TECHNISCHE UNIVERSITÄT KAISERSLAUTERN

A study of I/O behavior in Database Systems

by

Vítor Uwe Reus

A thesis submitted in partial fulfillment for the degree of
Master of Science

in the
Fachbereich Informatik
AG Datenbanken und Informationssysteme

September 2014

Declaration of Authorship

I, Vítor Uwe Reus, declare that this thesis titled, 'A study of I/O behavior in Database Systems' and the work presented in it are my own. I confirm that:

- This work was done wholly or mainly while in candidature for a research degree at this University.
- Where any part of this thesis has previously been submitted for a degree or any other qualification at this University or any other institution, this has been clearly stated.
- Where I have consulted the published work of others, this is always clearly attributed.
- Where I have quoted from the work of others, the source is always given. With the exception of such quotations, this thesis is entirely my own work.
- I have acknowledged all main sources of help.
- Where the thesis is based on work done by myself jointly with others, I have made clear exactly what was done by others and what I have contributed myself.

Signed: _____

Date: _____

The most exciting phrase to hear in science, the one that heralds the most discoveries, is not "Eureka!" but "That's funny..."

- Isaac Asimov

TECHNISCHE UNIVERSITÄT KAISERSLAUTERN

Abstract

Fachbereich Informatik
AG Datenbanken und Informationssysteme

Master of Science

by Vítor Uwe Reus

This thesis is a study about I/O behaviour in database systems. There has been little advance in terms of I/O delay times in magnetic disk for the past ten years. This is due to physical limitation of the mechanical components in magnetic disks. Flash devices are becoming more affordable. This new kind of storage offers a lower latency than traditional hard disk, but it still is a bottleneck in a system with a multi-core CPU and only one drive. The objective of this work is to understand the limitations of today's storage technology and its impact in recovery times in an ARIES System. We are also interested in the database recovery, since it is an I/O intensive operation, it is desirable to understand what factors influence its performance. We identified correlations between several aspects of a database system, and created a model that help to explain the effects in I/O. We have found some problems such as double caching and scattered calls for page propagation, that can fail the whole system, therefore another contribution of this work are the suggestions for better implementation of propagation control within the buffer manager.

Acknowledgements

I would like to express my very great appreciation to my advisor Caetano Sauer. His guidance, patience and innumerous advices were crucial to the development of this thesis. Not only is he an exceptional scientist but also a great friend that never misses a chance to amaze us with his brilliance.

I would like to offer my special thanks to my family. Thank you, Margret Alice Reus and Paulo Nestor Reus, for all the care with which you treated me and for all support you always gave me. I also would like to thank Johanna Luise Voget, who was always an example of determination and good will. I would also like to offer my special thanks to Paula Albrecht Corrêa, who even being far away, was crucial by bringing joy during tough times.

I would like to express my very great appreciation to Prof. Dr. Theo Härder, for the opportunity to work in his research group and realize this thesis efforts under the Technical University of Kaiserslautern.

Finally, I am particularly grateful for the support and good times given by my friends. This includes friends made during this course, and old friends that always remained in contact. You were crucial for the motivation, strenght and willpower to finish this work.

Contents

List of Figures

List of Tables

Dedicated to Margret and Paulo

Chapter 1

Introduction

1.1 Motivation

This thesis is a study about I/O behaviour in database systems. As seen in Figure 1.1, there has been little advance in terms of I/O delay times in magnetic disk for the past ten years. This is due to physical limitation of the mechanical components in magnetic disks.

To decrease the impact of high latency times, database systems use a memory hierarchy [1]. It is desirable to keep hot data in faster buffers. Hot data is data that is frequently accessed. The faster the buffers, the more expensive and smaller they are, as seen in Figure 1.2. Therefore, there is a price-performance trade-off when dealing with I/O.

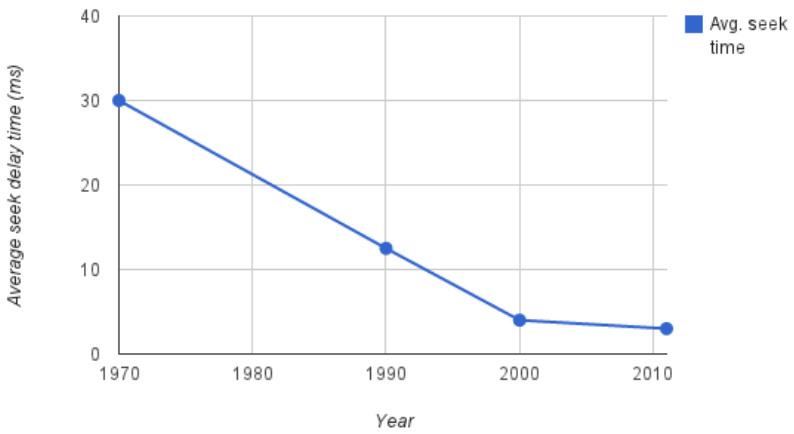

FIGURE 1.1: Evolution of Technical Characteristics of Magnetic Disks

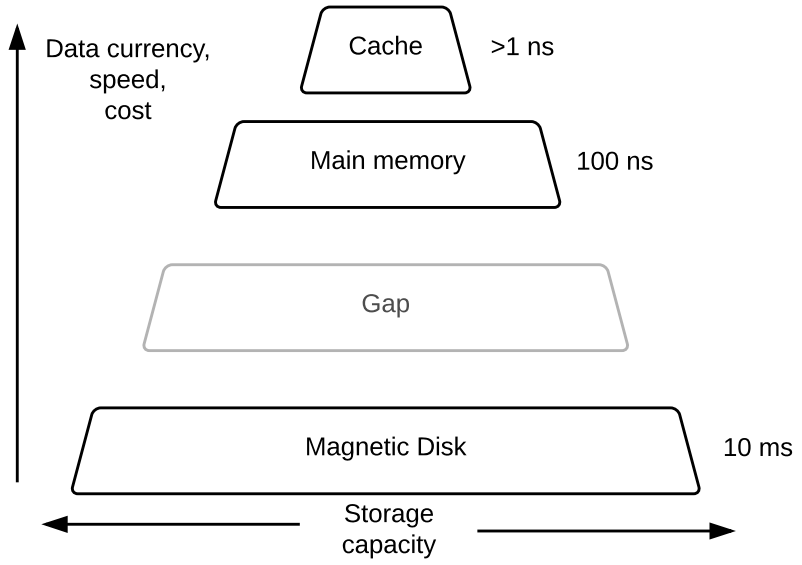

FIGURE 1.2: Evolution of Technical Characteristics of Magnetic Disks

The memory hierarchy of Figure 1.2 also shows a gap between Magnetic disk and Main memory. That is because access time from main memory is five orders of magnitude faster than magnetic disk. Even if adding a flash device, this difference is still three orders since typical delay times for such devices are 0.1ms.

Flash devices are becoming more affordable. This new kind of storage offers a lower latency than traditional hard disk, but it still is a bottleneck in a system with a multi-core CPU and only one drive. Multi-core and parallel programming allows a high processing capacity, widening even more the gap between CPU and storage. Therefore it is important to be careful not to neglected I/O while having high a transaction throughput.

1.2 System Recovery

Several database systems use the write-ahead logging technique to provide transaction atomicity and durability. The log is ideally stored in a different device as the database, therefore we refer as database device the disk that contains the database, and log device, the disk that contains the log. For a system based on write-ahead logging, the log device speed may limit the transaction throughput. This is because transactions can only commit after the system writes its changes in the log device. This shows that bottleneck for this kind of database system can be in I/O, more specifically, to the log device.

Additionally to write-ahead logging, a database system may use the ARIES method [2] to guarantee atomicity and durability. This method performs system recovery in three phases: Log analysis, REDO and UNDO. The log analysis pass gathers information

for the next phases such as dirty pages and active transactions during crash time from the log. REDO recovers all dirty pages by repeating log history. During UNDO, all modifications of active transactions during crash time should be undone, performing the opposite logged operations.

Since REDO needs to recover all dirty pages, it is expected that REDO performs at least one random read per page. This is because it should first fetch the page to be recovered from the database to then apply the transformations. These random reads ends up being the dominating cost of the REDO phase. Therefore, if a system has 16GB of buffer space with 8KB dirty pages occupying half of it, the total number of dirty pages will be around one million. Now, if this system fails, the REDO phase will need to perform one million random reads. Assuming a magnetic disk with four milliseconds delay time for random reads, this system will spend at least one hours in REDO phase.

This direct relation between recovery time and dirty ratio raises the desire to keep the number of dirty pages low during transaction processing. There are two ways of doing this. The first one is to keep propagating dirty pages during transaction processing, but this is, again, limited to the delay times of the database device. The second is to lower the page dirtying speed paying with a lower transaction throughput. A lower transaction throughput will have a lower dirtying speed, making it possible to clean more pages.

There is a tradeoff between transaction throughput and recovery times. One must decide if it is better to have a high transaction throughput or if it is acceptable to have large periods of down time. There is also the question if is it worth raising processing capacity using multi-core precessing to raise transaction throughput if there will be a higher number of dirty pages.

There are three kinds of I/O in an ARIES-based database system: recovery, the log and page cleaning. This work focuses more in the last two components. As explained before, REDO should perform random reads in the database for every page that was dirty at time of crash. Therefore to optimize recovery I/O we are interested in minimizing the number of dirty pages at runtime with page cleaning.

Page cleaning performs random writes to the database device. This kind of I/O is the most expensive if the database device is a lower cost high capacity device such as magnetic disks. The database device is also accessed for read during normal transaction processing, which may disrupt page cleaning.

The only moment when the log device performs read and writes is during recovery. During normal transaction processing, there are only write operations performed to the log. This allows a higher write throughput to this device since it does not compete

with read. This work investigates the I/O during normal system execution to study the effects of database, and log device I/O in transaction processing.

1.3 Contribution

The objective of this work is to understand the limitations of today's storage technology and its impact in recovery times in an ARIES System. Since database recovery is an I/O intensive operation, it is desirable to understand what factors influence its performance. This thesis also provide guidelines for the implementation of propagation control within the buffer management.

As seen in Section 1.1, I/O delay should be a major concern in a database system. The only way to minimize it, is to understand the inner works and details of the I/O subsystem. This includes the impact of different storage types such as magnetic disks and flash devices, the influence of using different file systems, the inner workings of the buffer manager, and even the role of the operational system kernel.

Chapter 2 will provide the theory background needed for this work. This work will use an ARIES system to perform experiments, therefore this chapter addresses the basic of recovery and ARIES. Details about dirty page propagation and its relation to recovery times are also discussed there.

Chapter 3 describes the experimental setup, methodologies, and tools used. This chapter also describes the Shore-MT system and methods for measuring transaction and disk throughput. The hypothesis and motivations for the experiments are also explained there.

Chapters 4 and 5 describe the experiments and its results. The first analyses the transaction throughput and its correlation to I/O. The second experiment measures the influence of different storage types and the effect in dirty pages propagation.

Chapter 6 discusses the findings. It analyses problems such as double caching and the Operating System Kernel interface and present solutions to them. This chapter also discuss the problems found with respect to the size of the database log.

Finally, Chapter 7 concludes this work. It summarizes all the findings and provides guidelines for future work in this area.

Chapter 2

Background Theory

2.1 ARIES recovery method

ARIES is a no-force, steal transaction recovery method. It guarantees Atomicity and Durability [3] using write-ahead logging (WAL), which logs any database element change before writing it to disk.

Different log record types represent distinct DBMS actions such as the Update, Commit, Abort, and Compensation log records. The set of all log records reflects the serialized actions of the DBMS at normal transaction processing. Since this log is sequential, every log record contains an always growing unique Log Sequence Number identifier, or LSN for short.

Normal transaction processing writes Commit and Abort log records to indicates transaction status at runtime. The recovery management uses this set of log records during recovery to know which transactions were active during the time of failure. This set of log records provide atomicity: all changes from transactions that do not have a Commit log record are undone.

Normal transaction processing writes Update log records to the log before any change to a page is applied. They describe the physical operation that was done to a page, such as deleting, adding or modifying a page record. This provides durability of the changes because, if a failure occurs, the actions of committed transactions can be redone if they did not make it to stable storage. Update log records also provides atomicity as failed transaction changes that were propagated to disk can be undone.

Every database page has a pageLSN which refers to the last Update log record written for that page. This means that whenever an update to this page is made, its pageLSN

changes. A page in the buffer is dirty if its pageLSN differs from the value in stable storage. The system should therefore propagate its contents to stable storage.

WAL allows a no-force policy because changes does not need to be propagated to the database device, only to the log. Therefore transactions do not wait for propagation which can occur at any moment. This means that dirty pages are allowed to remain in the volatile buffer during normal processing, which makes REDO necessary: if a failure occurs, the updates to the dirty pages are lost. Therefore, the recovery process must reconstruct these pages using the log information.

To minimize the time taken to recover, the system can periodically create a checkpoint. This avoids rescanning the whole logfile during log analysis. A checkpoint contains the active transactions and dirty pages at the time of the checkpoint. With this information, recovery does not need to analyse the log from the beginning, since it summarizes the needed information for the next phases that before the Checkpoint.

Checkpoints can be implemented in two ways: locking the whole database to ensure the active transactions and dirty pages do not change while taking the checkpoint, or to take a Fuzzy checkpoint. The Fuzzy checkpoint writes a Begin Checkpoint log record indication when the checkpoint began. The checkpoint then start gathering the active transactions and dirty pages table. The End Checkpoint record writes all the information, and represents a fuzzy set of the active transactions and dirty pages. Log analysis must then complement the checkpoint information using the log records in between the two Fuzzy checkpoint records to validate the checkpoint.

Crash recovery occurs in three phases: log analysis, REDO and UNDO. Since ARIES is a no-force method, recovery needs the REDO phase to repeat all actions that normal transaction processing executed at runtime but were not propagated to disk at the time of failure. The steal policy requires an UNDO phase, which undoes actions of transactions that did not commit, but which pages has been stolen, propagating its changes to disk. The log analysis phase will identify the active transactions and dirty pages in the buffer pool at time of crash, so the REDO and UNDO phases know which what actions they should take.

Log analysis is the first phase of recovery. It scans the log forward starting from log beginning or the last checkpoint if there are any. It will create the table of active transactions and dirty pages. These tables will be empty if starting from the beginning of the log, or will be the ones stored at the Checkpoint. Every time a Begin Transaction log record is found, the log analysis phase adds this transaction to the transactions table. If a commit or abort log record is found, the transaction is removed. Also, for every update to a page, log analysis will add it to the dirty pages table.

REDO will recover all changes logged by the Update log records for all the dirty pages. This phase scans the log forward starting from the minimal pageLSN from the dirty pages table. For every Update log records, if its LSN is larger than the pageLSN, the page should be recovered. If a page that needs recovery is not in the buffer, REDO fetches it from the disk. If REDO performs any change, a new Update log record is created and the LSN is applied to the page, so the system can also recover from a crash in the REDO phase. When REDO ends, the database is in the same state as in the time of crash, meaning that even changes from aborted transactions are redone. This is the repeating history [4] principle of ARIES: restore the system to the exact state of crash time, then UNDO the active transactions at crash time.

In the UNDO phase, all active transactions at time of failure should be aborted and its changes undone. For all the active transactions in the Transactions Table, the changes are undone starting at the last LSN from the transaction and then going backwards. To traverse the log backwards, the previousLSN log record field is used, which indicates the previous LSN of the current transaction. UNDO is logical, For every undone change, UNDO writes a Compensation Log Record, or CLR, describing this change, to make it possible to recover from a crash in this phase. The CLR are redone during the REDO phase in case of a crash during UNDO occurs, so the action is not repeated in case of repeated restarts. At the end of UNDO, the database is in a consistent state.

2.2 Page cleaning

Page cleaning is the propagation of dirty pages. Cleaning occurs in three places: Page stealing, log scavenging and background page cleaning. When a page is propagated, the recovery LSN of that page is set to the current pageLSN. This means that up to recovery LSN log changes from dirty pages are propagated to disk.

When fetching a page from disk into a full buffer, page replacement should select one of the buffered pages as a substitution victim. If the selected page is dirty, its contents should be propagated to disk in order to persist the changes. This is the reason for page propagation during page stealing.

Log scavenging also requires page propagation. New transactions need log space in order to write its respective log records and commit. When the log device is full, log scavenging deletes the older section of the log to make room for new log records. It is only possible to delete log records that will not be needed by recovery process. These are the log records that describe changes that were already propagated to stable storage,

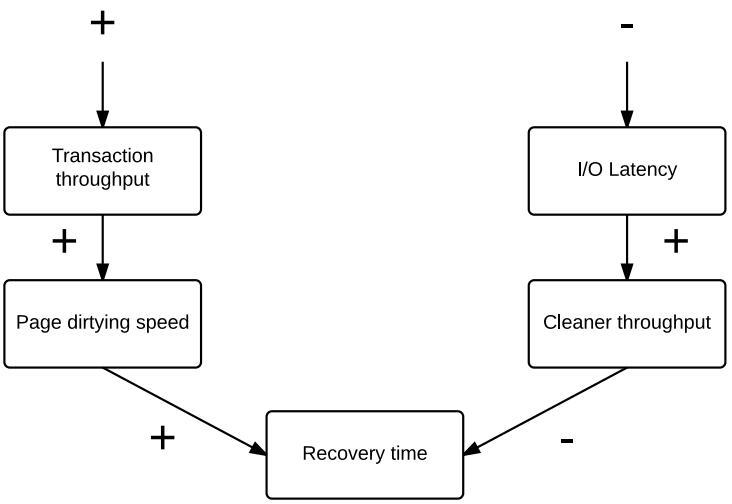

FIGURE 2.1: Race between cleaning and dirtying speeds

that is, the minimum recovery LSN from the dirty pages. Therefore, propagating the pages with lower recovery LSN allows log scavenging to delete more log segments.

It is safe to delete log records referring to clean pages, that is, the ones that have its content already reflected in stable storage. If a segment of the log is to be deleted, all dirty pages with recovery LSN smaller or equal than the last LSN in the log segment to be deleted should be cleaned to ensure Durability. Chapter 6 discusses more in log scavenging.

Background cleaning is a service that continuously propagates dirty pages. This minimizes recovery effort since REDO needs to restore fewer pages. It also allows lower latency for page stealing and log scavenging, since less work will need to be done in these critical situations. Chapters 4 and 5 study this balance between page propagation and page dirtying speed.

2.3 Recovery times

As seen in the introduction of this work, the recovery process might take several hours to finish depending on the system setup. In a short transactions scenario, REDO time dominates the recovery cost because UNDO chains will be short. In this case, the main work of recovery will be to redo the actions that were not propagated to disk. This means that recovery time will be proportional to the number of dirty pages at the time of crash [5]. This is because, as explained in Section 2.1, REDO must, at least once, fetch from the database each page that was dirty at time of crash. If the database is stored in a magnetic drive, the random read will cause a high I/O delay.

Large buffers contribute to higher recovery time. When the whole database, or at least the working set of the application fits in the buffer, it is much less likely page stealing. This decreases page cleaning caused by page substitution, therefore more dirty pages are likely to be in the buffer when a failure occurs. On the other hand, there will be less read activity, making room for more background cleaning. This shows a race between cleaning and dirtying pages seen in Figure 2.1. Chapter 4 will study this race condition.

A smaller buffer requires more page substitution, but more transactions must wait for read I/O, therefore there will be less opportunity for cleaning. Chapter 5 studies this equilibrium in depth.

Chapter 3

Experimental Setup

3.1 Shore-MT

The Shore-MT storage manager [6] is used in the experiments. It supports the basic ARIES method. The Shore-Kits [1] component, released with Shore-MT, implements the TPC-C benchmark used by the experiment. Shore-MT is a highly scalable multi-threaded version of the SHORE (Scalable Heterogeneous Object REpository) Storage Manager. Shore-MT scalability allows throughput increase up to 24 threads, which is the same amount of threads in CPU.

Our experiments will provide a better understanding of the I/O behaviour in Shore-MT, allowing the implementation of better alternatives for propagation control. Our focus will be on the page cleaner and the log manager.

Shore-MT divides the log into eight segments to facilitate log scavenging. A segment qualifies for scavenging if there is no dirty page with recovery LSN smaller than the last LSN of a segment. That means that all changes described by this log segment are already propagated to the database, thus it can be safely deleted. If a log scavenging wants to delete a partition that does not qualify, it should propagate all dirty pages with recovery LSN smaller than the last LSN of this segment.

Propagation of dirty pages is done by the page cleaner daemon. There are two main components in the cleaner, one manager thread, and multiple writer threads. The manager thread is responsible for coordinating the cleaning process, and the writer threads are responsible for writing the pages to the device. It is possible to have several writer threads, but since magnetic drives and flash does not take advantage of concurrent writes [7], we set the number of writer threads to one as the experiment only runs with

[1] `https://bitbucket.org/shoremt`

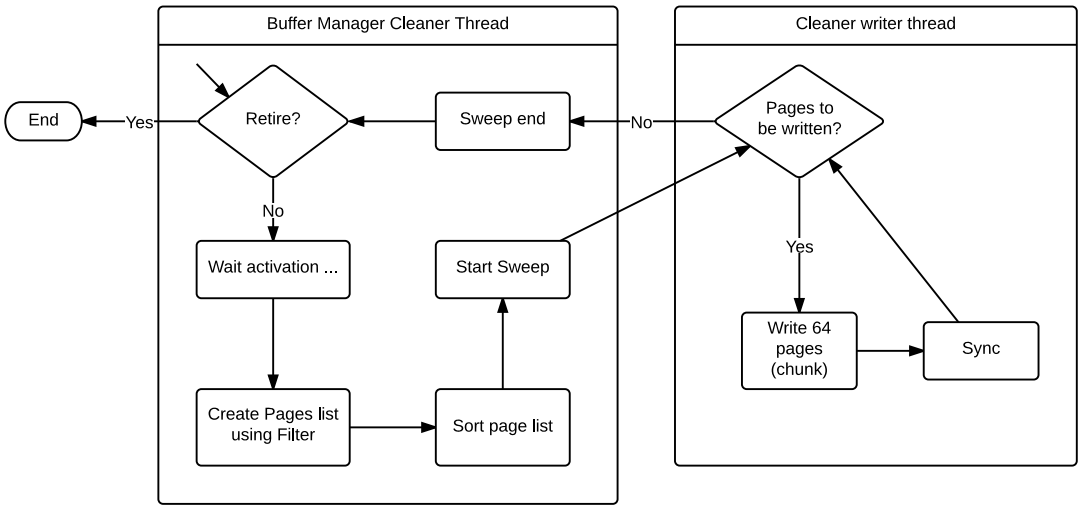

FIGURE 3.1: Cleaner flowchart

one database device. RAID configurations or multiple devices might take advantage of this option.

Figure 3.1 describes the main steps of the cleaning process. The cleaner starts by checking for the *retire flag*, which indicates termination. If set to true, cleaning should stop. Otherwise, waits for an activation signal. The cleaner is activated by events sent by other components such as the transaction and log managers. We have also extended the cleaner to activate based in configurable timeouts. Section 3.1.1 explains this extension.

The manager thread constructs a list of pages to be cleaned after it receives the activation signal. A page filter defines the selection criteria for the pages. For each page in the buffer, the cleaner asks the filter if the page qualifies for propagation. The default filter behaviour in Shore-MT is primary dictated by log scavenging. If the log size is small, and the dirty page ratio is more than 1/8 of the total number of pages, then all dirty which are not marked as *hot* hot pages qualify for propagation. If the dirty page ratio is larger than 3/4 of the total number of pages, the hot pages qualifies as well to prevent pool overfilling. If the log starts to grow, the cleaner tailors propagation to benefit log scavenging, therefore the oldest dirty non hot pages qualifies for cleaning. If the log becomes too big, also the hot pages from the oldest segment qualifies. Table 3.1 summarizes this complex behaviour.

After page selection, the manager thread sorts the list according to the page ID. This will provide a better probability of performing sequential writes, since the page ID translates directly to a page position in the database file. The sorted list is then encapsulated within a control object, which handles communication between manager and writer threads. Then, the manager thread signals the writer threads to start and waits for them to

	Case	Page qualifies if	Urgent (hot pages qualifies)
Empty log	Segments ≤ 2 Dirty ratio $< 1/8$	–	–
Normal log	Segments ≤ 2 Dirty ratio $\geq 1/8$	Dirty Not hot	Dirty ratio $> 3/4$
Full log	Segments >2	Dirty Not hot LSN $<$ Oldest segment	Segments > 4

TABLE 3.1: Shore default filter behaviour

finish. A *cleaning sweep* is defined as one iteration of this while process, from activation of the manager thread until completion of all writer threads.

The only job of the writer thread is to perform the actual writes. It will pool one chunk of 64 pages from the control object and submit it to the I/O manager. We also extended this behaviour by adding an *fsync* invocation after every chunk write. Section 6.1.1 explains the reason for this.

If there are no more pages to write, the writer threads stop and notify the manager thread. This is the end of the cleaning sweep. The manager thread then checks again for the *retire flag* to verify if it should wait for a new activation or terminate.

3.1.1 Cleaner Policies

We implemented novel cleaner policies to better understand the cleaner effects. The default cleaner policy selects pages based on a complex filter. The cleaner is then activated based on external calls and according to the dirty ratio. A modular cleaner policy system that allows the customization of this behaviour in a modular architecture was implemented.

All code related to page filters and activation criteria was removed to create a pluggable policy. A policy object contains all information for the cleaner behaviour, and can be selected at runtime. Backwards compatibility was also maintained creating the *normal policy*, that has the original Shore-MT cleaner behaviour.

As seen in Figure 3.2, the main components of a policy are its name, the waiting rules, the event activation rules, and the page filter. The name is primarily used for debugging and as policy selection option in runtime. The page filter dictates which pages should be cleaned. The event activation rules work together with the waiting rules to start sweep.

There are two waiting rules implemented as basis for any policy: *Passive flag* and *timeout*. The timeout is the number of miliseconds that the cleaner should wait between

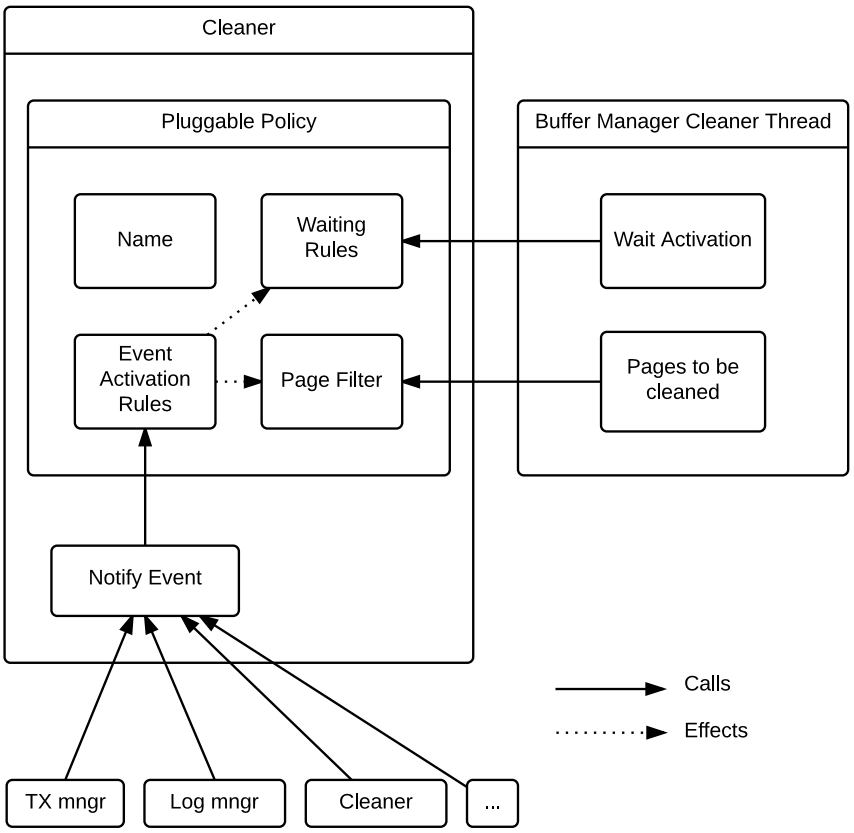

FIGURE 3.2: New modular cleaner architecture for novel policies

checking if it was externally activated. The passive rule defines if the cleaner should start only if activated by an external event. Values used for the aggressive policy are one second timeout and not passive. This means that the aggressive policy will start sweeping again at maximum one second after the last sweep end.

The waiting rules are also extensible. This means that new settings can be implemented by overriding the default `waitActivation` method. This provides a flexible framework to add more waiting rules to new cleaner policies.

The page filter of the policy will return what pages qualify for propagation. The aggressive policy, for example, selects all dirty pages from the buffer. The page filter can, however, change its behaviour according in order to the environment to optimize cleaning. Complex filter rules, such as the default cleaner policy depicted in Table 3.1 can also be expressed.

Originally, the Shore-mt cleaner does not have information about what caused activation. Therefore, an event-oriented activation system was added to the cleaner. That way, the caller can provide useful hints so a policy can decide whether or not the cleaner should be activated.

The cleaner events system can also define what kind of filter should be used. An event triggered by the log manager describing the need of log scavenging, for instance, can result in the creation of a filter that selects only pages from old log partitions. This is different from the current implementation where the cleaner tries to guess the Propagations need without knowledge of the events that caused activation. That way, the cleaner can work in a more precise way fulfilling the needs of the components that requires cleaning action.

Events can also affect waiting rules. The most evident effect is the signalling to end the wait. Currently, all policies cease the waiting when any event arrives. But this can be extended to change the passive flag, or waiting times, or any other user-defined behaviour.

3.2 Methodology

While Shore-MT runs the experiments, the `iostat` and `mpstat` system tools collects I/O and CPU usage. This will provide detailed information about system load. We also collect information from the Shore-MT, log such as the number of dirty pages, transactions and cleaner events during execution.

Shore-MT uses periodic checkpoints which save the table of dirty pages. It is possible to use these checkpoints but this measure would be imprecise since time between checkpoints is higher than the desired time precision. It is desirable to have a finer granularity page flush measure.

We modified the system to write a log record for every page write. As a small optimization, these log records are written in batches. For the experiments, a single page flush log record is generated for every 200 page writes.

Cleaner events were also captured using the log. Two new log record types were created, one for start and other for end. When the cleaner begins or ends a sweep, it inserts the respective log record. Such data allows us to identify the occurrence of event occurrence synchronized with experiment execution.

The `loginspect` tool recovers information from the log. It is a separate application created using the Shore-MT log manager to scan the entire log and returns a text file that summarizes system events. This tool is executed after the end of the experiment over the resulting log from transaction processing. For each log record of interest, the program outputs a new line with information such as page flushes, number of dirty pages,

beginning and end of transaction, beginning and end of cleaner seep, and beginning of checkpoint.

Events observed in the recovery log have, however, no time informatino attached. To provide some sense of time interval between events, we implemented a *tick log record*, which is an empty log record inserted every second by a system daemon.

Using tick log records, we are able to aggregate events at one-second intervals. It is possible, for instance, to compute the number of pages dirtied or cleaned per second. The experiments in Chapter 4 and 5 will make use of this facility to analyse various aspects of system behaviour.

3.3 Experiments motivation

As previously studied [5], the higher the number of dirty pages in the buffer pool at the time of failure, the higher will be the recovery time in case of failure. The higher the number of dirty pages, the higher will the REDO effort be, since this phase needs to recover all information that was not propagated to disk. There are several factors that may influence the dirty page ratio. The ones covered in this work are transaction throughput, I/O delay, buffer size, and maximum log size.

In a scenario of updating transactions, a higher transaction throughput, leads to a higher page dirtying speed. Section 4.1 will study this direct relation. In order to measure the effect of transaction throughput on the ratio of dirty pages, we vary the number of threads executing the benchmark in Shore-MT. The expected result is that the higher the number of threads, the higher the transaction throughput, therefore, the higher the dirtying speed. To measure the dirtying speed, we analyse the average number of dirty pages in buffer during execution.

The first part of this study will also analyse the relation between I/O delay and dirty pages ratio. Since page cleaning produces mainly random I/O operations, the lower the I/O delay, the higher the cleaner throughput, while with a lower I/O delay, it is expected that the average number of dirty pages is going to decrease. To verify this hypothesis, experiments with different storage devices and file systems were executed. This includes hard drive, flash, and we also study different file systems such as log-structured file system Ext2 and also raw device data access with no file system. The motivation for using a log-structure file system is that this kind of storage is optimized for write only activity. At the end, we compare the advantages of each one.

In the first experiment, 100% of the working set fits in memory. This is a realistic scenario, since memory prices are going cheaper, and in-memory databases are becoming common [8]. The advantage of having this configuration is that transactions do not need to fetch pages from disk, meaning that the database device will have no read activity. This provides the opportunity to perform much more write activity which the cleaner can exploit. Therefore, a large buffer implies a more idle disk, which provides a higher cleaner throughput that potentially lowers the dirty page ratio.

Another aspect of having a larger buffer is that transaction throughput is not limited by disk throughput since all page accesses are absorbed by the buffer pool. Therefore, a larger buffer provides a higher transaction throughput. But, as previously pointed, a higher transaction throughput causes a higher dirty page ratio.

The second part of this study tries to better understand the relation between buffer size and dirty pages. A lower buffer size will require database reads in order to fetch pages that are not in the buffer, therefore competing for I/O resources. Transaction throughput decreases, because transactions must wait for the disk to fetch pages, therefore lowering the dirtying speed. A lower buffer size also causes more page propagation because of page substitution, as explained in Section 2.2. Chapter 5 studies all these interactions.

The third aspect that influences dirty page ratio is the log size. As explained in Chapter 2, log scavenging needs to clean pages with low LSN in order to be able to throw older log segments away. Further execution of new transactions must wait for this cleaning process, which disrupts normal transaction processing, since they now need to wait for free log space. Therefore, log scavenging lowers transaction throughput, which should lower the dirty ratio.

We encounter some problems to measure the effect of log sizes on dirty pages because of the way Shore-MT implements log scavenging. The page filter effects page propagation depending on log size, but the log manager fails when waiting too much time for empty log space. Section 6.2 discusses this issue.

Other problems observed in Shore-MT include the double caching effect. This problem made it difficult to measure real device usage. We expect that this work will help to introduce the basic concepts of I/O and help the proper implementation and measuring of page cleaning.

Chapter 4

Experiment 1: Dirty Ratio

This chapter analyses the Shore-MT cleaner to understand what influences the dirty pages count. Two aspects will be studied: the effect of transaction throughput to the dirty pages count and the influence of I/O delay that different devices have.a

We ran the TPC-C benchmark [9] as implemented in Shore-Kits with initial scale factor 32, which yields an initial database size of 4.16GB. The hardware consists of a dual Xeon X5670 CPUs having 24 threads. The used operating system is a 64-bit Linux. The experiments uses a 256GB Samsung SSD 840 Pro flash device for logging. We use a 1TB Seagate ST1000VX000 for the database device for most experiments except on the last one which uses another SSD for the database device.

4.1 Transaction throughput

We first investigate the relation between transaction throughput and dirty pages count. A high transaction throughput is normally desired for any database system, since that way more requests can be handled by second. It might, however have negative impact in system recovery. Several transactions modifying data causes the page dirtying speed to raise.

With a high transaction throughput, more pages are modified by second, that is, the page dirtying speed raises. There will come a moment where the dirtying speed is higher than the device capacity to clean pages. When this occurs, the number of dirty pages rapidly grows. Such scenario is likely to happen when using large buffers, since transaction processing will then be limited by CPU or log device instead of database device.

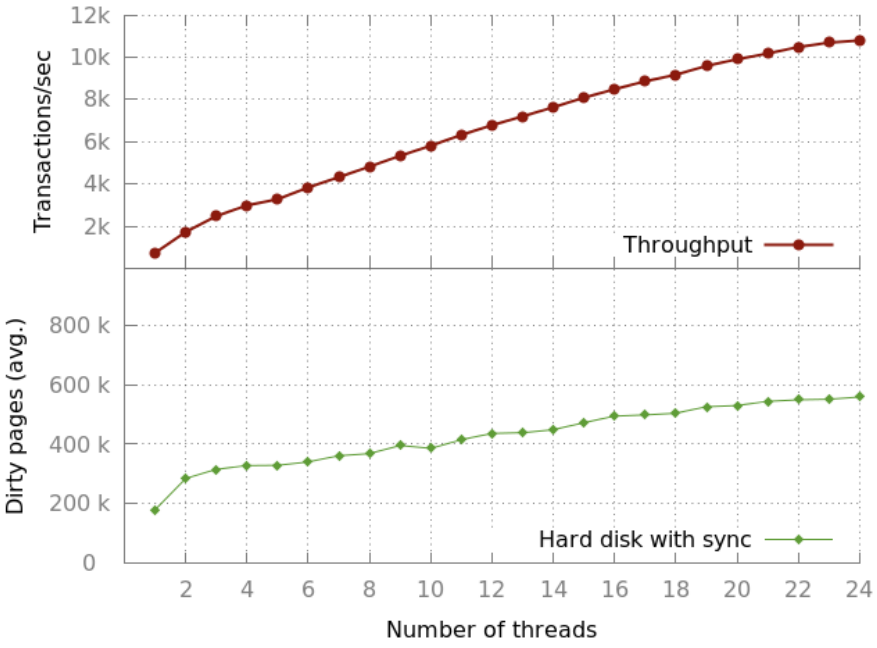

FIGURE 4.1: Average dirty ratio with different throughputs

We run an experiment to analyse the impact of transaction throughput in the number of dirty page. The TPC-C benchmark runs for ten minutes using a 1TB Seagate ST1000VX000 hard drive. The rest of configuration is as described in the beginning of this chapter. We vary the number of used threads to simulate various transactions throughputs. We also had to modify Shore-MT using `fsync` calls to ensure that the database device was really being used, so we could measure the real I/O delay impact. More about this fix is discussed in Section 6.1.

Figure 4.1 shows the results of the experiment. The upper section shows that the higher the number of threads, the higher the transaction throughput, thus, we were able to vary the transaction throughput by changing the hardware parallelism. The lower section confirms the hypothesis that the higher the transaction throughput the higher the average number of dirty pages. Therefore, the higher the transaction throughput, the higher will be the recovery time, since more random reads will have to be made during recovery.

4.2 I/O Delay

To analyse the influence of the storage system, we use the same configuration as above and fix the number of threads to 24, but this time the benchmark runs for one hour. Three different storage system for the database device will then be compared. We use the same 1TB Seagate ST1000VX000 hard drive with two different formats. First we

use block devices using `/dev/sd*`, wich provides access to raw data in the device, and then we compare formatting the device as NILFS. We also use another 256GB Samsung SSD 840 Pro for the database device when running benchmarks for flash storage.

4.2.1 Magnetic drives

For the first case, we use the HDD as a block device. Figure 4.2 shows the number of dirty pages aggregated by each second of the experiment, as measured by analysing the log. The *Checkpoint dirty count* line shows the number of dirty pages as measured by the checkpoint log records. The *Real dirty count* line shows the number of dirty pages as measured by logging each page flush. There is not much difference between this two measures for this experiment.

Figure 4.2 also shows that the cleaner does not retire immediately after ending the experiment. Normal transaction processing ends at 3600 seconds, as expected since the experiment should run for 1 hour, but the cleaner activity continues until 4400 seconds, when the *Sweep ended* event occurs. This is because the cleaner writer threads only check for retire after having written all the requested pages, as previously explained with Figure 3.1. One could argue that this is an implementation decision for graceful shut-down, since writing all dirty pages synchronizes the buffer with the stable storage, so the system does not need to recover on next start-up. This is, however, not true, since the pages to be cleaned are selected at the beginning of the sweep, therefore during the current sweep other pages will become dirty. The pages that became dirty during the current sweep will not be in the to-be-cleaned list because this list does not change during the sweep, therefore these pages will not be cleaned.

This also raises the question of whether it is worth to keep an unchanging list of pages to be cleaned. The cleaner is obviously unable to handle the amount of requested pages to be cleaned, making one single sweep more than the duration of the whole experiment. Also, the pages selected for cleaning at the beginning of the sweep might not be a good option at the middle of the sweep, where there are much more dirty pages options, such as pages that are in a more sequential pattern. A filter that selects only a limited number of pages could fix this *big sweep* problem.

Figure 4.3 shows the page flushes aggregated by each second of the experiment. There is clearly an irregular page flushing activity caused by the device. Since page cleaning causes random writes, the database can not keep a constant page flushing speed.

Figure 4.4 shows the write and read speeds for the log and database devices at each second of the experiment. It shows again the irregular database write speed in a magnetic

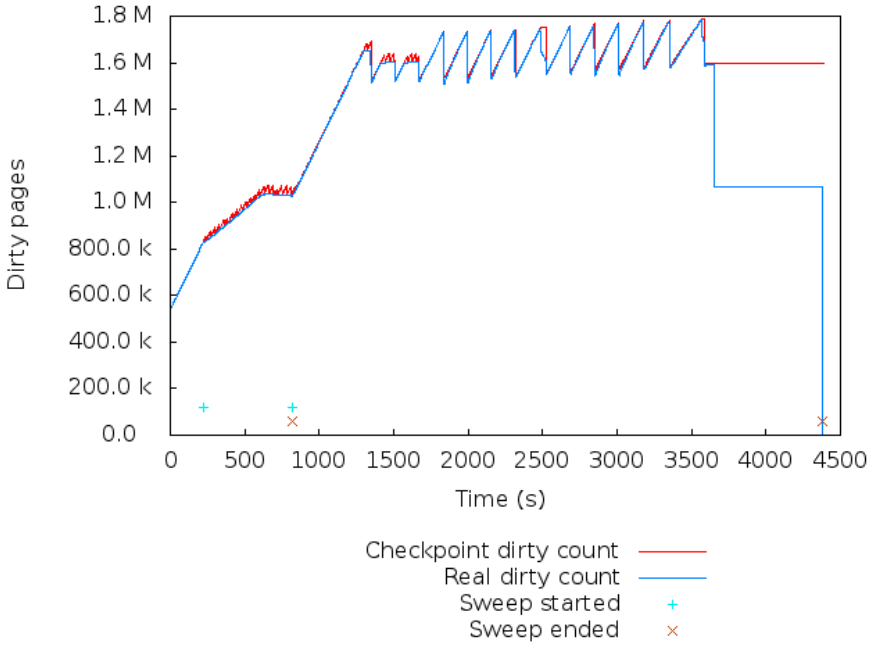

FIGURE 4.2: Number of dirty pages for HDD

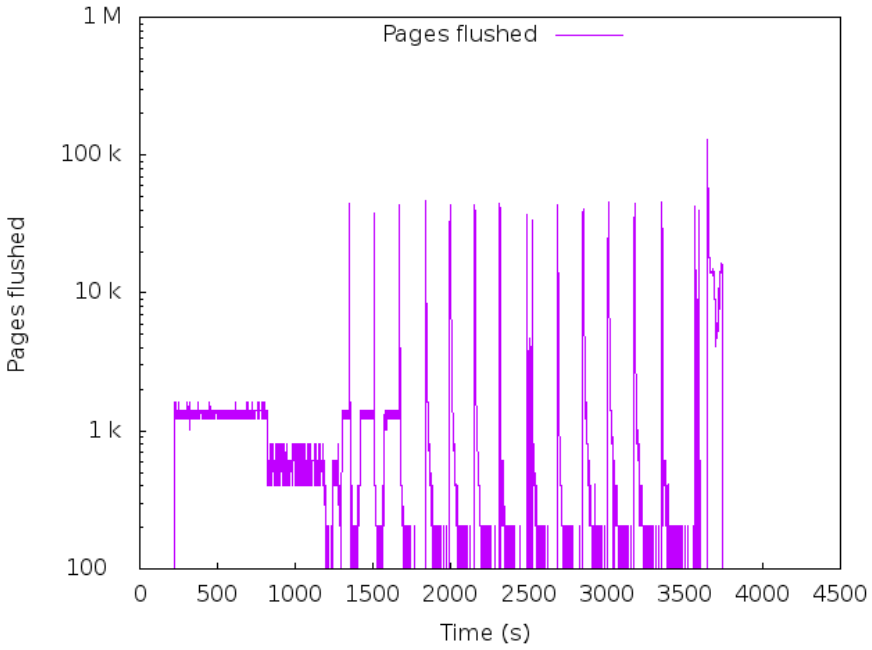

FIGURE 4.3: Page flushes for HDD

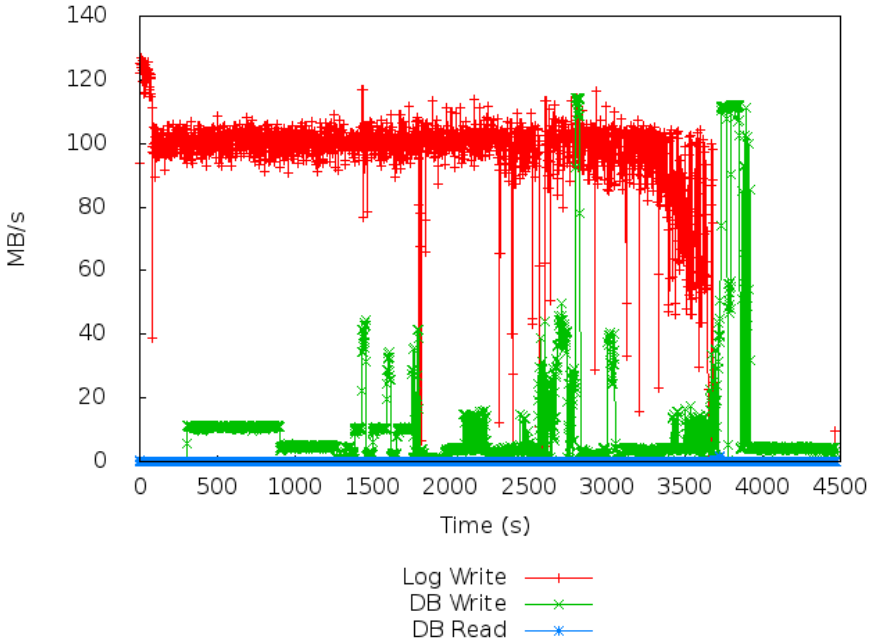

FIGURE 4.4: I/O status for HDD

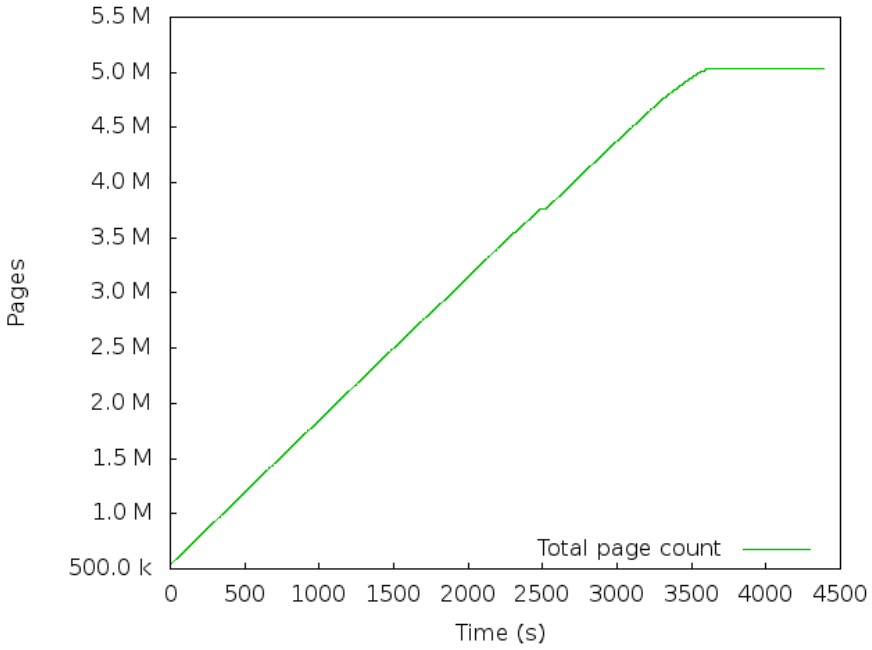

FIGURE 4.5: Total page count for HDD normal

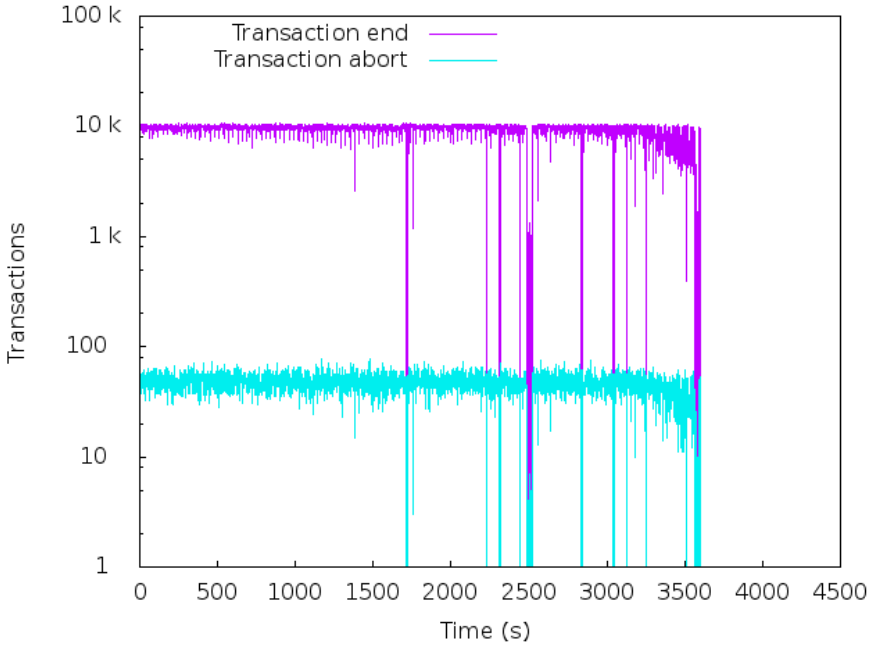

FIGURE 4.6: Transaction throughput for HDD normal

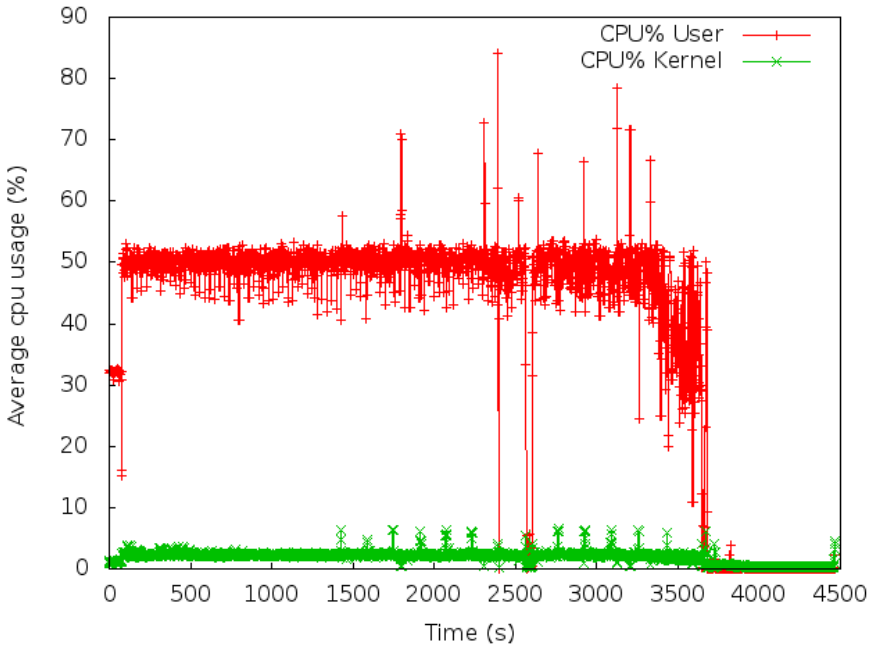

FIGURE 4.7: CPU status for HDD

drive. The database device speed is mostly low, but with peaks of 40MB/s. Since the cleaning is random, the HDD is not able to provide a constant cleaning speed. At the end there is an over 100MB/s activity, which is the cleaner continuing to work. This occurs after system shut-down,

The writing speed of the log is more or less constant at 100MB/s because it has a strong correlation with transaction throughput, which is constant in TPC-C. The log device activity starts, however, to decrease at the end of the experiment. The reason for that is because the log device is a 256GB SSD device formatted as Ext2. After one hour of writing activity, writing speed starts to decrease because the log device starts to degrade. If the experiment is performed again, write performance starts as bad as it ended in the last experiment and continues to decreases even more. Figure 4.23 shows the result in I/O activity after running several experiments without reformatting the log device. Reformatting the log device recovers the system performance. We tried using Ext4 without journalling but the same results occur. The exact reason for this degradation is unknown to us, an hypothesis is that the file system becomes fragmented, since formatting the device restores I/O throughput. A separate study may be conducted to further investigate this issue.

Since there is a strong correlation between log device write bandwidth and transaction throughput, the degradation of the log device directly affects it. Figure 4.6, which shows the number of committed and aborted transactions per second, shows this effect: whenever the log device activity lowers, the transaction throughput lowers as well. Because of the nature of TPC-C, which also inserts more records in the database, and since the transaction throughput is somewhat stable during the experiment, the total number of pages constantly grows. At the end, the database size is 10 times larger than the initial, as seen in Figure 4.5.

At last, Figure 4.7 shows the average CPU utilization by all 24 threads for each second of the experiment. It shows that the log device has a strong correlation with transaction throughput. The average CPU utilization is at half the maximum capacity, therefore CPU is not the bottleneck in this experiment. This image also shows the effect of the log device degradation: the CPU usage drops exactly at the same rate as the log device. Since none of the resources was saturated, that is, CPU was around 50% and the log device, which maximal bandwidth is 200MB/s was at 100MB/s, and the database device does not influence transaction throughput since all the working set of the application fits in the buffer, the bottleneck for this experiment might be lock contention, since TPC-C workload can cause concurrent access to the pages.

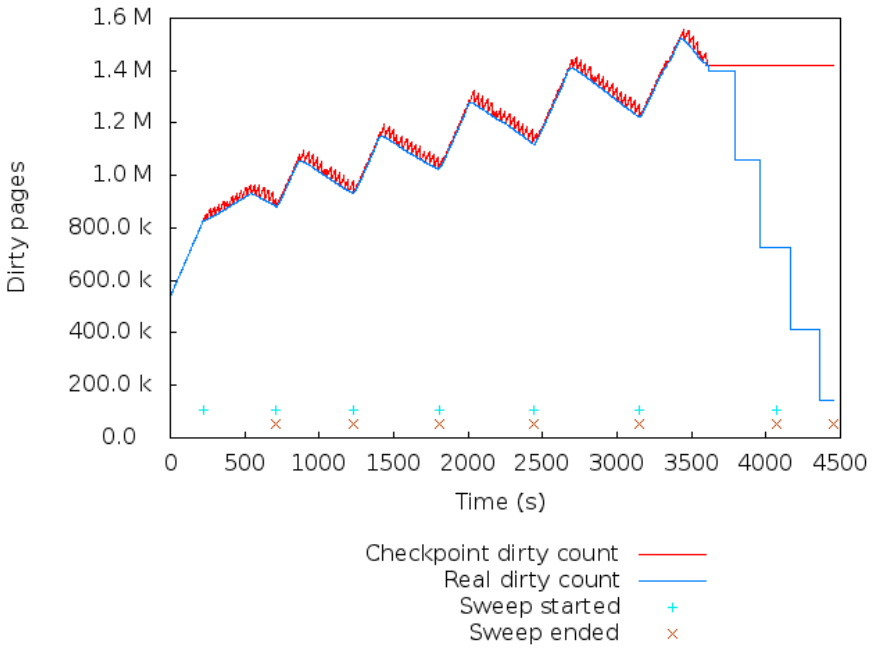

FIGURE 4.8: Number of dirty pages for NILFS

4.2.2 Log-structured File Systems

The second I/O experiment analyses the impact of using a log-structured file system. This experiment uses The New Implementation of a log-structured File System [10](NILFS). Figure 4.8 indicates a better cleaner throughput using this kind of file system. Thats confirmed with a much more constant number of page flushes per second, as seen in Figure 4.9, compared with the Ext2 file system page flushes, depicted in Figure 4.3. This is because the buffer holds 100% of the working set, hence there is more write than read activity, which is the scenario that log-structured file systems are optimized for. Figure 4.10 shows that there is a more constant writing speed, so the system can keep cleaning more pages.

Observing Figure 4.8 again, a sawteeth pattern in the dirty pages starts to form. These are the moments where the page cleaner starts and end, as seen in the *Sweep Start* and *Sweep End* events. This figure also shows that the time to perform one sweep becomes higher over time. That is because the total size of the database grows linearly in TPC-C, as seen on the previous experiment.

Also, Figures 4.11 and 4.12 shows that the Ext2 degradation still occurs here, since it uses the same log device as before.

The disadvantage of using this kind of file system is a higher physical storage usage. In this one hour experiment, the final logical size of the database was 30GB, whereas the

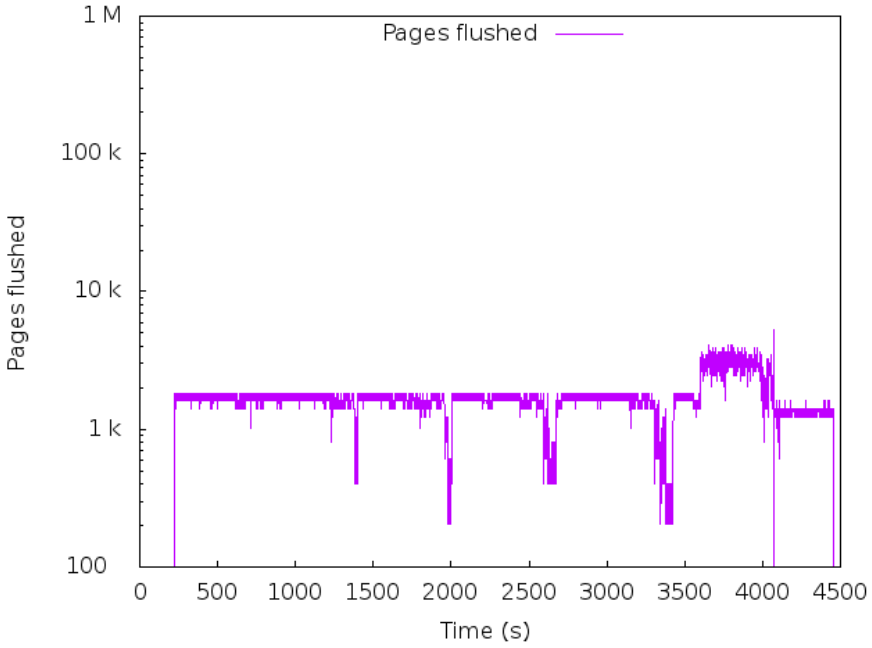

FIGURE 4.9: Page flushes for NILFS

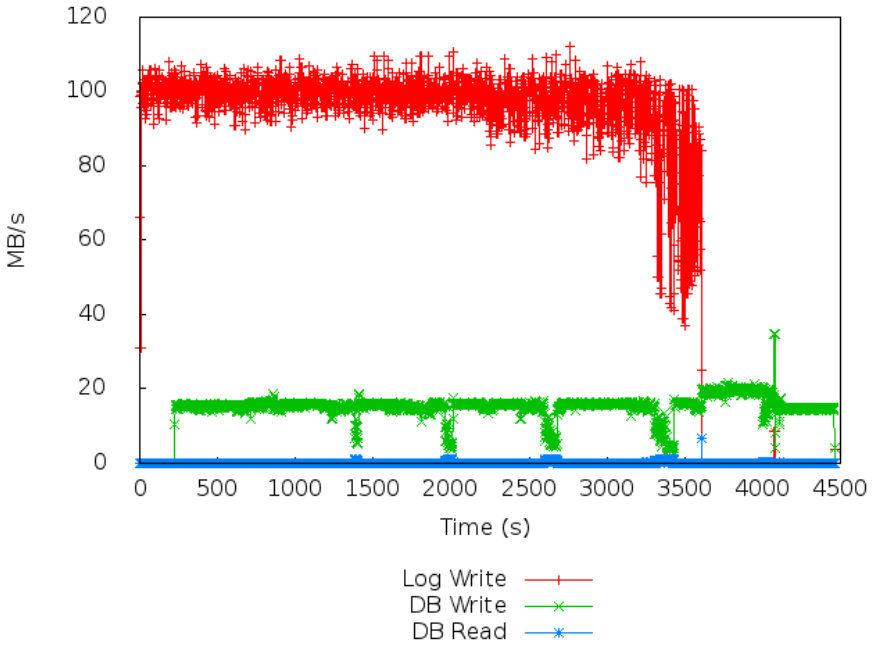

FIGURE 4.10: I/O status for NILFS

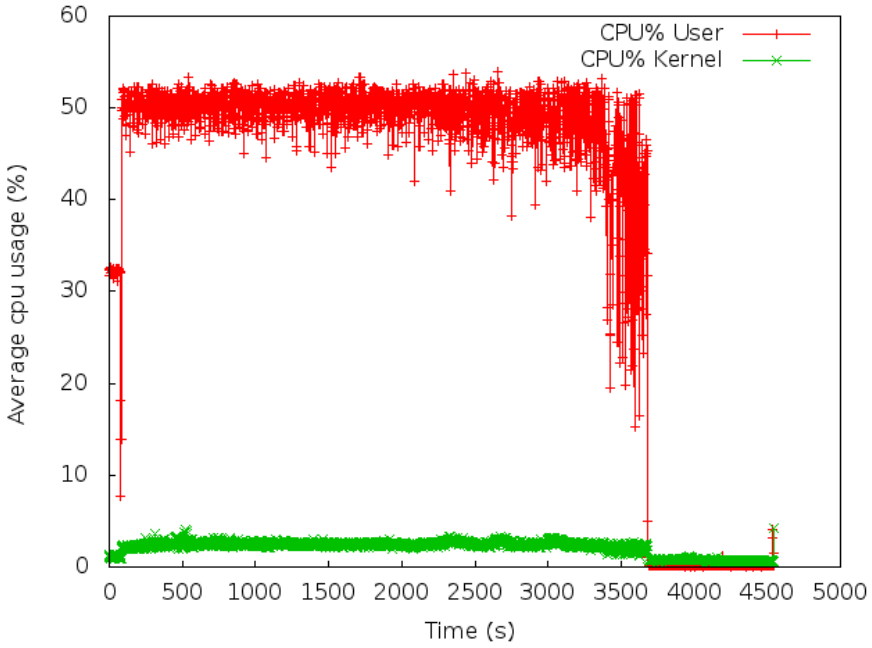

FIGURE 4.11: CPU status for NILFS

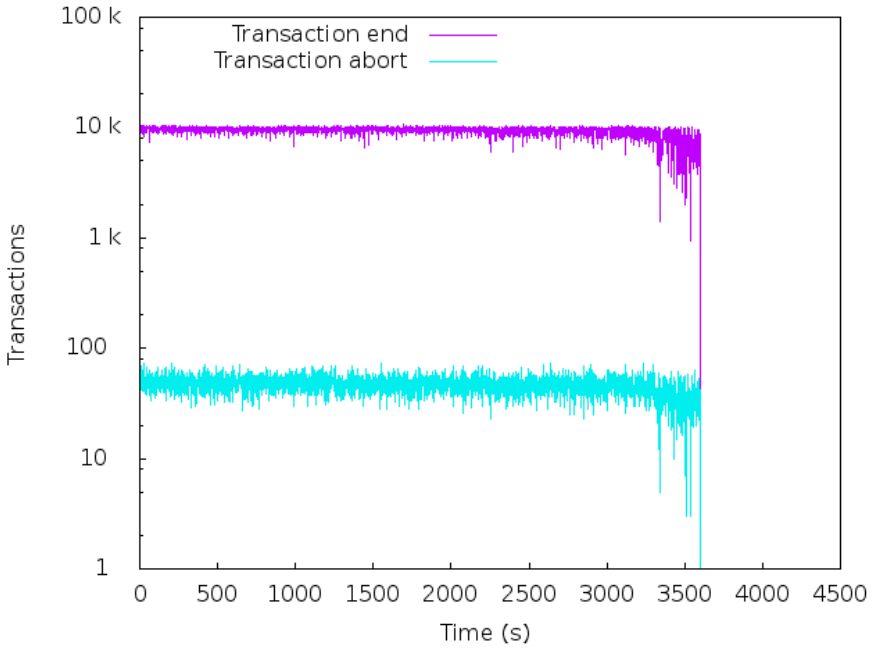

FIGURE 4.12: Transaction throughput for NILFS

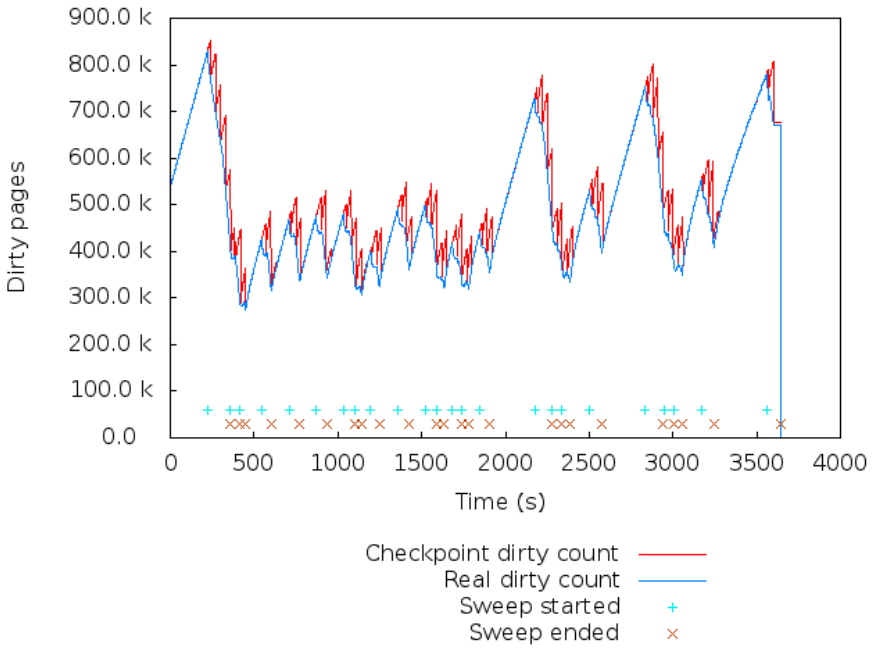

FIGURE 4.13: Number of dirty pages for flash

physical usage of the device was twice as large: 59GB. Another disadvantage of log-structured file system is that they are optimized for write-only activity. In Chapter 5 we discuss the effects of smaller buffers, which causes reads in the NILFS file system.

4.2.3 Flash

As flash-based solid state drives are becoming more affordable, it makes sense to use them as database storage devices [11] [12]. Figure 4.13 shows the dirty page counts in our first experiment with flash device. Here an interesting phenomenon occurs. Observing the *Sweep start* and *Sweep end* events, there are moments where the cleaner is idle. That is because the dirty ratio at this moment is lower than the activation threshold for the default cleaner. With flash, when the cleaner is activated, it propagates up to 10k pages per second, as seen in Figure 4.14, whereas this speed was only two thousand pages per second in NILFS. After that, dirty ratio becomes very low, so the cleaner keeps idle until this ratio grows.

Figure 4.15 shows the quick burst of writing activity in the database flash device. It is also possible to see the idle time of the database device. This is not the ideal behavior of the cleaner, since during these periods, the number of dirty pages might double, as seen in Figure 4.13 during second 2000. This quick cleaner behaviour, along with the too long cleaner of Section 4.2.1, serves as motivation for the implementation of new cleaner policies.

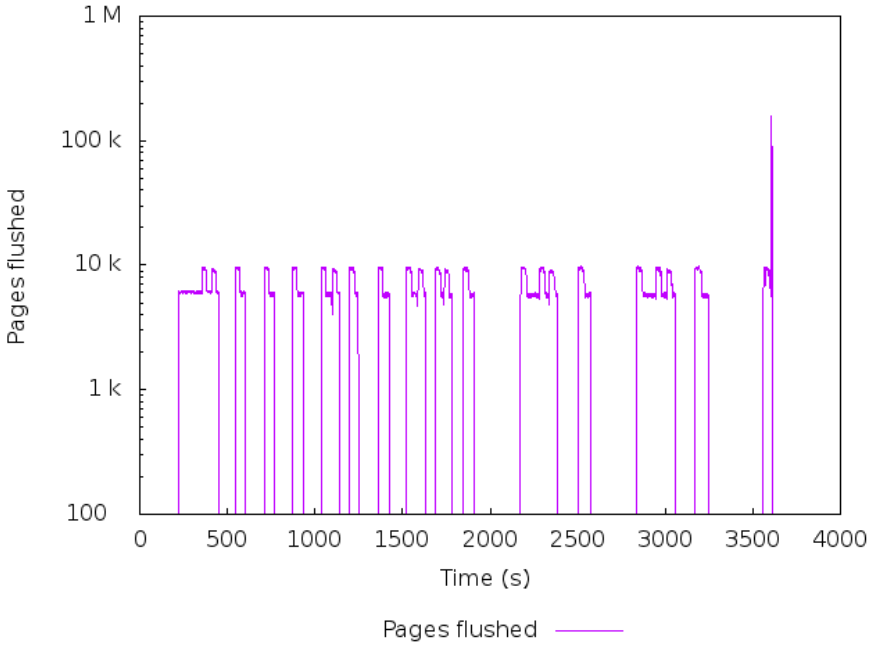

FIGURE 4.14: Page flushes for flash

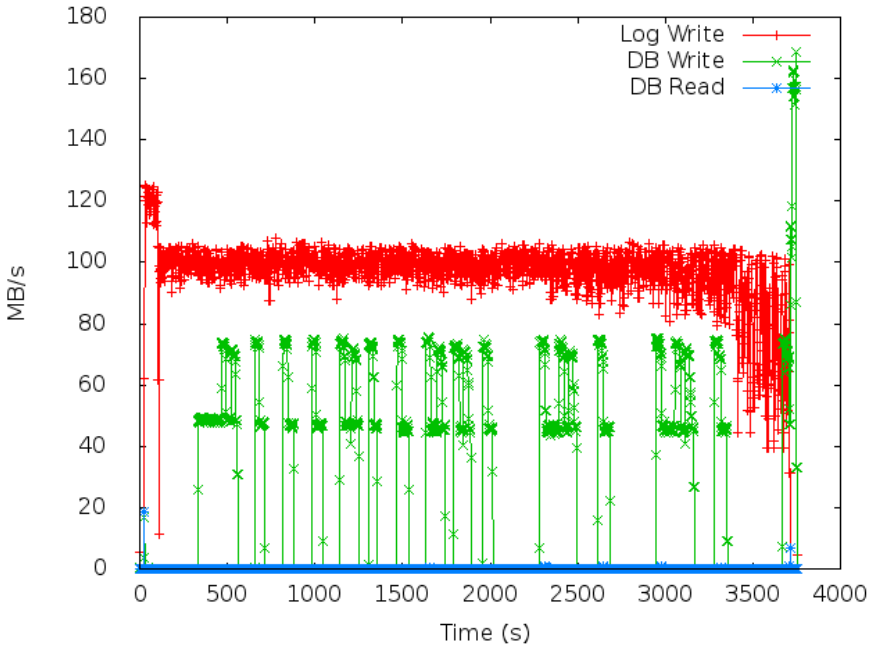

FIGURE 4.15: I/O status for flash

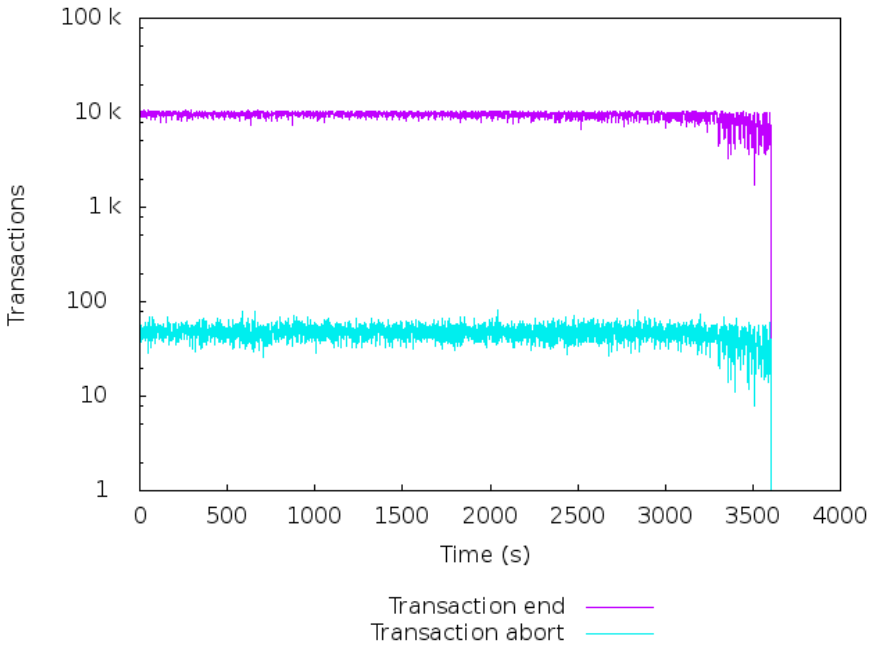

FIGURE 4.16: I/O status for flash

To take advantage of idle times observed in flash devices offers, a more aggressive cleaner policy was implemented. Instead of waiting indeterminately for an activation event, this policy instantly starts a new sweep after the previous ended. It also cleans all dirty pages in order to maximize I/O throughput. Figure 4.17 shows the result of such policy. This figure shows that now the cleaner constantly keeps the number of dirty pages low, with the maximal real dirty page count at 350k, while the maximal at normal policy was 850k. This is almost a 60% decrease in maximal dirty page count. Such difference might be responsible for a more than twice as longer recovery time, as discussed in Section 2.3.

Figure 4.17 also shows that, with a flash device, it is usefull to log individual page flush, and not only rely on checkpoints. The difference between these measures can go up to 100k pages, and mean a 30% increase in recovery time when the real dirty count is 350k, such as at second 3300. This difference will become bigger as new storage devices arise.

It is important to note that the absolute number of dirty pages in this experiment grows because the total number of pages grows as well. The number of dirty pages also decreases at the end because of the degradation of the log device file system which can be seen in Figure 4.18,in the same way as the transaction throughput in Figure 4.19.

With an aggressive policy, the sawteeth pattern appears again, and becomes very narrow. This is because the sweep times are much lower than the ones of NILFS. Figure 4.20 confirms that the cleaner only briefly stops between activations while selecting and ordering the pages to flush, and then continues to flush more pages.

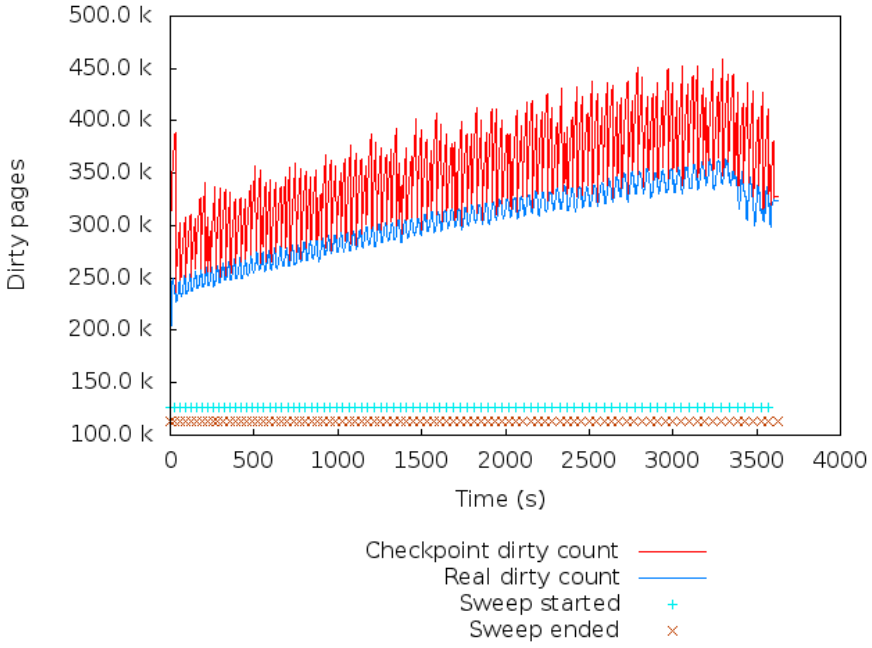

FIGURE 4.17: Number of dirty pages for flash aggressive

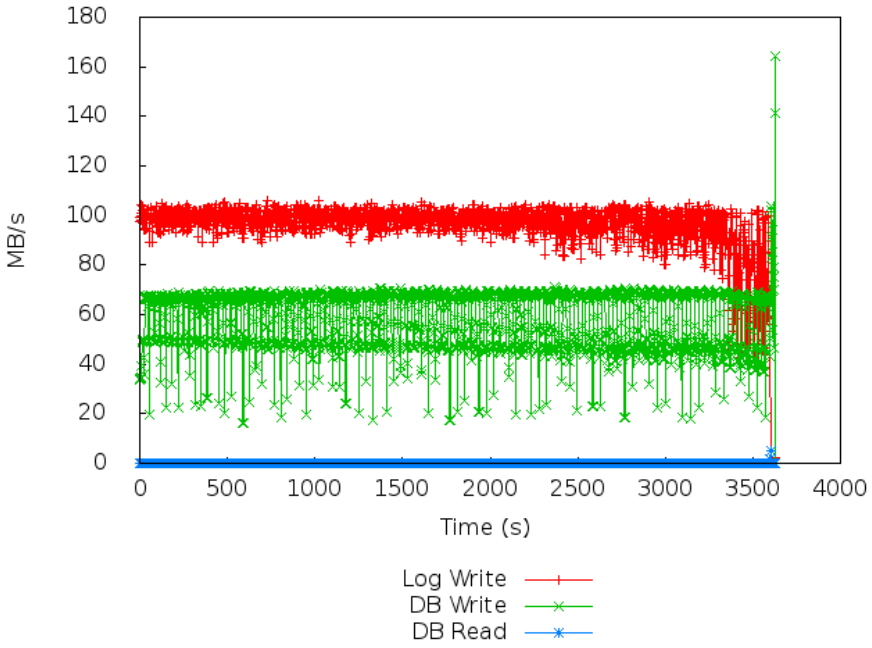

FIGURE 4.18: I/O status for flash aggressive

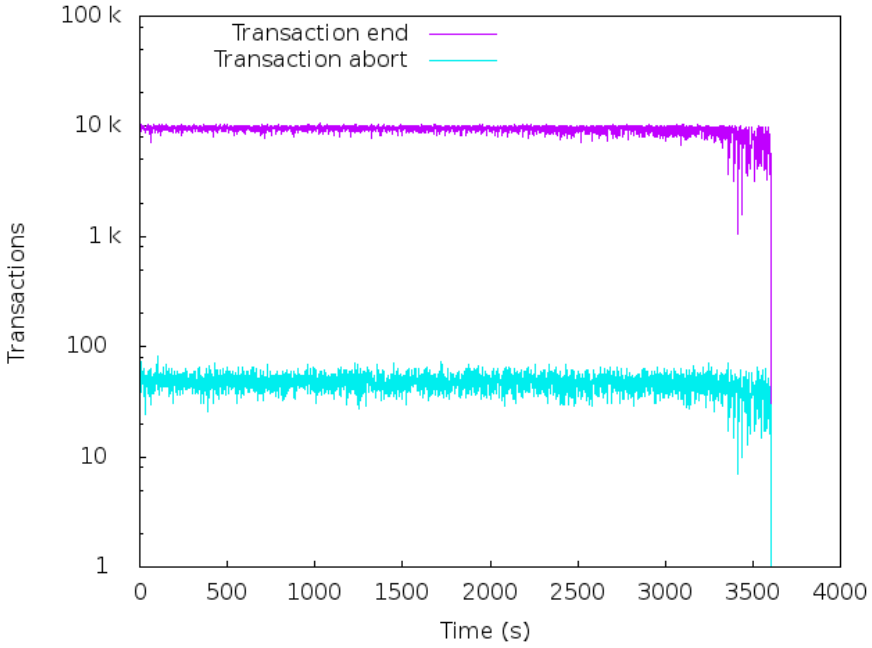

FIGURE 4.19: Transaction throughput for flash aggressive

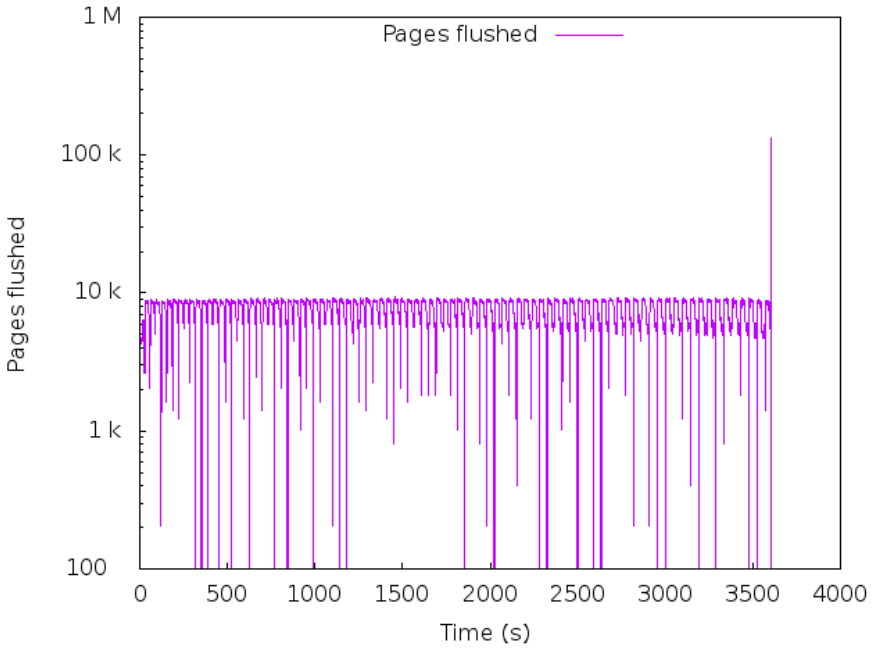

FIGURE 4.20: Page flushes for flash aggressive

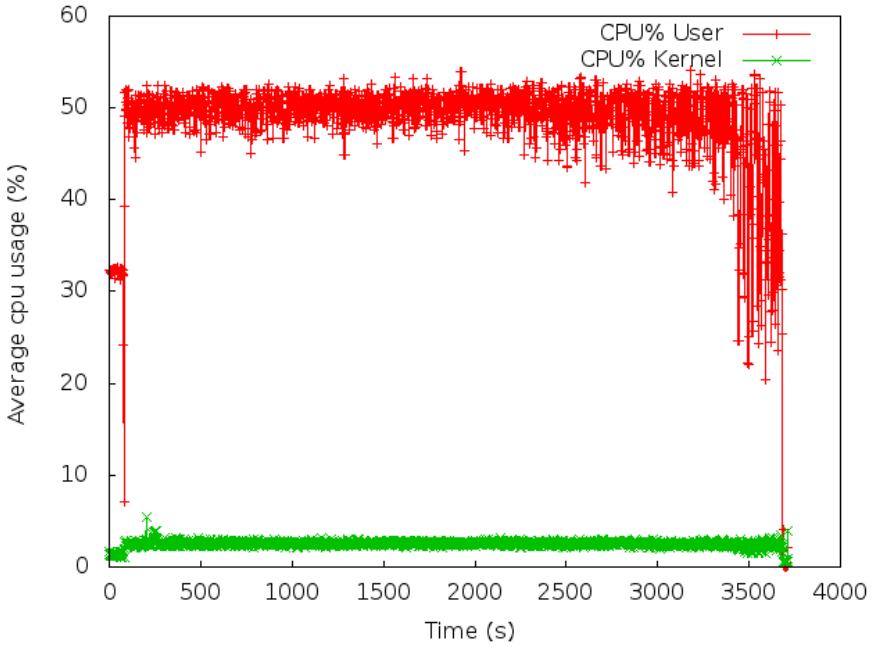

FIGURE 4.21: CPU status for flash aggressive

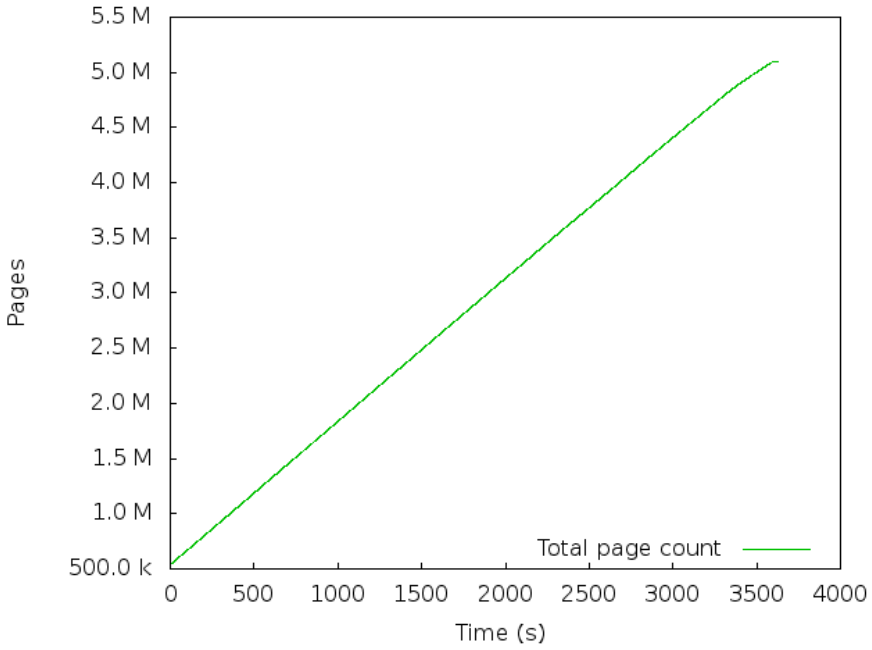

FIGURE 4.22: Total number of pages for flash aggressive

Figure 4.18 shows the I/O activity in the aggressive policy. Comparing to the normal policy, it has a much more constant activity. The isolated speed of individual writes in the normal policy is, however, 5MB/s faster. This is because flash devices still suffer from random writes. In flash devices, random writes to larger than 16MB areas are typically more expensive than sequential writes [7] [13]. Since the database is several GB large, random write performance decreases, while in the normal policy the total number of pages to be cleaned per sweep is higher, which means that there is a higher probability that sequential writes will occur.

At last, Figures 4.16 and 4.19 show that transaction throughput for both policies are the same, as well as for using magnetic drive. That is because the log device speed limits transaction throughput. This also means that the total number of pages grows at the same rate for all experiments, as seen in Figure 4.22 and 4.5. This is because TPC-C benchmark creates records, so if the transaction throughput is the same for all experiments, the database growth ratio will be the same. At last, Figure 4.21 shows that CPU usage is also independent of the device. The Kernel CPU usage variates a little between experiments because it reflects I/O management.

4.3 Discussion

In neither of the experiments, transaction throughput, log device behaviour, total number of pages and CPU usage is affected. Also, CPU usage and log write bandwidth does not reach 100% in any experiment.

Since there is a strong correlation between log write speed and transaction throughput, the Ext2 degradation is a significant problem. For the previous experiments, the log device had to be reformatted between experiments so that there were no variation in CPU usage, transaction throughput, or database size grow speed. Figure 4.23 shows the I/O activity after six executions using the same configuration as the previous flash experiments, but without formatting the log device between runs. This degraded file system shows a significantly lower log write speed compared to previous measures.

Degradation of I/O throughput to the log device influences transaction throughput. Figure 4.24 shows this effect . The points with lower transaction throughput match the points with lower log device write speed.

The new aggressive policy proved to be useful when using flash devices, lowering the number of dirty pages. Other policies can be implemented, such as an HDD policy, with a filter that returns fewer pages, so that sweeping times are not so high, as suggested in Section 4.2.1.

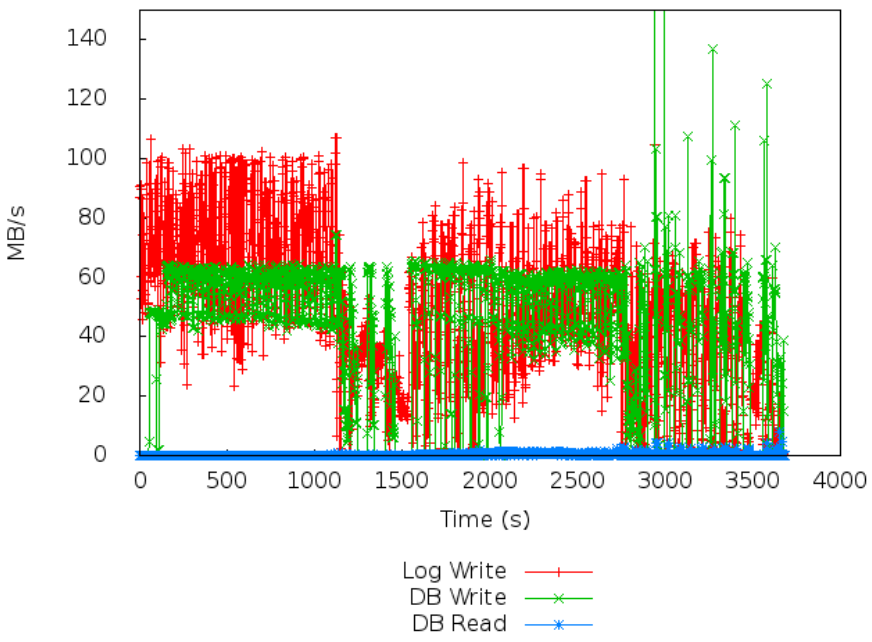

FIGURE 4.23: I/O status for degradated Ext2 log device

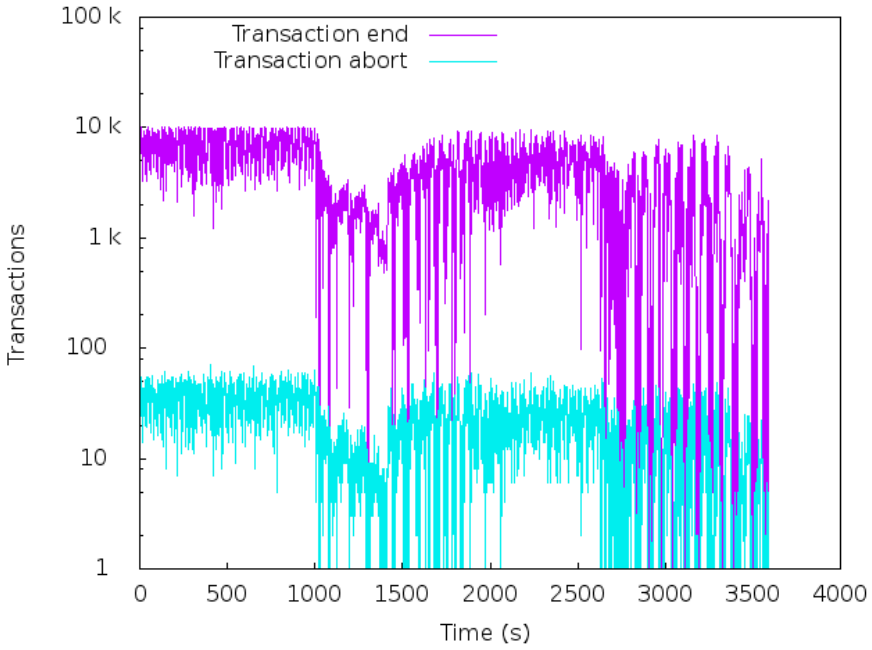

FIGURE 4.24: Transaction throughput for degradated Ext2 log device

Finally, I/O effects several aspects of the overall database performance. A different cleaning strategy for flash devices helps to keep a lower dirty ratio. The log device degradation affects transaction throughput. And the I/O delay affects the dirty pages count. All these correlations are summarized in the Conclusion. Being able to see in detail what happens during experiment runtime allowed us to identify this new aggressive policy, and we suggest the HDD policy as future work.

Chapter 5

Experiment 2: Cleaner throughput

The previous chapter studied the impact of different device configurations in the number of dirty pages. The experiments had 100% of the working set in the buffer. Since all the relevant pages fits in the buffer, there is no reason to ever perform a page read from the database device except for the first read of each page. Therefore, since the experiment measurement starts after the database load, the only database device activity that we had at the previous experiments was generated by the page cleaner. Having only device write activity allowed us to study the cleaner behaviour in detail, with less variables to worry about between experiments.

In practice, it might not be possible to accommodate all the working set in memory, either because of a system with lower amounts of RAM, or because of data-intensive applications [14]. With a lower *buffer/database size* ratio, more read activity will occur. That way, the cleaner write activity will compete with buffer read for the database device. This is because page replacement happens when a transaction requests to read data that is not in the buffer. Since the transaction will have to wait for the page to be fetched, transaction throughput now also depends on the database device read speed.

Another effect of a lower buffer ratio is a lower number of dirty pages. Even though the background cleaner writes must compete for database device usage with page reads, cleaning will happen as a consequence of replacing dirty pages in the buffer pool. But the real reason for a lower number of dirty pages is the transaction throughput. As seen in Section 4.1, the lower the transaction throughput, the lower the dirty pages count. Since a lower buffer ratio requires transactions to wait for device read, the transaction throughput lowers, resulting in a lower dirty ratio.

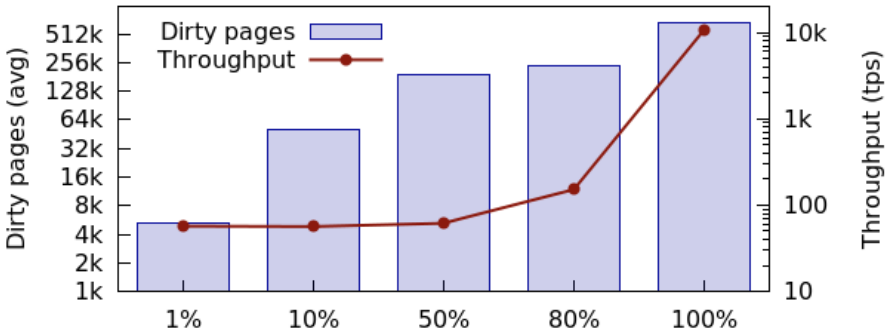

FIGURE 5.1: Throughput at several buffer ratios in Ext2

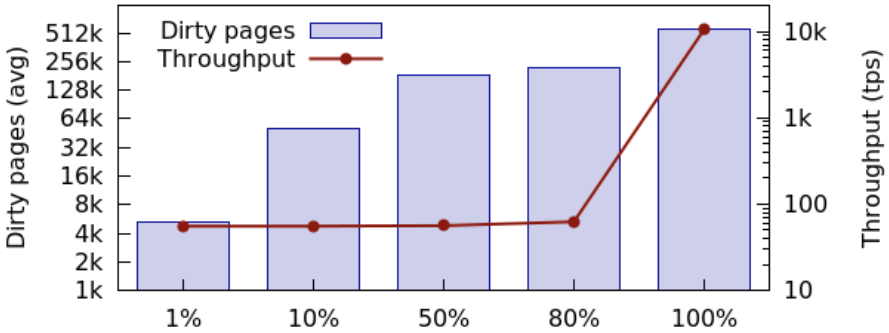

FIGURE 5.2: Throughput at several buffer ratios in NILFS

We want to measure the effect of different buffer ratios in dirty ratio and transaction throughput. Since the database grows, the buffer ratio is the size of the buffer in relation to the initial database size. Since our initial database size is 4.17GB, a 10% buffer ratio will be 417MB. The TPC-C benchmark runs for ten minutes with the same configuration as in the previous experiment. We then vary the size of the buffer in relation to the database size. As explained above, with a lower buffer ratio, we expect a lower transaction throughput, and thus a lower number of dirty pages.

We also vary the storage type in this experiment. On the previous experiment, NILFS was more efficient keeping the dirty page count low. We will compare the NILFS performance with Ext2, measuring transaction throughput and average dirty pages count. Since NILFS is write-optimized, we expect a worse performance than Ext2 with lower buffer ratios. The performance impact of flash devices is also measured. To perform precise measures of device read activity, we drop the OS page cache every second. This is better explained in Section 6.1.2.

Figure 5.1 shows the effect of varying the buffer size in Ext2. The x-axis contains the buffer ratios. The y-axis of the bars represents the average dirty pages, and the line represent the transaction throughput in transactions by second. As expected, the lower the buffer ratio, the lower the transaction throughput and the dirty pages count. The

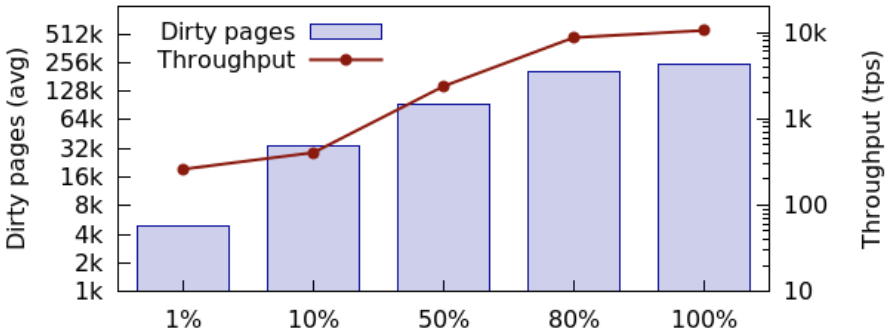

FIGURE 5.3: Throughput at several buffer ratios in Flash

transaction quickly decreases from 100% to 80% buffer ratio, because transactions must now wait for I/O. At less than 50% buffer, transaction throughput does not decrease as fast as from 100% to 80%.

Figure 5.2 shows the performance of NILFS/@. It has a much worse performance than Ext2 at lower buffer sizes. Since there is more read activity in this case, this write-optimized file system does not show the advantages that it had with 100% buffer ratio. At 100% buffer ratio the transaction throughput is the same, but the dirty pages is a little lower. The difference between dirty pages count in these two file systems is not as huge as in the previous experiment because the experiment only runs for 10 minutes, so there is not much opportunity for the difference in cleaner behaviour to appear.

Figure 5.3 shows that using a flash device for the database provides a huge advantage with lower buffer ratios. First, at 100%, the transaction throghput remains the same as in the magnetic disk, but the dirty ratio is lower. Chapter 4 explains in detail the reason for same transaction throughput and lower dirty page count in flash with 100% buffer ratio. Then, at 80%, transaction throughput does not lower significantly compared to the magnetic drive. Transaction throughput continues to fall when lowering the buffer ratio more, but not as significantly as in the magnetic drive, ending with about 20K transactions per second at 1% buffer, whereas on magnetic drive it was around 5K/@. This is because transactions must wait for page reads in less than 100% buffer ratio, and since flash provides a lower I/O latency, the waiting time is lower, resulting in a higher transaction throughput.

Having an overview of the system behaviour, we now want to investigate exactly how I/O effects transaction throughput and Dirty pages count at different buffer ratios. For that, the I/O status from the extreme buffer cases, 1% and 100%, for each storage type are inspected. This provide a better understanding of the difference between having a 100% in-memory database, and having to read pages.

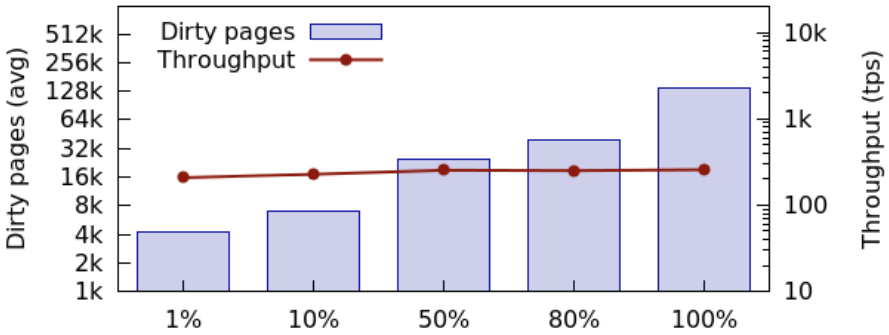

FIGURE 5.4: Throughput at several buffer ratios with log on magnetic drive

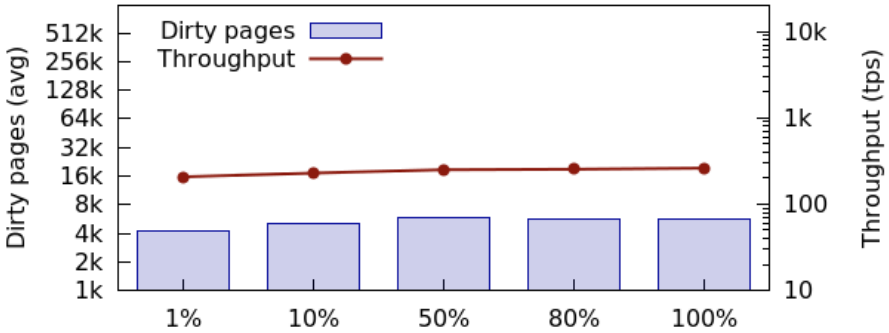

FIGURE 5.5: Throughput at several buffer ratios with log on magnetic drive and aggressive cleaner

We now look at 1% and 100% buffer ratios of Ext2. Figure 5.1 shows that a lower buffer ratio will lower transaction throughput because of page reads. Figure 5.6 shows the I/O activity with 100% buffer ratio. It is similar to the results of the previous experiments: High log device speed, some database device write because of background cleaning and no database device read because everything is always in the buffer.

Figure 5.7 shows I/O activity in the other extreme: 1% buffer ratio. The main activity is the database device read. The second more frequent activity is database device write. This indicates that there is much more page reads than page writes. The write speed is around five times higher. Therefore, we can assume that there are more page reads, since the buffer manager is able to select clean pages for replacement. The lower activity is log device. As explained before, log activity is directly proportional to transaction throughput. Since in these experiments transactions are mostly waiting for page reads, the transaction throughput will be low, therefore there is a low log device write activity.

Figure 5.7 also shows that with a lower buffer ratio, a flash device is not required for a high transaction throughput anymore. In this experiment, the bottleneck is the database device read bandwidth.

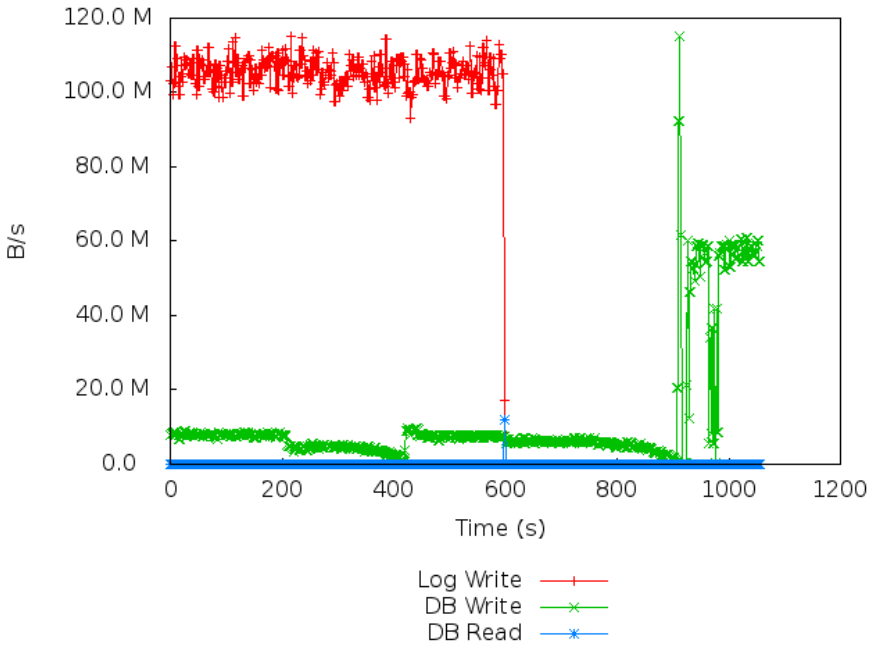

FIGURE 5.6: Iostat for 100% buffer ratio Ext2

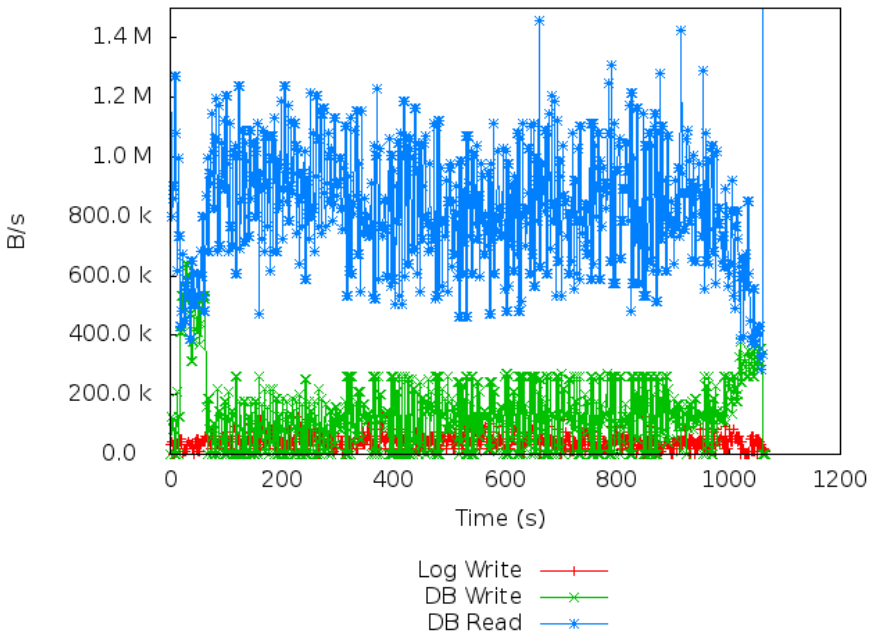

FIGURE 5.7: Iostat for 1% buffer ratio, database on magnetic drive Ext2

Figure 5.8 shows the advantage of using a flash device for the database device. Since the lower buffer ratio requires lots of random reads to the database, may be the system bottleneck, when using the magnetic drive. Therefore, with a lower buffer ratio, it might make more sense to use an SSD as database device.

One could even argue that the log could be stored on a magnetic device when working

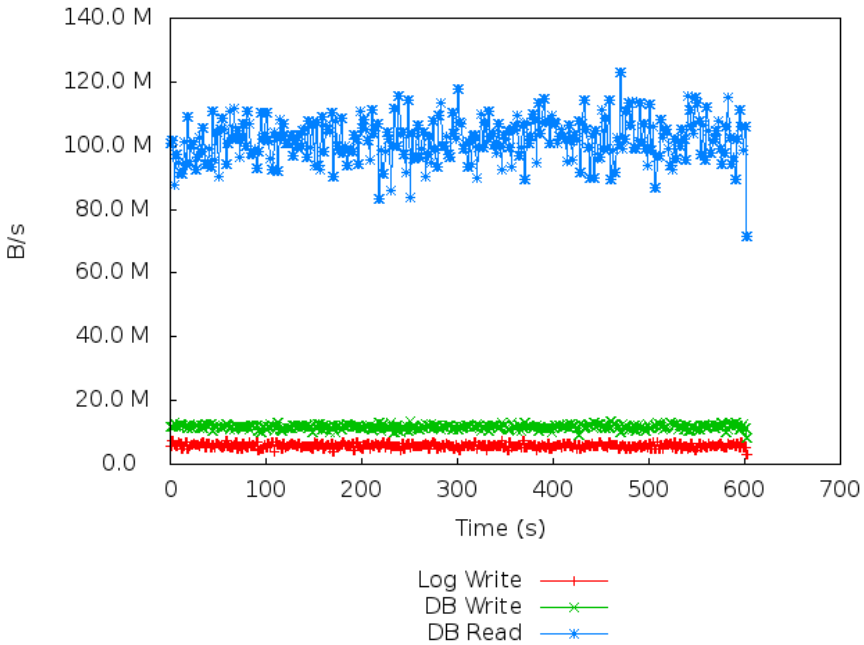

FIGURE 5.8: Iostat for 1% buffer ratio, database in Flash

with a low buffer ratio, since it barely goes over 5MBp/. The log is also written sequentially, therefore a magnetic drive could be a cheaper alternative. The disadvantage of this is a higher recovery time, specially for UNDO, since it has to traverse the log backwards for each transaction, which results in a more random access pattern than REDO, which reads the log sequentially forwards.

We run the same experiment with a magnetic drive as the log device and an SSD as database device. We will expect a low transaction throughput with 100%, but a similar performance at 1%. Figure 5.4 confirms this hypothesis: the transaction throughput remains almost constant between different buffer ratios. This is because log write bandwidth is now only sufficient for the transaction throughput of the 1% buffer ratio. Higher buffer does not provide a higher transaction throughput when using a magnetic drive for logging Since we are using the original Shore-MT cleaner policy, the number of dirty pages goes up when using a higher buffer ratio. The aggressive policy prevents the dirty pages count to rise. This was experimented and is shown in Figure 5.5.

Figure 5.9 shows the I/O activity at 1% buffer ratio with the database on flash and log on magnetic drive. The behaviour is similar to the scenario of 1% with SSD, but with a little less log I/O. Transaction throughput is only 20% lower than SSD, with 156K with LOG at SSD and 124K commits for LOG at magnetic drive, which can is acceptable given that SSD devices with the same capacity are around six times more expensive as magnetic drives.

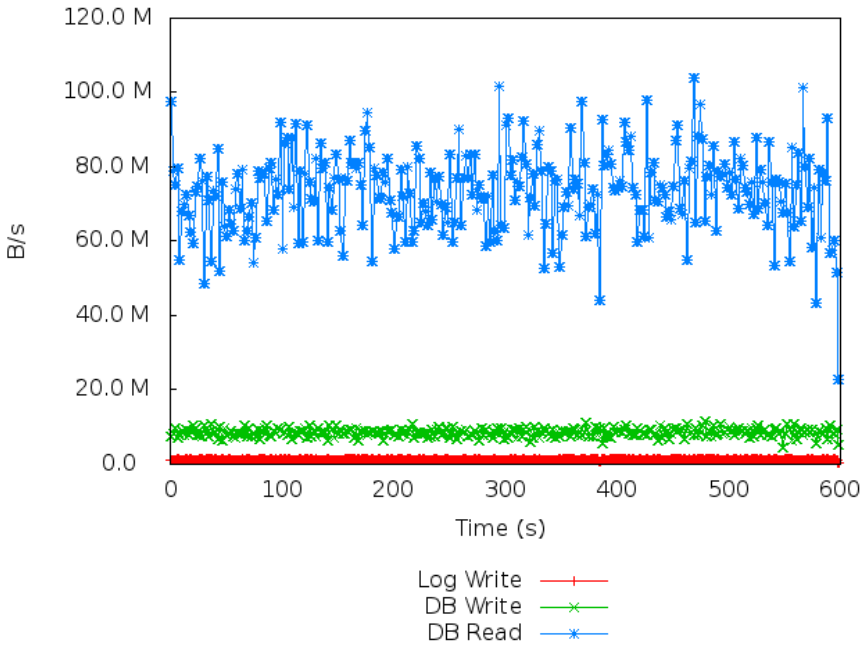

FIGURE 5.9: Iostat for 1% buffer ratio, database in flash, log in magnetic drive

With these experiments, we see that I/O requirements change depending on buffer ratio. The database device bandwidth quickly becomes the bottleneck when lowering buffer ratio. In-memory databases are becoming more common with large quantities of RAM available, but performance quickly drops when having page replacement.

Chapter 6

Discussion

This chapter will discuss the major findings of this work. The main contribution is the better understanding of I/O behaviour and its effects in transaction processing. This chapter summarizes these details as a guideline for further development of the propagation control subsystem.

6.1 Double caching

The first problem found is double caching. Shore-MT implements its own page buffer, but the Linux operating system also has an I/O buffer, known as *OS page cache*. Thus, if no additional care is taken, any I/O operation performed by Shore-MT must go through two caches, as seen in Figure 6.1.

By default, Shore-MT suffers from double caching for the database device. Unrealistic I/O delay times in the first experiments indicated this. Measurements showed less than 1ms delay in a magnetic drive, which was not expected for our 7200 rpm device. Therefore, Shore-MT was only writing database pages to the kernel buffer.

There are two distinct problems when dealing with double caching: read and write. The experiments of Chapter 4 performed only writes to the database, due to page cleaning activity. Therefore, we focus only on write caching problems in Section 6.1.1. The second experiment, on the other hand, also performed reads to the database, due to page fault in small buffer. Therefore, we focus in read caching problems in Section 6.1.2.

The Shore-MT log device does not suffer from double caching. It has the *flush until LSN* method which fsyncs the log until the given LSN. Log flushes occurs after transaction commits or aborts to ensure transaction durability. Therefore the log device does not need a fix for double caching.

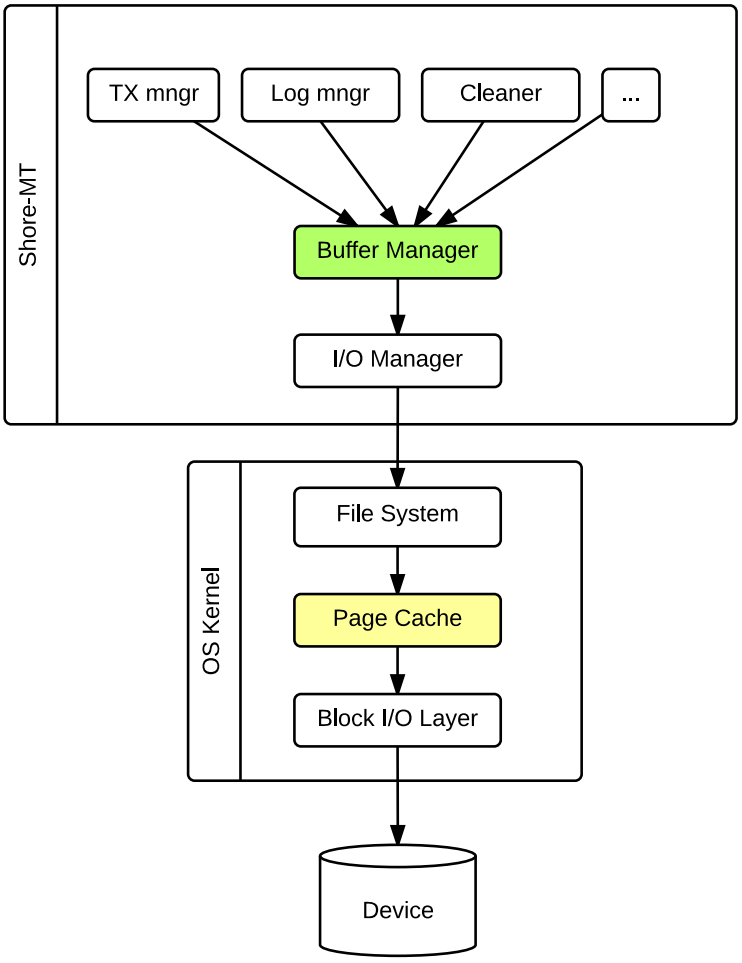

FIGURE 6.1: Shore architecture suffers from double caching

6.1.1 Write caching

Besides providing wrong measurements, propagation only to the kernel buffer also implies in no durability guarantees. After a page write, Shore-MT marks the written page as clean, thus removing it from the dirty pages table. Therefore, a checkpoint will not include it on the list of dirty pages to be recovered. Thus, if a failure occurs between the checkpoint and the propagation to disk, the changes will be lost, even if the log manager logs them since recovery will ignore this page.

To fix this problem, the page cleaner must issue *fsync* calls in order to tell the operating system to commit buffer cache to disk. This will fix the double caching for write operations in the database allowing more realistic I/O measurements.

There are three main options in invoking fsync, as seen in Table 6.1. The options are: to perform one fsync per sweep; an fsync every chunk, which is the set of pages that each writer thread writes at once; and an fsync after every page write. To guide

When	Number of fsyncs per sweep
Sweep fsync	1
Chunk fsync	Number of chunks
Page fsync	Number of pages

TABLE 6.1: Manual fsync frequency

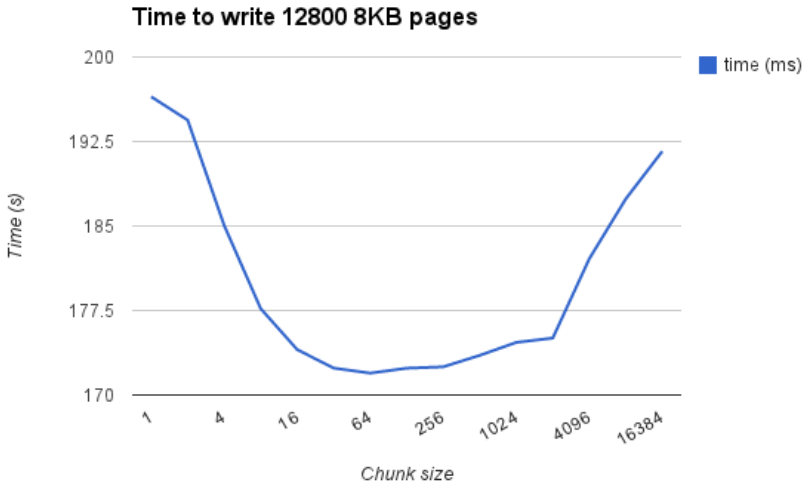

FIGURE 6.2: Total write time versus chunk size

our decision, we run an experiment concerning the performance of fsync. A separate program was created to randomly write 12800 pages of 8KB in a 4GB range. The experiment exponentially varies the fsync frequency, and measures the total execution time. Figure 6.2 shows that the lower total time to fsync occurs when syncing 64 pages at once. Therefore, the best place to fsync our data in Shore-MT is in the buffer writer thread since we measured that Shore-MT writes exactly one chunk of 64 pages at once.

The total time is higher when performing an fsyc for every page write because of more random page writes and system call overhead. The higher time to perform might have several causes: higher overhead for sorting pages prior writing, poor file system implementation, small device buffers which can be flushed between chunks.

6.1.2 Read cache

After fixing write operations on the page cleaner, we fix the double read cache. Fixing read behaviour was required for the experiment of Chapter 5 to allow a proper measuring of the real cost of read and write operations.

Four alternatives to fix the double caching are identified: Periodic fsync and cache drop, Direct device I/O, Memory mapping and Raw devices and

The raw device solution performs raw I/O using `/dev/rawctl`. This is, however, a deprecated interface since Linux kernel 2.6.3. The direct I/O interface should be used instead to achieve similar results.

We decided to use periodic fsync and cache drop to be able to measure the real cost of write operations. In this approach, fsync is invoked after each chunk write, as explained in Section 6.1.1, and the OS page cache is invalidated every second in a separate process. This solution requires the least changes in the system. The Linux kernel provides the sysctl files in `/proc/sys/vm` that tunes the operation of the virtual memory (VM) subsystem [15]. The `drop_caches` file invalidates the read cache. Writing the special value 3 to this file will empty the page cache. This is a non-destructive operation that only invalidates clean objects. That means that data which was not propagated to de device with an fsync call will remain in the buffer.

Although the periodic fsync and cache drop offers the quickest implementation alternative, it does not fully fix the problem. It works for performance measuring because now the devices performs real I/O, so the measurements are more precise. This solution is, however, not correct because it still does not guarantee durability.

To ensure correctness, the remaining two implementation options should be considered: Direct device I/O or Memory mapping.

To perform direct I/O, a file must be opened with the `O_DIRECT` flag. There are some limitations, as it is not possible to open any arbitrary file using `O_DIRECT`. Some file systems do not support direct I/O, such as Ext4 with journalling and NILFS, because physical data representation does not correspond to the logical data. Therefore the experiments of Chapter 5 cannot use this solution to compare NILFS and Ext2 performances because `O_DIRECT` is not supported by NILFS/@. This was one of the reasons to use periodic cache drop in the experiments instead of direct I/O.

Another limitation of `O_DIRECT` is that write buffers must be aligned to the device sector size, which is usually 512 bytes. This is because direct I/O uses Direct Memory Access (DMA) to write to the device. The `posix_memalign` system call allows aligned memory allocation for the write buffers to enable `O_DIRECT` in Shore-MT. It is used in place of the standard `malloc` system call.

`O_DIRECT` can present correctness issues because concurrent access to a file opened with `O_DIRECT` might cause unexpected behaviour. Therefore, `O_DIRECT` works better with exclusive access of a block device with no file system. Most of the problems go away with exclusive drive access, at the cost of not being able to share the device, nor take advantage of the benefits of some file systems. Since exclusive access for a disk is a

Solution	OS Friendly	Flexibility	Fewer changes	Cache control	Correctness
Direct IO	-	-	+	+	+
mmap	+	+	-	-	+
Periodic sync and cache drop	+	+	+	-	-

TABLE 6.2: Alternatives to fix double caching

typical scenario for a database system, loosing such flexibilities is acceptable in such solution.

The second alternative for fixing the buffer manager is memory-mapped I/O, or simply *mmap*. Memory mapped I/O allows usage of kernel space buffer as user space buffer, providing zero-copy I/O. Zero-copy I/O does not perform the task of copying data from the OS buffer to application buffer. The performance gain of zero-copy I/O will offer a slight speed increase, but not significantly since this operation is by definition I/O bound. The real advantage is less RAM usage, since main memory will have only one copy of the data.

The `mmap` system call implements memory-mapped I/O. It binds a file descriptor to a memory region, which alows to read and write to this file as if it were in RAM. This makes the process of I/O transparent. The kernel will handle the device reads and buffer management.

The main disadvantage is that the user loses control of the buffer manager, giving it to the kernel. The `madvise` function can minimize this problem. This provides high-level hints to the kernel in order to optimize buffer management. The `MADV_SEQUENTIAL` and `MADV_RANDOM` hint tells the kernel if the page reference pattern in a certain region is expected to be random or sequential, which influences how much read-ahead is performed by the kernel. Other useful hints are `MADV_WILLNEED` and `MADV_DONTNEED`, which tell the kernel if the specified area is expected to be accessed in the near future or not, making it respectively less or more susceptible for page replacement.

Three solutions were provided to the double caching problem. The periodic fsync and cache drop is a quick fix for performance measurement and direct I/O is a correct alternative that is not possible to use in the experiment. The memory mapping technique is the most OS friendly solution but loses cache control and will yield the most changes in the existing system. Table 6.2 summarizes the discussed solutions and advantages of each one. It should provide a guideline for implementing proper buffer management.

6.2 Log Failures

Another issue identified with Shore-MT is a system failure that occurs when the log grows beyond a limit specified in the experiment configuration. Analysing the error output, it provided the "No empty partitions" error. This means that the buffer was not able to clean pages to free old log segments.

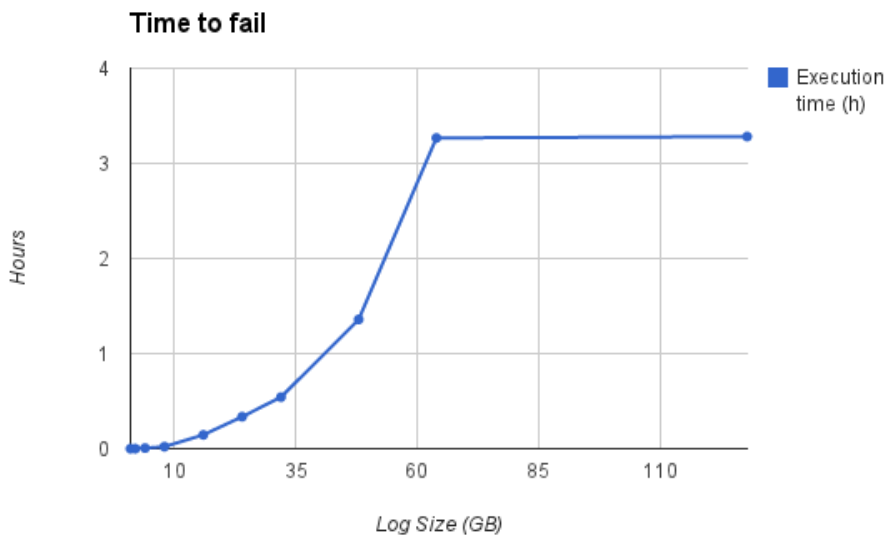

FIGURE 6.3: Time to fail depending on log size

We then run an experiment to understand the relation between log size and time to failure. It runs the TPC-C with scale factor 72 (10GB) benchmark set to run for ten hours. The experiment then exponentially varies the log size from 1 to 128GB. Figure 6.3 shows a super-linear correlation. After 64GB, the time to failure is constant. Therefore, Shore-MT is failing because of small log sizes at least until 64GB.

The reason for the constant failure time after 64GB is the database size. As seen in the experiments from Section 4, the database size grows linearly with the TPC-C benchmark. The experiment uses a 333GB partition the database device. Since the database from previous experiments grows around ten times in one hour, it should take our initial 10GB database arount 3.3h to reach 333GB. That is exactly the constant time to fail after 64GB. This failure is acceptable, since there is nothing that the DBMS can do if there is no database device space left.

The second reason for the failure is, however, avoidable. The processing fails because the log manager does not wait enough time when closing old partitions. Figure 6.4 shows the logic of closing a partition: the log manager will activate the cleaner at the

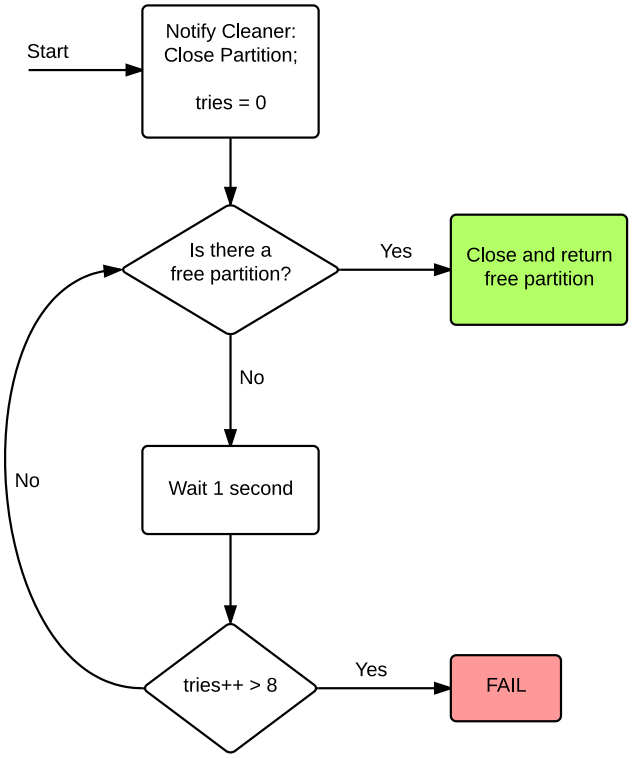

FIGURE 6.4: Log Scavenging flowchart

beginning of the `close_partition` method. If there is no free partition, it waits one second and tries again. But if it has tried more than eight times it simply fails. This means that the cleaner has only eight seconds to clean the partition. And as seen in previous experiments, some sweeps can take hours to complete, therefore this is not an ideal behaviour.

Ideally, the whole system should not crash if there is no log space. The log manager should wait until the cleaning process completes. Failing because of I/O related problems should be the cleaner's concern, not the log manager. There are other places that calls the buffer manager to clean pages. These places, such as the transaction manager, also yield failures when it is not possible to propagate the pages.

A full log also causes failure of the transaction manager when there is no space on log. It tries to reserve log space in order to be able to commit or rollback transactions because with no log space left, transactions can not write the commit nor abort log records, preventing them to finish. When there is no space, the transaction manager asks the buffer core to force old pages, but this method is not implemented, and therefore it always fails. Again, this should not happen. Transactions should wait or abort when there is no log space, not crash the whole system.

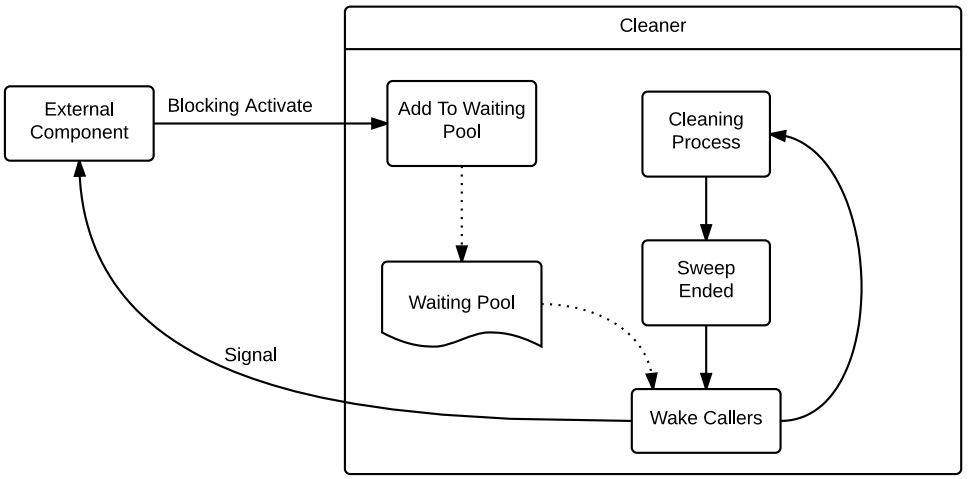

FIGURE 6.5: Cleaner Daemon

Ideally, there should be no scattered buffer manager calls that clean pages directly. We propose a higher separation of concern to achieve a better architecture. Figure 6.5 illustrates this. The cleaner service should be the only component allowed to flush pages, and other system services –such as the log or transaction manager– should only generate events. The cleaner must then listen to such events in order to react accordingly. For example, a full log could generate a high priority event which requests the cleaning of pages whose persisted version is older tan a given LSN. These calls can use the event-oriented architecture introduced in Section 3.1.1 to send additional information for the cleaning process. The cleaner then adds the caller to a Waiting pool, and waits the normal cleaning process to finish. When the current sweep ends, the cleaner signals all registered callers in the waiting pool so they can continue to perform its own operations.

Chapter 7

Conclusion

This work studied the influence of I/O in transaction processing. Figure 7.1 summarizes our findings. It shows the correlation between several aspects. Each box represent an aspect that can be either a parameter, a hardware or a software effect. Each arrow represents the effect of raising an aspect, that can cause either a positive or negative effect in other aspect. A plus sign in an arrow indicates a raising correlation, and negative sign a reducing correlation. For example, raising the hardware parallelism will raise transaction throughput, and raising the log device write latency will lower transaction throughput.

The *parameter* boxes are easily controllable and were varied in this work, with the exception of workload concurrency, which was always the same since we only used one workload type: the TPC-C benchmark. The *software effects* are logical aspects of the database. Software effects can be optimized with better algorithms, such as the cleaner throughput, which in this work was raised by modifying the cleaner policy logic. *Hardware effects* are usually end nodes of the graph and represent physical effects in the system. It is not possible to directly lower physical effects through software optimization.

The 5 *parameter* aspects are bottleneck sources: database device write latency, log device write latency, hardware parallelism, buffer size, and workload concurrency. We omit database and log device read latencies for clarity. Therefore, as in our previous experiments, bottleneck can be the workload concurrency if there is enough hardware parallelism, log write latency and a large buffer. On the other hand, a small buffer raises lowers transaction throughput

This model can be used to identify what actions should be taken to lower an aspect. To do this, the arrows should be followed backwards in order to identify causes. For example, if one wants to lower recovery time, the dirty ratio should be lowered. Then,

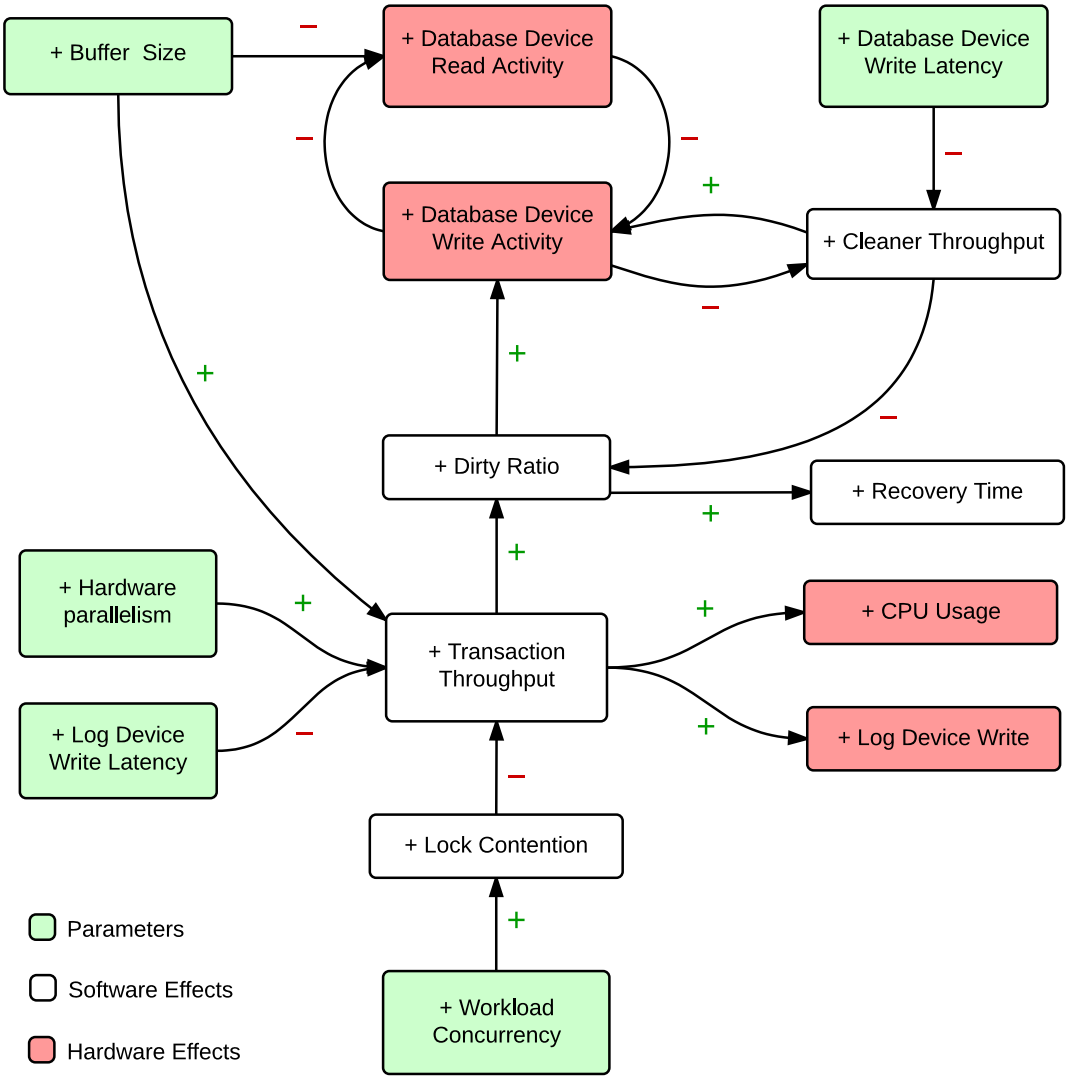

FIGURE 7.1: Correlations of system resources and parameters

there are two options to lower dirty ratio: to lower transaction throughput or raise cleaner throughput. A lower transaction throughput is trivial and not desirable, so to raise cleaner throughput, one could lower the database device write latency. This can be achieved by changing a high latency database device such as a magnetic drive with a low latency device, such as flash.

The buffer size parameter was one big focus of this work. With the lower prices of random access memory, in-memory database systems are expected to become common [8]. Such databases achieve a higher transaction throughput since the database device is not the bottleneck anymore.

Higher throughput raises the number of dirty page. More dirty pages during runtime means that if a failure occurs, more pages should be redone during REDO, therefore

increasing recovery time. Therefore, such systems require an efficient recovery method.

Keeping the number of dirty pages low during runtime is a way to reduce recovery time. Lower latency devices can help reduce the number of dirty pages. Flash devices are a good alternative since they have lower I/O delays than magnetic drives and are becoming more affordable. Write-optimized file systems such as NILFS can also help to reduce the number of dirty pages when using large buffers.

Software optimization can also help to change aspects. One of the objectives of the Shore-MT project was to lower lock contention of the SHORE storage manager. Therefore, they manage to achieve a higher transaction throughput by lowering lock contention only with software changes.

We also implemented a software change: the cleaner policy system. This allows the implementation of novel page propagation behaviour. The aggressive policy was one useful implemented policy. It was useful to reduce the number of dirty pages in faster devices such as flash, when the default Shore-MT policy allowed database device to become idle, raising the total number of pages.

Another contribution of this work are the suggestions for better implementation of propagation control. We have found some problems such as double caching and scattered calls for page propagation, that can fail the whole system.

The periodic fsync and drop cache fixes the double caching problem, allowing accurate I/O measurements. This is, however, only a quick fix. We also analyse the advantages and disadvantages of definitive solutions: Direct I/O and memory mapping. Direct I/O is better used directly with block device interface, while the memory mapping is a more OS-friendly approach which any file system can use.

We also studied the failures that small log sizes cause in the system. Such failures occur because some Shore-MT components directly calls the buffer manager when requiring page propagation. These places then manually handle time-outs and failures, which should not be their concern. We propose a new cleaner architecture to meet page propagation requirements. These will provide a centralized cleaning process and simpler code in other components, which would not need to manually handle page propagation.

At last, we conclude that while transaction throughput is important, I/O should not be neglected. It is expected that this work will be useful as guideline for implementation and better understanding I/O and its effects in transaction processing, buffer management and propagation control.

Bibliography

[1] Theo Härder and Erhard Rahm. *Datenbanksysteme – Konzepte und Techniken der Implementierung*. Springer, 2001. ISBN 3540421335.

[2] C. Mohan, Don Haderle, Bruce Lindsay, Hamid Pirahesh, and Peter Schwarz. ARIES: a transaction recovery method supporting fine-granularity locking and partial rollbacks using write-ahead logging. *ACM Transactions on Database Systems*, 17(1):94–162, March 1992. ISSN 03625915. doi: 10.1145/128765.128770. URL http://portal.acm.org/citation.cfm?doid=128765.128770.

[3] Theo Haerder and Andreas Reuter. Principles of transaction-oriented database recovery. *ACM Computing Surveys*, 15(4):287–317, December 1983. ISSN 03600300. doi: 10.1145/289.291. URL http://portal.acm.org/citation.cfm?doid=289.291.

[4] C Mohan. Repeating history beyond ARIES. *Proceedings of the 25th VLDB Conference*, pages 1–17, 1999. URL http://ece.ut.ac.ir/classpages/F83/AdvancedDataBaseSystem/Paper/Hot_Topic_Paper/P1.pdf.

[5] Caetano Sauer, G Graefe, and Theo Härder. An empirical analysis of database recovery costs. *First International Workshop on Reliable Data Services and Systems*, 2014.

[6] Ryan Johnson, Ippokratis Pandis, Nikos Hardavellas, Anastasia Ailamaki, and Babak Falsafi. Shore-MT. In *Proceedings of the 12th International Conference on Extending Database Technology Advances in Database Technology - EDBT '09*, page 24, New York, New York, USA, 2009. ACM Press. ISBN 9781605584225. doi: 10.1145/1516360.1516365. URL http://dl.acm.org/citation.cfm?id=1516365.

[7] Luc Bouganim, Björn Jónsson, and Philippe Bonnet. uFLIP: Understanding Flash IO Patterns. *arXiv preprint arXiv:0909.1780*, September 2009. URL http://arxiv.org/abs/0909.1780.

[8] H. Garcia-Molina and K. Salem. Main memory database systems: an overview. *IEEE Transactions on Knowledge and Data Engineering*, 4(6):509–516, 1992.

ISSN 10414347. doi: 10.1109/69.180602. URL http://ieeexplore.ieee.org/lpdocs/epic03/wrapper.htm?arnumber=180602.

[9] Transaction Processing Performance Council. TPC Benchmark C sepecification, 2010. URL http://www.tpc.org/tpcc/spec/tpcc_current.pdf.

[10] Ryusuke Konishi, Yoshiji Amagai, Koji Sato, Hisashi Hifumi, Seiji Kihara, and Satoshi Moriai. The Linux implementation of a log-structured file system. *ACM SIGOPS Operating Systems Review*, 40(3):102–107, July 2006. ISSN 01635980. doi: 10.1145/1151374.1151375. URL http://portal.acm.org/citation.cfm?doid=1151374.1151375.

[11] Philippe Bonnet and L Bouganim. Flash Device Support for Database Management. *CIDR*, pages 1–8, 2011. URL http://www.cs.umd.edu/class/spring2011/cmsc724/cidr-abstracts.pdf.

[12] Radu Stoica, Manos Athanassoulis, Ryan Johnson, and Anastasia Ailamaki. Evaluating and repairing write performance on flash devices. *Proceedings of the Fifth International Workshop on Data Management on New Hardware - DaMoN '09*, page 9, 2009. doi: 10.1145/1565694.1565697. URL http://portal.acm.org/citation.cfm?doid=1565694.1565697.

[13] Volker Hudlet and Daniel Schall. SSD!= SSD-An Empirical Study to Identify Common Properties and Type-specific Behavior. *BTW*, 2011. URL http://www.researchgate.net/publication/221324011_SSD_!_SSD_-_An_Empirical_Study_to_Identify_Common_Properties_and_Type-specific_Behavior/file/9fcfd507e53f3dc3c6.pdf.

[14] Tony Hey, Stewart Tansley, and Kristin Tolle, editors. *The Fourth Paradigm: Data-Intensive Scientific Discovery*. Microsoft Research, Redmond, Washington, 2009. ISBN 978-0-9825442-0-4. URL http://research.microsoft.com/en-us/collaboration/fourthparadigm/.

[15] Rik van Riel and Peter W. Morreale. Linux Kernel Documentation for the sysctl files, July 2008. ISSN 01635980. URL https://www.kernel.org/doc/Documentation/sysctl/vm.txt.