An empirical analysis of database recovery costs

Caetano Sauer
University of Kaiserslautern
Germany
csauer@cs.uni-kl.de

Goetz Graefe
Hewlett-Packard Laboratories
Palo Alto, CA, USA
goetz.graefe@hp.com

Theo Härder
University of Kaiserslautern
Germany
haerder@cs.uni-kl.de

ABSTRACT

The time required for recovery from a failure is heavily influenced by hardware setup and workload characteristics. In bad but still realistic cases, the recovery required during restart can take hours. For a database system based on write-ahead logging, we performed a qualitative study of how hardware and software configurations affect the behavior of the database and, consequently, how this behavior affects recovery time after a system crash. With the relevant parameters identified in the qualitative study, we performed an empirical quantitative analysis of recovery costs in multiple scenarios. We show that recovery costs tend to get worse as hardware and software improve in efficiency, and we discuss possible approaches to make recovery time independent of system configurations and workload characteristics.

Categories and Subject Descriptors

H.2.2 [**Physical design**]: Recovery and restart

General Terms

Databases, transactions, failure, recovery, availability

Keywords

Write-ahead logging, log analysis, redo, undo, rollback

1. INTRODUCTION

The standard approach for providing transaction atomicity and durability in database systems is based on write-ahead logging, in many cases following the ARIES design [7]. During restart after a system failure (e.g., a crash of the operating system), the recovery procedure involves scanning the log three times: log analysis, REDO pass, and UNDO pass. Each of these phases has different cost factors, but a common characteristic is that all tend to increase with the efficiency of both hardware and software.

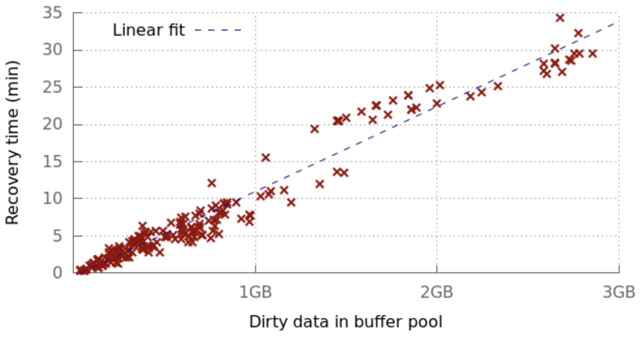

Figure 1: Recovery times vs. dirty data in buffer

As shown in Figure 1, a typical transactional workload such as TPC-C may incur crash recovery times close to thirty minutes even for relatively small working sets. In larger production systems, it is thus required to take hours or even days into account. To obtain a detailed picture of the behavior observed in this experiment, this paper investigates the main factors that determine recovery times. To understand what such factors may be in the first place, a brief review of the restart procedure in ARIES is required.

During restart, log analysis scans the log from the last checkpoint onwards in order to determine which pages need to be redone (i.e., database pages possibly dirty in the buffer pool at the time of failure) and which transactions need to be undone (i.e., active transactions at the time of failure, also known as “loser” transactions).

During the REDO pass, the log is scanned forward from the oldest log record affecting a page identified during log analysis. All log records affecting dirty pages are redone unless the database page is already up-to-date. The goal is to restore, at least in the buffer pool, the physical state of all database pages immediately before the crash, i.e., ensuring durability by repeating history [7]. To apply the updates of a log record, the page must first be fetched into the buffer pool, which incurs a random page read. The number of dirty pages cannot exceed the capacity of the pre-crash buffer pool, as dirty pages selected for eviction must first be written back to persistent storage. Therefore, we may assume that there is one random read for each dirty page, and thus the dominating cost of the REDO pass is the number of dirty pages at the time of crash. The time required for REDO after a system failure can be reduced by flushing dirty pages during transaction processing, i.e., prior to any crash.

During the UNDO pass, each loser transaction is rolled back starting from its most recent log record. This can be done with a single backward log scan, undoing (or at least queuing for UNDO) each log record of active transactions in reverse chronological order. The goal is to restore the logical consistency of the database, ensuring the atomicity property. Pages touched by loser transactions must first be redone during REDO, even if they do not contain any update from committed transactions. Hence, they are very likely to remain in the buffer pool after REDO, assuming loser transactions were not long-running and that page replacement avoids evicting pages recently referenced (e.g., LRU or CLOCK [1]). Therefore, the UNDO pass is expected to incur very little, if any, I/O cost on database pages. On the other hand, the time required for UNDO cannot be shortened by any kind of checkpoint, because all updates of aborted transactions must be undone, including any extremely long transactions. Therefore, UNDO time depends solely on the amount as well as the duration of user transactions.

In this paper, we investigate the hardware, software, and workload parameters that affect the recovery costs sketched above. Having established, for instance, that REDO time depends mostly on the number of dirty pages, we further investigate what system parameters most heavily influence the number of dirty pages. Section 2 performs a qualitative analysis of the parameters that affect crash recovery. In Section 3, we perform a series of experiments that quantify the influence of such parameters in a running system. Finally, Section 4 concludes our findings and presents opportunities for future research.

2. CRASH RECOVERY

The factors that influence recovery time can be classified into three categories: (i) workload characteristics, such as intensity (i.e., demand for throughput), duration of transactions, and skew in the distribution of page and record accesses; (ii) software and system parameters, such as frequency and type of checkpoints, page cleaning activity, and size of the log buffer (to allow the relevant portion of the log to remain in main memory during recovery); and (iii) hardware parameters, such as amount of main memory, capacity of persistent storage devices, level of parallelism, and device latency.

For the first category, we assume the workload characteristics of a typical transaction-processing scenario (e.g., TPC-C). In terms of throughput, we assume optimal utilization, meaning that as many transactions per second will be processed as the system can deliver (neither backlogs nor idle time). As common in OLTP scenarios, we also assume that transactions are very short, which implies that REDO costs dominate UNDO costs during crash recovery. Lastly, we assume that page access is not highly skewed, in which case the REDO time would be much shorter, because the ratio of dirty pages in the buffer pool is then much lower than non-skewed scenarios.

Our study focuses on software and hardware parameters. The former includes most importantly checkpoints and page cleaning activity. By experimenting with these, a system administrator can achieve an optimal trade-off between performance during normal transaction processing and time required for recovery. The key factor to analyze in our study is how intensively the system flushes dirty pages during normal processing, and what effect that has on performance and recovery time. We discuss this in Section 2.1 below.

Among hardware parameters, we consider three factors: the number of CPU cores, the size of main memory, and storage media. To deliver high transaction throughput, database servers usually run on multi-core machines with large memories and flash devices to store the log. High throughput, on the other hand, increases the time required for recovery in case of failures, because clean pages tend to get dirty faster (assuming non-skewed updates) and more active transactions must be rolled back. The media used to store the database pages also plays a key role in recovery time, because random page reads are the limiting factor for REDO performance. These issues are discussed in Sections 2.2 and 2.3.

2.1 Page Flushing Policy

Flushing dirty pages in the buffer pool back to disk during normal processing lowers the amount of REDO necessary during recovery, because only updates not reflected in the persistent database must be redone. Pages are flushed in the following three cases:

Buffer replacement: When the buffer pool is full and a page must be fetched from disk, an existing buffered page must first be evicted. Replacement algorithms usually try to pick clean pages, but if a dirty page must be evicted, then it must first be flushed to disk.

Page cleaner: Typical database system implementations employ a page cleaning service that runs as a background thread, periodically flushing dirty pages. The page cleaner sweeps through the buffer pool, selecting pages to be flushed based on various criteria. One possibility is to collect pages that are "too old". Old, in this case, can be measured either as a time threshold or as an amount of log records inserted in the log. It is also possible to flush pages in a particular order, such as "oldest first". Such measures are important because they not only reduce REDO time, but also the length of the log scan required for recovery. Flushing old pages also allows portions of the log to be discarded, and thus this kind of page cleaning can also be enforced when available log space gets critical. Another possibility is to flush pages considered "hot", i.e., those with a high update frequency and thus a large amount of related log records. The various policies can also be combined with different schedules and priorities, depending on system state and configuration parameters, making the page cleaner a very flexible tool.

Checkpoints: One important technique to limit recovery time is the use of checkpoints. They are typically of the fuzzy type [3], which means that the checkpoint process is carried on without disrupting normal transaction processing, generating multiple log records enclosed by "begin" and "end" marks[1]. A checkpoint is then only considered complete if its "end" log record is flushed to the persistent log.

In order to establish boundaries for the log scans required during recovery, checkpoints usually record information about which pages are dirty in the buffer pool and which transactions are active at the moment. Because checkpointing does not disrupt transaction processing, this information is

[1]This definition of fuzzy checkpoints is from Gray and Reuter [2]. The survey paper by Härder and Reuter [3] defines fuzzy checkpoints as those that only record information about dirty pages in the buffer pool and active transactions (also called indirect checkpoints), as opposed to flushing pages back to persistent storage. In practice, both definitions capture the characteristics of the basic checkpoint technique used by the vast majority of implementations.

(as the name suggests) fuzzy, and the actual state of the database at the time of crash can only be determined by examining the logged operations since the checkpoint started, hence the need for the log analysis pass.

During checkpoints, pages can be flushed to establish a fixed boundary to the REDO pass. A common technique is to flush all the pages that became dirty before the last checkpoint. The checkpoint "end" record is then only generated once the flushing process completes. This measure provides a guarantee that only two consecutive checkpoint intervals worth of log are needed in order to REDO all outstanding updates. This technique, sometimes referred to as *second chance*, can further be generalized by considering the last k checkpoints.

2.2 Transaction Throughput

The effect of transaction throughput on recovery time is quite obvious: the more updates the system is able to apply in a fixed amount of time, the higher the required recovery effort will be to restore the changes of that time window. The effort, in this case, translates to the number of random reads during recovery—one for each dirty page in the buffer pool between the last checkpoint and the crash. If no dirty pages are flushed during normal processing, the ratio of dirty pages steadily increases, perhaps even approaching 100% (although slowly for highly skewed accesses). The higher the transaction throughput, the faster the ratio grows.

More aggressive page flushing policies may slow down the growth of the dirty ratio, or even manage to lower it over time, but this depends on a delicate balance between transaction throughput and disk bandwidth. To better understand this, we assume two extreme scenarios, relying for now on intuitive interpolation to interpret the cases in between.

First, we assume that the buffer pool is large enough to fit the complete database. This is the ideal case for flushing policies, because they can then fully exploit the device write bandwidth to flush pages back to the persistent database. In this case, the behavior of the dirty page ratio depends on a race between transaction throughput and I/O flushes. As we empirically demonstrate in Section 3, the I/O bandwidth eventually saturates due to pages being dirtied more quickly as the number of worker threads increases. After saturation, a backlog of dirty pages may accumulate over time, and hence page flushing policies would not contribute to lowering the recovery time in case of a crash.

One may argue that employing disk arrays may solve this problem, but, as we argue in Section 2.3, this adds a new set of drawbacks. Moreover, it is debatable whether adding extra disks to keep up with CPU bandwidth is a reasonable solution, because the historic trend is that this gap increases over time.

For the second extreme scenario, consider a large database (and a large working set), such that only 1% of the pages can be kept in the buffer pool. In this case, throughput will most likely be limited by I/O latency, as most transactions must wait for pages to be replaced. Thus, page flushing must now compete for the same resource (disk I/O) rather than trying to keep up with the in-memory transaction throughput. In this scenario, the dirty page ratio can only be kept low by lowering transaction throughput, a compromise that must be considered if acceptable restart performance is desired.

We conclude that increasing transaction throughput unavoidably hurts recovery time. A modern database system design should be able to exploit the low disk latencies of flash devices for log storage as well as the high amount of cores in modern CPUs [4, 5], delivering higher throughput and better scalability, which makes recovery time a greater concern.

2.3 Storage Media

Hardware influences the costs of recovery by the kind of media device used for both the database contents and the log. Because flash drives have a latency up to 100 times faster than traditional disk drives, it is highly recommended to use it as log device, as the throughput is improved significantly due to faster log flushes at commit time. Therefore, our study will assume flash devices for the log on all experiments.

For database storage, we assume traditional magnetic disks. If the database is small enough to fit in a not-too-expensive flash device, as well as a large portion of it in main memory, then indeed the time required for crash recovery will be less of an issue (although still critical in the case of recovery from media failures). As discussed earlier, large buffer pools and database media with high bandwidth allow aggressive page flushing to keep a very small dirty page ratio, thus partially mitigating the problem of restart time. Our study, on the other hand, focuses on cases where it becomes an issue, namely when large magnetic disks are used.

Another important but less obvious aspect of storage media is the capacity of individual devices. Given that the total database size is fixed by the workload, a system administrator is faced with the choice between few large devices or many smaller ones. During crash recovery, smaller devices allow the REDO pass to fetch pages in parallel, which seems like a good idea at first. However, the mean time to a media failure also decreases as the number of devices increases, and thus media recovery is expected to be invoked more frequently. One may argue that the net effect on the system is the same, because media recovery will be also shorter for a smaller device. However, the disadvantages manifest as increased administrative overhead incurred by frequent media failures, higher frequency of outages, and costly maintenance of servers with many drives.

The principal advantage of larger devices, nevertheless, is from the economic perspective. At the time of writing, the cost per byte of a 4TB desktop hard disk is about 60% as that of an equivalent 1TB disk. Furthermore, because energy consumption does not depend on the device capacity, the 4TB drive would have only a quarter of the energy costs of four 1TB drives. Lastly, because large devices have higher density and usually employ more recent technology, they either provide faster transfer rates or allow the same transfer rate at a lower rotational speed (which has a positive effect on energy consumption and mean time to failure). Larger devices, therefore, tend to be more cost-effective in server infrastructures.

One last argument against large devices is that smaller devices on disk arrays deliver higher I/O bandwidth. However, because we focus on transaction processing and not on analytical processing, the bandwidth of the storage media is not of crucial importance, as I/O operations are mostly random single page accesses. In this case, the determinant factor for I/O performance is device latency, which is independent of the level of device parallelism. Therefore, our analysis focuses on the case of a single device.

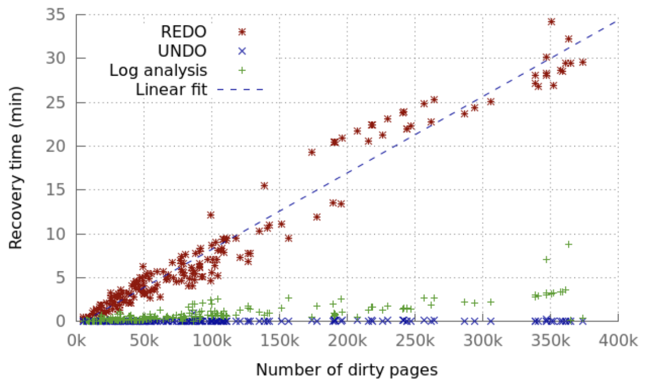

Figure 2: Number of dirty pages vs. REDO time

3. EXPERIMENTS

3.1 Environment

We performed the experiments using the Shore-MT storage manager [4], which implements the basic ARIES algorithm for logging and recovery. We used the TPC-C benchmark as implemented in the Shore-Kits component released with Shore-MT[2]. Our hardware is a 64-bit Linux 3.11.0 server with dual Intel Xeon X5670 CPUs, providing a total of 24 thread contexts. In all experiments, we use a flash device to store the database log, namely a 256GB Samsung SSD 840 Pro. As the database device, we use a Seagate ST1000VX000 hard disk, which has a capacity of 1TB. The page size is 8KB.

3.2 Results

Dirty pages vs. REDO time. To confirm our hypothesis that REDO time is determined almost exclusively by the number of dirty pages in the buffer pool at the time of the crash, we performed various TPC-C benchmark runs with different configurations. After a random amount of time between 1 and 5 minutes, a crash is simulated by killing the process and we count the number of dirty pages as computed by log analysis, as well as the duration of recovery. Figure 2, which is a detailed version of Figure 1, shows the result of each run as a point in the graph, with dirty page count in the x-axis and recovery time—divided into log analysis, UNDO, and REDO—in the y-axis. Because our goal is simply to study the correlation between dirty pages and recovery time, the configurations of individual runs is irrelevant.

The graph shows that REDO costs dominate recovery time, at least in typical short-transaction workloads such as TPC-C. In such scenarios, the UNDO costs are negligible, as shown in our experiment. Given the large amounts of log generated in high-throughput scenarios, it turns out that the cost of log analysis becomes significant, or at least more so than that of the UNDO phase. To keep the log analysis cost under control, it is enough to increase checkpoint frequency. Nevertheless, the most important result of this experiment is the strong correlation between the number of dirty pages in the buffer pool at the time of crash and the REDO time. The line represents a best fit for the data points collected, computed with least-squares linear regression. It gives us a cost of 4–6ms per dirty page, which is what one would expect for a random read on a modern hard disk. Note that this experiment does not write any pages, which means that the buffer pool is large enough to absorb all updates. If writes are required, the recovery cost per dirty page is expected to raise. Given the results obtained here, the following experiments consider the number of dirty pages as the sole determinant of crash recovery time.

[2]`https://sites.google.com/site/shoremt`

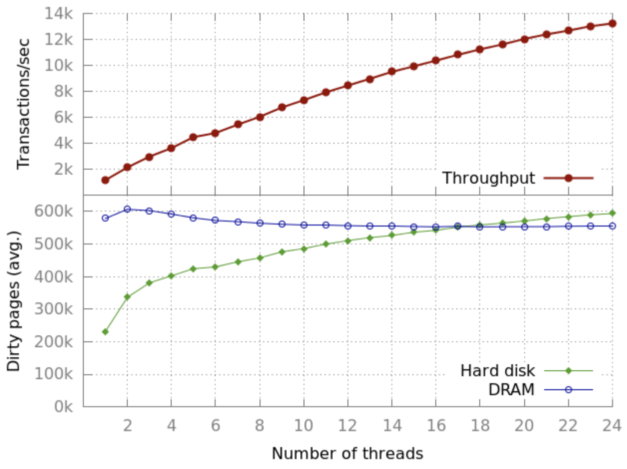

Figure 3: Effect of throughput on cleaning behavior

Maximum throughput. As discussed in Section 2.2, if the buffer pool (or at least the working set of the application) fits completely in main memory, then the database storage device can be fully exploited for flushing dirty pages using the page cleaner. Since copies of pages are taken into a separate write buffer prior to flushing, only the minimal overhead of latching a page for the duration of a copy is incurred. In this scenario, there is a "race" between in-memory transaction processing, which turns clean pages into dirty, and the page cleaner, which performs the opposite transformation. As the transaction throughput increases, the page cleaner is expected to hit a "saturation point", which is when pages get dirty faster (assuming non-skewed updates) than disk bandwidth allows them to get cleaned. To demonstrate this effect, we perform five-minute TPC-C benchmark runs on a warm buffer, varying the number of worker threads from 1 to 24.

Figure 3 shows the measurements of this experiment. On the x-axis, the number of worker threads is shown. The graph at the top shows the transaction throughput, which reaches about 13,000 transactions per second at 24 threads. On the bottom graph, we compute the average number of dirty pages in each run for two different scenarios: one for a hard disk with a write latency of ~4ms; and a second one which uses a ramdisk, meaning that pages can be flushed as fast as a DRAM copy operation. The total database size is ~8GB (1 million pages), which is given by a TPC-C scale factor of 64. As we can see, the amount of dirty pages increases with the throughput. At the best performing scenario, 60% of the pages are dirty on average, which means that recovery would have to perform 600 thousand random I/Os. In a single-disk scenario, this amounts to 40 minutes of REDO time. Notice, however, that the database used in the experiment is relatively small (~8GB). For a 1TB database (125 million pages), the expected REDO time under such high throughput would be about 90 hours!

We also ran the experiment with all database pages in a ramdisk, shown by the "DRAM" data series in Figure 3. In this scenario, the dirty page ratio stays always at about 55%,

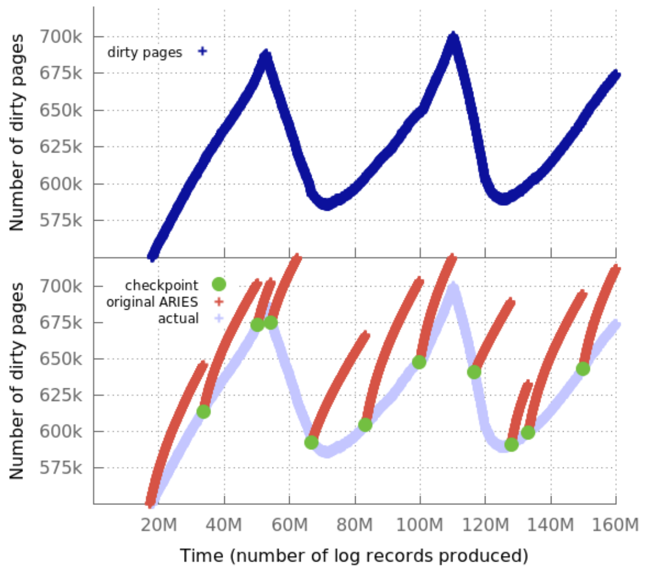

Figure 4: Dirty page ratio during experiment

with a curious increase for low-throughput configurations. The reason for that is the policy used by the cleaner, which is driven by log space reclamation and by checkpoints. Since we use a fairly large log (80GB), checkpoints are the main trigger for the cleaner, which means it is invoked at regular time intervals (30s), instead of eagerly collecting dirty pages for cleaning. Therefore, we conclude that the cleaner service should employ various policies depending on the workload and on I/O setup. We believe the implementation and evaluation of such policies would be a valuable contribution for future work.

To better understand what happens to the dirty page ratio in a single benchmark run, we collected the number of dirty pages at regular intervals and plotted it on a graph, shown in Figure 4. The x-axis shows a timescale in terms of log activity (data points are collected at every ten thousand log records). The data here was collected from the 24-thread experiment of Figure 3. The data points show the count of dirty pages as computed by log analysis. This means that, for each point, the y-axis value is the dirty page count that would be fetched from disk if a crash were to happen at the point in time given by the x-axis value. On the top graph, we plot the actual dirty page count, computed by inspecting the log file produced by the experiment. On the bottom graph, we plot the amount computed by log analysis following the original ARIES algorithm. The difference between them will be explained later on.

As the plot shows, the amount of dirty pages oscillates as the page cleaner is triggered. Once activated, the cleaner sweeps through the buffer pool to collect copies of dirty pages (not necessarily all of them), flushes them to disk, and goes back to idle. The points where such sweeps happen can be identified in Figure 4 by the decrease in the number of dirty pages. If the crash happens shortly after a sweep is completed, recovery time will be slightly shorter. On the other hand, if it happens right before the cleaner is triggered—i.e., when the graph is at a peak—longer recovery times are expected.

Logging page flushes. So far, our analysis emphasized the strong relationship between the number of dirty pages and REDO time. However, it is important to note that the ARIES design cannot precisely determine at restart time which pages were actually dirty and which not. The consequence is that some pages which are thought to be dirty after log analysis may be fetched only to realize that they do not require any REDO, i.e., they were clean at the time of crash. This happens because the dirty page set is computed by log analysis in two phases: first, a list of dirty pages is extracted from the checkpoint log records, and then this list is updated as the analysis moves forward and finds updates to pages which were so far considered clean. The inaccuracy comes from false positives in the list—pages which are cleaned but not removed from the dirty page list afterwards.

A simple technique to achieve significantly higher accuracy in computing the dirty page set is to log page flushes. The idea is to introduce a special log record that contains a list of page identifiers that were recently flushed. When log analysis finds such log records, false positives can be removed from the list of dirty pages, whereas without them the list can only grow. A concern with this technique, which was mentioned in the original ARIES design [7], is the increase in log volume. However, modern storage technology makes log volume less of a problem, especially if we consider a high-throughput scenario, where the number of page-flush log records is expected to be negligible relative to the amount of data-related log records. Furthermore, because these log records are not required for transaction consistency, they can be held aside and placed on fragmented space inside log pages. In this case, there is no increase in log volume at all.

To demonstrate the effect of page-flush logging, we refer back to Figure 4. The data points labeled "original ARIES" in the bottom graph correspond to the standard method of computing the dirty page set in ARIES, while the "actual" series shows the amount computed by taking page-flush log records into account. These points are the same as in the top graph, plotted again for visual comparison. To better understand the differences, we also plot the values computed by each checkpoint. Note that the set of dirty pages in the original ARIES procedure, as expected, increases monotonically from the starting amount computed by a checkpoint, while the optimized method allows the set to decrease along with page flushing activity.

Note that when the page cleaner is triggered, only the optimized method takes the decrease in the dirty page ratio into account. If a crash were to happen during these flush periods, many pages in the dirty set computed by ARIES can be false positives. Also note that the value is corrected when a checkpoint occurs, which means that the likelihood of higher false-positive ratios is increased as checkpoints become less frequent. As the bottom plot of Figure 4 shows, there are certain points in time (e.g., shortly after 60M on the x-axis) where about 150 thousand pages in the ARIES dirty set are actually false positives, resulting in many unnecessary random I/O reads and further worsening the problem of REDO time.

One last remark on page-flush logging is relevant for scenarios in which rarely-updated pages tend to linger in the buffer pool. If such pages occur as false positives in the dirty page set, the length of the REDO log scan will likely be larger than necessary, because isolated old updates may be found to already be reflected on the persistent database. In such scenarios, logging page flushes makes the length of REDO scan much closer to the minimum necessary.

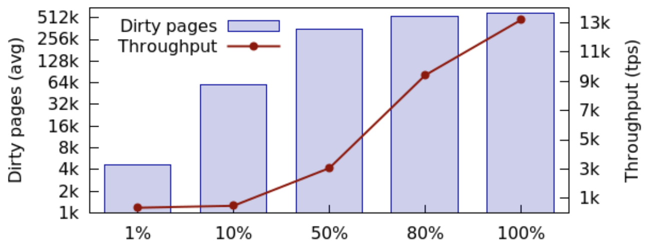

Figure 5: Average dirty page count vs. buffer size

Limited buffer pool. The maximum throughput scenario discussed above represents one extreme of the spectrum of available main memory for the buffer pool, namely when it is large enough to fit the entire database (or the complete working set). The next experiment analyzes the problem of recovery time on the other extreme, namely when a very small portion of the database pages can be cached in the buffer pool. Because this scenario involves intensive buffer replacement, a visible effect on transaction throughput is expected. This means that pages will get dirtied much less frequently than observed in the experiment of Figure 3, but at the same time, page replacement is likely to keep the database device busy, which limits opportunities for page cleaning. Therefore, even in such low throughput scenarios, it is safe to assume that recovery time is still an issue.

Figure 5 shows the results of an experiment involving one-minute TPC-C runs with varying buffer sizes on a dataset of 8GB (1 million pages). Along the x-axis (not to scale), we have different buffer sizes expressed as percentages of the total database size. On the left y-axis, whose values are plotted in bars, we show the average number of dirty pages throughout the experiment, computed with page-flush logging. The right y-axis, whose values are plotted as a line, shows the transaction throughput. We observe that smaller buffer sizes indeed require less recovery effort, since the absolute number of dirty pages is lower, but this obviously comes at the cost of a significantly lower transaction throughput.

4. CONCLUSION

The results presented in this paper clearly demonstrate that recovery time represents an important issue for database systems expected to deliver high availability. Even though many techniques can be employed to remedy that situation, the only way to guarantee fast recovery times is to trade it for transaction processing performance, a measure that goes against the historic trend that led to write-ahead logging and the omnipresent ARIES design. An even greater concern, as we demonstrated in our study, is that the problem is aggravated by hardware improvements such as larger memories and disk drives as well as multi-core CPUs. In other words, without new software techniques, the problem will continue to get worse, not better.

To solve this problem, we believe that logging and recovery algorithms must be designed to allow recovery work to happen concurrently with normal transaction processing and incur minimal overhead. Ideally, such an algorithm would make the time required for the system to be ready for new transactions independent of the amount of UNDO and REDO work necessary to restore a consistent database state. We hope that this can be achieved requiring only moderate modifications to the ARIES design already implemented by the majority of transactional systems.

Some improvements on that direction are already proposed within the context of ARIES itself. The original design [7] suggests the admission of new transactions as soon as the REDO phase is completed. This can be achieved if loser transactions reacquire their locks during REDO, which allows the UNDO phase to be performed by means of transaction rollbacks, i.e., concurrently with other transactions. As we established in our study, REDO costs dominate typical workloads, and thus a noticeable improvement is only expected if there are many long-lived transactions. Further improvements in ARIES [6] allow new transactions to be processed concurrently with the REDO phase, but with severe limitations. First, new transactions may not touch in-doubt pages. These are pages with an LSN value greater than the global commit LSN, which is the lowest LSN value produced by all active transactions. Furthermore, locks requested by new transactions may not conflict with those of loser transactions. These limitations imply that new transactions may not touch the working set of the application. Such transactions are, by the definition of working set, unlikely to occur. Therefore, new techniques must be proposed to achieve the goals laid out by our study.

Acknlowledgments

Pinar Tözün generously answered our questions about Shore-MT and Shore-Kits, which we appreciate very much. We are also grateful to Vitor Uwe Reus for his dedicated assistance in running the experiments.

References

[1] Effelsberg W, Härder T (1984) Principles of Database Buffer Management. ACM TODS 9(4):560–595

[2] Gray J, Reuter A (1993) Transaction Processing: Concepts and Techniques. Morgan Kaufmann

[3] Härder T, Reuter A (1983) Principles of transaction-oriented database recovery. ACM Computing Surveys (CSUR) 15(4):287–317

[4] Johnson R, Pandis I, Hardavellas N, Ailamaki A, Falsafi B (2009) Shore-MT: a scalable storage manager for the multicore era. In: Proc. EDBT, ACM, pp 24–35

[5] Kuno H, Graefe G, Kimura H (2013) Making Transaction Execution the Bottleneck. In: Databases in Networked Information Systems, Springer, pp 71–85

[6] Mohan C (1993) A Cost-Effective Method for Providing Improved Data Availability During DBMS Restart Recovery After a Failure. In: Proc. VLDB, pp 368–379

[7] Mohan C, Haderle D, Lindsay B, Pirahesh H, Schwarz P (1992) ARIES: a transaction recovery method supporting fine-granularity locking and partial rollbacks using write-ahead logging. ACM TODS 17(1):94–162