Q,S)$, where Q is an absolute path of node labels and S is a *sort value expression*. We assume that path Q of key definition k is unique among all elements in K. We distinguish between *keyed* and *unkeyed* nodes. Keyed nodes have a path that matches path Q of a $k \in K$. The sort expression S determines the values on which nodes having path Q are sorted. We refer to these values as *sort keys*.

[1]For sorting trees that exceed main memory, we assume that, though arbitrarily long, node keys do not exceed the size of available memory.

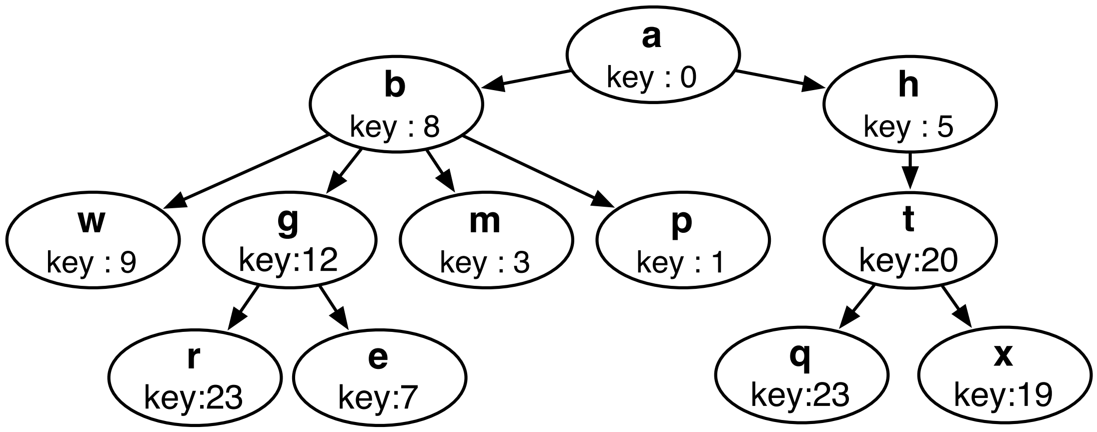


Figure 6.2: A hierarchical dateset annotated with the local key values for its nodes.

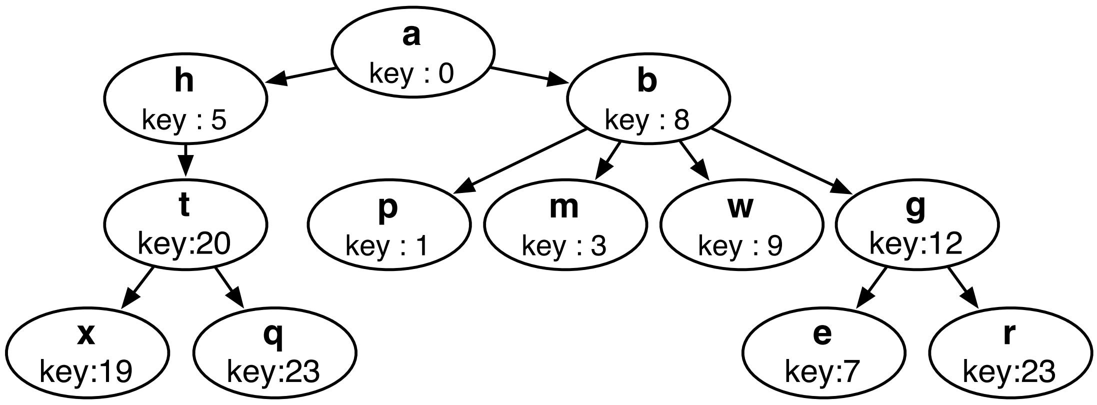


Figure 6.3: The sorted dataset of Figure 6.2.

For sorting, every node has an additional sort key attribute. Given a key specification K, we assign each keyed node its key value in a preprocessing step as described in [Buneman et al., 2004]. For unkeyed nodes, the sort key is the maximum value of the sort domain, followed by a placeholder denoting its position in the input. For each node n, let its *local key* (or simply *key*) $k(n)$ be the value of its sort key attribute. The key is the *local ordering criterion* by which we decide the rank of a node with respect to its siblings (of the same type, if different types are present). To sort the entire tree, one has to recursively sort the children of every non-leaf node, starting from the root. We assume the local key of a node to be unique among all of its siblings of the same type (we can always ensure uniqueness by appending the position or the identifier of the node to the local key). Let the *absolute* key $a(n)$ of a node be the concatenation of the local keys of all its ancestors: the concatenation of the local keys for all nodes from the root of the tree up to n. Therefore, the absolute key for a node is given by its unique path from the root if we replace each node label in the path with the corresponding local key value.

Using absolute node keys we can define the *total ordering criterion* for all nodes in the tree. Two nodes x and y are *equal iff* they have the same absolute key. In all other cases, and for any two nodes x and y, their absolute keys, $a(x)$ and $a(y)$ respectively, share a proper prefix (at the very least the value of the local key of the root node). Suppose that the common prefix of $a(x)$ and $a(y)$ is of length c. Then x is *less than y iff* $a_{c+1}(x) < a_{c+1}(y)$,[2] or $a(x)$ is of length c, where a_i denotes the i^{th} local key component of $a(x)$. Correspondingly, node x is *greater than y iff* they are not equal and x is not less than y. Naturally, any node is considered less than any of its children to preserve hierarchical relationships. The definition of local keys for unkeyed nodes means that all unkeyed nodes will follow their keyed siblings in input order in the total ordering. The children of unkeyed nodes may be keyed, in which case we need to recursively sort them. A tree is sorted if the children of all nodes are sorted on their local keys (if they are keyed).

Example 3.1: A hierarchical dataset is shown in Figure 6.2. Each node is annotated with its local key. The absolute key for node `r` is `/0/8/12/23`; for node `t`, it is `/0/5/20`. In Figure 6.3, the same dataset is shown sorted. □

Note that the local key value of a node may be a subtree. In such cases we serialise the subtree into a string and perform string comparison, when comparing keys. Also, sibling nodes may be of different type and therefore keyed on different sort value expressions. To address this, we prepend every local key with the type of the node. We then define a total order on node types to distinguish between multiple types of key and for grouping siblings of the same type in the output tree. If keys are not unique, two siblings may have the same key value even though they are different nodes. In such cases, we append the position of the node to its key. The same applies if we want to preserve the order of unkeyed nodes: we set their local key to be their position, prepended by a symbol denoting that the node is unkeyed.

In the case that complex keys are present (*i.e.*, if a node has two or more key paths), the local key of a node consists of the concatenation of all its key values (possibly prepended by their path or type) separated by a special character. For instance, if a node is keyed by the values of attributes `firstname` and `lastname` (possibly found in its payload), its local key can be recorded as `firstname:john,lastname:smith`.

[2]Here, "<" denotes an arbitrary ordering of local key values. If the key values are character strings, this is their lexicographical order; if the key values are numbers, "<" corresponds to arithmetical comparison.

When comparing two such nodes, corresponding components of the complex key can be identified and be compared. If the key of a node n has q key paths, then the value of the key value for the i-th key path is denoted as $k(n)[i]$. For two such nodes n_1 and n_2, of the same type, we define n_1 to be less than n_2, *i.e.*, $k(n_1) < k(n_2)$, when for some j with $1 \leq j < q$, $n_1[i] = n_2[i]$ for all $1 \leq i < j$ and $n_1[i+1] < n_2[i+1]$.

6.3.2 The HERMES Algorithm

During the first phase of the algorithm, we create sorted runs using a hierarchy-aware adaptation of replacement selection. Our goal is to exploit the hierarchical structure. This can reduce the number of possible sorting outcomes from $N!$ (for a flat file of N records) to $(F!)^{\lfloor (N-1)/F \rfloor} \cdot ((N-1) \mod F)!$ for a tree of N nodes and a maximum fan-out of F [Silberstein and Yang, 2004]. Sorted runs contain the keys in a compressed form (to eliminate redundancy). During the second phase, sorted runs are merged to create the sorted output.

6.3.2.1 Standard external mergesort

External mergesort uses replacement selection to create the initial runs. For flat data, replacement selection reads the input record by record and starts filling a *min* priority heap. When the heap is full, the first (and thus smallest) item is removed and written to the first run. Then, it is replaced in the heap by the next record from the input. The (new) smallest item in the heap is examined. If it has a key greater than the one just written, it is written to the current run and replaced in the heap by the next record from the input. Otherwise, the item cannot be included in the current run and is therefore marked for the next run. Marking a record implies placing it at the end of the heap and considering it greater than unmarked ones during heap comparisons. At some point, all records in the heap will have been marked for the next run. Then, the current run will be closed and the algorithm will start creating the next run. Repeating this process until the input has been exhausted yields runs that contain sorted subsets of the input. In the next phase the runs are merged using a priority heap. The priority heap is initially filled with the first item of each run and the smallest item is selected and written to the output run. Subsequently, it is replaced by the next record of the same run and the process continues. If we use one memory page for each run being read and one for the output run, it is possible that the amount of available memory is not sufficient for all runs to be simultaneously processed. In this case multiple merge levels are necessary. At each such level l, as many runs of level l as the memory can accommodate are merged into a single run of level $l+1$, until only one run is obtained at some level l_n. This run contains all the records in sorted order.

6.3.2.2 Hierarchy-aware replacement sort

For hierarchical data, one needs to sort the children of the root of the tree on their key values, and then recursively repeat this process for the root's children until the whole tree is sorted. The children of a node can be sorted, however, independently of other node keys in the tree. Thus, sorting in the traditional sense, *i.e.*, ordering a group of items on their values, only needs to be performed "locally" at a node. Our algorithm is based on employing replacement selection using a priority heap locally at a node to produce a sorted run of its children.

We use a tree serialisation protocol much like XML, *i.e.*, the tree is stored in a depth-first manner: the start and end of a node are specified with starting and ending tags and all its children lie within. All node-specific information (*i.e.*, its type, name, local key annotation (if any), and payload) follow its starting tag. The input tree is thus retrieved in depth-first fashion and for each node we take appropriate action. The output is a file that contains the tree in the same serialised format, except that children of all nodes are ordered by their key values (or by their position in the input tree, if they are not keyed).

The algorithm operates on an in-memory representation of tree nodes, termed SortNodes. A SortNode holds: (*a*) the type of the node (and possibly its position in the document), (*b*) the key of the node which can be constructed when the node is first read into memory,[3] (*c*) a pointer to the payload of the node (the payload might be text associated with the node, an attribute value, *etc.*), (*d*) an array of pointers to the SortNodes corresponding to the children of the node, which is used as a priority heap and is referred to simply as *heap* hereafter, and (*e*) one bit called *read state bit*. The read state bit is set to 0 when the start of the node is read and is set to 1 when the end of the node has been encountered (*i.e.*, when the whole subtree rooted at the node has been fully read). The payload of a node can be reached using the pointer mentioned above, but plays no role during sorting. For this reason it is copied elsewhere in memory; further discussion of payloads is deferred to Section 6.4.2. A typical SortNode is shown in Figure 6.4.

[3] If the key of a node is the value of one of its descendants, we assume the node has been annotated with this value in a previous annotation step, as in [Buneman et al., 2004], so that its key can be found locally.

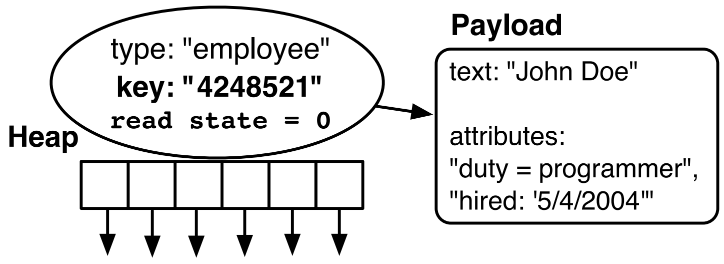


Figure 6.4: An example of a SortNode

Sorting. The sorting phase of our algorithm for the creation of the initial runs is shown in Figure 6.5. The input is read tag by tag. Stack *inputPath* holds the SortNodes of all ancestors of the last read node (initially it contains the SortNode for the root). Stack *outputPath* holds the SortNodes for the ancestors of the last output node. Traversal of the input starts at the root node. When the start tag of a node is encountered, a SortNode is created in main memory. If the available memory is full, one or more nodes need to be written to the current run to make room for the newly read node (lines 7-11). The nodes to be output are the nodes that form the subtree rooted at a node that (*a*) has the least absolute key among all subtrees in the tree, and (*b*) has been completely read from the input. The root of this subtree is located by findLeastKey and output by outputSubtree; both procedures will be explained later on. Nodes are output to the current run until enough memory has been freed for the new node.

When enough memory becomes available (lines 12-16), we identify the local key for the node and create a SortNode for it (with its read state bit set to 0). The new SortNode is inserted as a child to its parent's SortNode: it is pointed to by an element of the heap array of its parent's SortNode. This is performed by procedure insertNode of Figure 6.5. If the first place in the heap of *inputPath*.top() is free, then a pointer to the new SortNode is inserted there; otherwise the pointer is inserted at the end of the heap. If the first place of the array is free, it means that the array had the heap property at some point in time, after which its first element was removed (*i.e.*, it was output to a run). Thus, placing the new SortNode in the first place of the array enables us to maintain the heap property with a single call to heapify. We use the terminology of [Cormen *et al.*, 2001] with respect to heap operations: buildHeap constructs a heap from an array, while heapify (0) maintains the heap property of the array when its first element is removed and substituted by a new one. Once the SortNode is connected to its parent, it is pushed to the top of *inputPath*, so that the top of *inputPath* always holds the SortNode for the currently processed node. When the end of a node is encountered, the top of *inputPath* corresponds to its SortNode. The SortNode's read state bit is set to 1, marking that the subtree rooted at the node has been fully read; the *inputPath* stack is then popped (lines 17-20).

The path to the parent node of the most recently output subtree is maintained in stack *outputPath* (initially this stack only contains the SortNode for the root of the tree). For each node in *outputPath*, the algorithm stores the key value of its last output child. This enables heapify to identify children nodes that have key values less than that key value and mark them for the next run by moving them to the end of the heap (*i.e.*, by considering them "greater" than their siblings with keys greater than the last output key). Note that the memory space required for storing this information is equal to the size of the *outputPath* stack. When the array of a SortNode has the heap property, the children of the node that should be written to the next run are all placed at the end of the array.

Subtree output during sorting. We now turn to procedure findLeastKey (shown in Figure 6.6), which selects the next node (or subtree) to be output when memory is full. The algorithm locates the root of the subtree that (*a*) has the least absolute key that is greater than the last key output to the current run, and (*b*) has been fully read. When findLeastKey is called, the top of *outputPath* holds the parent of the previously output node. Using getLeastChild, we obtain the minimum child of the top of *outputPath*. Procedure getLeastChild, shown at the bottom of Figure 6.6, operates on some SortNode *p*. It initially checks if the first element in *p*'s heap array is free. This happens when no new child of *p* was read since the last time a child of *p* was output; then, the last element of the array is brought to the first position. Otherwise, a new node has been inserted as the first element of *p*'s heap array. In both cases, heapify is called to adjust the heap. A more complex case arises when (*a*) *p* is the top of both the *inputPath* and the *outputPath*, (*b*) at least one child of *p* has been output, and (*c*) two or more children of *p* are read consecutively into *p*'s heap (which has already been constructed using buildHeap) before any child is output. In this case, buildHeap would have to be called again. To avoid this we force a child of *p* to be output before the second consecutive child of *p* is inserted as the first element of *p*'s heap array. This case is not shown in insertNode, however, to keep the presentation simple. Procedure getLeastChild returns the first item of the heap, thus the node with the least key value. The only exception is the case when all children of *p* have been marked for the next run; this marks *p* for the next run as well.

Returning to findLeastKey, we first dispose of nodes of the *outputPath* that have

```
Algorithm 1: HERMES - Sorting Phase (Tree T)

1.  Stack inputPath
2.  Stack outputPath
3.  while (the input has not been exhausted)
4.     Read the next tag from T
5.     if (a start tag was encountered)
6.        Read the new node n
7.        while (not enough memory left for n)
8.           SortNode min = findLeastKey ()
9.           outputSubtree (min)
10.          free(min)
11.       end while
12.       Extract the local key of n
13.       Create a SortNode sn for n
14.       sn.readstate = 0
15.       insertNode (inputPath.top(), sn)
16.       inputPath.push(sn)
17.    else /* an end tag was encountered */
18.       inputPath.top().readstate = 1
19.       inputPath.pop()
20.    end if
21. end while

Procedure insertNode (SortNode p, SortNode sn)
22. if (the first place in p.heap is free)
23.    insert sn at the first place of p.heap
24. else
25.    insert sn at the end of p.heap
26. return

Procedure outputSubtree (SortNode root)
27. Write root to current run
28. sort (root.heap)
29. for each (SortNode cn child of root)
30.    outputSubtree (cn)
31. free(root)
32. return
```

Figure 6.5: The sorting phase of HERMES

```
Procedure: findLeastKey ()

1.  SortNode sn = getLeastChild (outputPath.top())
2.  while (all children of sn have been written)
3.      free(sn)
4.      sn = getLeastChild (outputPath.top())
5.      outputPath.pop()
6.  end while
7.  while (sn == null)
8.      outputPath.pop()
9.      sn = getLeastChild (outputPath.top())
10. end while
11. if (outputPath is empty)
12.     Start a new run
13.     outputPath.push(root of the tree)
14.     Write the root of the tree to the new run
15. end if
16. while (the end of sn has not been read)
17.     sn.buildHeap()
18.     outputPath.push(sn)
19.     Write sn to current run
20.     sn = the first item of sn.heap
21. end while
22. return sn

Procedure getLeastChild (SortNode p)
23. if (the first place in p.heap is free)
24.     move the last element of the heap to the front
25. p.heapify (0)
26. if (all children of p are marked for the next run)
27.     mark p for the next run
28.     return null
29. else
30.     return the first item of the heap
```

Figure 6.6: Procedures findLeastKey and getLeastChild

been fully output (lines 1-5). A node has been fully output if it has been fully read and its heap array is empty. Next, while the top of the *outputPath* stack has all its children marked for the next run, we ascend the tree by popping the stack to find the greatest ancestor that does not have all its children marked for the next run (lines 7-10). If no

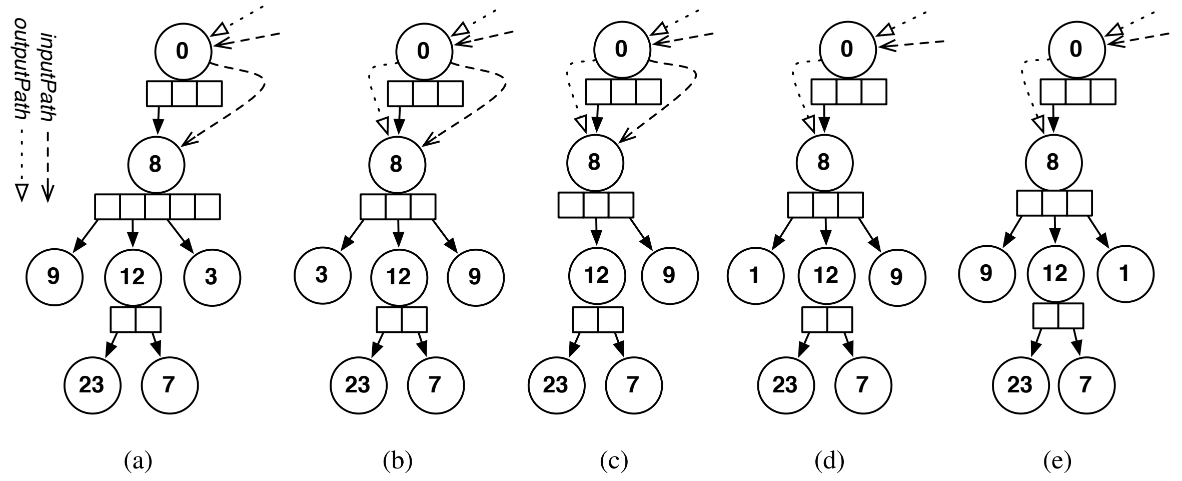


Figure 6.7: An example of the sorting phase for the creation of the initial runs

such node has been found after the root of the tree has been popped, the current run is closed and a new run has to be created (lines 11-15). The root of the tree is output to the new run and pushed onto the *outputPath* stack. At this point, we have reached a node some children of which are eligible to be written to the current run. However, if the subtree rooted at this node has not been fully read yet we need to descend into the tree to find such a complete subtree (lines 16-21). While descending, nodes are visited for the first time since the creation of the current run; therefore, we call buildHeap for each and push onto the *outputPath* stack their child with the least key value (without calling getLeastChild– it will always be the first element of the heap array). As we traverse a path of the tree we output visited nodes (line 19). The procedure returns the node closest to the root after the iteration (line 22).

The fully read node returned by findLeastKey is passed to outputSubtree (shown in Figure 6.5). The absolute key for the root of the subtree is smaller than all the subtree's nodes (as it is their prefix) so it precedes them in the output run. After the root of the subtree is output, outputSubtree sorts the root's children (all of which are present – line 28). It then recurses until the whole subtree has been sorted and written to the current run (lines 29-30). At the same time, the memory occupied by the subtree is freed (line 31).

Example 3.2: For the dataset of Figure 6.2 we show a part of the sorting phase in Figure 6.7, assuming that seven nodes fit in main memory. In Figure 6.7(a), node 3 has just been read and *inputPath*.top() points to node 8, the parent of the last read node. Since memory is full, the tree is traversed from the root towards the leaves, at each step heapifying and following the pointer to the child with the least key value. In Figure 6.7(b), the state of the system when the heap for node 8 has been constructed is shown. Stack *outputPath* points to that node, as it is the parent of the node to be output. In Figure 6.7(c), that node has been output and at the next step (Figure 6.7(d)), node 1 has been read and placed in the first place of its parent's heap. Node 8 has then been fully read and *inputPath* is popped. A node needs to be output again and heapify is called for the heap of node 8. Since 1 is less than 3, the key of the last written node, node 1 is placed at the end (shown in Figure 6.7(e)). Node 9 is then output and the next node is read from the input. Note that at the next output step, the subtree rooted at node 12 will be output as a whole, while node 1 will be written to the next run. □

Node serialisation in runs. Keys are written to disk runs in a compressed form, as all absolute keys in the tree share common prefixes. Had absolute keys been written uncompressed, the run files would be polluted with redundant information. This would not only be wasteful in terms of secondary storage, but also would heavily increase the I/O cost of writing each run to disk and subsequently reading it back in during merging. We use a typical tree compression scheme. Each time a node is output to the current run, its type, local key value and payload are serialised to the disk, preceded by the following special characters:

- A "|" if the node is a sibling of the last written one.
- A "/" if the node is a child of the last written one.
- A "^" for each level in the tree that the node is higher than the last written one.

Merging. During the merging phase, the sorted runs will be merged to produce the final output. One memory page is used as the input buffer for each input run and one page as the output buffer of the resulting merged run. As with external mergesort for flat data, it is possible that the available physical memory does dot suffice for all sorted runs to be merged in one merging phase. In this case more than one merging levels are required [Graefe, 1993]. Merging at each level is identical from an algorithmic perspective, thus we only describe the algorithm for a single merging phase.

The merging algorithm is shown in Figure 6.8. Nodes are read from the sorted runs into memory. When a node needs to be output a hierarchical priority queue, similar to the one used during the sorting phase, is used to locate the node with the least absolute key. This node is output and the next one is read from the run to which the output node belonged (lines 6-8). Identifying the node to output (line 5) is performed by findLeastKey, but with some modifications with respect to the sorting phase. The main difference is that the first element of a SortNode's heap array is always greater than the one previously written from that heap to the current run. This means that if a node *n* is visited by findLeastKey, then the subtree rooted at *n* will be written entirely to the output run before findLeastKey leaves the node. Thus, findLeastKey only pops a node from *outputPath* (line 8 in Figure 6.6) when the subtree rooted at this node has been written completely. Note that there is no way one can tell if the subtree rooted at a node has been entirely read or not without reading the next symbol of all runs that contain children of that node. Therefore, findLeastKey only returns leaf nodes during the merging phase, and outputs internal nodes as it descends to find the leaf nodes. The node returned by findLeastKey is written to the output run and the next node from the run from which the output node came is read. The process ends when all runs have been exhausted.

Note that each time we read from a run, we read as many nodes as required to reach a leaf, accomplished through procedure readNextLeaf of Figure 6.8. Always reaching a leaf ensures that when an internal node has been read, at least one of its children will have been read as well. For each input run we maintain a stack, termed *path*, that holds the ancestors of the last read node from that run. Initially, *path* contains the SortNode for the root of the tree (which is the first node written to all runs given the run serialisation protocol). Note that an internal node may appear in more than one run (*i.e.*, its descendants may appear in multiple runs). When an internal node is read, we check if it is already present in the heap of its parent's SortNode (line 16). If the node is already there, no new SortNode is created; otherwise, a SortNode is created and inserted into its parent's heap (lines 14-17).

6.4 Algorithm Analysis

In this section we study the theoretical properties of HERMES with respect to the size of the initial sorted runs and the I/O cost of the algorithm. We show that our algorithm maintains the most important advantages of external mergesort with replacement selection for the creation of initial runs. At the same time it does not repeat redundant information both in main memory and on disk, and does no redundant comparisons between nodes. We also present how the core algorithm takes advantages of the hierarchical structure to boost its performance.

```
Algorithm 2: HErMeS - Merging Phase (Run [] runs)
1. for each run file r in runs
2.    readNextLeaf (r)
3. end for each
4. do
5.    SortNode min = findLeastKey ()
6.    let r' = run from which min was read
7.    Write min to the output run
8.    readNextLeaf (r')
9. while (not all runs have been exhausted)
10. return

Procedure readNextLeaf (Run r)
11. while (a leaf node has not been reached)
12.    Read the next node n from r
13.    sn = lookUp (r.path.top(), n)
14.    if (sn == null)
15.       Create a SortNode sn for n
16.       insertNode (r.path.top(), sn)
17.    end if
18.    inputPath.push(sn)
19. end while
20. return sn
```

Figure 6.8: The merging phase of HErMeS

6.4.1 Run Size

For flat data, the average size of a run produced by replacement selection is twice the size of the memory used [Knuth, 1998]. Hereafter, we refer to the "size" or "length" of a run not in terms of bytes, but in terms of the number of nodes that it contains. The same applies to the size of main memory. For standard external mergesort each run is expected to contain twice as many records as can fit in a full main memory priority heap. After the priority heap becomes full, its size remains constant: before a new record is inserted into the heap, one is first output to the current run (and the heap shrinks when the input has been exhausted). We now prove that the initial runs created by HErMeS have the same property:

Theorem 4.1: *The average size of a run is twice the size of available main memory for sorting.* □

Proof. Consider the priority heap h_n of a node n in main memory, which holds pointers to the children of n. The size of h_n, which we denote as $|h_n|$, grows as children of n are read from the input. When the first child of n is to be output to the current run, h_n stops growing and from that point on its size remains constant. This is because from that point on, as explained in Section 6.3.2, before a new node is to be inserted into h_n, one node from h_n is output to the current run. When all children of n have been read, h_n begins to shrink. In other words, HERMES outputs the children of n in the same order that standard replacement selection would output them if they were records of a flat file *and* in the same number of runs. Consequently, for each node n the expected number of n's children written to the current run is twice the size of h_n, *i.e.*, twice the size of h_n when the first child of n is output.

Let m be the maximum number of nodes that fit into memory and consider the point in time t_0 at which a new run is created. At that point in time the memory is full, *i.e.*, a node needs to be written to the run (the first node of the run). At t_0 the size of the heap h_n of a node n $(1 < n < m)$ is $|h_n|_0$. All $|h_n|_0$ children of n will be written to the new run, since each one of them is neither marked for the next run at t_0, nor will ever be marked for the next run (as all of them are present when output to the run starts). When the first child of node n is to be written to that run at a later time t_i $(t_i > t_0)$, the size of h_n will be either equal to $|h_n|_0$ (if no child of n was read in the interval $[t_0, t_i]$, *i.e.*, the node had been fully read at t_0) or greater than $|h_n|_0$ if more children of n were read during that interval. In both cases $|h_n|_i \geq |h_n|_0$ holds. However, the expected number of children of n that will be written to that run is $2|h_n|_i$, *i.e.*, the total number of nodes written to that run is:

$$\sum_{n=1}^{n=m} 2|h_n|_i = 2\sum_{n=1}^{n=m} |h_n|_i \geq 2\sum_{n=1}^{n=m} |h_n|_0$$

At time t_0 the memory is full and holds m nodes. Of these nodes, all but the root of the tree are pointed to by some heap h_ℓ, that is $\sum_{n=1}^{n=m} |h_n|_0 = m - 1 \simeq m$ for large values of m. Hence,

$$\sum_{n=1}^{n=m} 2|h_n|_i \geq 2m$$

holds. □

From the description of the algorithm it follows that each node appears only once in main memory, not only during the creation of the initial runs, but also during the merging phase. Also, nodes appear only once in the sorted runs; only the key value of some internal nodes may be written to more than one runs, at most once in each run, and only when it is necessary for the reconstruction of the subtree that the run represents. More importantly, the algorithm only compares local key values of sibling nodes. Never are absolute keys used to compare two nodes that belong to different subtrees.

6.4.2 I/O Behavior

Regarding the I/O cost of HERMES, suppose that available memory is M memory pages. During the merging phase, $M-1$ memory pages are used for reading the input and 1 page is used to write to the output run. If the total size of the input tree is T memory pages then each initial run will have an average size of $2(M-1)$. Also, since $M-1$ buffers can be used for merging, there are going to be $\lceil \log_{M-1} \lceil \frac{T}{2(M-1)} \rceil \rceil$ levels of merging. Adding the first pass over the input to create the initial runs, we have that there will be $1 + \lceil \log_{M-1} \lceil \frac{T}{2(M-1)} \rceil \rceil$ passes over the input. In each of these passes, the whole tree is read and written to disk. This makes a total I/O cost of $2T \cdot \left(1 + \lceil \log_{M-1} \lceil \frac{T}{2(M-1)} \rceil \rceil \right)$.

As pointed out in [Graefe, 1993], one of the main concerns with replacement selection is how one can handle the payloads of the nodes that reside in main memory at any given time, *i.e.*, the payloads of the nodes whose keys are in the heap at that time. If these nodes are kept in the original buffer pages there is a great waste of space: only half of the nodes of any given page are expected to be in the priority heap of their parent node at any given point in time. This would mean that half of the available memory is not effectively used for sorting keys. Therefore, the benefits from replacement selection are cancelled (and quicksort could be used instead, probably yielding better results). As also pointed out in [Graefe, 1993], the solution to this problem is to copy the payloads of those nodes to a temporary space in memory until they are written to the run, so that no space is wasted. Assuming that nodes of the same type have similar size, this can be a viable solution. However, large variations in the size of the nodes of the tree require complex and potentially overhead-inducing in-memory management primitives.

Double buffering certain pages during the merging process is also a technique that can improve the performance of our algorithm. For instance, using more than one memory pages for the output run at each merge level can eliminate the need for the

CPU to wait for a write I/O call to complete after the output buffer is flushed (as is the case if a single output buffer is used). Regarding the input buffers, the situation is somewhat different. Reserving two memory pages (or more) per input run would reduce the number of runs we can merge by half. What we can do is reserve a number of k memory pages in order to prefetch the next page from the k input buffers that contain one of the k smallest maximum keys among all buffers (since we then know that the next page to be read will be the next from one of those k runs).

6.4.3 Improvements

We now present how the hierarchical structure has been further exploited to improve the core algorithm.

Processing entire subtrees. A useful optimisation arises when all descendants of a node n have been read into main memory and the first needs to be output. In that case, the whole subtree rooted at n is output, with the children of nodes of that subtree being sorted (see Section 6.3.2). As n will be the first node of the subtree to be written, one can place a mark on the run file indicating that the whole subtree follows, *i.e.*, all descendants of n are written to that same run, following n. That way, when n is read during the merging phase, we know that it is followed by the entire subtree rooted at n. Therefore, only n needs to be brought to memory and be placed in the heap of its parent. The rest of the subtree need not be constructed in main memory. When n is to be output, the subtree of n is copied from the input run to the output run without any in-memory processing, as it is already sorted. Furthermore, if this subtree spans many pages, these pages can all be prefetched.

Properties of runs. We now turn to the properties of initial runs.

Lemma 4.1: *A group of nodes that co-exist in memory at some point in time will be written either to the same run, or to two consecutive ones.* □

Proof. During the creation of the initial runs, a node that has been read from the input will be written either (a) to the current run, if it has a greater key than all its siblings that had been written to the current run when the node was read, or (b) to the next run otherwise. Thus, if two nodes n_1 and n_2 co-exist in memory at some point in time, they will eventually be written either to the same run or to consecutive ones. That is, if r_i is the current run, n_1 and n_2 will either both be included in r_i, or both in r_{i+1}, or one of them in r_i and the other in r_{i+1}. Following an inductive argument, it is easy to see that the same applies for any number of nodes that at some point in time co-exist in memory. □

We shall now show that this is the case for all groups of sibling nodes. Recall that the input tree is read depth-first. For this reason, all nodes with the same parent will be read as a *batch*. Let n_f be the first node of the batch written to the output and n_l the last one.

Lemma 4.2: *Every node of the batch other than n_f and n_l will at some point in time co-exist in memory with some other node of the batch, i.e., with one of its siblings.* □

Proof. The statement holds due to depth-first traversal. Assuming it does not hold, there must be at least one run of length 1. Any other case implies that at least one node that does not belong to the batch has been read between two nodes of the batch. The latter is eliminated by the depth-first traversal of the tree. The former is negligible since it implies that the memory allocated for the heap can only hold one node (*i.e.*, it has a size of 1); fortunately, we have been able to use larger memory sizes for building heaps! □

Thus, all co-existing nodes will eventually be written to consecutive runs, say, $r_w \ldots r_z$. The only nodes of the group not accounted for so far are n_f and n_l. To show that *all* groups of sibling nodes will be placed in consecutive runs, it suffices to prove the following.

Lemma 4.3: *Node n_f will be written either to r_{w-1} or to r_w and n_l will be written either to r_z or to r_{z+1}.* □

Proof. The trivial cases are (a) when n_f and n_l co-exist with some other node of the batch in their parent's heap at some point in time, or (b) they are written to r_w and r_z respectively; then the statement holds. If neither case holds, we need to show that n_f will be written to r_{w-1} and n_l will be written to r_{z+1}. Therefore, n_f is the only node of the batch written to $r_{w'}$ ($w' < w$) and n_l is the only node of the batch written to $r_{z'}$ ($z' > z$). Then, for n_f we have that when it was to be written to a run, it was the only entry in its parent's heap (otherwise we fall into the first trivial case mentioned above). This implies that memory became full just when n_f was read. After it was output to the run, the next node read, say, n_g, was a sibling of n_f, as input is processed in a depth-first manner (if n_f has no siblings the lemma holds). If the key of n_g is greater than the key of n_f, then n_g is written to that same run, *i.e.*, $w \equiv w'$. Otherwise, it is marked for the next run and outputting continues from a sibling of n_g's parent. The parent of n_g is visited again by the algorithm when the next run is being produced and n_g is output to that run. In that case, $w \equiv w-1$. Similar arguments show that n_l will either be written to r_z or r_{z+1}. □

A different merging scheme. The previous lemmata prove that all children of a node are written in consecutive runs. During the merging phase, one can utilise this fact when not all siblings of a node have been read into its parent's heap and the first sibling is written to a run. A bit of extra book-keeping is needed to identify the first and last of these consecutive runs (*i.e.*, runs pertaining to the set of siblings). Assuming these runs are no more than k, we can prefetch at least one page from each run and in this way avoid having the CPU wait for I/O to complete while merging the siblings.

We shall now present a different merging scheme that takes advantage of all nodes in a batch being written to consecutive runs. In the following we assume that all children of a node n have been written to k consecutive runs $r_w, \ldots, r_{w+k}$. Such information can be recorded for n during the replacement selection phase and stored in the SortNode for n, which remains in memory until the last child of n is flushed to a run. During the merging phase, runs $r_w, \ldots, r_{w+k}$ can be merged into a single run $r_{w,w+k}$. Before we start merging these runs, we can be certain that $r_{w,w+k}$ contains all children of n in the proper order. We therefore record this when the first node of the batch is written during merging. During the next merging level, when n is read, we can identify that this run contains the whole subtree rooted at n, so that we can output all nodes of the batch consecutively, similarly to what we mentioned for the case of merging initial runs. Again, in this way we avoid the cost of re-constructing the subtree in memory and inserting its nodes into the priority heaps of their parents. Considering the prefetching techniques mentioned earlier this can turn into a significant advantage. The only caveat is that this technique is applicable only if multiple merging levels are needed. In case of a single merging level, this variation will probably have worse performance since it will introduce another pass over the records of $r_{w,w+k}$, if these runs $r_w \ldots r_{w+k}$ are merged individually. If multiple merging levels would be used, however, we can use this technique for disjoint groups of k runs. The larger the fan-out of the tree, the greater the length of a batch will be and, thus, the more efficiently this technique is expected to perform.

6.5 Experimental Results

HERMES has already been deployed as part of the archive management system XARCH, detailed in [Müller et al., 2008] and [Koltsidas et al., 2008]. XARCH is a stand-alone Java application that allows one to maintain, populate, and query archives of hierarchical data with a key specification. XARCH uses HERMES for sorting incoming versions before applying nested merge. In this section, we evaluate the performance of HERMES under various workloads as a stand-alone hierarchical data sorting solution. To that end we re-implemented it in C++. HERMES was compiled using the GNU C++ compiler, version 4.1.2. All our experiments were run on an Intel Core 2 Duo processor clocking at 2.33GHz with 2GB of physical memory. The box was running Ubuntu Linux 7.10 with the 2.6.22 kernel. For each experiment we report the average wall clock time of five runs over cold data.

We implemented the algorithm of Section 6.3 and the improvements of Section 6.4. To create the input data we used a custom data generator. Each input file was an XML document, each node of which had a randomly generated character string as its label. We used the label of a node as its key. Node label lengths, and thus key value lengths, were variable. The generator allowed us to specify the maximum depth of the tree and a maximum fan-out for all nodes. The fan-out of each node was uniformly distributed between 0 and the specified maximum. As a result, the average fan-out was half the maximum. We compare the performance of HERMES to that of NEXSORT using the original implementation of NEXSORT in combination with the *Transparent Parallel I/O Environment* [D. E. Vengroff, 1994], as in [Silberstein and Yang, 2004]. As suggested in [Silberstein and Yang, 2004], we set the sorting threshold for NEXSORT to be roughly twice the block size, which, for our system was 64KB, making the threshold equal to 128KB. All experiments, apart from the ones of Section 6.5.5, we run over a magnetic disk.

6.5.1 Impact of Input Size

We first measured the impact of input size on the running times of HERMES and NEXSORT. We used input trees of a depth of six, which is a typical depth for most real-world datasets. For each depth, we generated trees of different sizes by varying the average node fan-out. For both algorithms, the size of available main memory for sorting was set to 10MB. The sizes of the input trees varied between 30MB (or, 1.2 million nodes) and 2.2GB (or, 83 million nodes). The average length of a node key was set to ten

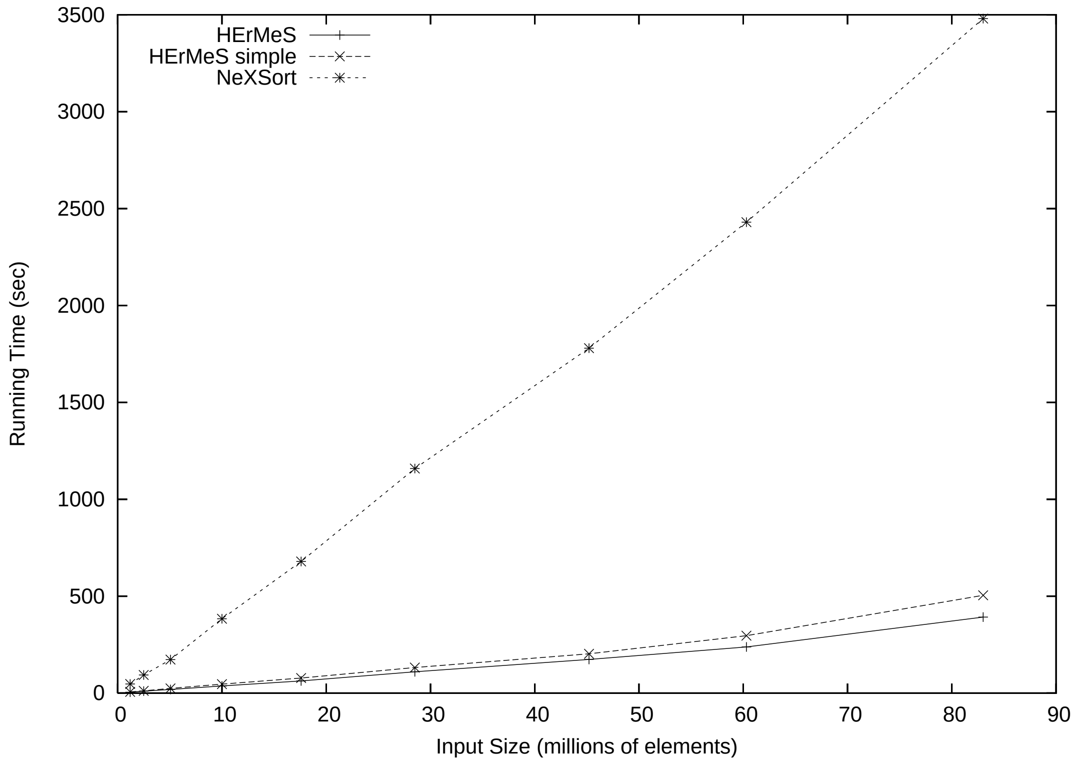


Figure 6.9: Impact of input size

bytes. The results are presented in Figure 6.9 (the raw data are given in Figure 8.37). We also report under the "HErMeS simple" plot the response time for our algorithm if the improvements of Section 6.4 are not used.

In all cases, HErMeS performs 8.5 to 10.8 times faster than NeXSort. This is due to the way NeXSort writes sorted runs. NeXSort employs a stack over secondary storage to store the next subtree to be sorted. When a subtree residing in the stack has been sorted, the sorted subtree is written to disk and a pointer is written back to the stack. At any point in time there are multiple such secondary storage stacks used for book-keeping purposes. Pushing nodes onto these stacks and popping them involves disk accesses. Therefore, during its sorting phase, NeXSort reads and writes both to the on-disk stack pages and to the current sorted run at the same time. This introduces a severe performance penalty. In addition, NeXSort accesses the disk in a random pattern when reconstructing the output tree: it follows pointers to sorted runs that have not been sequentially written to disk (*i.e.*, one run after an other). Hence, a lot of time is spent with the CPU stalling for I/O.

On the other hand, during replacement selection for the creation of initial runs, HErMeS accesses secondary storage only for the purpose of flushing data to the cur-

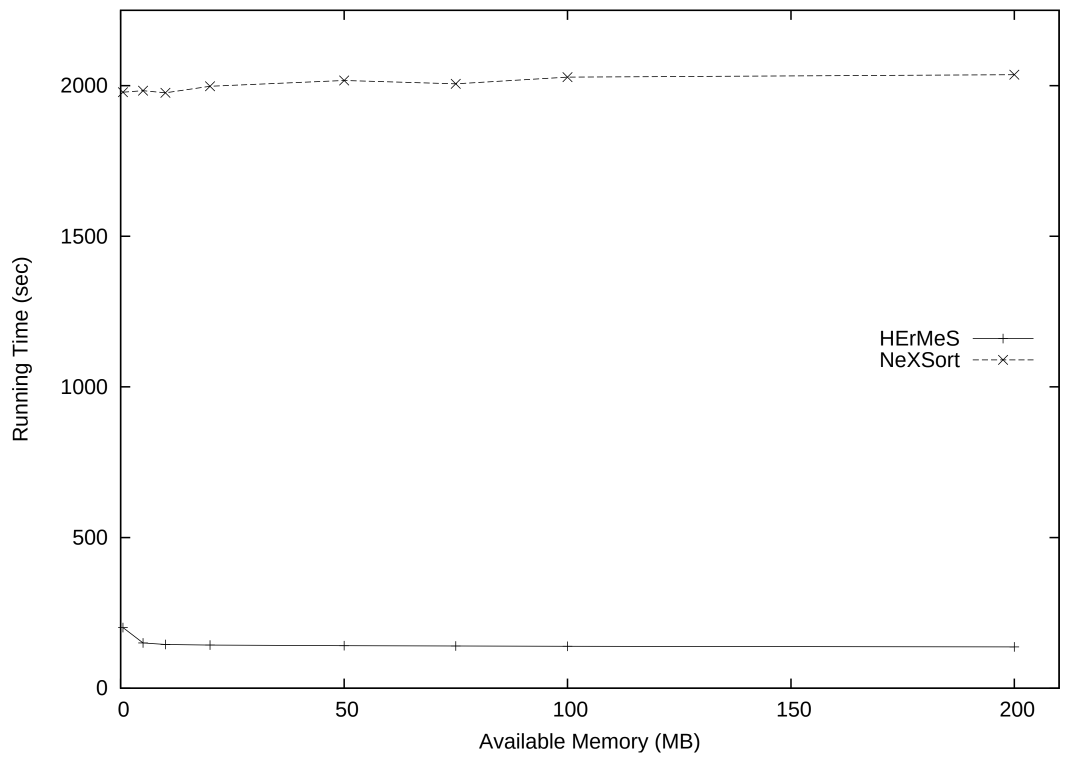


Figure 6.10: Impact of available main memory

rent run. As additional evidence, during our experiments we observed that NeXSort's reconstruction time amounted to almost 40% of its total running time. On the contrary, the merging phase for HErMeS took no more than 20% of the total running time (when a single merging level is required). As one can observe from the results, the improvements discussed in Section 6.4, give another 15% to 25% performance boost to HErMeS. Thus, we believe such improvements are a worthwhile addition to the main algorithm and have used them in all experiments.

6.5.2 Impact of Available Main Memory

In our next experiment, we examined the performance of both algorithms for different sizes of main memory available for sorting. We generated an 1GB input tree with 41 million nodes. This tree was six levels deep and the average node fan-out was set to 35. We varied the available memory size to seven different values ranging between 0.5MB to 200MB and measured the running time for both HErMeS and NeXSort. The results are presented in Figure 6.10 (the raw data are given in Figure 8.38).

When only 0.5MB of physical memory are used, HErMeS requires two merging levels. We observed that processing each merging level takes about 45 seconds, or, about 25% of the total running time. For larger sizes of available main memory only one merging level is required and thus the total running time is almost constant. When the amount of available memory grows much larger than the amount needed to achieve a single merge level, running time drops slightly as available memory increases. This is because the average size of a sorted run grows much bigger and the number of sorted runs drops, *i.e.*, merging has a smaller fan-in. When only one merging level is needed HErMeS performs almost ten times faster than NeXSort, irrespective of the amount of available memory. When two merging levels are required, HErMeS performs about eight times faster than NeXSort.

6.5.3 Impact of Tree Depth

We next examined how the depth of the input tree affects the performance of our algorithm. We experimented with trees three, five, seven, and nine levels deep. For each of these depths we generated trees of different sizes (by varying the average fan-out). The results are presented in Figure 6.11 (the raw data are given in Figure 8.39). As shown, the running time of our algorithm is not heavily affected by the depth of the tree, especially for trees deeper than five levels. One can observe that the deeper a tree is, the more efficiently it is sorted by HErMeS.

To understand why this is the case, consider trees of the same size (*i.e.*, equal numbers of nodes) but of different depths. To keep the number of nodes fixed, shallow trees will have a much greater fan-out than deep trees. For instance, the largest tree we generated for each depth had a size of about 1.6GB. However, the average fan-out for a tree of depth three is 7,500 nodes, while the average fan-out for trees of depth five, seven, and nine are 85, 21 and 9 respectively. When sorting using HErMeS, the larger the fan-out of a tree is, the greater the average size of a heap is in main memory. As a result, each heapify operation takes longer. Moreover, for a fixed memory size, the probability of a subtree having been fully read during the sorting phase grows are the fan-out shrinks. Thus, the algorithm processes entire subtrees more frequently, albeit for smaller subtrees. Thus, the improvements described in Section 6.4.3 are more frequently applicable. This behaviour verifies our claim that HErMeS takes advantage of the hierarchical structure of a dataset: it performs better for deeper hierarchies. Also, these results verify the theoretical expectations of [Silberstein and Yang, 2004]: the possible sorting outcomes for a tree with a fixed number of nodes increases as the maximum fan-out of the tree grows.

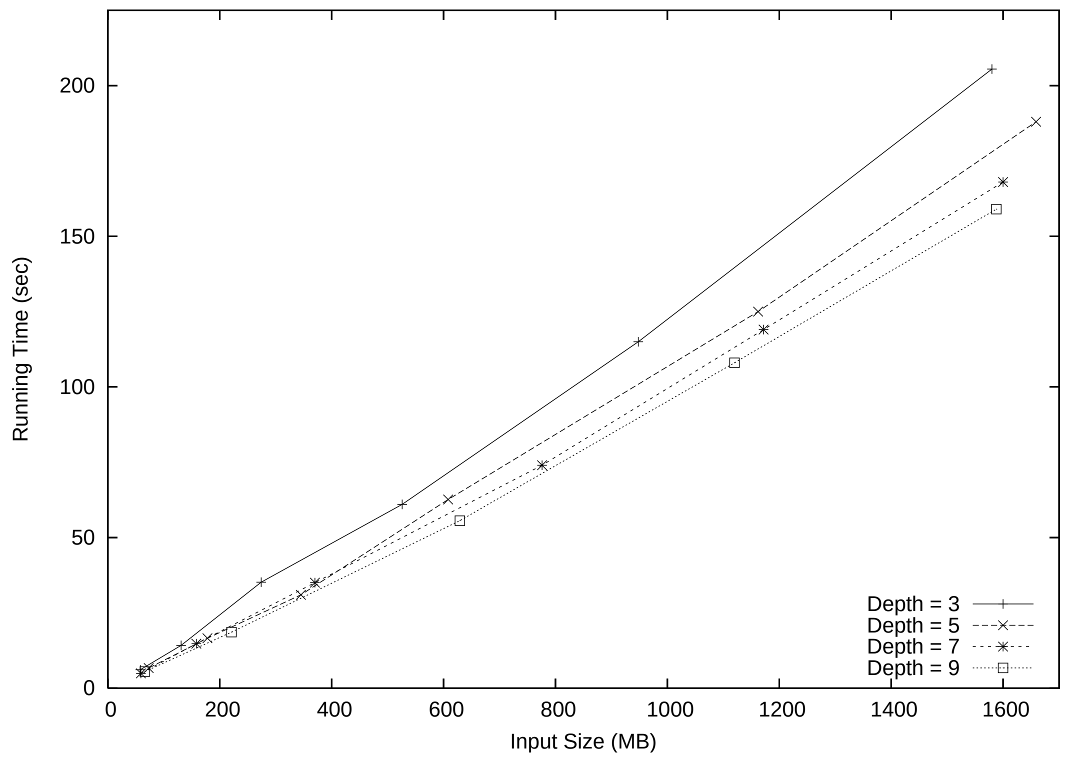


Figure 6.11: Impact of tree depth

6.5.4 Impact of Key Length

We then moved on to evaluate the performance of HERMES with respect to different key lengths. We used our generator to create regular trees with the same number of nodes. All trees had a depth of five and a constant fan-out of eighty. They only differed in the average length of node keys. We created trees with the average key length varying from 5 bytes to 180 bytes. For each such tree, we ran HERMES having set the available main memory to 10MB. The resulting running times are shown in Figure 6.12 (the raw data are given in Figure 8.40).

As expected, the running time increases linearly with the length of the key. When longer keys are used all in-memory operations on keys, such as in-memory copying and comparisons, take longer to execute. The same applies to secondary storage operations, since the amount of data to be read/written for each key increases. The increase in execution time is linear to the length of the keys, which is also expected since all the aforementioned operations take linear time with respect to key length.

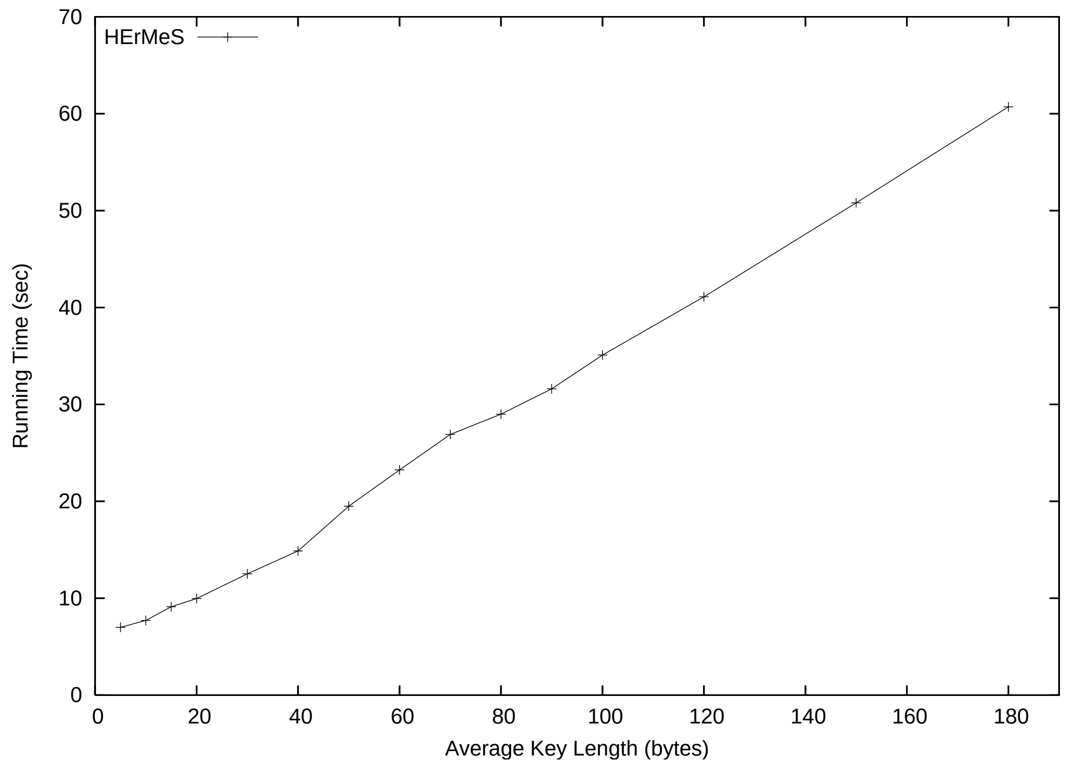


Figure 6.12: Impact of key length

6.5.5 Performance over a flash disk

We also experimented with HErMeS and NeXSort over flash disks. We created an input tree of depth 4; the average node key length was 50 bytes, while the total size of the tree was 1.75GB. For this set of experiments we used the two systems, System A and System B, described in Section 4.7; each system is equipped with a flash disk. For each one of the two systems and for each algorithm, we used three setups: (*a*) only the magnetic disk was used for storage, (*b*) only the flash disk was used for storage and (*c*) the magnetic disk was used to store the input and output files, while the flash disk was used for the intermediate sorted runs. We measured the running time for all configurations and plotted it on Figure 6.13 (the raw data are given in Figure 8.41).

For System A, the running time of HErMeS dropped by about 50% both when using the flash disk only and when using both disks. The reason for this is that the access pattern was favourable for the flash disk, as all writes to the flash disk where sequential and reads were random, *i.e.*, the full potential of the flash disk was exploited. Recall that the flash disk of System A is a high-end one and as such this result is not surprising. An interesting thing to note here is that, when running HErMeS on System A over the magnetic disk, the CPU utilisation was at about 55% when creating the sorted runs and

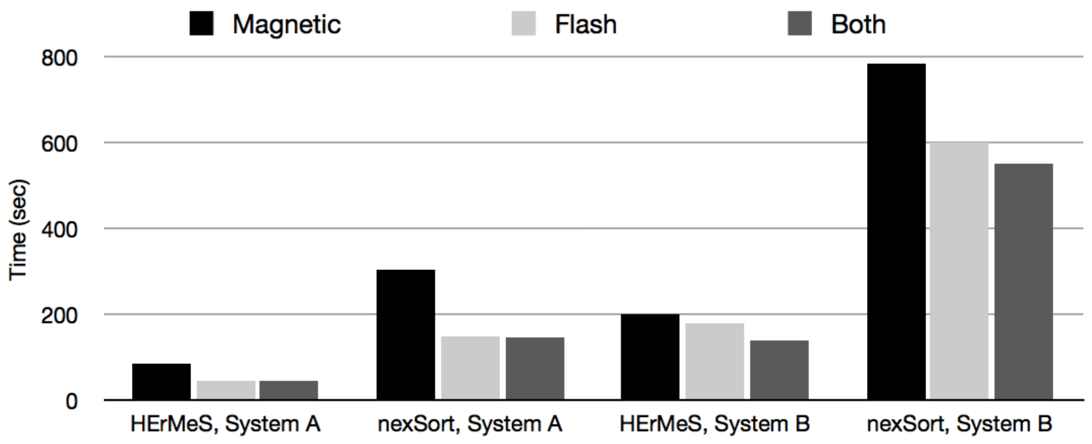


Figure 6.13: Sorting over flash memory.

34% during the merging phase. On the other hand, when running HERMES over the flash disk, the CPU utilisation was steadily over 95% for both phases. In other words, the disk almost saturated one processing core even at random read patterns. Similar observations hold for NEXSORT: the running time dropped by about 55% when using the flash disk. Still, the performance of NEXSORT remained more than 4 times worse than the performance of HERMES in all cases.

Regarding HERMES on System B, as shown in the figure, the drop in the running time is about 10% when running from the flash disk only and about 30% when the input and the output are stored on the magnetic disk. The particular flash device was a low-end one and, thus, the performance increase was not as much as for System A. The access pattern was favourable for the flash disk; however, at sequential operations the bandwidth of the magnetic disk was slightly better than that of the flash disk. Reading the input and writing the output file are both done sequentially; this is why HERMES performed better when both the input and the output were stored on the magnetic disk. Also, when only the flash disk was used, the device saw both reads and writes during both the sorting and the merging phase. Considering the lack of parallelism and native command queueing of the particular device [Lee et al., 2009], this behaviour was also a result of reads stalling until erase operations for writes finished. When using both disks, on the other hand, the flash disk was only written to during sorting and only read from during merging. For NEXSORT on System B, the performance improvement due to the flash disk was greater as NEXSORT performs much more random I/O than HERMES. However, it still remained more than 4 times worse than HERMES. Using both disks also helped NEXSORT for the same reason as for HERMES: due to the better sequential bandwidth of the magnetic disk. As a conclusion, we found that flash disks can substantially enhance sorting performance. Of course, the better the I/O costs of the flash disk, the more the performance improvement will be. Nevertheless, even for low-end devices, the improvement is important when the flash disk is used as temporary space only.

Chapter 7

Conclusions

With the emergence of flash memory as a viable storage alternative to traditional storage media, several database storage design choices need to be revisited. In this thesis, we have explored the design of a hybrid storage system, *i.e.*, one that utilizes both magnetic and flash disks for data storage. Our starting point was the observation that the term "flash disk" incorporates many different classes of storage devices, with widely disparate price and performance characteristics. Devices currently on the market exhibit random read speeds that vary across two orders of magnitude, while their random write speeds range within four orders of magnitude. As such, it comes as no surprise that their per-gigabyte prices range within two orders of magnitude as well. What is more, one can classify flash disks neither as "better" than magnetic ones nor as "worse". Flash disks certainly dominate magnetic ones at random reads. When writing randomly, on the other hand, some flash devices outperform magnetic disks, while others are orders of magnitude slower. Even for flash disks that totally outperform magnetic ones, however, the situation becomes quite more complicated when their price/performance ratio is taken into account.

In our work we have identified the different classes of flash devices and proposed algorithms and techniques to take advantage of their I/O characteristics with respect to performance, most notably to exploit their random read efficiency. When dealing with inexpensive flash disks, which are inherently inefficient at writing but large at their capacities, we proposed that they be used as persistent storage devices using novel data placement algorithms. Enterprise-class, high-end flash disks can be used as cache for the underlying storage very effectively. In both cases, main memory buffer allocation has a significant impact on performance and therefore the I/O costs of devices need to be taken into account. As such, all techniques presented in this thesis are cost-aware and adaptive. Our work has studied hybrid system setups in the context of database workloads. However, since no assumptions are made regarding the data model or access methods, extending our proposals to generic storage systems is rather straightforward.

Data Placement. In the first part of this thesis we studied how low-end flash disks can be efficiently used in a commodity system. Such disks typically use MLC flash memory and have a capacity comparable to that of magnetic disks. However, although they outperform magnetic disks at random reads, they are slower when writing randomly. In such cases, we propose using the two disks at the same level of the memory hierarchy, with each data page residing in either of the two media, but not on both, *i.e.*, the flash disk is utilized as persistent storage.

The storage manager decides the optimal placement for each page according to the workload of the page. Pages with a read-intensive workload are placed on the flash disk, while pages with an update-intensive workload are placed on the magnetic disk. Thus, reads are faster than a magnetic-disk-only system, and writes are faster than a flash-disk-only system. In this manner the total I/O cost is reduced. The main challenge we dealt with, was how one can predict the future workload of a page based on past accesses to the page with confidence. Of paramount importance was the ability to self-tune, *i.e.*, adapt the placement choice for each page when its workload changes from read-intensive to update-intensive and vice-versa. Considering that moving a page from one disk to another incurs significant I/O cost, the prediction of a page's future workload had to be as accurate as possible. Failure to achieve an acceptable level of accuracy would result in the I/O cost being heavily penalized.

The system keeps track of the read and write operations a page has seen and decides the placement of the page as soon as it is evicted from main memory. We modelled the decision problem for the two placement alternatives as a two-state task system. We proposed a family of on-line algorithms that take into account both the physical history of the page, *i.e.*, the physical I/O on the page, and the history of logical operations on the page (that is, operations served in-memory) to decide the optimal medium on which the page should be stored. If the page is found to be on the wrong medium, it migrates. By continuously evaluating the placement decision for pages, the system adapts to changing workloads. The theoretical study of the proposed algorithms showed our algorithms to be 3-competitive with respect to the optimal offline adversary.

We have implemented all proposed algorithms in a real system and conducted an extensive experimental study. The results proved our techniques to significantly improve I/O performance over magnetic-disk-only and flash-disk-only setups for database workloads that frequently occur in practice. Our algorithms adapted well to changing workloads. Clearly, the hybrid system can significantly outperform both magnetic-only and flash-only systems; more so, as hot pages have read-intensive workloads. Evidently, a system that uses only the low-end flash disk is unsuitable for real-world data-processing applications. On the other hand, the performance of traditional magnetic-only systems can be substantially enhanced using a low-end flash disk with efficient data placement. Therefore, we believe, a hybrid system assisted by techniques like the ones we presented is essential when pursuing high-performance data processing.

Buffer Allocation. After addressing the problem of data placement, we went on to study the problem of buffer allocation in databases that store data across multiple storage devices with varying I/O characteristics. Storage devices in a multi-device configuration compete for main memory buffer space. The system needs to drive this competition based on informed decisions that carefully exploit the I/O cost discrepancy between devices. Thus, our goal was to improve the performance of such a system by allocating the optimal number of main memory buffers to each device. The techniques we proposed take into account not only the I/O characteristics of devices, but also the impact of caching the data of a device across different cache sizes.

We started by experimenting with a static allocation algorithm, that uses only the I/O costs of devices to decide page replacement. We found the proposed algorithm to have a lot of potential, as in many cases it reduces the I/O cost paid by the system substantially. However, we observed that the wise choice of user specified parameters was critical for the performance of the system. Therefore, we went on to design a system that decides buffer allocation dynamically and adaptively. Towards informed decisions and high-performance, we found that the system needs to know the expected hit ratio for the pages of each device under all possible cache sizes. To that end we proposed a novel technique for measuring hit distances in the cache utilizing page timestamps. This technique has practically no overhead and enables fast and accurate tracking of the hit ratio curve for each device using Mattson's algorithm. The latter was otherwise not practically applicable at such fine a granularity; what is more, our technique is applicable to all systems that can benefit from hit ratio curve tracking, *e.g.*, allocating memory to different processes. Towards optimal memory allocation, we introduced the notion of *device caching utility*, which captures the cost savings realized by allocating page frames to a device. Essentially, this metric is determined by the hit ratio for the data of a device and the random I/O costs of the device. The caching utility was subsequently used by a low-overhead greedy algorithm to decide the optimal number of page frames that should be allocated to a device. Allocation decisions are also fine-grained ones and are constantly re-evaluated by the system, leading to adaptive behavior under real-world access patterns.

We implemented the proposed algorithms and experimented with both synthetic and real-world workloads. Our experimental results show that (*a*) our algorithm for measuring hit distance is both efficient and accurate, (*b*) utility-aware cache partitioning substantially improves I/O throughput in a variety of workloads and device setups, and (*c*) using our techniques, wrong data placement decisions can be effectively reversed.

Caching on flash memory. Next, we explored how a system can use a flash disk as a cache layer between the main memory buffer pool and the underlying magnetic disk(s). A crucial decision for the designer of such a system is which data will be cached on the flash disk. In contrast to buffering in main memory, pages do not need to be brought into the flash cache before they are processed. That is, a page may go directly from the magnetic disk to the memory and may well never be written to flash. Thus, deciding how data should flow from one level of the memory hierarchy to the others is not straightforward.

Let $P_{RAM}(t)$ be the set of pages stored in the RAM cache at some point in time t, and $P_{FLASH}(t)$ be the set of pages on flash. We identified the following three potential invariants:

1. $\forall t \; P_{RAM}(t) \bigcap P_{FLASH}(t) = P_{RAM}(t)$
   Whenever a page is in RAM, it is also cached on flash (*inclusive* cache hierarchy).
2. $\forall t \; P_{RAM}(t) \bigcap P_{FLASH}(t) = \emptyset$
   No page is stored in *both* RAM and flash at any time. A page brought from FLASH to RAM is removed from FLASH (and vice versa), resulting in an *exclusive* cache hierarchy.
3. $\forall t \; P_{RAM}(t) \bigcap P_{FLASH}(t) \subseteq P_{RAM}(t)$
   Decisions are made on a *lazy* per-page basis. A page in RAM may or may not be cached on FLASH, depending on criteria either set by the user or decided based on the current workload, *e.g.*, caching only clean pages.

Enforcing any one of the above invariants results in a different flow of pages across the levels of the memory hierarchy; thus, we presented three different *page flow schemes*. Each scheme incurs a different I/O cost for a given workload and is independent of the replacement policy for each cache; our system calculates the I/O cost for each one of these schemes and adaptively switches to the most efficient. We also studied several implementation issues that arise when using a flash disk as a cache: (*a*) the page directory for the cache, (*b*) the size of flash pages, and (*c*) caching only pages that satisfy specific predicates.

Our experimental results showed that most questions regarding flash-resident caches cannot be given universally optimal answers; rather, a cost model like the one we propose should be used to answer such questions for each individual case with confidence. High-performance flash devices can substantially improve the I/O performance of a system using any page flow scheme. On low-end, write-inefficient flash disks, however, either the dirty pages should not be cached, or a scheme that avoids many writes to the flash cache should be used. In the general case, analytical tools such as the ones we provide are necessary to achieve maximum performance.

Sorting hierarchical data. In the last part of our work, we studied the problem of sorting hierarchical data in external memory. Our purpose was to (*a*) experiment with sorting on flash memory, as external sorting algorithms like merge-sort generate access patterns that can exploit the full potential of flash memory and, (*b*) generalise the existing sorting algorithms to hierarchical datasets. Sorting hierarchical data has emerged as a salient operation for many applications, most notably for archiving scientific data.

We proposed an algorithm that generalizes the most widely-used techniques for sorting flat data in external memory, namely replacement selection and external merge-sort. The algorithm efficiently exploits the hierarchical structure in order to minimize the number of disk accesses and optimize the utilization of available memory. We extracted and verified the theoretical bounds of the algorithm with respect to the structure of the hierarchical dataset. The experimental study of our algorithm included a comparison to the state-of-the-art approaches. Our results showed that our algorithm outperforms the competition by almost an order of magnitude and its performance is the one expected from its theoretical analysis. Though motivated by sorting scientific datasets for archiving purposes, the algorithm is general and efficient enough to be applicable in a variety of problems where the need for sorting arbitrary hierarchical datasets arises. Also, it is not geared towards systems equipped with flash disks only, but also suitable for use in traditional systems with magnetic disks.

To summarize, we consider the most important contributions of this thesis to be the following:

- We found that low-end flash disks can improve performance in a hybrid setup substantially and provided on-line algorithms to decide data placement.

- We introduced per-device memory allocation and the concept of caching utility and proposed novel techniques towards high-performing allocation.

- We studied caching on flash memory and found it to be a multi-dimensional problem. Our results showed that there is no optimal design across all workloads and flash devices; rather, analytical tools such as the ones we provide are necessary.

- We designed and presented a novel algorithm for sorting hierarchical data in external memory, suitable both for flash and magnetic disks. Our proposals were found to outperform competition by an order of magnitude.

7.1 Future Work

Certain future research directions can be identified. In the context of data placement an interesting question is on what principle one should decide data placement when the flash disk does not have enough capacity for all the pages with read-intensive workloads to be stored. In such cases, a ranking algorithm should be considered, *i.e.*, one that captures the utility of each read-intensive page being stored on the flash disk: then only the subset of pages with the highest utility should be placed on the flash disk. Data placement becomes an interesting problem when the set of queries will be executed on a database and their frequency is known *a priori*. In that case, one can take advantage of that knowledge to statically predict the probability of a page having a read-intensive or an update-intensive workload; thus, placement can be decided statically and, possibly, with more confidence.

We have found that the use of a flash disk for external sorting can substantially improve performance. It would be interesting to explore what other database operations, such as join evaluation, can be more efficient using flash memory as an extended memory buffer and the effect of the I/O costs of the particular device(s) used on each such operations. Query optimizations should also account for the new storage medium. In [Ramamurthy and DeWitt, 2005], the authors show that query optimisers should not ignore the contents of the buffer pool when selecting access methods or evaluating execution plans; rather, since a significant portion of a table may be cached in-memory at some point, they suggest that the optimiser takes advantage of it. Similarly, when using a flash disk as a cache, the query optimiser should use the I/O costs of the flash disk for the portion of the data that is cached; more so the capacity of the flash cache grows.

Lately, manufacturers of flash disks consider offering a richer interface to the users, most notably by allowing users to explicitly invalidate logical pages, using the so-called TRIM command. Using this command the user can essentially erase specific physical blocks at will. Also, in some very recent products, the controller logic and the FTL algorithm for the disk run at the operating system level, *i.e.*, inside the driver for the device, using the CPU and main memory of the system. The market trend is towards giving users more power and moving critical operations outside of the disk enclosure: in this way such operations can be customised for specific applications. This is particularly interesting for the database community, as currently all flash devices are geared towards user filesystems. Therefore, we believe that implementing database-friendly FTL algorithms and tuning the flash disk internals towards databases will be critical for the suitability of flash disks for database workloads in the future.

Chapter 8

Experimental Data

In this chapter we present the raw data collected in our experiments in tabular form.

Execution #	M/F	M	F
1	212	212	12
2	212	214	11
3	214	212	11
4	414	212	11
5	130	213	11
6	130	212	12
7	858	212	11
8	11	212	11
9	11	212	11
10	11	212	11
11	11	212	11
12	11	212	11
13	11	212	11
14	11	212	11
15	11	212	11

Figure 8.1: Raw data for Figure 3.6 (a).

Execution #	M/F	M	F
1	235	245	770
2	234	243	765
3	234	241	771
4	233	243	770
5	233	243	770
6	233	243	770
7	234	243	770
8	233	243	770
9	232	243	770
10	233	243	770
11	234	243	770
12	235	243	770
13	234	243	770
14	233	243	770
15	234	243	770

Figure 8.2: Raw data for Figure 3.6 (b).

Execution #	M/F	M	F
1	917	941	1647
2	938	930	1630
3	875	928	1625
4	876	929	1630
5	674	928	1630
6	673	929	1630
7	885	930	1629
8	617	931	1630
9	614	932	1631
10	614	930	1632
11	614	930	1630
12	614	931	1630
13	614	931	1630
14	614	931	1630
15	614	931	1630

Figure 8.3: Raw data for Figure 3.7 (a).

Execution #	M/F	M	F
1	839	835	1071
2	888	825	1058
3	652	824	1060
4	600	824	1057
5	398	825	1055
6	398	826	1054
7	650	825	1058
8	356	824	1058
9	356	825	1058
10	355	825	1058
11	356	825	1059
12	361	825	1058
13	354	825	1057
14	354	826	1058
15	354	826	1058

Figure 8.4: Raw data for Figure 3.7 (b).

Execution	Conservative	Optimistic	Hybrid	Optimal
r	320	1213	362	1211
r	342	21	741	21
r	383	21	178	21
r	522	21	499	21
r	104	21	21	21
r	104	21	21	21
r	597	21	21	21
r	21	21	21	21
r	21	21	21	21
r	20	21	21	21
u	1040	508	859	335
u	506	376	446	432
u	351	432	376	432
u	431	432	431	432
u	433	432	432	432
u	432	432	431	432
u	432	432	431	432
u	432	432	432	432
u	431	432	431	432
u	432	432	430	755
r	219	565	232	21
r	217	230	272	21
r	220	108	228	21
r	225	21	244	21
r	264	22	170	21
r	147	21	57	21
r	344	21	172	21
r	65	21	45	21
r	65	21	68	21
r	64	21	35	21
r	66	21	21	21
r	65	21	21	21
r	323	21	21	21
r	21	21	21	21
r	21	21	21	21
r	21	21	21	21
r	20	21	21	21
r	21	21	21	21
r	21	21	21	21
r	21	21	21	21

Figure 8.5: Raw data for the top graph of Figure 3.8.

Execution	Conservative	Optimistic	Hybrid	Optimal
r	0	24691	1054	24691
r	807	24691	13017	24691
r	4113	24691	14836	24691
r	12808	24691	24691	24691
r	12808	24691	24691	24691
r	12808	24691	24691	24691
r	24691	24691	24691	24691
r	24691	24691	24691	24691
r	24691	24691	24691	24691
r	24691	24691	24691	24691
u	21586	14072	19099	9734
u	16592	9734	14107	9734
u	9757	9734	9761	9734
u	9757	9734	9761	9734
u	9757	9734	9761	9734
u	9757	9734	9761	9734
u	9757	9734	9761	9734
u	9757	9734	9761	9734
u	9757	9734	9761	9734
u	9757	9734	9761	24968
r	9757	16795	10208	24968
r	9766	21893	12072	24968
r	9837	24661	13924	24968
r	10405	24717	16701	24968
r	12891	24776	18819	24968
r	12915	24839	19945	24968
r	18701	24839	22757	24968
r	18701	24849	22757	24968
r	18701	24916	24000	24968
r	18701	24938	24984	24968
r	18742	24948	24984	24968
r	18742	24963	24984	24968
r	24987	24963	24984	24968
r	24987	24963	24984	24968
r	24987	24963	24984	24968
r	24987	24963	24984	24968
r	24987	24963	24984	24968
r	24987	24963	24984	24968
r	24987	24963	24984	24968
r	24987	24963	24984	24968

Figure 8.6: Raw data for the bottom graph of Figure 3.8.

Execution	Conservative	Optimistic	Hybrid	Optimal
r	150	720	157	160
r	155	10	285	146
r	160	9	97	146
u	293	185	323	289
u	253	292	291	289
u	292	291	290	289
r	150	598	28	353
r	148	221	145	145
r	146	75	148	146
u	290	186	288	290
u	291	291	291	291
u	292	291	290	290
r	146	519	147	146
r	145	178	145	144
r	145	93	147	144
u	291	185	290	291
u	292	292	290	290
u	293	291	290	291

Figure 8.7: Raw data for the top graph of Figure 3.9.

Execution	Conservative	Optimistic	Hybrid	Optimal
r	0	14946	238	350
r	103	14946	4022	350
r	737	14946	4292	350
u	63	350	350	350
u	63	350	350	350
u	63	350	352	350
r	350	9650	146	146
r	350	12923	354	350
r	164	14946	438	350
u	350	350	350	350
u	350	350	350	350
u	350	350	350	350
r	350	8376	350	350
r	350	11372	350	350
r	364	14946	438	350
u	350	350	350	350
u	350	350	350	350
u	350	350	350	350

Figure 8.8: Raw data for the bottom graph of Figure 3.9.

Set Size	100%-0%	90%-10%	80%-20%	70%-30%	60%-40%
0.1	8804	8928	9129	9332	9509
0.2	8043	8340	8711	9121	9465
0.3	7279	7723	8266	8876	9502
0.4	6528	7125	7852	8646	9365
0.5	5808	6568	7460	8442	9342
0.6	5063	5968	7030	8205	9346
0.7	4378	5397	6602	8001	9350
0.8	3690	4841	6195	7779	9280
0.9	3025	4285	5788	7562	9320
1	2463	3763	5320	7335	9370

Figure 8.9: Raw data for the graph of Figure 3.10.

Chunk Number	Region 1	Region 2
1	2515	5372
2	29373	5119
3	171737	5389
4	66297	3958
5	46188	3587
6	36871	3232
7	35392	3823
8	37789	5653
9	27667	3583
10	22900	3227
11	16301	4293
12	16664	5598
13	14218	4705
14	13884	4201
15	11260	3839
16	9770	5150
17	8906	3752
18	8089	3963
19	7971	3615
20	6157	3123

Figure 8.10: Raw data for the top graph of Figure 4.15.

Region Size	Region 1	Region 2
1	2515	5372
2	31888	10491
3	203625	15880
4	269922	19838
5	316110	22425
6	352981	25657
7	388373	28480
8	426162	34133
9	453829	37716
10	476729	40943
11	493030	45236
12	509694	50834
13	523912	55539
14	537796	59740
15	549056	63579
16	558826	68729
17	567732	72481
18	575821	76444
19	583792	80059
20	589949	83182

Figure 8.11: Raw data for the bottom graph of Figure 4.15.

Epoch Length	NAIVE	GROUPS	TS
10	0	0.2	1.28
100	0	1.4	1.28
1000	0	4.23	1.28
10000	0	12.19	1.28

Figure 8.12: Raw data for the top graph of Figure 4.24.

Epoch Length	NAIVE	GROUPS	TS
10	25	105	1.3
100	25	21.1	1.3
1000	25	12.9	1.3
10000	25	8.1	1.3

Figure 8.13: Raw data for the bottom graph of Figure 4.24.

Cache Size	NAIVE	GROUPS	TS
100	1.08	3.5	1.1
200	1.17	3.42	1.12
500	1.44	3.38	1.09
1000	2.21	3.3	1.12
5000	5.94	3.32	1.17
10000	14.6	3.2	1.2
20000	55.1	2.9	1.14
30000	102	2.6	1.17
40000	175	2.4	1.12

Figure 8.14: Raw data for the top graph of Figure 4.25.

# of groups	GROUPS
5	3.3
10	4.4
20	6.7
50	13.1
100	25.1
200	45.2

Figure 8.15: Raw data for the bottom graph of Figure 4.25.

	Linear	Exponential
CBR	20.27%	78.92%
RUC	8.54%	77.05%
HUC	21.10%	79.24%
HRCA	21.04%	78.18%

Figure 8.16: Raw data for the graph of Figure 4.26.

	Setup 1	Setup 2	Setup 3
CBR	0.00%	6.12%	64.30%
RUC	16.70%	7.82%	61.90%
HUC	17.91%	8.11%	67.20%
HRCA	17.56%	7.93%	67.10%

Figure 8.17: Raw data for the graph of Figure 4.27.

	Setup 1	Setup 2	Setup 3	Setup 4	Setup 5
CBR	0.28%	9.25%	90.47%	0.00%	47.50%
RUC	3.82%	8.91%	89.14%	-3.51%	20.03%
HUC	29.41%	9.33%	92.51%	14.74%	48.12%
HRCA	29.60%	10.13%	92.87%	13.61%	48.20%

Figure 8.18: Raw data for the graph of Figure 4.28.

	20.00%	50.00%	80.00%
CBR	40.74%	54.52%	58.53%
RUC	27.59%	29.63%	24.34%
HUC	41.25%	55.44%	59.65%
HRCA	41.27%	55.52%	59.58%

Figure 8.19: Raw data for the graph of Figure 4.29.

	Setup 1	Setup 2
CBR	9.75%	8.72%
RUC	7.91%	9.23%
HUC	9.77%	12.82%
HRCA	9.79%	12.85%

Figure 8.20: Raw data for the graph of Figure 4.30.

	System A	System B
CBR	9.32%	14.51%
RUC	7.64%	9.05%
HUC	8.25%	10.85%
HRCA	12.71%	17.06%

Figure 8.21: Raw data for the graph of Figure 4.31.

	System A	System B
CBR	7.10%	1.73%
RUC	1.42%	5.72%
HUC	4.03%	3.35%
HRCA	1.42%	3.33%

Figure 8.22: Raw data for the graph of Figure 4.32.

	System A	System B
CBR	7.69%	10.53%
RUC	8.74%	10.55%
HUC	9.62%	11.05%
HRCA	9.79%	11.58%

Figure 8.23: Raw data for the graph of Figure 4.33.

S	IRP	S	TPC-C	S	TPC-H
0.0002	0.0002	0.0020	0.0030	0.0001	0.1130
0.0003	0.0003	0.0040	0.0070	0.0003	0.1580
0.0008	0.0007	0.0060	0.0420	0.0006	0.1890
0.0018	0.0017	0.0080	0.1790	0.0015	0.2160
0.0033	0.0033	0.0100	0.3120	0.0029	0.2390
0.0068	0.0066	0.0120	0.3630	0.0059	0.3010
0.0133	0.0132	0.0150	0.4010	0.0117	0.3830
0.0200	0.0198	0.0170	0.4330	0.0176	0.4320
0.0267	0.0264	0.0210	0.4750	0.0235	0.4570

Figure 8.24: Raw data for the graph of Figure 5.6.

FLASH / RAM	IRP	TPC-C	TPC-H
2	0.981	0.888	0.996
4	0.963	0.783	0.991
6.67	0.937	0.654	0.984
10	0.907	0.519	0.976
13.33	0.877	0.415	0.968
33.33	0.707	0.191	0.908

Figure 8.25: Raw data for the graph of Figure 5.7.

FLASH / RAM	IRP	TPC-C	TPC-H
6	1.069	1.065	1.058
12	1.072	1.068	1.061
20	1.068	1.065	1.056
30	1.072	1.063	1.055
40	1.069	1.063	1.058

Figure 8.26: Raw data for the graph of Figure 5.8.

Workload	Inclusive	Exclusive	Lazy
IRP	1	1.19	1.14
TPC-C	1	1.3	1.16
TPC-H	1	1.2	1.14

Figure 8.27: Raw data for the graph of Figure 5.9.

FLASH / RAM	Inclusive	Exclusive	Lazy
6	354	349	351
12	346	341	343
20	338	335	336
30	326	323	324
40	310	307	308

Figure 8.28: Raw data for the graph of Figure 5.10.

FLASH / RAM	Inclusive	Exclusive	Lazy
6	1166	1385	927
12	851	1352	697
20	565	1334	528
30	435	1324	417
40	357	1317	344

Figure 8.29: Raw data for the graph of Figure 5.11 (a).

FLASH / RAM	Inclusive	Exclusive	Lazy
6	191	146	165
12	138	116	128
20	92	90	90
30	70	72	69
40	56	65	55

Figure 8.30: Raw data for the graph of Figure 5.11 (b).

SSD	Inclusive	Exclusive	Lazy
Samsung	1	1.8	0.93
Intel X25-M	1	1.06	0.91
Intel X25-E	1	0.84	0.95
FusionIO ioDrive	1	0.87	0.95

Figure 8.31: Raw data for the graph of Figure 5.12.

FLASH / RAM	Inclusive	Exclusive	Lazy
2	69.23	65.64	92.92
4	64.54	62.02	90.69
6.67	61.45	59.81	86.87
10	59.68	58.33	85.44
13.34	58.57	57.34	84.53
33.34	55.64	54.65	82.04

Figure 8.32: Raw data for the graph of Figure 5.13(a).

Scheme	Lookups	Updates	Total
Inclusive	27.62	32.06	59.68
Exclusive	27.56	30.77	58.33
Lazy	52.77	32.67	85.44

Figure 8.33: Raw data for the graph of Figure 5.13(b).

	Hit Ratio		HDD Reads	
Block Size	Inclusive	Lazy	Inclusive	Lazy
4K	0.28	0.28	23825	23785
8K	0.28	0.21	23566	49146
16K	0.32	0.18	22051	48581
32K	0.42	0.17	19021	45699
64K	0.55	0.17	14771	41515
128K	0.7	0.21	10622	36060

Figure 8.34: Raw data for the graph of Figure 5.14.

	λ		HDD Reads Ratio	
Block Size	TPC-C	TPC-H	TPC-C	TPC-H
4K	1.00	1.00	1.00	1.00
8K	1.03	1.13	1.87	1.85
16K	1.09	1.34	1.66	1.71
32K	1.10	1.39	1.59	1.59
64K	1.10	1.32	1.56	1.47
128K	1.12	1.27	1.56	1.33

Figure 8.35: Raw data for the graph of Figure 5.15.

	Hit Ratio		Time	
Dirtyness Ratio	All	Clean Only	All	Clean Only
0.1	0.19	723	0.19	672
0.4	0.19	855	0.19	684
0.7	0.2	953	0.17	662
0.9	0.19	1011	0.07	725

Figure 8.36: Raw data for the graph of Figure 5.16.

Input Size (elements x 10^6)	Input Size (MB)	HerMeS	HerMeS simple	NeXSort
1.2	30	4.81	6.14	47
2.5	61	9.9	11.5	93.3
5.06	130	19	24.7	173
10	270	37	46.3	384
17.6	455	63	78	679
28.5	740	110	132	1159
45.2	1175	174	203	1780
60.3	1565	238	296	2430
83	2175	392	505	3480

Figure 8.37: Raw data for the graph of Figure 6.9.

Main Memory (MB)	HerMeS	NeXSort
0.5	201	1978
5	150	1983
10	145	1976
20	143	1998
50	141	2017
75	140	2006
100	139	2028
200	137	2036

Figure 8.38: Raw data for the graph of Figure 6.10.

Input Size (MB)	Depth = 3	Input Size (MB)	Depth = 5
58	6.1	73	6.7
131	14.2	178	16.6
274	35.2	345	31.0
526	61.0	608	62.6
948	115.0	1162	125.0
1580	205.5	1659	188.0

Input Size (MB)	Depth = 7	Input Size (MB)	Depth = 9
59	4.9	66	5.5
158	14.8	221	18.6
370	35.0	629	55.6
776	74.0	1120	108.0
1172	119.0	1588	159.0
1600	168.0		

Figure 8.39: Raw data for the graph of Figure 6.11.

Key Length (bytes)	Time
5	7.0
10	7.7
15	9.1
20	10.0
30	12.5
40	14.9
50	19.5
60	23.2
70	26.9
80	29.0
90	31.6
100	35.1
120	41.1
150	50.8
180	60.7

Figure 8.40: Raw data for the graph of Figure 6.12.

	Magnetic	Flash	Both
HErMeS, System A	86	46	46
nexSort, System A	304	148	147
HErMeS, System B	201	180	140
nexSort, System B	785	600	550

Figure 8.41: Raw data for the graph of Figure 6.13.

Bibliography

[Agrawal *et al.*, 2008] Agrawal *et al.*, N. (2008). Design tradeoffs for ssd performance. In *ATC USENIX Annual Technical*.

[Ailamaki et al., 2002] Ailamaki, A., DeWitt, D. J., and Hill, M. D. (2002). Data page layouts for relational databases on deep memory hierarchies. *The VLDB Journal*, 11(3):198–215.

[Ajwani et al., 2008] Ajwani, D., Malinger, I., Meyer, U., and Toledo, S. (2008). Characterizing the performance of flash memory storage devices and its impact on algorithm design. In McGeoch, C., editor, *Proc. 7th Intern. Workshop on Experimental Algorithms (WEA)*, volume 5038 of *Lecture Notes in Computer Science*, pages 208–219, Provincetown, USA. Springer.

[AnandTech, 2008] AnandTech (2008). Intel X25-M SSD: Intel Delivers One of the World's Fastest Drives. `http://anandtech.com`.

[AnandTech, 2009] AnandTech (2009). The SSD Anthology: Understanding SSDs and New Drives from OCZ. `http://www.anandtech.com/storage/showdoc.aspx?i=3531`.

[Avila-Campillo *et al.*, 2002] Avila-Campillo *et al.*, I. (2002). XMLTK: An XML Toolkit for Scalable XML Stream Processing. In *PLANX 2002*.

[Axboe, 2009] Axboe, J. (2009). Flexible I/O Tester. `http://freshmeat.net/projects/fio`.

[Bansal and Modha, 2004] Bansal, S. and Modha, D. S. (2004). Car: Clock with adaptive replacement. In *FAST '04: Proceedings of the 3rd USENIX Conference on File and Storage Technologies*, pages 187–200, Berkeley, CA, USA. USENIX Association.

[Belady et al., 1969] Belady, L. A., Nelson, R. A., and Shedler, G. S. (1969). An anomaly in space-time characteristics of certain programs running in a paging machine. *Commun. ACM*, 12(6):349–353.

[Berglund, 2007] Berglund, A. (2007). XML Path Language (XPath) 2.0.

[Birrell et al., 2007] Birrell, A., Isard, M., Thacker, C., and Wobber, T. (2007). A design for high-performance flash disks. *SIGOPS Oper. Syst. Rev.*, 41(2).

[Black and Sleator, 1989] Black, D. L. and Sleator, D. D. (1989). Competitive algorithms for replication and migration problems. *Technical Report CMU-CS-89-201*.

[Boag, 2007] Boag, S. (2007). XQuery 1.0: An XML Query Language.

[Borodin et al., 1992] Borodin, A., Linial, N., and Saks, M. E. (1992). An optimal on-line algorithm for metrical task systems. *J. ACM*, 39(4).

[Bouganim et al., 2009] Bouganim, L., r Jnsson, B., and Bonnet, P. (2009). uflip: Understanding flash io patterns. In *CIDR*.

[Buneman et al., 2001] Buneman, P., Davidson, S., Fan, W., Hara, C., and Tan, W.-C. (2001). Keys for xml. In *WWW '01: Proceedings of the 10th international conference on World Wide Web*, pages 201–210, New York, NY, USA. ACM.

[Buneman et al., 2004] Buneman, P., Khanna, S., Tajima, K., and Tan, W.-C. (2004). Archiving scientific data. *ACM Trans. Database Syst.*, 29(1):2–42.

[Chawathe et al., 1996] Chawathe, S. S., Rajaraman, A., Garcia-Molina, H., and Widom, J. (1996). Change detection in hierarchically structured information. In *SIGMOD '96: Proceedings of the 1996 ACM SIGMOD international conference on Management of data*, pages 493–504, New York, NY, USA. ACM.

[Chen, 2009] Chen, S. (2009). Flashlogging: exploiting flash devices for synchronous logging performance. In *SIGMOD '09: Proceedings of the 35th SIGMOD international conference on Management of data*, pages 73–86, New York, NY, USA. ACM.

[Choi et al., 2000] Choi, J., Cho, S., Noh, S., Lyul, S., and Cho, Y. (2000). Analytical prediction of buffer hit ratios. *Electronics Letters*, 36(1):10–11.

[Chou and DeWitt, 1985] Chou, H.-T. and DeWitt, D. J. (1985). An evaluation of buffer management strategies for relational database systems. In *VLDB*.

[Chung et al., 2006] Chung, T.-S., Park, D.-J., Park, S., Lee, D.-H., Lee, S.-W., and Song, H.-J. (2006). System software for flash memory: A survey. In *EUC*.

[Cobena et al., 2002] Cobena, G., Abiteboul, S., and Marian, A. (2002). Detecting changes in xml documents. In *ICDE '02: Proceedings of the 18th International Conference on Data Engineering*, page 41, Washington, DC, USA. IEEE Computer Society.

[Consortium, 2007] Consortium (2007). The Universal Protein Resource (UniProt). *Nucleic Acids Res.*, 35(Database Issue):D193–197.

[Cormen *et al.*, 2001] Cormen *et al.*, T. H. (2001). *Introduction to Algorithms (Second Edition)*. MIT Press.

[D. E. Vengroff, 1994] D. E. Vengroff (1994). A Transparent Parallel I/O Environment. In *DAGS Symposium on Parallel Computation*.

[Denali, 2009] Denali (2009). NAND Forward Price Drops will Slow Significantly. `http://www.denali.com/wordpress/index.php/dmr/2009/07/16/nand-forward-prices-rate-of-decline-will`.

[Do and Patel, 2009] Do, J. and Patel, J. M. (2009). Join processing for flash ssds: remembering past lessons. In *DaMoN '09: Proceedings of the Fifth International Workshop on Data Management on New Hardware*, pages 1–8, New York, NY, USA. ACM.

[European Bioinformatics Institute, 2007] European Bioinformatics Institute (2007). EMBL Nucleotide Sequence Database.

[Fagin, 1977] Fagin, R. (1977). Asymptotic miss ratio over independent references. *J. Comput. Syst. Sci.*, 14:222–250.

[FusionIO, 2008] FusionIO (2008). Fusion-io Validates Groundbreaking Solid-State Storage Performance. `http://www.fusionio.com/PressDetails.php?id=49`.

[Gal and Toledo, 2005] Gal, E. and Toledo, S. (2005). Algorithms and data structures for flash memoies. *ACM Comput. Surv.*, 37(2):138–163.

[Ghanem, 1975] Ghanem, M. Z. (1975). Dynamic partitioning of the main memory using the working set concept. *IBM Journal of Research and Development*, 19(5):445–450.

[Graefe, 1993] Graefe, G. (1993). Query Evaluation Techniques for Large Databases. *ACM Comput. Surv.*, 25(2).

[Graefe, 2007] Graefe, G. (2007). The five-minute rule twenty years later, and how flash memory chenges the rules. *DAMON*.

[HP Labs, 2006] HP Labs (2006). Getting smart about data center cooling. `http://www.hpl.hp.com/news/2006/oct-dec/power.html`.

[Hu et al., 2009] Hu, X.-Y., Eleftheriou, E., Haas, R., Iliadis, I., and Pletka, R. (2009). Write amplification analysis in flash-based solid state drives. In *SYSTOR '09: Proceedings of SYSTOR 2009: The Israeli Experimental Systems Conference*, pages 1–9, New York, NY, USA. ACM.

[Jonathan Corbet, 2009] Jonathan Corbet (2009). The trouble with discard. `http://lwn.net/Articles/347511/`.

[Kim and Ahn, 2008] Kim, H. and Ahn, S. (2008). BPLRU: a buffer management scheme for improving random writes in flash storage. In *FAST*.

[Kim et al., 2002] Kim, J., Kim, J. M., Noh, S. H., Min, S. L., and Cho, Y. (2002). A space-efficient flash translation layer for compactflash systems. *Transactions on Consumer Electronics.*

[Kim et al., 2000] Kim, J. M., Choi, J., Kim, J., Noh, S. H., Min, S. L., Cho, Y., and Kim, C. S. (2000). A low-overhead high-performance unified buffer management scheme that exploits sequential and looping references. In *OSDI'00: Proceedings of the 4th conference on Symposium on Operating System Design & Implementation*, pages 9–9, Berkeley, CA, USA. USENIX Association.

[Knuth, 1998] Knuth, D. (1998). *The Art of Computer Programming, Volume 3: Sorting and Searching, 2nd edition.* Addison-Wesley.

[Koltsidas et al., 2008] Koltsidas, I., Müller, H., and Viglas, S. D. (2008). Sorting hierarchical data in external memory for archiving. *Proc. VLDB Endow.*, 1(1):1205–1216.

[Koltsidas and Viglas, 2008] Koltsidas, I. and Viglas, S. D. (2008). Flashing up the storage layer. *Proc. VLDB Endow.*, 1(1):514–525.

[Lee and Moon, 2007] Lee, S.-W. and Moon, B. (2007). Design of flash-based DBMS: An in-page logging approach. In *SIGMOD*.

[Lee et al., 2009] Lee, S.-W., Moon, B., and Park, C. (2009). Advances in flash memory ssd technology for enterprise database applications. In *SIGMOD '09: Proceedings of the 35th SIGMOD international conference on Management of data*, pages 863–870, New York, NY, USA. ACM.

[Lee et al., 2008] Lee, S.-W., Moon, B., Park, C., Kim, J.-M., and Kim, S.-W. (2008). A case for flash memory ssd in enterprise database applications. In *SIGMOD '08: Proceedings of the 2008 ACM SIGMOD international conference on Management of data*, pages 1075–1086, New York, NY, USA. ACM.

[Lee et al., 2007] Lee, S.-W., Park, D.-J., Chung, T.-S., Lee, D.-H., Park, S., and Song, H.-J. (2007). A log buffer-based flash translation layer using fully-associative sector translation. *Trans. on Embedded Computing Sys.*

[Leventhal, 2008] Leventhal, A. (2008). Flash storage memory. *Commun. ACM*, 51(7):47–51.

[Li et al., 2008] Li, X., Da, Z., and Meng, X. (2008). A new dynamic hash index for flash-based storage. In *WAIM '08: Proceedings of the 2008 The Ninth International Conference on Web-Age Information Management*, pages 93–98, Washington, DC, USA. IEEE Computer Society.

[Li et al., 2009] Li, Y., He, B., Luo, Q., and Yi, K. (2009). Tree indexing on flash disks. In *ICDE '09: Proceedings of the 2009 IEEE International Conference on Data Engineering*, pages 1303–1306, Washington, DC, USA. IEEE Computer Society.

[Mattson et al., 1970] Mattson, R. L., Gecsei, J., Slutz, D. R., and L., T. I. (1970). Evaluation techniques for storage hierarchies. *IBM Systems Journal*, 9(2):78–117.

[Megiddo and Modha, 2003] Megiddo, N. and Modha, D. S. (2003). Arc: A self-tuning, low overhead replacement cache. In *FAST '03: Proceedings of the 2nd USENIX Conference on File and Storage Technologies*, pages 115–130, Berkeley, CA, USA. USENIX Association.

[Micron Technology Inc., 2008] Micron Technology Inc. (2008). Micron Collaborates with Sun Microsystems to Extend Lifespan of Flash-Based Storage, Achieves One Million Write Cycles. http://www.micron.com/about/news/pressrelease.aspx?id=5F432D92EFA2B68E.

[Müller et al., 2008] Müller, H., Buneman, P., and Koltsidas, I. (2008). Xarch: archiving scientific and reference data. In *SIGMOD '08: Proceedings of the 2008 ACM SIGMOD international conference on Management of data*, pages 1295–1298, New York, NY, USA. ACM.

[Myers, 2007] Myers, D. (2007). On the use of nand flash memory in high-performance relational databases. Master Thesis. MIT.

[Narayanan et al., 2009] Narayanan, D., Thereska, E., Donnelly, A., Elnikety, S., and Rowstron, A. (2009). Migrating server storage to ssds: analysis of tradeoffs. In *EuroSys*.

[Nath and Gibbons, 2008] Nath, S. and Gibbons, P. B. (2008). Online maintenance of very large random samples on flash storage. *Proc. VLDB Endow.*, 1(1):970–983.

[Nath and Kansal, 2007] Nath, S. and Kansal, A. (2007). FlashDB: Dynamic Self-Tuning Database for NAND Flash. In *IPSN*.

[Ng et al., 1995] Ng, R., Faloutsos, C., and Sellis, T. (1995). Flexible and adaptable buffer management techniques for database management systems. *IEEE Trans. Comput.*, 44(4):546–560.

[Nyberg et al., 1995] Nyberg, C., Barclay, T., Cvetanovic, Z., Gray, J., and Lomet, D. (1995). Alphasort: a cache-sensitive parallel external sort. *The VLDB Journal*, 4(4):603–628.

[Ou et al., 2009] Ou, Y., Härder, T., and Jin, P. (2009). Cfdc: a flash-aware replacement policy for database buffer management. In *DaMoN '09: Proceedings of the Fifth International Workshop on Data Management on New Hardware*, pages 15–20, New York, NY, USA. ACM.

[Park et al., 2006] Park, S.-y., Jung, D., Kang, J.-U., Jinsoo, K., and Lee, J. (2006). CFLRU: a replacement algorithm for flash memory. In *CASES*.

[Rajkumar et al., 1997] Rajkumar, R., Lee, C., Lehoczky, J., and Siewiorek, D. (1997). A resource allocation model for qos management. In *RTSS '97: Proceedings of the 18th IEEE Real-Time Systems Symposium*, page 298, Washington, DC, USA. IEEE Computer Society.

[Ramamurthy and DeWitt, 2005] Ramamurthy, R. and DeWitt, D. J. (2005). Buffer-pool aware query optimization. In *CIDR*, pages 250–261.

[Ross, 2008] Ross, K. A. (2008). Modeling the performance of algorithms on flash memory devices. In *DaMoN '08: Proceedings of the 4th international workshop on Data management on new hardware*, pages 11–16, New York, NY, USA. ACM.

[Silberstein and Yang, 2004] Silberstein, A. and Yang, J. (2004). Nexsort: Sorting xml in external memory. In *ICDE '04: Proceedings of the 20th International Conference on Data Engineering*, page 695, Washington, DC, USA. IEEE Computer Society.

[Soundararajan et al., 2008] Soundararajan, G., Chen, J., Sharaf, M. A., and Amza, C. (2008). Dynamic partitioning of the cache hierarchy in shared data centers. *Proc. VLDB Endow.*, 1(1):635–646.

[Soundararajan et al., 2009] Soundararajan, G., Lupei, D., Ghanbari, S., Popescu, A. D., Chen, J., and Amza, C. (2009). Dynamic resource allocation for database servers running on virtual storage. In *FAST '09: Proccedings of the 7th conference on File and storage technologies*, pages 71–84, Berkeley, CA, USA. USENIX Association.

[Stoica et al., 2009] Stoica, R., Athanassoulis, M., Johnson, R., and Ailamaki, A. (2009). Evaluating and repairing write performance on flash devices. In *DaMoN '09: Proceedings of the Fifth International Workshop on Data Management on New Hardware*, pages 9–14, New York, NY, USA. ACM.

[Stone et al., 1992] Stone, H. S., Turek, J., and Wolf, J. L. (1992). Optimal partitioning of cache memory. *IEEE Trans. Comput.*, 41(9):1054–1068.

[Stonebraker, 1981] Stonebraker, M. (1981). Operating system support for database management. *Commun. ACM*, 24(7):412–418.

[StorageSearch.com, 2006] StorageSearch.com (2006). Flash Memory vs. Hard Disk Drives - Which Will Win? `http://www.storagesearch.com/semico-art1.html`.

[StorageSearch.com, 2008] StorageSearch.com (2008). Flash vs DRAM Price Projections - for SSD Buyers. http://www.storagesearch.com/ssd-ram-flash%20pricing.html.

[Sun Microsystems., 2008] Sun Microsystems. (2008). The Solaris ZFS filesystem.

[TGDaily, 2008] TGDaily (2008). FusionIO - the power of 1000 hard drives in the palm of your hand. http://tgdaily.com.

[Thiébaut et al., 1992] Thiébaut, D., Stone, H. S., and Wolf, J. L. (1992). Improving disk cache hit-ratios through cache partitioning. *IEEE Trans. Comput.*, 41(6):665–676.

[TigerDirect.com, 2009] TigerDirect.com (2009). OCZ Core Series 64GB SATA II 2.5" Solid State Drive. http://tigerdirect.com.

[Tsirogiannis et al., 2009] Tsirogiannis, D., Harizopoulos, S., Shah, M. A., Wiener, J. L., and Graefe, G. (2009). Query processing techniques for solid state drives. In *SIGMOD '09: Proceedings of the 35th SIGMOD international conference on Management of data*, pages 59–72, New York, NY, USA. ACM.

[Tufte and Maier, 2001] Tufte, K. and Maier, D. (2001). Aggregation and Accumulation of XML Data. *IEEE Data Eng. Bull.*, 24(2).

[Wang et al., 2003] Wang, Y., DeWitt, D., and Cai, J.-Y. (2003). X-diff: an effective change detection algorithm for xml documents. In *Data Engineering, 2003. Proceedings. 19th International Conference on*, pages 519–530.

[Wanxia Wei and Mengchi Liu and Shijun Li, 2004] Wanxia Wei and Mengchi Liu and Shijun Li (2004). Merging of XML Documents. In *ER*.

[Westbrook, 1992] Westbrook, J. (1992). Randomized algorithms for multiprocessor page migration. *DIMACS*, 7:135–150.

[Wikipedia, 2008] Wikipedia (2008). Hybrid drive. http://en.wikipedia.org/wiki/Hybrid_drive.

[Wikipedia, 2009] Wikipedia (2009). Solid State Drive. http://en.wikipedia.org/wiki/Solid-state_drive.

[Wu et al., 2007] Wu, C.-H., Kuo, T.-W., and Chang, L. P. (2007). An efficient b-tree layer implementation for flash-memory storage systems. *Trans. on Embedded Computing Sys.*, 6(3).

[Zeinalipour-Yazti et al., 2005] Zeinalipour-Yazti, D., Lin, S., Kalogeraki, V., Gunopulos, D., and Najjar, W. A. (2005). Microhash: an efficient index structure for fash-based sensor devices. In *FAST'05: Proceedings of the 4th conference on USENIX Conference on File and Storage Technologies*, pages 3–3, Berkeley, CA, USA. USENIX Association.

[Zheng and Larson, 1996] Zheng, L. and Larson, P.-A. (1996). Speeding up external mergesort. *IEEE Trans. on Knowl. and Data Eng.*, 8(2):322–332.

[Zhou et al., 2004] Zhou, P., Pandey, V., Sundaresan, J., Raghuraman, A., Zhou, Y., and Kumar, S. (2004). Dynamic tracking of page miss ratio curve for memory management. *SIGPLAN Not.*, 39(11):177–188.